The Dissertation Committee for Hanbing Liu
certifies that this is the approved version of the following dissertation:

Formal Specification and Verification of a JVM and its Bytecode Verifier (draft)

Committee:

J Strother Moore, Supervisor

William Cook

David Hardin

Warren Hunt

Greg Lavender

Formal Specification and Verification of a JVM and its Bytecode Verifier (draft)

by

Hanbing Liu, B.E.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2006

Formal Specification and Verification of a JVM and its Bytecode Verifier (draft)

Publication No. ____________________

Hanbing Liu, Ph.D.
The University of Texas at Austin, 2006

Supervisor: J Strother Moore

How do we know that a bytecode-verified Java program will run safely? This dissertation addresses the question by building a precise model of the JVM and its bytecode verifier. We also built a "small" machine and its bytecode verifier to demonstrate an approach to solving this problem. We proved that for any program on the small machine that has been vetted by the small bytecode verifier, then that program will run safely on the small machine. We created substantial libraries of ACL2 definitions and lemmas towards specifying and proving that the JVM safely executes verified programs.

The fundamental problem is to connect the abstract execution of the bytecode verifier with the concrete execution of the JVM. These *diverge* in two ways: (1) the bytecode verifier executes on more abstract states and (2) its execution of INVOKE-family and BRANCH-family instructions differs from their execution by the JVM.

Our contribution was identification of a critical “on-track” property that, despite these divergences between the bytecode verifier and the JVM, enables one to use the success of bytecode verification to predict the safety of concrete execution.

The second difficulty is that the official specification describes many “procedural” aspects of the bytecode verification process. These aspects obscure the checks conducted by the bytecode verifier.

We introduce an alternative bytecode verifier without such “procedural” aspects. We use the new bytecode verifier as a stepping stone for proving that the official bytecode verifier is effective.

Following this methodology allowed us to prove, on our ”small” machine, that executions of bytecode-verified programs never overflow the operand stack.

We note that significant effort is required in order to extend this result from our “small” machine to the full JVM. We have formulated appropriate stronger notions of “safe” execution for programs on the full JVM. We introduced an alternative bytecode verifier. We proved the “reduction theorem” that relates the official bytecode verifier with the alternative bytecode verifier. We completed proofs of several thousand lemmas towards proving the “safe” execution of bytecode-verified programs on the full JVM. Our results are organized into supporting lemma libraries.

Contents

List of Figures

Chapter 1

Introduction

We face the challenge of building *good* (correct, useful, reliable, secure) software and hardware. Addressing the challenge, people design abstraction layers to separate concerns of different kinds. People use computer-aided tools to help them understand and build more complex artifacts within each abstraction layer.

In this dissertation, we study the Java Virtual Machine (JVM) [45] as an abstraction layer. We formalize the safety guarantee provided by this layer. We study how the safety guarantee may be correctly and efficiently provided by a specification compliant JVM implementation. We use a computer-aided reasoning tool, ACL2 [28], to help us explore these two aspects. We develop lemma libraries for reasoning about the JVM.

1.1 Background and Motivation

1.1.1 General Challenges

We face the challenge of building *good* (correct, useful, reliable, secure) software and hardware.

Everyday, our lives depend more and more on various computing devices. For example, not only are computers, cell phones, PDAs, and mp3 players computing devices, but cars, airplanes, and hospitals contain ever more computing devices as well. That is, we depend on computing devices not just for dispensable conveniences but for essentials. Sometimes our lives depend on the correct functioning of such devices.

The two challenges to building good software and hardware are:

- Individual systems are becoming more complex.

  For example, more transistors are being put on a chip; software is developed to do ever more tasks.

- With computing systems being connected to each other, a computing system may need to function in a hostile environment — because it is almost inevitable that the device is connected somehow to other devices under an attacker's control.

A major task of computer science and engineering is to manage the growing complexity and ensure the correct functioning of computing systems.

There are two major avenues for managing the growing complexity in hardware and software and assuring their correctness:

- Devising proper layers of abstractions to partition a problem and isolate different kinds of considerations to different layers;

  Development of virtual machines, high level programming languages, and programming libraries are examples of such layers.

- Using computer-aided tools to extend our capacity to manage complexity directly.

  Software development environments, compilers, simulation tools, computer-aided reasoning tools (model checkers, theorem provers) are among the tools that people use.

In this thesis, we study the Java Virtual Machine (JVM) as an abstraction layer. We formalize the safety guarantee of this layer. We study how the safety guarantee may be provided by a specification compliant JVM implementation. We use a computer-aided reasoning tool, ACL2, to do that. We develop lemma libraries for reasoning about the JVM. In particular, we study the effectiveness of class loading and class verification in proving the JVM safety guarantees.

1.1.2 Java and Java Virtual Machine as a Response

The Java programming language [16] and the Java Virtual Machine [45] emerged as responses to the challenge of creating reliable and secure software. They are designed for creating software that runs on a great number of hardware platforms in an interconnected (and possibly hostile) environment.

Java is designed for writing portable software. It is designed originally for creating portable software for embedded consumer electronic devices, such as cellphones and TV set-top boxes. Java programs are first compiled into an architecture independent class file format for executing on the Java Virtual Machine. The JVM is designed to provide the abstraction layer that hides peculiarities of different hardware platforms and operating system environments.

In addition to portability, more importantly, Java is the first widely available programming language for writing reliable and secure software. The Java language environment white paper [17] declares prominently:

> "The Java programming language is designed for creating highly *reliable* software .... Java technology is designed to operate in distributed environments .... With security features designed into the language and run-time system, Java technology lets you construct applications that can't be invaded from outside."

Java is designed to support the conventional security model [13]. The conventional computer security model generally talks about managing and enforcing security policies [3]. Security policies describe a set of actions that specific entities can perform. Policies are enforced by a small and reliable "reference monitor". Java allows software users to specify elaborated access control policies. An example policy may be that a Java program `P` from site `S` can read `A`'s file on the local disk only when the entity that executes `P` has successfully authenticated as `B` and possesses some suitable credential `C`. Besides allowing users to specify the access control policy, the execution environment of the Java programs guarantees that policies are always enforced. In addition to enforcing the users supplied policies, the Java language also guarantees that Java programs respect a strong set of default guarantees. Such guarantees are defined as an inherent part of the language. For example, objects are always used in a way consistent with their types and access permissions of data members and methods are respected.

The JVM plays a critical role in fulfilling the security promises of the Java programming language [46]. Java programs are compiled into the class file format for execution on the JVM. Before a class file is interpreted for execution on a JVM, it is examined by the bytecode verifier component of the JVM [45]. The bytecode verifier rejects any programs that are either predicted to be "unsafe" to execute or difficult to predict to be "safe". Assuming that a JVM only executes programs which the bytecode verifier predicts to be safe, a correct JVM implementation is expected to be able to enforce the security policy on program executions.

In summary, the Java language and the JVM have emerged as tools for building portable security systems that can operate in a distributed environment. They are designed to provide a clearly defined high-level abstraction layer that enables the application writers to focus on the actual problem domain — instead of spending their time on low level details such as memory allocation, enforcing security policies, and other orthogonal aspects of programming.

1.1.3 Why Study the JVM and the Bytecode Verifier?

The Java programming language and the JVM have been successful and popular. Java is publicized as "the first language to provide a comprehensive solution to the challenges of programming for the Internet, providing portability, security, advanced networking and robustness without compromising performance" [44]. Java and the JVM have largely lived up to their claims.

Portability, security, and rich supporting libraries have attracted many people to develop their programs in Java. SourceForge.net listed 17893 software development projects that use Java, more projects than in any other programming language (February 10, 2006). NASA/JPL uses Java in its Mars rover project, where Java is used for writing a substantial part of data visualization, collaboration, command and control software on the ground.

Java and the JVM also make strong security claims. As mentioned in the previous section, they are designed to allow software users to specify security policies (access control policies, what entities can do what kinds of operations to what resources). The JVM guarantees that the policies are enforced. In addition, Java and the JVM are designed to guarantee that operations are always applied to data of suitable types, respecting access permission declaration of their class, method, and data members. For example, if a class method M of C is marked with "private", only the objects of exactly type C can invoke method M — the access permission to M is respected.

It is non-trivial for a JVM implementation to guarantee that user supplied security policies will be enforced while still providing good execution efficiency. The JVM specification describes a delicate static type checking process as well as a set of dynamic checks that the JVM must do. The hope is that if these mechanisms are implemented properly, the JVM implementation can enforce the security policies without relying on any special hardware protection mechanism, while emulating the Platonic JVM efficiently.

Typically, to enforce a security policy on program executions, one needs a reliable and trusted "reference monitor" to execute in parallel with the program. The reference monitor examines the execution history, checks the next execution step against the policy, and aborts execution if the next step will violate some security policy.

A conventional reference monitor either executes on physically separate hardware, or the reference monitor relies on some hardware assisted mechanism (for example, privileged mode or hardware supported virtual memory mapping) to protect its own execution environment from being modified by the processes that it monitors [3].

However, JVM architects have designed the JVM specification so that it is possible to implement a JVM without relying on the availability of any hardware protection mechanisms. One major design goal of Java is portability. Programs written in Java should be "compiled once and run everywhere". Consequently, the JVM needs to be implementable on widest selection of hardware platforms. Not all hardware platform (especially, the consumer electronic devices for which Java was designed) have the necessary hardware-based protection mechanism.

The JVM bytecode verification mechanism is the central piece of the JVM design for permitting a pure software-based reference monitor implementation. It is the cornerstone of the JVM security architecture [46]. The JVM bytecode verifier is responsible for examining a program statically before a JVM implementation executes it. The bytecode verifier rejects any program whose runtime behavior is either predicted to be unsafe or difficult to predict. The intuition is that by only accepting well behaved programs, the security policy enforcing component can be implemented entirely in software and the JVM implementation can efficiently emulate the ideal JVM.

The bytecode verification mechanism is complex. The original description of bytecode verification in the JVM specification [45] is vague and complicated. This has led to a series of vulnerabilities in past JVM implementations

[13, 29].

In fact, to address complexity and performance issues in the bytecode verification algorithm as presented in the JVM specification [45], the JSR139 specification committee has produced a rigorous specification [11] for a simplified bytecode verifier in April, 2003, for the JVM that fits the CLDC (Connected Limited Device Configuration) profile [42]. The bytecode verification process is simplified and the specification is expressed formally as a set of derivation rules in Prolog.

It is both interesting and of great value to find out whether the new bytecode verifier specification can provide the safety guarantees. It is a good opportunity for applying a proof-based approach to reason about the soundness of the new specification. The central quest of this research is to formalize the safety guarantee of the JVM and to study how to prove that the new bytecode verification mechanism provides the safety guarantee.

1.1.4 Why ACL2?

Computing systems can be described as formal systems. We humans are limited in our ability reason flawlessly about hugely complex formal systems. We introduce abstract models to simplify the problems. We rely on computer-aided reasoning tools to extend and amplify our abilities.

ACL2 is the tool that we use to model the JVM and its bytecode verifier and to reason about the models. ACL2 is a programming language with precisely defined semantics [28]. Most importantly, it comes with an industrial strength automated theorem prover [20]. People build formal models of the computing systems [1, 15, 19] in ACL2. People also specify and prove properties of the models in the ACL2 system [14, 4, 33].

1.2 Contributions

The goal of the Java Virtual Machine (JVM) specification is to describe a standardized safe execution environment that application programmers (who write programs to run in the environment) and system programmers (who write the JVM implementations) can agree on. A *correct* JVM specification should ensure that a compliant implementation is safe.[1] A *useful* JVM specification should allow an implementation to achieve reasonable execution efficiency.

Among others, the current official JVM specification in English [45] contains the following type of information:

- the behaviors of the JVM operations under "safe" conditions,
- mechanisms designed with the intention to ensure JVM operations are always executed with their "safe" conditions met.

The safety guarantee of the JVM is declaratively asserted: (1) all reachable state meets a list of static and structural constraints and (2) all JVM operations are executed with their "safe" conditions met.

Given a JVM implementation, it is difficult to check whether it provides the safety guarantee as specified because we need to reason about *all reachable state* and consider all scenarios for executing each JVM operation. One must also trust that when the mechanisms described in the JVM specification

[1]A *correct* JVM specification should ensure that a compliant implementation is safe for both the programs executing on the JVM and the underlying system that runs the JVM implementation.

are implemented, the resulting JVM implementation will provide such safety guarantees. We view this as "weaknesses" of the current JVM specification.

This dissertation makes the following contributions addressing the above mentioned "weaknesses".

- It formalizes the safe execution explicitly in ACL2 by modeling a defensive JVM [5].

- It identifies and formalizes an alternative safety guarantee: (1) the JVM execution preserves a *consistent state* property and (2) the JVM operations can be *guard-verified* with the *guards* that we attached to them. [2]

  The alternative safety guarantee is better that (1) it is stronger and (2) it is more explicit and easier for JVM implementors to use, that is to check whether their JVM implementations are safe, they do not need to reason about all reachable states and consider all scenarios for executing an operation.

- It provides a framework for verifying that the bytecode verification mechanism (as published in JSR139 [11]) is effective – when the bytecode verification mechanism functions together with the JVM dynamic class loading and runtime checks.

[2] Writing *guard* is a systematic way (i.e. supported by ACL2) to define safe conditions for executing an operation. In order for a operation to be *guard verified*, one needs to prove that when operation is invoked with inputs that meet its guard, all sub operations invoked by it will be invoked with inputs that meet their specific guards. For details about ACL2 guards, see Chapter 3.

1.2.1 Detailed JVM Model

I completed an executable formal model, M6, of a realistic bytecode interpreter [15].

M6 is a formal, carefully-considered CLDC JVM implementation [42].[3] M6 implements the class loading and class initialization processes. M6 also implements field and method resolution. Details such as synchronization, object initialization, and exception handling are also faithfully implemented.

I implemented 21 out of 41 native methods in the CLDC class library. Java programs that do not use any of unimplemented native methods can be executed on M6.

Moore and I have used M6 as the formal operational semantics for studying properties of concrete Java programs [14]. Similar but simpler operational models [31, 32] have been used in the assertion-based Hoare style program verification approach [30] by Moore. A formal operational machine model, coupled with ACL2's support for symbolic simulation, permits inductive assertion style code proofs without the need for customized verification condition generator [30, 18]. The assertion-based approach has been applied to a single-threaded restriction of M6 by Moore [private communication].

1.2.2 Useful JVM Safety Specification

The official JVM specification describes the JVM safety guarantee in the following fashion:

- Identify the properties that *must* hold on a JVM state; assert that all

[3]CLDC stands for "connected limited device configuration" CLDC specification describes a profile of a cut-down version of the JVM for use in devices such as PDAs and cellular phones.

reachable states have these properties.

- Identify the preconditions for invoking JVM operations; assert that during program execution, all JVM operations are invoked with their preconditions met.

The JVM specification describes a list of properties of the JVM state and the preconditions for executing selected JVM operations. The JVM specification also asserts that the JVM will preserve the properties on the JVM state and that the preconditions will be met.

The list is useful to a programmer who writes programs for executing on the JVM, because the list describes the guarantees that a Java programmer can assume about executions on the JVM.

However, the specification is not useful in guiding people to write a safe and efficient JVM implementation. [4] The list of constraints only describe what is required, not how they may be provided efficiently.

- There is no guarantee that for any JVM state that has all asserted properties, the state produced by executing one more step will also satisfy all the assertions. It is not obvious that the mechanisms (class loading and class verification process) described in the JVM specification ensure that the above property holds.

  In order to show all reachable states have the specified property, a JVM implementor is obliged to identify a stronger property, which (1) is pre-

[4]One way to create a safe implementation is to check all the stipulated assertions and preconditions defensively before the JVM implementation executes any step. However the efficiency penalty is not acceptable. In fact, the JVM is designed with the intention that all such properties and preconditions can be proved to be true and thus the checks can be skipped.

served over a JVM execution step, (2) is satisfied by all valid initial JVM states, and (3) entails the official JVM safety assertions.

- There is no guarantee that when the preconditions of some top level JVM operation O are met, preconditions of the lower level JVM operation o will be met, where o is invoked as a sub-operation for accomplishing O. It is not obvious that the mechanisms described in the JVM specification ensure that all of the preconditions for every sub-operation will be satisfied.

  A JVM implementor is obliged to identify stronger preconditions for JVM operations, such that the desirable relation holds between the preconditions of the top level JVM operation and the preconditions of the lower level JVM operations.

This thesis presents a more useful specification of the JVM safety guarantee by identifying what is sufficient for a JVM implementation to provide the safety guarantees described in the official JVM specification.

The new safety guarantee is formalized as

- JVM executions preserve a global consistency predicate on the JVM states and
- JVM executions always meet the local safety assertions (*guard*) of low level JVM operations.

The global consistency state predicate is phrased as a conjunction of assertions on the JVM state. I expect that the consistency state predicate (1) is preserved over every JVM execution step, (2) is satisfied by the initial JVM states that one cares about, and (3) entails the official JVM safety assertions.

The local safety assertions are expressed as preconditions on the inputs of JVM operations. I expect that if preconditions on the JVM operation O are met, preconditions on the JVM operation o are also met, where o is a sub operation invoked directly as part of accomplishing O.

For a cut-down version of M6, M6', I structured an ACL2-checked proof that M6' satisfies this stronger specification. The proof depends on a substantial set of conjectures that we have not yet proved. We developed the lemma library for verifying M6 is safe and its bytecode verifier is effective.

1.2.3 Detailed Bytecode Verifier Model

I wrote an executable CLDC [42] bytecode verifier in ACL2.

The official CLDC bytecode verifier specification is given as a set of Prolog-style derivation rules [11]. Whether a bytecode program can be "verified" depends on whether a corresponding syntactic term can be derived or not.

The following is one derivation rule from the official bytecode verifier specification:

```
methodIsTypeSafe(Class, Method) :-
    methodAccessFlags(Method, AccessFlags),
    methodAttributes(Method, Attributes),
    notMember(native, AccessFlags),
    notMember(abstract, AccessFlags),
    member(attribute('Code', _), Attributes),
    methodWithCodeIsTypeSafe(Class, Method).
```

The rule encodes one sufficient condition for a method `Method` being "verified" in `Class` — when one can find suitable substitutions of `AccessFlags`, `Attributes`, _, such that every term on the right hand side of `:-` is derivable.

I built my executable bytecode verifier model by translating the Prolog-style derivation rules from the official CLDC specification into ACL2 functions. The translation is systematic and stylized. I solve the problem of mapping a complex declarative specification into a procedural program.

Being executable, the bytecode verifier model is useful as an algorithmic implementation of the declarative specification. I tested the bytecode verifier on the CLDC class library and JDK1.3 class library, as well as hand-crafted bytecode programs.

Written in ACL2, the bytecode model is amenable to mathematical reasoning. One simple property that I proved is that whether a method passes the bytecode verification is independent of the bytecode instructions contained in other methods. I also proved that the CLDC bytecode verifier can be *reduced* to a simpler bytecode verifier. That is if the CLDC bytecode verifier succeeds in verifying a bytecode program, the simple bytecode verifier will also verify the bytecode program[5]. The proof is presented in the section 7.3 of this dissertation. The third result that I proved is that a bytecode-verifier-like static checker guarantees that verified programs never overflow their operand stacks during program execution on a simpler machine. All proofs are mechanically checked by the ACL2 theorem prover.

[5]This proof is one important step towards proving our final goal that verified program executes safely on M6.

1.2.4 A Framework for Proving a JVM is Safe

I created a framework for proving that the JVM is safe and for demonstrating that a JVM can be efficiently implemented. Since M6 is my formal model of the JVM, the framework is in fact used for proving that M6 is safe and supplies some informal evidence that M6 can be efficiently implemented.

The framework is comprised of two parts: the overall approach and an ACL2 lemma library for pursuing the approach. The overall approach explains (1) how to formalize that M6 is safe and how to justify that M6 is efficient, (2) what "leaf-level" properties one needs to prove for each individual JVM instructions to show that M6 is safe and efficient. The lemma library configures the ACL2 theorem prover to prove the leaf-level properties.

I formulate the JVM safety guarantees as follows: (1) JVM executions preserve a global invariant, and (2) JVM executions meet the local safety assertions (preconditions for executing the JVM operations safely). Thus one may be misled to think that the approach for formalizing the M6 safety property and identifying proof obligations is as straightforward as follows:

- Formalize the M6 safety guarantee as:
  - M6 executions preserves a global invariant
  - M6 executions meet the local safety assertions
- Identify the leaf-level proof obligations as:
  - For each M6 instruction, when executed with its safety conditions met in a *consistent* state, the resulting state is also a *consistent* state. Here a consistent state is any state that satisfies the global invariant.

  - For each M6 instruction, if executed as part of a *verified* method and executed in a consistent state, the safety preconditions for executing the M6 instruction are always met.

However, in reality, the approach for formulating that an M6 is safe is more complicated. The global invariant, the one that I identified that a safe JVM must preserve, requires that values in the state are of proper types. The local preconditions for executing JVM operations also assert that the operands are of the proper type. However an M6 state does not maintain enough type information to express these requirements.

In order to state that M6 is safe, I first augment the structure of an M6 state with type information. I update the M6 operations to maintain the type information. I call such an augmented M6 state a *defensive JVM* (DJVM) state. I updated M6 operations as DJVM operations. Before executing a bytecode instruction, the DJVM checks the preconditions for executing the instruction. [6] I define the global invariant on the DJVM state and local safety assertions on the DJVM operations. In order to prove M6 is safe for executing verified programs, one needs to prove that

- When executing verified programs, M6 behaves like the DJVM with the extra type information stripped.
- The DJVM's execution is safe (in the sense that its execution preserves the global invariant and never violates local assertions).

[6]The approach of using a defensive JVM to characterize explicitly the safety requirement for executing an instruction is first explored by Cohen in his original work *Defensive Java Virtual Machine* [5]. My defensive JVM model is more realistic. I also identify the invariant that our DJVM will maintain. The preconditions that I identify for the DJVM operation are also stronger.

To argue that M6 is efficient (that is the JVM can be efficiently implemented on today's hardware platform), I would first urge the critic to examine the definition of the consistent state predicate and the preconditions for executing M6 operations. I would expect the critic to reflect on how such a JVM state can be correctly represented and how such JVM operations can be emulated on that representation. The critic will find that M6 states contain no extraneous data and its operations require no unexpected runtime checks.

In addition to describing how to formalize the concept of M6 being safe, the overall approach also describes how to reduce the big proof obligation that M6 is safe into proof obligations that correspond to leaf-level properties of the JVM instructions.

We observe that the key differences between a bytecode verifier's execution and DJVM's execution can be summarized as the following two *divergences.*

- **Divergence 1** The bytecode verifier sees more general states than DJVM, which sees *assignment compatible* ones, i.e. the type signature of the runtime states is no more general than the abstract state seen by the bytecode verifier.

- **Divergence 2** The bytecode verifier does not handle *INVOKE*-family instructions the way DJVM does. DJVM pushes a new frame and proceeds into the code for the invoked method; the bytecode verifier just assumes the invocation returns and proceeds to the next instruction in the caller's frame.

To prove that bytecode verification is effective in ensuring the JVM execution safety, we need to bridge these two divergences. We need to identify the close relation between reachable JVM states with the abstract state seen during the bytecode verification. We need to relate the result of the bytecode verifier's checks to the safety of executing the JVM operations when such a relation holds.

In our work towards proving that bytecode verifier is effective, the following two insights are worthy of special mention. They are the keys for reducing the proof that the JVM is safe into proofs of leaf-level lemmas about each JVM instruction.

- We need to identify an *on-track* property as one of the requirement for a *consistent state.* The on-track property asserts that the executions of verified programs remains within the state space explored by the bytecode verifier's execution. In particular, the *on-track* property requires that each call frame in a runtime state is *approximated* by the abstract state observed by the bytecode verifier at the corresponding `INVOKE`-family instruction.

  We need to show that the *on-track* property is satisfied by all the reachable states of the JVM while executing verified programs. We rely on this *on-track* property is preserved, to *bridge the Divergence 2* (page 18) between the JVM execution and the bytecode verification — that is although the abstract execution diverges from concrete execution, they are still closely related by our *on-track* requirement.

- To express the *on-track* property, we need to introduce an alternative bytecode verifier to separate the procedural aspects of the bytecode verifier specification from what is checked by a bytecode verifier. We need to prove a *reduction theorem* that asserts that all programs verified by the original bytecode verifier can be verified by the alternative bytecode verifier.

I have identified the following set of leaf level properties that one needs to prove about each JVM instruction. Take `AALOAD` for example:

- (1) The DJVM operation `execute-AALOAD` is *guard verified.*

  If the precondition for executing the `AALOAD` instruction — `AALOAD-guard` — is met in a DJVM state `s` , no precondition of any JVM operation will be violated as a result of invoking `execute-AALOAD` in state `s`

- (2) If the precondition for executing `AALOAD` is met, executing it will preserve the *consistent-state* property.

- (3) In a consistent state, if the DJVM's runtime check `check-AALOAD` succeeds, the preconditions for executing `AALOAD`, `AALOAD-guard`, will be met.

- (4) Extract the type information from the current state to form a type signature state. If the bytecode verifier asserts that it is safe to execute `AALOAD` in the current type signature state, the DJVM's runtime checking `check-AALOAD` will succeed.

- (5) If the bytecode verifier asserts it is safe to execute `AALOAD` in some type signature state, the bytecode verifier will assert it is safe to execute `AALOAD` in a more specific type signature state when both type signature states are *good* with respect to some *consistent* type hierarchy.

- (6) If the current state satisfies the global invariant and the program being executed is "verified", there exists some type signature state that is more general than the type signature state of the current state. The bytecode verifier is known to assert that it is safe to execute `AALOAD` in that type signature state.

- (7) If the precondition for executing `AALOAD` is met, executing the instruction on M6 produces the same behavior of executing the instruction on the DJVM, with type information stripped.
- (8) The type signature state of a consistent state is *good* and the type hierarchy encoded in the consistent state is *consistent*.

This research so far has produced an incomplete lemma library for assisting people to prove leaf-properties as listed above. I used the lemma library to prove the leaf level properties for a small subset of the JVM operations.

1.3 Limitations

We formalized the JVM safety guarantees. We carefully modeled the JVM mechanisms (runtime checking, bytecode verification, and dynamic class loading). We designed an approach for proving that the JVM mechanisms provide the safety guarantee. We created a lemma library for proving that the JVM is safe.

However, the usefulness of this work is limited by its incompleteness.

The JVM safety guarantee formulation is not complete. We defined a strong `consistent-state` predicate as the global invariant that a safe JVM shall maintain. However the consistent-state predicate is not strong enough to capture some desirable properties. For example, the consistent-state predicate does not assert that there is at most one active thread in any critical section protected by locks. We only defined strong safety preconditions for executing a small subset of the JVM instructions and preconditions on the related JVM operations. Although we do not expect new challenges in identifying the strong safety preconditions for executing the remaining JVM operations, it should be noted that it takes effort to identify the safety preconditions — because these preconditions need to be strong enough so that one can prove that they entail the safety preconditions of the lower level operations.

The overall approach has not been put to the test with the complete JVM model, M6. In experimenting with the overall approach, we defined a separate machine model, `Small`. The small machine resembles a much simpler JVM. Its state has a stack of call frames (instead of a thread table containing stacks of call frames) and a list of method definitions (instead of an elaborate class table). A call frame contains an operand stack and a local variable array. We defined a JSR139-style bytecode verifier for the `Small` machine. The desired (prototypical) safety guarantee is that verified code never overflows the operand stack. We succeeded in proving that verified code executes safely on the `Small` machine by following the approach described above. The complete proof has been mechanically checked by ACL2.

The lemma library for proving that leaf-level properties is incomplete. We have only used the lemma library to prove leaf-level properties for a limited set of instructions, `AALOAD`, `AASTORE`, `ANEWARRAY`, `ALOAD`, `ASTORE`, `GETFIELD`, and `IFEQ`. We have not used the lemma library to prove the leaf-level properties of other kinds of instructions such as `INVOKEVIRTUAL`, `ARETURN`, `ATHROW`, and `NEW`. Furthermore, the lemma library itself contains unproved lemmas. Such lemmas are explicitly marked with "skip-proofs".

We have not used the lemma library to prove leaf-level properties about the `INVOKE`-family instructions from the M6. We argue that we have sufficiently explored the two essential aspects about an `INVOKE` instruction separately. The first aspect is about the method resolution — how the bytecode verifier's static check ensures that dynamic method resolution will succeed at runtime. We argue that our lemma library for reasoning about the field resolution for executing a `GETFIELD` is a sufficient template of a lemma library for reasoning about the method resolution process. The second aspect of the `INVOKE` instructions is that the M6's execution diverges from the execution of the bytecode verifier (See *Divergence 2*, page 18). To execute an `INVOKE`, M6 creates a new call frame to represent the callee's execution state, while the bytecode verifier finishes the `INVOKE` in one step. We have faced the same kind challenge in bridging such difference between the `INVOKE` instruction on the `Small` machine and its static checker. We have addressed the problem in the `Small` machine by (1) introducing an alternative bytecode verifier (2) requiring an *on-track* property to be part of the *consistent state* criteria. We think that these two key elements of how we structure the main proof into leaf-level proofs are equally effective in bridging the difference between how M6 and its bytecode verifier executes operations of `INVOKE`-family.

The usefulness of this work is also limited by the complexity of the lemma library. To prove that a cut-down version of M6, M6' is safe while executing a verified program, the 130,600 lines of proof input to ACL2 is organized into 283 files. We have proved

The dependency graph (figure 1.1) between these files contains 1728 edges. On average, each file depends on 7.32 other files. Thus, our work establishes that verification of the safety guarantees is possible, but it does not make it routine or easy.

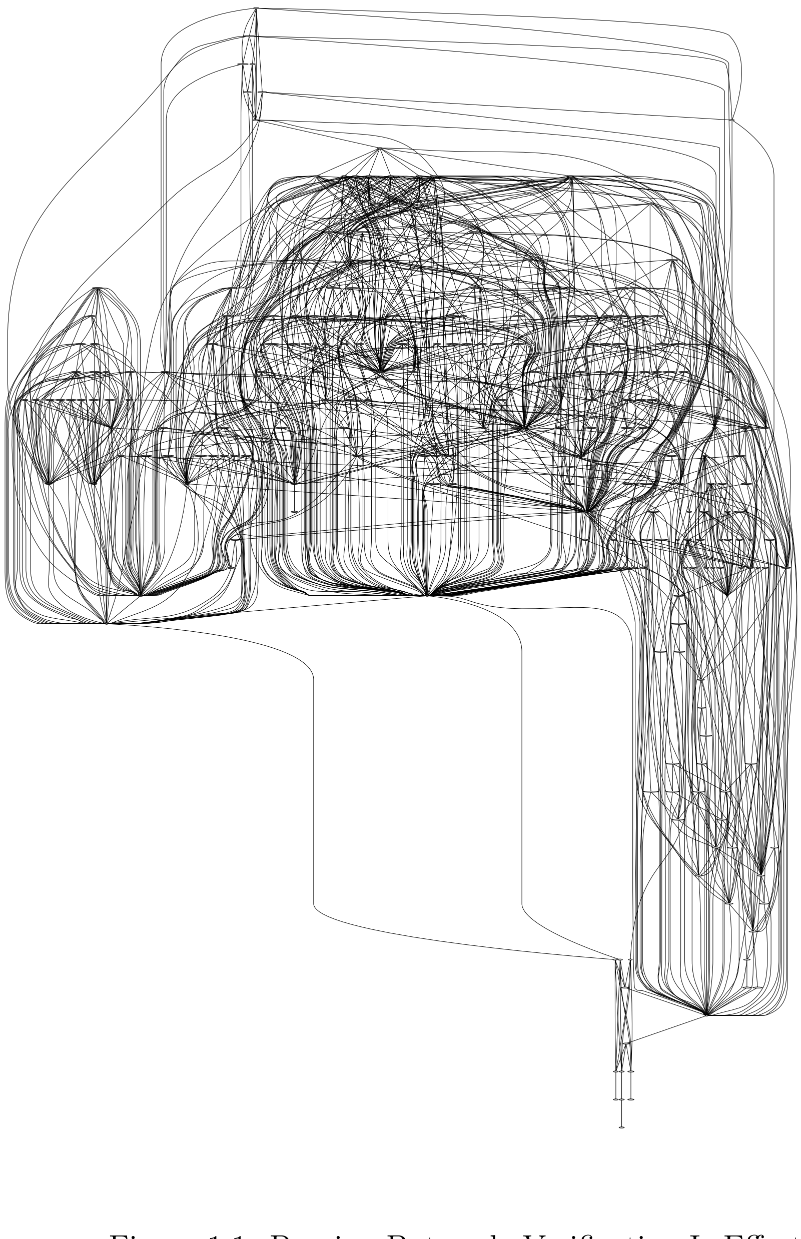


Figure 1.1: Proving Bytecode Verification Is Effective

We have finished the most academically-interesting aspects of the proof, though, some properties have not been proved. What remains to be done has more of an engineering flavor. The interesting aspects that we have done include (1) a proof decomposing scheme — tested to be effective on the simpler `Small` machine; (2) a lemma library for reasoning about the operations such as field resolution, dynamic class loading, read and write to call frames and heaps.

Should a researcher wish to apply the methodology to M6 in its entirety, I recommend that they first build a set of ACL2 books corresponding to commonly used abstract data types with operations that define and manipulate values of these types, as well as lemmas for reasoning about them. Then they should consider rebuilding M6 and the type checker more systematically with the data structures and operations of these abstract data types. Before they start, they need to understand and appreciate how to use ACL2 books to organize their proofs — importing, summarizing and exporting relevant lemmas. Our existing ACL2 books in `DJVM/INST/` from [22] may be a good case study to understand how to use books to organize proofs. Then they can follow the our framework to prove the leaf-level lemmas for each instruction. In the process, they will improve on our existing supporting libraries for proving these leaf-level lemmas. They should study our completed proof that the `Small` machine is safe, and understand how to put together the leaf-level lemmas to prove the top level goal that the bytecode verifier is effective and the JVM is safe for executing verified programs. In the concluding chapter (Chapter 8, page 302) of this dissertation, we also recommend a few suitable simplifications to M6. A researcher may consider those recommendations to simplify our JVM model before trying to prove that the bytecode verifier is effective.

1.4 Related Work

The work presented in this dissertation is related to several areas of research, including: (1) formalizing JVM semantics, (2) program specification and verification, (3) designing sound type systems for the JVM bytecode language, and (4) proving the correctness of the bytecode verifier.

Several people have formalized the JVM as a state machine [2, 5, 10, 35]. Semantics of the JVM instructions are given as state transition rules describing how the state is updated and conditions under which it is updated.

My JVM model, M6, is the most accurate formal JVM model that we know of. The model faithfully models class loading and class initialization, as well as synchronization operations at the JVM level. It can execute any bytecode program that does not rely on certain native methods from the CLDC class library. M6 is specified in ACL2.

JBook [35] specifies an executable JVM model of comparable complexity. Their model is presented as a set of state transition rules in ASM notation [7]. The model is executable by an interpreter, ASMGofer [39].

Compared to the JVM model described in JBook, we have the advantages of being able to specify the properties of M6 in ACL2. We can interact with the ACL2 theorem prover to find proofs of these properties and have the proofs mechanically checked by the ACL2 theorem prover.

Our effort towards identifying the consistent-state predicate is comparable to projects that aim to design *sound* type systems for the JVM bytecode language [41, 36, 34]. In a conventional type system, the typing rules will be rather complicated to capture the fact that the available class definitions can be dynamically extended. Whether a specific term (a machine state) is well-typed depends on delicate consistency requirements between subterms. One such consistency requirement can be that the subterm representing a value in the operand stack needs to be of a valid class according the runtime class table — another subterm. The evaluation relation of the bytecode language corresponds directly to our JVM model, M6.

By identifying the consistent state predicate, we are essentially defining what a well-typed term is in this type system. We want to first show that executing a verified (well-typed) program in some consistent (well-typed) state will ensure that preconditions for executing a step are met. This corresponds to the *Progress* requirement of a sound type system: well-typed programs will not get stuck. We also want to show that a verified (well-typed) program executing a step in a consistent (well-typed) state will yield a program and state is also consistent (well-typed). This corresponds to the *Preservation* requirement of a sound type system.

Unlike the efforts to define sound type systems for the JVM bytecode language, we started by constructing a faithful model of the JVM directly. The evaluation rules for the Java bytecode language are incorporated into the definition of the executable JVM model. Our executable JVM model is easily recognizable as a JVM. Our approach also exposes some low level details in the JVM — allowing us to study the important properties of the class loader inside the JVM.

We have aimed at proving a formal claim that roughly says "verified programs execute safely on the JVM". Stärk et.al. present a mathematical proof that their bytecode verifier model in their JBook [35] is correct. Klein and Nipkow have completed a mechanically checked proof of this nature for their model of the JVM and bytecode verifier [12, 21].

The uniqueness of our work is that we modeled the class loading process in the JVM. The type hierarchy information is explicitly encoded in the superclass field and superinterface field of loaded classes. We do not assume that loaded classes form a lattice with respect to some assignment compatible relation. Instead, we are obliged to prove explicitly that available classes form a lattice with respect to the assignment-compatible relation. We are also obliged to prove that dynamic class loading as specified in the JVM preserve this fact.

Another difference worth noting is that our bytecode verifier is derived directly from the official bytecode verifier specification [11]. We have strived to make the connection between the official specification and the ACL2 model as direct as possible. Our bytecode verifier is directly executable without needing a separate refinement step to create an executable bytecode verifier.

1.5 Organization

In Chapter 2, we give a simple introduction to the Java Virtual Machine and its bytecode verifier. We highlight the two styles of specification (operational and declarative) that informally coexist in the official JVM specification. We formulate our task of proving that the JVM is safe as to show that our operationally specified bytecode verifier ensures that all verified program execute in a way that meet declaratively specified constraints.

In Chapter 3, we give an extended introduction to ACL2, its programming language, mathematical logic, and mechanical theorem prover. We use a "Hanoi Tower" example throughout the chapter to explain the concepts and usage patterns. We explain in detail: (1) how to model an interpreter in ACL2, (2) how to write operational, functional, and safety specification for such an interpreter, and (3) how to use the ACL2 theorem prover to prove properties of functional correctness and safety. We introduce the ACL2 concepts of `skip-proofs` and `books` that one uses to organize a top-down proof. We describe the concepts of guards and guard verification as a method for specifying and verifying the interpreter safety.

We present our JVM model, M6, in Chapter 4. We describe its state representation and state transition functions. We describe our class loader and class initializer model in M6. We also touch on how the model can be used in proving properties about the Java programs.

In Chapter 5, we present our bytecode verifier model written in ACL2. It is constructed by systematically translating the Prolog-style rules from the official CLDC bytecode verifier specification into ACL2 functions. The bytecode verifier model is executable. We can also prove theorems about it. We present a simple property that we proved about the bytecode verifier. The property asserts that whether a method can pass the bytecode verification does not depend on how other methods are implemented.

We present our alternative JVM safety specification in Chapter 6. Our alternative JVM safety specification is originally motivated as a necessary intermediate step to prove that a JVM implemented by following the operational specification will meet the declaratively specified safety constraints inductively. We argue that our alternative JVM safety specification is a useful specification in its own right. It helps in guiding JVM implementors to create safe JVM implementations.

Our framework for proving that the JVM is safe and the bytecode verifier is effective is presented in Chapter 7. We start by describing the overall proof scheme. We describe the scheme using the `Small` machine. We show that how we proved that verified programs will never overflow the operand stack when executing on the `Small` machine. We then continue to present a "reduction" theorem that we proved about the CLDC bytecode verifier and an alternative version of the bytecode verifier. We also present our supporting ACL2 library for proving the necessary leaf-level lemma for instructions of M6.

We then summarize and conclude with some remarks on the limitations and additional work necessary for proving that M6 is safe for executing verified programs.

Chapter 2

Aspects of the Problem

We introduce the basic concepts of the Java Virtual Machine (JVM) and the JVM implementation. We describe the bytecode verification process briefly. We explain the scope of this dissertation — what we study and what challenges are.

2.1 JVM and JVM Implementation

Niklaus Wirth nicely summarized "programs" as being no more than "algorithms + data structures". More generally, *programs* can be understood as describing sequences of *operations* (algorithms) to be applied to *objects* (data structures) from a certain *universe*. *To execute a program on a machine* is to have the "machine" mechanically carry out the corresponding sequence of operations on the input objects.

For x86 assembly programs, the universe of discourse contains machine registers and the array of memory cells that record 0s and 1s. Machines for executing the assembled x86 programs are physical x86 computers.

In contrast, the universe of the Java bytecode programs includes entities such as operand stacks, local variable arrays, Java objects, and references to Java objects. These entities are generally much more complex than machine registers and memory cells. The *Java virtual machine (JVM)* is the machine that carries out the operations on these more complex entities.

In reality, such a Platonic virtual machine for executing Java bytecode programs does not exist. Instead, to execute a Java bytecode program, one resorts to representing the abstract entities as collections of 0s and 1s, and emulating the JVM operations by manipulating these low level representations. The methods for representing a JVM state and for emulating JVM operations are often coded as in a low level machine language (such as for an x86 computer). We refer to such a program as *a JVM implementation* on the low level machine.

A JVM implementation thus bridges the semantic gap between the universe of the JVM and the 0s and 1s of the low level machine. It is desirable for a *good* JVM implementation to have the following properties:

- *Correctness (Accuracy)*

  A JVM implementation shall behave like the JVM on all bytecode programs;

- *Safety*

  A JVM implementation shall always execute safely on the low level machine. No inputs (i.e. bytecode program and inputs to the bytecode program) can induce an execution that violates the safety constraints of the low level machine;

- *Efficiency*

  Additionally, a JVM implementation shall emulate the JVM with good efficiency for all bytecode programs.

Whether a good JVM implementation is feasible depends on the definition of the JVM — how different a JVM is from a conventional x86 computer. For example, emulating an x86 computer with a Turing machine will never be efficient, because x86 machines have random access memory, while the Turing machine only has a tape that can be accessed by moving its head one cell left or right [1]

It is the JVM designer's goal that the JVM design shall allow for a good (correct, safe, and efficient) implementation on the today's general purpose processors.

This dissertation is a formal study of the Platonic JVM – what are the guarantees of the Platonic JVM and how such guarantees are provided. We hope that this formal study of the JVM can provide useful insights for why a correct, safe and efficient JVM implementation is possible and what properties a JVM implementation needs to meet. This is not a study of efficient JVM implementation techniques.

2.2 JVM Specification and its Objectives

A specification of the JVM serves two purposes. Application programmers, who write Java programs to run on the JVM, rely on the specification to think about their programs and predict the programs' behavior. The system pro-

[1]Assuming that the Turning machine has a limited number of internal states, and the set of input symbols is small. By *efficient*, we mean a slow down by a small constant factor.

grammers, who write the JVM implementations, rely on the specification for descriptions of what operations need to be implemented and what constraints need to be met. For the specification to be useful, it should be feasible to implement the specified operations and meet the listed constraints. A specification will be particularly useful to JVM implementors if, when operationally specified operations are implemented, the declarative specified constraints will automatically be satisfied.

The official JVM specification (JVMSpec) [45] is written with consideration for the needs of both application programmers and system programmers in mind. JVMSpec specifies the behavior of the Platonic virtual machine both operationally and declaratively.

Operationally, JVMSpec describes the JVM as an interpreter. The interpreter recognizes a set of instructions and is able to execute the corresponding operations. These operations manipulate entities in the environments. The operations also cause the interpreter to update its own internal state, affecting what instruction will be interpreted next — some operations may cause the interpreter to locate the definition of a new program (dynamic loading) and prepare the new program for execution (linking).

The effects of interpreter's operations are described, however, only when the operations are used to manipulate entities of suitable types. The interpreter's behavior is not specified if the interpreter attempts to apply some operation to operands of wrong type. For example, JVMSpec does not specify how the JVM will execute the `IADD` instruction when the top item on the operand stack is not a JVM 32-bit integer. Nor does it specify how the machine will execute `IADD` when the operand stack contains one item.

If JVMSpec had only had the above operational specification, the specification would have been *incomplete* in describing the behavior of the ideal JVM [5]. An incomplete JVM specification is not useful to people who write programs for executing on the JVM. They cannot rely on the specification to predict the behaviors of their programs because:

- there is no obvious way to predict whether some program will induce the interpreter to apply an operation to operands of wrong type, and
- the result of executing that operation is unpredictable.

One way to make JVMSpec effectively complete, is to require that no program will ever induce the interpreter to apply an operation to operands of the wrong types. Section 4.8 of JVMSpec includes a list of constraints which *intends to* assert just that.

The following are a few constraints from JVMSpec.

- Each instruction must only be executed with the correct number of arguments with suitable types, regardless of the execution path that leads to its invocation.
- At no point during execution can the operand stack grow to a depth greater than the declared maximum size. At no point during execution can more values be popped from an operand stack than it contains.
- No local variable can be accessed before it is ever assigned a value.
- There must never be a reference to some uninitialized object on the operand stack or in a local variable array when any backward branch is taken.

- If GETFIELD or PUTFIELD is used to access a protected field of a superclass of the current class, the type of the instance being accessed must be the same as or a subclass of the current class. If INVOKEVIRTUAL or INVOKESPECIAL is used to access a protected method of a superclass, then the type of the class instance being accessed must be the same as or a subclass of the current class.

- The target offsets of jump and branch instructions never fall in the middle of an instruction.

The declarative aspects of the specification make a correct JVM difficult to implement. In general, given an arbitrary set of declarative constraints, it is not clear whether a correct implementation is possible at all – the set of constraints may be inconsistent.

To guide the JVM implementation, JVMSpec describes a bytecode verification process. Before the interpreter starts interpreting a program, it invokes the bytecode verification algorithm to examine the program statically. The hope is that any verified program's behavior is well-defined, that is, none of the declarative constraints are violated during execution.

However, a JVM implementation is constructed by following its operational specification (semantics of instructions and the bytecode verification process). The natural question is whether an implementation that complies with the operational specification also satisfies the declarative specification. This is one of the questions that we seek to answer in this work.

2.3 Bytecode Verification and its Goal

JVMSpec specifies the bytecode verification process operationally.

Before a method is interpreted by the JVM, several passes of static analysis of the code are carried out by the bytecode verifier [45, 11]. Some passes focus on the syntactical correctness of the class file. The most complex pass of the static analysis can be understood in the framework of abstract interpretation [6]. Instead of executing the program on a concrete state, a bytecode verifier executes on an abstract state. The underlying belief is that the properties obtained about the abstract execution is correlated to some corresponding properties of some concrete execution. In particular, if the abstract execution succeeds in the bytecode verifier, no constraints stipulated in the JVM specification will be violated during actual execution. In the language of abstract interpretation, the abstract interpretation of bytecode programs using the bytecode verifier is a *safe simulation* of the concrete interpretation [40] with respect to the properties that we are interested in. The abstract execution traces need to be a *consistent abstract interpretation* with respect to the concrete executions.

The abstract execution of a method starts from an abstract state. The initial abstract state is constructed from assumptions about the possible input to the method being verified. The abstract state records the types of values that would be present in the actual activation record (call frame). The abstract state also records other information such as the name of the current method, the method's permissions for accessing other classes, and the type hierarchy of the known classes.

Before the abstract interpreter makes a state transition, it checks the precondition for making the transition. If any errors are detected, the bytecode method is rejected and the JVM will refuse to execute the program.

The checks done by the bytecode verifier are closely related to the constraints that the JVM specification stipulates — except that these checks are performed on the abstract states. Comparing the following (incomplete) list of checks done by the bytecode verifier with the list of constraints that a JVM must met from the previous section, one can see a close match.

- Bytecode instructions are only applied to correct number of operands of compatible types.
- There are no operand stack overflows or underflows. The size of the operand stack depends on the offset of the instruction being executed, and is independent of how the instruction is reached during execution.
- The local variable being read must contain a valid type.

  Initially, the local variables are marked with a special signature that indicates an invalid type.
- When a backward branch may be taken, there is no uninitialized reference type on the operand stack or in the local variables.
- Accesses to protected fields and method to a superclass of the current class are properly limited.
- The targets of potential branches are *valid*, i.e. within the method, and never fall in middle of an instruction.

It is a goal of this dissertation to show these two kinds of checks are in fact related — if the checks succeed on the abstract state, the corresponding constraints on the runtime states are met.

The first challenge is to identify the conditions under which the two kinds of checks are related:

- Given a runtime state, identify when it is *approximated* by some abstract state — certain ill-formed runtime state will not have a corresponding abstract state that approximates it.
- Given a constraint on a well-formed runtime state, examine what is checked against its corresponding abstract state, and confirm that the checks are sufficient for satisfying the constraints on the concrete state.

Suppose one views an abstract state as a type for the runtime states that it approximate and that violating any constraint at runtime as is *getting stuck*. [2] Then the first challenge roughly reduces to designing a type system that has the *progress* property — well typed programs do not get stuck.

The second challenge is to show such a type system also has *preservation* property, that is, executing a step of well typed program produces a well-typed program. Given a runtime state known to have an abstract state that approximates it, we need to show that there exists some abstract state that approximates the state produced by executing one step from the given runtime state.

A third challenge is of a different nature. The official CLDC bytecode verification specification [11] describes the bytecode verification process with over 100 Prolog-style derivation rules. Although the specification is declarative

[2]Imagine a small-step semantics for the Java bytecode language; a program configuration is *stuck* if there is no valid transition from that configuration.

in form (as derivation rules), the verification process described there is very algorithmic. The essential reasons for the bytecode verifier being effective are obscured with the procedural aspects of description.

We need to create a simpler bytecode verifier that is equivalent to the algorithmic official CLDC bytecode verifier. The abstract execution of the simpler bytecode verifier needs to be directly useful for predicting the concrete JVM's execution.

Chapter 3

Using ACL2

The acronym "ACL2" stands for *A Computational Logic for Applicative Common Lisp*. The meaning of "ACL2" is overloaded. We use ACL2 to refer to three different things: (1) the ACL2 programming language – a dialect of the Lisp programming language; (2) the ACL2 logic — a first order logic with induction and equality [1]; (3) the ACL2 theorem prover — an interactive theorem prover for exploring the properties of ACL2 programs and proving these properties as theorems in the ACL2 logic.

The ACL2 programming language is described in Chapter 3 of the ACL2 book *Computer-aided Reasoning: An approach* [28]. We write the Java virtual machine models as ACL2 programs. The models are executable.

The ACL2 logic is what we use to formalize the properties of our JVM model and to conduct proofs in. Being a mathematical logic, it describes a set of axioms and derivation rules. Kaufmann and Moore's *A Precise Descrip-*

[1]ACL2's "logic" is technically both a first order logic and a first order theory, the latter being the former with the addition of a set of axioms for certain symbols such as `car` and `cons`. But in this dissertation we consistently refer to the mathematical formalism as a "logic."

tion of the ACL2 Logic [25] paper and *Structured Theory Development for a Mechanized Logic* paper [26] together is a definitive guide to the logic.

Our reason for specifying the JVM and bytecode verifier properties in the ACL2 logic is that there is a close connection between the ACL2 logic and the ACL2 programming language — it is possible to assign meaning to ACL2 logical formulas in a such way that: (1) the meanings of axioms turn out to be true facts in the universe of ACL2 programming language; and (2) the derivation rules are truth preserving.

Relying on this connection between the ACL2 programming language and the ACL2 logic, we study properties of our JVM model (an ACL2 program) by proving corresponding syntactic forms as theorems in the ACL2 logic. Given a property that we want to show about the JVM model, we first encode the property as a syntactic form — a formula — in the logic. If the syntactic form happens to be a theorem in the logic (i.e. it can be derived from axioms with repeated application of derivation rules), we can conclude that the ACL2 program does has the corresponding property.

We use the interactive ACL2 theorem prover to help us prove such theorems. The interactive ACL2 theorem prover functions like a simple-minded critic with great attention to details. The critic examines mathematical claims of the form "formula P is a theorem". It either verifies the claim or requests the user to supply additional justifications. We typically supply additional justifications of the form "formula Q is a theorem" where "Q" is simpler and thus more plausible to the critic than "P". The critic can "learn" from claims accepted and move on to check more difficult claims. Other forms of "additional justification" may be supplied by the user, including hints to enable or disable the automatic use of previously admitted definition or lemma (so-called "`:in-theory`" hints), hints to use some previously proved lemma via some explicit instantiation ("`:use`" hints), or even explicit proof scripts for certain subgoals ("`:instructions`").

3.1 Modeling

3.1.1 The ACL2 Programming Language

To communicate any thought about a computing system, we need to be able to first describe what the computing system is. Our language needs to be understood by the people to whom we want to communicate.

We often resort to describing a computing system in some non-ambiguous artificial language. We have the following requirements on the artificial language: (1) people can agree on how to interpret the basic constructs in the language; (2) the description is non-ambiguous; (3) the artificial language is rich enough to succinctly describe the system.

The ACL2 programming language is the simple, precise, expressive language that we use to describe computing systems in this study. Its parenthesis-laden syntax is very uniform. To add two constants, `1` and `2`, we write `(+ 1 2)`. To apply the operation `op` to the value of the variable `o`, we write `(op o)`. The first symbol after an open parenthesis can generally be viewed as denoting some operation, the rest of elements till the close parenthesis are terms describing the operands.

The operation of ACL2 is *side-effect free*. Take any `op` and a term `o`, executing `(op o)` will always produce the same output. ACL2 is a functional subset of the Lisp programming language. That is, `op` is a function. It is possible to understand `(op o)` as denoting some pre-determined value of the function `op` on its operand — without thinking in terms of the operation actively constructing the answer from the operand.[2]

If we view ACL2 programs as mathematical functions, the domain of every ACL2 function has at least five kinds of value objects: numbers, strings, characters, symbols, and cons pairs.[3]

We have many built-in ACL2 primitive functions. Function application `(if test a b)` is equal to `a`– if `test` is not equal to `nil` – otherwise, `(if test a b)` is equal to `b`. Function application `(cons a b)` is equal to the ordered pair (or "cons") containing the values of `a` and `b` respectively. For example, `(cons 1 2)` produces the pair that is written `(1 . 2)`.

To define a new operation, we use the special `defun` form.

```
(defun push (v stk)
  (cons v stk))
```

This syntactic form can be read as: define function `push` that takes two arguments `v` and `stk`, `(push v stk)` denotes a cons cell whose first element is `v` and the second element is `stk`. The intuitive picture is that the `push` operation pushes the value `v` onto some stack `stk`, and then returns the extended stack. The stack is represented as a cons pair.

We can also define functions recursively. The following defines the factorial function.

[2]For example, the programmer might think of `(expt 2 n)` as computing 2^n by multiplying 2 by itself n times or by using some other algorithm; but one can also think of `expt` as a function that maps from pairs of operands to values, as might be done via an infinitely large pre-determined table.

[3]The universe of ACL2 objects is open and so there may also be objects not of these five kinds.

```
(defun factorial (n)
  (if (zp n) 1
      (* n (factorial (- n 1)))))
```

One may view this recursive definition as explaining how to execute the function operationally — if `(zp n)` is not `nil`, executing `(factorial n)` returns 1, otherwise, it executes the operation again on a different input `(- n 1)`. When the execution of `(factorial (- n 1))` finishes, it returns the result of multiplying `n` times the result of executing `(factorial (- n 1))`.

One may also view this recursive definition as describing a constraint that relates that the value of function `factorial` at different points in the domain of the function. The constraint says that (1) function `factorial`'s value at point `n` is 1 if `(zp n)` is true; (2) otherwise, function `factorial`'s value at `n` is `n` times its value at `(- n 1)`.

After defining functions, we can group function definitions (and other information) into separate files, called ACL2 *books*. Each book defines a set of operations. The operations of one book may be "included" in another book by writing an appropriate `include-book` form in the latter book. For example, the `misc/records.lisp` file is an ACL2 book that comes with the ACL2 distribution. The book defines *get* and a*set* operations: `(g key dict)` and `(s key val dict)`. Before we can use it to define other functions, we need to use `(include-book m̈isc/records ̈:dir :system)` to import their definitions. We will show a concrete example of using this construct in the next section.

The ACL2 programming language diverges from the Lisp programming language by demanding that all operations must terminate on arbitrary inputs.

3.1.2 Modeling the Hanoi Tower Mover

We can write ACL2 programs to model interpreters. We take the "Towers of Hanoi" problem as an example to show how one may model a simple Hanoi Tower mover that can interpret a sequence of instructions and carry out the moves.

In the "Tower of Hanoi" problem, we have three pegs — `A`, `B`, and `C` — and `n` disks of different sizes. The state of the system is determined by the arrangement of the disks on the pegs. The disks are all initially on peg `A` with larger disks beneath smaller disks. The goal is to move all disks to peg `C`. We need to comply with the following rules:

1. Only the topmost disk disk on a peg may be moved at a time.
2. A disk can only be placed as a topmost disk and it must be smaller than the current topmost disk on the destination peg, if the destination is peg is not empty.

In ACL2, we model a Hanoi tower operator that can follow a sequence of instructions and carry out the corresponding operation as shown in figure 3.1.

The `play` function is our interpreter that takes a list of moves (instructions) and executes the moves one by one.

```
(defun play (moves st)
  (if (endp moves)
      st
      (play (rest moves)
            (do-move (first moves) st))))
```

```
------------------------------------------------------------
File: hanoi-model.lisp
------------------------------------------------------------
; The semicolon starts a comment that extends to the end
; of the line.

(in-package "HANOI")

(include-book "stack")
; defines stack: new-stack, stackp, push, top, has-more
(include-book "state")
; defines the state: new-state, statep, set-peg, and get-peg
(include-book "move")
; defines move: new-move, movep, src, and dest

(defun do-move (m st)
  (let* ((from (src m))
         (to   (dest m))
         (from-stk (get-peg from st))
         (to-stk   (get-peg to st)))
    (set-peg from
             (pop from-stk)
             (set-peg to
                      (push (top from-stk)
                            to-stk)
                      st))))

(defun play (moves st)
  (if (endp moves)
      st
      (play (rest moves)
            (do-move (first moves) st))))
```

Figure 3.1: Modeling a Hanoi tower operator

moves is assumed to be a cons structure representing a list. To play a sequence of moves, the interpreter first checks whether the list of moves satisfies endp. If the list has ended — (endp moves) — the interpreter returns the state st unmodified. Otherwise, the interpreter executes the first move with (do-move (first moves) st) updating the state, then plays the rest of moves until the end.

We represent the state st of the system as a record structure. Among others, the state data structure defines set-peg and get-peg operations. The move data structure defines src and dest operations. (src m) specifies the name of the peg from which the top most disk is to be removed. (dest m) specifies the name of peg to which the disk is to be placed.

The do-move operation defines how the interpreter executes one move.

```
(defun do-move (m st)
  (let* ((from-peg (src m))
         (to-peg   (dest m))
         (from-stk (get-peg from-peg st))
         (to-stk   (get-peg to-peg st)))
    (set-peg from-peg
             (pop from-stk)
             (set-peg to-peg
                      (push (top from-stk)
                            to-stk)
                      st))))
```

Let from-peg be the source peg that we are instructed by the move m to take the disk from. Let to-peg be the destination peg. We first set the destination peg in the state `st` to have one additional disk from the source peg with

```
(set-peg  to-peg
          (push (top from-stk)
                to-stk)
          st)
```

We then update the source peg with `(set-peg from-peg (pop from-stk) ...)` to indicate that the top disk has been removed from the stack. The resulting state is returned.

This definition of `play` has the implicit assumption that the sequence of moves supplied to it are legal moves — moves that do not attempt to put a bigger disk onto a smaller disk; moves that do not attempt to remove a disk from an already empty peg. Nor does `play` check whether the state is well formed, that is, the pegs themselves exist. In the next section, we will continue with this example, and show how we specify this type of implicit assumptions about the interpreter. We also show what are the desired properties of the interpreter and how we specify these properties in ACL2.

3.2 Formal Specification

3.2.1 The ACL2 Logic

It is not surprising that one can use a general-purpose programming language to write executable models of various computing systems. Nor is it surprising that one can write assertions that can be evaluated at runtime for checking against unexpected scenarios.

What makes the ACL2 programming language unique is its close connection with the ACL2 logic. The ACL2 logic is designed in such a way that the set of definitions in an ACL2 program corresponds a specific logical theory of the ACL2 logic. Theorems in that theory correspond to true properties of the ACL2 program. [4] Given a runtime assertion, it is possible to treat the assertion as a logical formula in the ACL2 logic. If the formula can be shown to be a theorem, one can conclude that the executions of the ACL2 program will not violate the assertion. More accurately, executing the ACL2 program will never violate the assertion if the syntactic form of the assertion is a theorem of a specific theory — a theory obtained by extending the *ground zero theory* with user-defined ACL2 functions from the program.

This connection between the semantics of the ACL2 programming language and the ACL2 logic is rooted in the choice of (1) the basic ACL2 axioms and axiom schemata and (2) the inference rules (including its induction principle), and (3) the definitional principle for extending a theory.

The remaining of the section gives a short description of the ACL2 logic and its connection with the ACL2 programming language.

The ACL2 logic builds on the traditional propositional calculus with equality. The ACL2 logic describes the *Propositional Axiom schema* and four inference rules often used to characterize the propositional calculus. Much of the material below is taken (with permission) from Kaufmann and Moore's *A Precise Description of the ACL2 Logic* [25].

[4]Logicians often use "valid" to mean true in *all* models. In a sound system, theorems (provable) correspond to valid properties of all models. In this dissertation, we use the ACL2 logic to reason about ACL2 programs. We assume the standard model of the ACL2 logic. So we often use "true" instead of "valid".

Axiom Schema (*the Propositional Axiom*).

$$\phi \vee \neg\phi$$

Rule of Inference.

- *Expansion* derive $\phi_1 \vee \phi_2$ from ϕ_2 ;
- *Contraction* derive ϕ from $\phi \vee \phi$;
- *Associativity* derive $((\phi_1 \vee \phi_2) \vee \phi_3)$ from $(\phi_1 \vee (\phi_2 \vee \phi_3))$; and
- *Cut* derive $(\phi_2 \vee \phi_3)$ from $(\phi_1 \vee \phi_2)$ and $\neg\phi_1 \vee \phi_3$.

The ACL2 logic includes the following axiom schemata and inference rule to capture the concept of equality.

Axiom Schema (*Reflexivity*)

$$x = x$$

Axiom Schema (*Equality Axioms for Functions*)

For every function symbol of arity n, we add:

$$((X_1 = Y_1) \rightarrow \ldots((X_n = Y_n) \rightarrow f(X_1 \ldots, X_n) = f(Y_1 \ldots, Y_n)))\ldots)$$

Axiom (*Equality Axiom for =*)

$$((X_1 = Y_1) \rightarrow ((X_2 = Y_2) \rightarrow ((X_1 = X_2) \rightarrow (Y_1 = Y_2)))).$$

Rule of Inference *Instantiation*

Derive ϕ/σ from ϕ, where σ is a substitution from variables to terms. [5]

[5] A substitution pairs distinct variable symbols, called the *keys*, with terms and the

In addition to the above axioms and inference rules from propositional calculus with equality, the ACL2 logic defines the other axioms and inference rules that are specific for formalizing the universe and operations of the Common Lisp programs.

For example, the first such axiom from *A Precise Description of the ACL2 Logic* [25] is `T` $\neq$ `NIL`. This axiom dictates that any model for the ACL2 logic has at least two distinct objects. Other axioms dictate the existence of other kinds of objects in the Common Lisp universe, such as strings — objects that make `(EQUAL (STRINGP obj) T)` true — but are distinct from `T` or `NIL`), numbers, and cons cells.

These additional ACL2 axioms and inference rules are chosen to specify the value of every evaluable term in the ACL2 subset of the Common Lisp language. The specified value — dictated by the axioms and inference rules — is expected to be consistent with the result of evaluating the corresponding term in Common Lisp. For example, axiom 47 (from [25]) states that for all `X`, `Y`, `(EQUAL (CAR (CONS X Y)) X)`. This axiom corresponds nicely to how the Lisp primitive `CAR` operates on a `cons` cell. With these axioms and inference rules, one can reason about the behavior of ACL2 programs that use built-in operations.

To use the ACL2 logic for reasoning about more complicated ACL2 programs, the ACL2 logic also introduce a definitional principle that describe how to extend a theory with axioms when a `defun` is accepted.

Adding a new `defun` form defines a new operation. In the ACL2 logic world, the addition of the `defun` form extends the logical theory (that corre-

notation "ϕ/σ denotes the uniform replacement in ϕ of the key variable symbols of σ by their corresponding terms. In the ACL2 logic, all variables are implicitly universally quantified. Thus, the procedure for instantiation is particularly simple.

sponds to the original program) by (1) adding a new function symbol with a fixed arity and (2) introducing a new axiom into the theory. For example, the previously discussed `defun` defining the `factorial` program

```
(defun factorial (n)
  (if (zp n) 1
      (* n (factorial (- n 1)))))
```

introduces a new axiom `(factorial n) = (if (zp n) 1 (* n (factorial (- n 1))))` into the current theory.

The expectation is that the extended theory with the new axiom can be used to predict the behavior of the program. One should note that some operational "definitions" can not be (and should not be) admitted. The following example shows an operation that should be not admitted into the logic. Suppose, ACL2 admits the following form

```
(defun f (a)
  (not (f a)))
```

the logical theory is extended by the corresponding axiom `(f a) = (not (f a))` will make the theory inconsistent — because we have reflexivity schema `x = x` and the conventional rules of inference for propositional calculus. Thus all well-formed formulas are theorems in this extended theory. The theorems in the theory will no longer correspond to true properties of the ACL2 programs.

The definitional principle of the ACL2 logic is designed to only admit `defun` forms that the extended theory can be used to predict the behavior of the corresponding program. The definition principle demands that all admissible ACL2 operations terminate on arbitrary inputs.

3.2.2 Kinds of Specifications

There are different kinds of specifications that we can write with the ACL2 programming language and the ACL2 logic.

Operational specification

We write ACL2 programs to model computing systems. Such an ACL2 program can be viewed as an *operational specification* for the computing system. As an operational specification, the ACL2 program precisely describes how the computing system shall behave. [6] An operational specification is useful to guide the implementation of the actual computing system.

Functional specification

Another kind of specification that we can write is a *functional specification.* A functional specification describes the desired effect of a computing system. Take the specification for a sorting algorithm for example. An operational specification may be the pseudo code for describing a sequence of steps involved in implementing the algorithm, and a functional specification may assert that the output of the algorithm is an ordered permutation of the input.

In ACL2, we can write precise functional specifications for ACL2 programs. We may even be able to prove that arbitrary executions of our ACL2 program will meet its functional specification — if we can prove that the syntactic form of the functional specification is a theorem. [7]

[6]Sometimes, people prefer to call such an operational model a *reference implementation* instead of an operational specification — especially when the operational model is efficiently executable.

[7]Note: some true properties of ACL2 programs may not be provable as evident from the Gödel's incompleteness result.

Safety specification

A *safety specification* is the third kind of specification that we can write with ACL2. If we view a computing system as a state transition system, states are nodes, transitions are arcs that connect one node to others. A safety specification may identify (1) a subset of possible states as "bad" states, and (2) the "guard" conditions for taking a state transition — conditions that may refer to the current state or even the history of transitions for reaching the current node. A safety specification will assert that execution of the state machine does not encounter any bad states and no state transition violates the guards.

We can write an ACL2 program that can check whether a state is "bad". The ACL2 programming language also has built-in support for attaching "guards" to the ACL2 operations, which we explain in some detail in the next section.

3.2.3 Guard and Guard Verification

Programmers write assertions in their programs. Assertions are evaluated at runtime. A correct program should not fail any of its assertions.

Well-written assertions capture the design intention. People can inspect them to understand the assumptions, guarantees, and invariants of a program and its components. Well written assertions make it easier to maintain the program.

The ACL2 programming language defines syntactic constructs for writing assertions. Programmers can attach *guards* (assertions) to the input of an ACL2 function. Guards can be any valid snippet of an ACL2 program. The guards can be checked at runtime. The following is an example.

INT32-ADD is an operation that implements 32-bit fixed-width addition.

```
(defun INT32-ADD (x y)
  (declare (xargs :guard (and (INT32p x)
                              (INT32p y))))
  (int-fix (+ x y)))
```

We attach a guard asserting that operands x and y are actually 32 bit integers.

Assertions are useful to programmers in debugging their programs. Instead of detecting an error late in the execution, failed assertions at runtime cause the program to "fail hard" early. Failed assertions help programmers pinpoint where their programs diverge from their intended behavior.

In conventional programming languages, the use of assertions is limited to the above two areas: testing and documentation.

By making use of the ACL2 theorem proving environment for ACL2 programs, assertions in ACL2 can be made directly useful in specifying and verifying the safety of the program execution.

As Dijkstra famously said, testing can only show the existence of bugs, not their absence. The most rigorous assurance we can get concerning assertions is to prove that assertions will never fail when the functions are used consistently with their guards. However, it is difficult to prove that a set of assertions never fails. Assertions are usually attached at different points in a program. These assertions encode what needs to hold when a program execution reaches the point. These assertions usually are not strong enough — satisfying an assertion, say the precondition for executing an top level operation, does not guarantee that the low level operations (for implementing top level operation) will have their their preconditions met.

To make it easier to prove that a set of assertions on a program never fail, we need to identify the set of assertions more systematically. The ACL2 system has a built-in mechanism for helping to identify the set of assertions and verify that the assertions never fail.

The ACL2 system can be configured to only accept programs if their guards can be *guard verified.* For a program to achieve the "guard verified" status, ACL2 demands that if the initial input satisfies the guard of the top level operation, all sub level operations invoked as a result of invoking the top level operation will have their guards met.

To verify the guard of `INT32-ADD`, one needs to show both the guard for operation `+` (in `(+ x y)`) and the guard for `int-fix` (in `(int-fix (+ x y))`) are satisfied when both `x` and `y` are `INT32p`.

ACL2 generates the following proof obligations:

- `(INT32p x)` $\wedge$ `(INT32p y)`
  $\Rightarrow$ `(acl2-numberp x)` $\wedge$ `(acl2-numberp y)`

  The guard of `+` demands both operand be `acl2-numberp` (i.e., to satisfy the predicate `acl2-numberp`, which recognizes the ACL2 numbers)

- `(INT32p x)` $\wedge$ `(INT32p y)` $\Rightarrow$ `(integerp (+ x y))`

  The guard of `int-fix`, which returns the lower 32 bits of an integer, expects the sole operand to satisfy `integerp`

In order to accept `INT32-ADD` as a guard verified operation, one needs to prove these two obligations. [8] In this example, the ACL2 theorem prover can

[8] In addition, one also needs to show functions such as `int-fix`, `+`, `INT32p`, and `acl2-numberp` are guard verified.

prove these two obligations automatically (without human guidance). In more complicated cases, an ACL2 user often needs to interact with the theorem prover — providing hints to help ACL2 to prove the guard conjectures.

If all operations described by an ACL2 program are guard verified, it is easy to prove that an execution will never fail any guard. It then suffices to prove that the initial state satisfies the guard of the top-most operation where the execution starts.

In our study of the Java Virtual Machine and its bytecode verifier, we specify guards for our JVM operations. We rely on ACL2 for generating proof obligations for guard-verifying the JVM operations. We prove theorems to relieve these proof obligations. The process of relieving proof obligations also guides us in identifying programming bugs and defining more accurate guards.

We have guard-verified the class loading operation in our JVM model.

3.2.4 Specifying the Properties of Hanoi Tower Mover

In the section 3.1.2, we described how to use the ACL2 programming language to implement the Tower of Hanoi example. We continue with this simple example to show how one may write operational, functional, and safety specifications in ACL2.

Operational Specification

The following is a simple recursive algorithm for moving a tower of height `n` from peg `A` to peg `C`, using peg `B` as a temporary peg

- If `n` is 0, there is nothing to do

```
(defun play-hanoi (from to temp n st)
  (if (zp n) st
    (let* ((st1 (play-hanoi from temp to (- n 1) st))
           (st2 (do-move (new-move from to) st1))
           (st3 (play-hanoi temp to from (- n 1) st2)))
      st3)))
```

Figure 3.2: Operational specification for Hanoi tower mover

- Otherwise, `n` is greater than 0, and we divide the task of moving tower of `n` disks into three small sub tasks to be executed in sequence:
  - First move the top most `n - 1` disks from peg `A` to peg `B`, using peg `C` as a temporary peg
  - move the one remaining disk on peg `A` to peg `C`
  - move the top most `n - 1` disks from peg `B` to peg `C`, using peg `A` as a temporary peg

To implement this algorithm, one may write the ACL2 program `play--hanoi` in figure 3.2, which may also be considered as an operational specification of the informal algorithm.

Functional Specification

There is a different solution for moving `n` disks by reusing our Hanoi tower operator `play` function. Instead of writing a recursive program that moves the disks on the fly, we write a "planner" algorithm. Given a task for moving Hanoi tower, the planner generates a sequence of moves for the interpreter `play` to carry out separately.

```
(defun h (from to temp n)
  (if (zp n) nil
    (app (h from temp to (- n 1))
         (cons (new-move from to)
               (h temp to from (- n 1))))))
```

`(h from to temp n)` generates a sequence of moves. It works as follows: if `n` is not an integer or it is not positive, `h` returns the empty sequence, otherwise, the returned sequence is the concatenation of three segments: (1) the sequence returned by `(h from temp to (- n 1))`, (2) the sequence that contains one move `(new-move from to)`, and (3) the sequence returned by `(h temp to from (- n 1))`.

The expectation for `(h from to temp n)` is that when one follows the sequence of moves specified by it, a tower of `n` disks will be moved from peg `from` to peg `to`.

With the additional definitions in figure 3.3. Our functional specification for the composition of the "planner" algorithm and the operator is:

```
(defthm hanoi-correct
  (equal (play (Hanoi n) (init-state n))
         (final-state n)))
```

where `(Hanoi n)` uses `(h 'A 'B 'C n)` to create a sequence of moves; `(init-state n)` creates a state with a tower of `n` disks on peg `A`; and `(final-state n)` creates a state with a tower of `n` disks on peg `C`.

Safety Specification

To specify the execution safety of the "Tower of Hanoi" operator, we need to first separate the good states from the bad states. When a bigger disk is

```
(include-book ‘‘hanoi-model’’)
(defun app (a b)
  (if (endp a)
      b
    (cons (car a) (app (cdr a) b))))

(defun h (from to temp n)
  (if (zp n) nil
    (app (h from temp to (- n 1))
         (cons (new-move from to)
               (h temp to from (- n 1))))))
; ‘‘the planner’’

(defun Hanoi (n)
  (h 'A 'B 'C n))

(defun tower (n)
  (if (zp n)
      nil
    (app (tower (- n 1)) (list n))))

(defun init-state (n)
  (s 'A (tower n)
     (s 'B nil
        (set-peg 'C nil nil))))
; Initial state has a tower of n disks on peg A

(defun final-state (n)
  (s 'A nil
     (s 'B (tower n)
        (s 'C nil nil))))

(defthm hanoi-correct
  (equal (play (Hanoi n) (init-state n))
         (final-state n))
; Functional specification
```

Figure 3.3: Functional specification for Hanoi tower mover

placed on top of a smaller disk in a state, the state configuration is obviously a bad one. Additionally, the set of disks on the pegs shall be the same set of disks — no disks are missing nor are new disks added.

We define the good state as follows

```
(defun safe-state (st n)
  (and (disks-perm st (tower n)); no missing disks nor new ones
       (disks-inorder st)))     ; stacks of disks are in order
```

where `(disk-perm st (tower n))` asserts that lists of disks from peg `A`, `B`, `C`, together form a permutation of the original list of disks of `(tower n)`.

```
(defun disks-perm (st l)
   (perm (app (get-peg 'A st)
              (app (get-peg 'B st)
                   (get-peg 'C st)))
         l))
```

`(disks-inorder st)` asserts that disks on all three pegs are in order.

```
(defund disks-inorder (st)
  (and (stack-inorder (g 'A st))
       (stack-inorder (g 'B st))
       (stack-inorder (g 'C st))))
```

For a sequence of moves to be safe, we expect that all intermediate states satisfy `safe-state`. Before we can talk about properties of all intermediate states, we need to define a "monitor" that can record the intermediate states.

```
(defun collect-states (moves st)
  (if (endp moves)
      nil
    (let ((nx (do-move (car moves) st)))
      (cons nx (collect-states (cdr moves) nx)))))
```

`Collect-states` behaves like our original Hanoi tower operator `play` on page 47. However, instead of returning the final state as `play` does, `collect-states` collects all the intermediate states into a list.

We formalize the requirement that all intermediate states are safe with the following ACL2 form.

```
(defthm safe-play
  (all-safe-states (collect-states (Hanoi n) (init-state n)) n))
```

We assert that if we start from the initial state with `n` disks, executing according to the plan `(Hanoi n)`, and collect all the intermediate states into the list `(collect-states ....)`, then all such states are safe states, that is `(all-safe-states ...)`.

As shown above, we have formalized the safety requirement for the Hanoi tower operator and the planner as a formula in the ACL2 logic. If we can establish the formula is theorem, we have shown that our model of the Hanoi tower operator and the move planner is safe.

Verifiable guards as safety specification

In this simple example, the above safety requirement (that all intermediate states are good states) is, perhaps, assuring enough to most people. It captures the very important property that we want.

However, many desirable (though perhaps less essential) properties are not directly addressed by the simple `safe-play` theorem. For example, how do we know that during the execution of the moves, the `POP` operation is never used for removing a non-existent disk from an empty peg? To prove this new property, we could write a different "monitor" that operates like "collect-states" but collects all the stacks that a `POP` operation is ever invoked upon. One is then obliged to show that each such stack has at least one element in it.

When we are specifying more complicated computing systems, using one theorem or a few theorems becomes less assuring — a few theorems are not likely to cover many of the desirable safety requirements. The approach for expressing a safety requirement — by writing "monitors" that record the inputs to operations — also becomes rather tedious. We need a more systematic way for writing down such requirements.

We write the ACL2 guards (assertions) to specify the safety requirement for executing individual operations. We make use of ACL2's guard verification mechanism to ensure that the set of guards are comprehensive.

Recall the guard conjectures generated in the `INT32-ADD` example of section 3.2.3. Similarly, in our Hanoi tower example, for every operation that we introduce, we attach a guard assertion that identifies the safety conditions for executing the operation. Take `do-move` for example:

```
(defun do-move (m st)
  (declare (xargs :guard (and (statep st)
                              (legal-movep m st))))
  (let* ((from (src m))
         (to   (dest m))
```

```
          (from-stk (get-peg from st))
          (to-stk   (get-peg to st)))
     (set-peg from
              (pop from-stk)
              (set-peg to
                       (push (top from-stk)
                             to-stk)
                       st))))
```

In order to execute `do-move`, we expect that

- `(statep st)` — `st` represents a Hanoi tower state;
- `(legal-movep m st)` — `m` is a legal move in state `st`

Let us assume

- the guard for `(src m)` is that `m` satisfies `movep`.
- the guard for `(get-peg peg st)` asserts that `st` satisfies `statep`, and `peg` satisfies `pegp`.
- the guard for `(pop stk)` asserts that `(stackp stk)` and `(has-morep stk)`
- the guard for `(set-peg peg stk st)` asserts that `(pegp peg)`, `(stackp stk)`, and `(statep st)`.
- ...

To guard verify `do-move`, ACL2 generates and must prove formulas that assume `(statep st)` and (legal-movep m st) and conclude with the following claims:

- `(movep m)`

  — to meet the guard of the sub-operation `(src m)` and `(dest n)` used for implementing `do-move`.

- `(statep st)` ∧ `(pegp (src m))` ∧ `(pegp (dest m))`

  — to meet the guard of `(get-peg (src m) st)` and `(get-peg (dest m) st)`

- `(has-morep (get-peg (src m)) st)`
  ∧ `(stackp (get-peg (src m)))`

  — to meet the guard of `(pop (get-peg (src m) st))`.

- `(pegp (dest m))` ∧ `(pegp (dest m))`
  ∧ `(stackp (push (top ...)  (get-peg (dest m) st)))`
  ∧ `(statep st)`

  — to meet the guard of
  `(set-peg (dest m) (push (top ...)  (get-peg (dest m) st)) st)`

- ....

Identifying a verifiable guard for an operation is difficult. First we need to identify a set of necessary safety conditions for executing the operation. We then need to strengthen the conditions for the current operation so that we can prove that all low level operations that the current operation invokes will have their guard met, assuming the strengthened safety condition is met. We also need to make sure that the strengthened guard condition is not unnecessarily strong — because choosing an unnecessarily strong guard of an operation may make it is impossible to define verifiable guards for higher level operations that use the current operation.

The verifiable guard for a recursive function that invokes itself is particularly difficult to write. For example, to define a verifiable guard for the `play` function,

```
(defun play (moves s)
  (declare (xargs :guard (play-guard moves s)))
  (if (endp moves)
      s
    (play (cdr moves)
          (do-move (car moves) s))))
```

we need to show:

```
(play-guard moves s) ∧ (consp moves) ⇒
                  (statep s) ∧ (legal-movep (car moves) s)
                ∧ (play-guard (cdr moves) (do-move (car moves)))
```

To define a suitable guard for an operation, we often start with a conjecture. We then attempt to guard verify the operation. The failed guard verification attempts often point to conditions for strengthening the initial conjecture. The process of identifying a suitable guard is an iterative process which often involves a few rounds of interaction with the theorem prover.

In summary, the safety specification for the Hanoi tower mover can be specified as two properties:

- All intermediate states are safe states:

  ```
  (defun safe-state (st n)
  ```

```
      (and (disks-perm st (tower n)); same set of disks
           (disks-inorder st)))   ; stacks of disks are in order

    (defthm safe-play
      (all-safe-states
         (collect-states (Hanoi n) (init-state n)) n))
```

- The following function guard-witness can be guard-verified.

```
     (defun guard-witness (n)
        (declare (xargs :guard (integerp n)))
        (play (Hanoi n) (init-state n)))
```

A defensive play – a hint at things to come

Another way we could approach the safety assertion for play is to define a "defensive" version of play that returns nil (instead of a normal state) if it detects anything "unexpected" during the execution of the moves. It might look like this:

```
(defun dplay (moves s)
  (declare (xargs :guard (play-guard moves s)))
  (if (endp moves)
      s
     (if (and (good-state s)
              (legal-movep (car moves) s))
        (play (cdr moves)
              (do-move (car moves) s))
        nil)))
```

Let us say that `dplay` "raise a red flag" when it returns `nil`.

We might then define a move checker that inspects a list of moves to insure certain very subtle properties are met. We might then prove that

- `dplay` *is safe*
  - `dplay` preserves the invariant of `good-state`, and
  - `dplay` is guard verified.
- *static checker is effective*
  - when the move checker approves a list of moves, that `dplay` does not "raise a red flag", and
  - when `dplay` does not "raise a red flag", its result is the same as `play`'s in all steps.

We argue that this is a good safety specification.

Firstly, one can inspect the definition of `dplay` to confirm that the `dplay` will in fact "raise a red flag" in all unexpected scenarios.

Secondly, if a thorough inspection were not assuring enough in itself — because the `dplay` definition had been too complicated, the safety specification also demands that (1) `dplay` preserves a `good-state` property and (2) `dplay` is guard verified. One can inspect the definition of `good-state` and the definitions of guards (both of which will be simpler than `dplay` itself) to check that the two theorems capture the expected safety requirement.

We argue that our *move checker is effective* specification is also good. It shows that the runtime checks of `dplay` never fails while executing *verified* programs. It shows that `play`, an efficient version of `dplay`, produces the same results as the inefficient `dplay` while executing verified programs — that is (1) `play` also maintains the `good-state` property and (2) `play` does not violate the runtime guard of any of the operations that it invokes.

This introduction of `dplay` to express safety is quite analogous to what we will do with the JVM. In this analogy, the move checker is the bytecode verifier, `dplay` is the defensive JVM, and `play` is the efficient model of the JVM. The reason we did not approach safety this way in the Hanoi example is that it is not easy to define a static checker for a list of moves that does not make quite strong assumptions about the initial state.

Now we return to the main flow of the discussion of *Using ACL2*, we continue on to explain how to use the ACL2 theorem prover to prove properties of ACL2 programs.

3.3 Formal Verification

3.3.1 The ACL2 Theorem Prover

We use the ACL2 theorem prover to check whether a formula is a theorem. As we explained briefly before, the ACL2 theorem prover functions like a simple-minded critic who pays great attention to details, "learns" from previously proved theorems, and can take certain kinds of hints and advice.

One essential component of the ACL2 theorem prover is a symbolic *rewriting engine*. The rewriting engine rewrites one term into another by applying *rewrite rules* of the form `LHS = RHS`. To rewrite a *target term*, x`LHS`, the rewrite engine matches x`LHS` with the pattern, `LHS`, if possible, finding a substitution σ for the variables in `LHS`. Provided such a σ can be found, the occurrence of x`LHS` is replaced by `RHS`$/\sigma$. Often the rewrite rules also contain hypotheses that need to be *relieved* before a term of the shape `LHS` can be replaced with another term of shape `RHS`. Rules with hypotheses are called *conditional rewrite rules.* The ACL2 rewriter attempts to relieve hypotheses by rewriting them (under the given σ); if a hypothesis rewrites to `T`, it is relieved, and if all the hypotheses are relieved, the replacement of x`LHS` by `RHS`$/\sigma$ is done.

The ACL2 theorem prover maintains a database of available rewrite rules to use with the rewriting engine. The theorem prover's ability to prove a difficult theorem depends on the set of rewrite rules that it has.

When we are using the ACL2 theorem prover to reason about a specific model, we also need to extend the database by adding rewrite rules specific to the model. The theorem prover maintains the property that if it rewrites a formula using rules from the database, then the rewritten formula will be logically equivalent with the original formula.

Instead of allowing a user to introduce arbitrary rewrite rules directly into its rule database, the ACL2 theorem prover demands the proposed rewrite rule are phrased in terms of a claim. Only after the theorem prover can accept the claim is a theorem (possibly via rewriting using existing rules), the ACL2 theorem prover will then derive a corresponding rewrite rule from the claim and add it to the rule database.

This description of the ACL2 theorem prover is an over-simplification. ACL2 also uses other techniques to prove theorems. Such techniques include type-based reasoning, decision procedures for proving linear arithmetic lemmas, as well as the use of induction to prove theorems.

3.3.2 Proofs, Skip Proofs and Books

Proofs in ACL2

To prove that a formula `P` is a theorem, we interact with the theorem prover to find a sequence of lemmas that guide the theorem prover to eventually accept that `P` is a theorem. We often refer to this sequence of lemmas as an *ACL2 proof* of the theorem. One should note that an ACL2 proof is not a formal proof. [9] However, for any theorem that ACL2 verifies, we believe that a corresponding formal proof exists.

Skip proofs

The process for proving a theorem with the ACL2 theorem prover often naturally calls for a top down approach. To prove a top level theorem, we often need to "convince" the prover that it is valid to use certain specific rules by proving additional theorems; to prove these secondary theorems, we may need to repeat the process to identify necessary rules for proving these secondary theorems, and then prove even lower level theorems so that these new rules can be added.

Skip-proofs and *books* are the ACL2 theorem prover's mechanisms for supporting the top down process of guiding the theorem prover to prove a theorem. *Skip-proofs* is a construct that an ACL2 user uses to experiment with different rules and study what kinds of rules are useful for the prover to prove the goal theorem. *Books* are a mechanism allowing an ACL2 user to organize a set of rules into a module (a knowledge base) for proving a certain

[9]A formal proof of some theorem is a valid derivation that is built from the axioms and repeated application of rule of inferences.

```
(skip-proofs
  (defthm skip-proof-example
    (implies (h x y)
             (equal (foo x y z) (bar y z)))))
```

Figure 3.4: Using `Skip-proofs`

kinds of theorems.

The ACL2 theorem prover relies on the rules from its database to prove theorems. An ACL2 user guides the theorem prover to prove a goal theorem. She has two distinct tasks: (1) identify useful rules that the ACL2 theorem prover can use to prove the goal theorem, and (2) convince the theorem prover that it is sound to use these rules.

The skip-proofs mechanism allows an ACL2 user to focus on the first task. One could wrap the key-words `skip-proofs` around a `defthm` term, and the ACL2 theorem prover will accept the theorem as if it has been proved and add the corresponding rule to its rule database. For example, once we submit the following term (figure 3.4) to the ACL2 theorem prover, the theorem prover will have one additional rule that it can use to rewrite any term of shape `(foo x y z)` into `(bar y z)`, if it can show that `(h x y)` is not `nil`.

By "skip-proofing" some of the theorems, an ACL2 user can experiment with possible rules and understand how the prover will use these rules to prove the top level theorem.

By using skip-proofs, an ACL2 user also make very explicit what is assumed about the model. If our final theorem asserts that the computing system has a certain property, the skip-proofs often assert what are assumed about the components, low lower level operations of the model.

Lemma Books

The ACL2 theorem prover relies on the rules from its database to prove theorems. However, it is not true that the more rules that the database has, the more efficient or effective the ACL2 theorem prover will be in proving new theorems.

Adding many irrelevant rules will always slow down the prover in proving new theorems. Furthermore, different rules may interfere with each other — different rules may guide the theorem prover to follow conflicting strategies in proving a theorem.

As it is often the case, the strategy for proving a top level theorem can be quite different from the strategy for proving a supporting lemma. We need to separate the supporting rules used for justifying adding a target rule from the proof where the target rule will used. We do not want the rules for proving a lemma to interfere with rules for proving top level goal theorem.

The ACL2 theorem prover describe a module system called *books*. An ACL2 book exports a set of function definitions (as we explained earlier) and rules. The theorems for justifying the exported rules are "localized" (hidden) inside the book. The supporting rules for proving these theorems (and lemmas for justifying these rules) are not exposed to the user of the "books".

To follow a top down methodology in proving a theorem, we first identify the obviously helpful rules for the prover to prove the theorem. These rules can be added by introducing "skip-proofs" forms. We continue to interact with the theorem prover to identify other necessary rules, usually by studying the failed attempt at proving the goal theorem. Once we identify the sufficient set of rules for proving the goal theorem, we can then move on to show that rules that we have introduced are justifiable — we need to prove the "skip-proofs" forms are in fact theorems themselves. We can create a book for each "skip-proofed" conjecture. The new goal theorem of each book is the corresponding form that we "skip-proofed" in the first stage. Each book only exports the rule associated with the goal theorem.

The theoretical foundation for this module system is discussed in *Structured Theory Development for a Mechanized Logic* [26] by Kaufmann and Moore. The practical aspects of organizing lemma books to follow a top down approach in developing ACL2 proofs are presented in *Modular Proof: The Fundamental Theorem of Calculus* [24] by Kaufmann.

In my work of modeling the JVM and verification of the JVM safety, I have followed the top down methodology and created an extensive set of ACL2 books (over 200) for reasoning about different aspects of the JVM.

3.3.3 Proving Properties of the Hanoi Tower Mover

We continue with the Hanoi Tower example to show how one may use ACL2 to prove theorems. My Hanoi Tower proof presented here is derived from the Kaufmann and Moore's proof about the Towers of Hanoi problem as described in [27]. My model for the Hanoi tower mover is different from their model for the mover. My model does not check whether a move is legal before executing it. We also differ on how the desired properties are stated. In particular, I separate the *functional specification* from the execution *safety specification.* Consequently, the proofs are different in non-trivial ways.

We first prove that a planner generates effective moves for the Tower mover to move a tower of arbitrary disks — functional correctness. We then

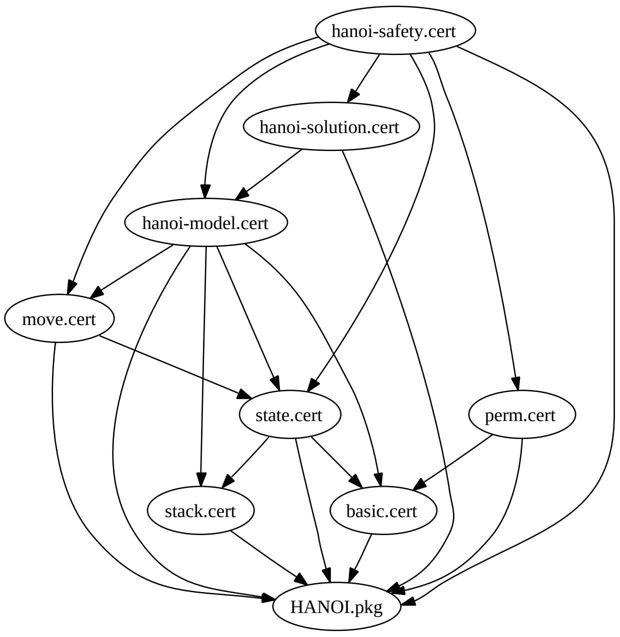


Figure 3.5: Organizing correctness proofs for Hanoi tower mover

prove that during execution of the planner-generated moves: (1) the state of system remains "good" and (2) all moves are legal moves with respect the state encountered — execution safety.

We make use of ACL2 books to organize the proofs. Figure 3.5 gives an overall picture of how the definitions and proofs are organized into this set of books. We define the basic data structures and operations on them in the files: `move.lisp`, `stack.lisp`, `state.lisp`. The operational model of the Hanoi Tower mover is described in `hanoi-model.lisp`. File `hanoi-solution.lisp` contains the definition of the planner model and the planner's functional specification. It also contains the lemmas that the ACL2 theorem prover can use to show the planner is correct. File `hanoi-safety.lisp` defines the "good-state" and specifies the execution safety of the mover. The same file also describes an ACL2 proof that asserts that the Hanoi tower mover will execute safely by following the moves generated by the planner. The files are available in `hanoi` directory of the dissertation supporting material [22].

Functional correctness

We modeled the Hanoi Tower mover `(play moves st)` as an ACL2 program. We wrote the Hanoi Tower move planner `(Hanoi n)`. We specified the functional correctness of the composition of the mover and planner with the following theorem.

```
(defthm hanoi-correct
  (equal (play (Hanoi n) (init-state n))
         (final-state n)))
```

To prove that the composition of Hanoi tower mover `play` and the move planner `Hanoi` is functionally correct, we first observe that we can rewrite `(play (app moves1 moves2) st)` into `(play moves2 (play moves1 st))`. We prove a lemma to introduce this rewrite rule into the ACL2 theorem prover's rule database.

```
(defthm play-app
  (equal (play (app a b) s)
         (play b (play a s))))
```

The ACL2 theorem prover can prove the above theorem from the definition of `app`, `play`, with no additional guidance from us.

We then observe that any non-empty sequence generated by `(h from to temp n)` is a concatenation of three segments: `(h from temp to (- n`

```
(defun do-move (m st)
  (let* ((from (src m))
         (to   (dest m))
         (from-stk (get-peg from st))
         (to-stk   (get-peg to st)))
    (set-peg from
             (pop from-stk)
             (set-peg to
                      (push (top from-stk)
                            to-stk)
                      st))))

(defun play (moves st)
  (if (endp moves)
      st
      (play (rest moves)
            (do-move (first moves) st))))
```

Figure 3.6: Modeling Hanoi tower operator (same as figure 3.1)

```
(defun h (from to temp n)
  (if (zp n) nil
    (app (h from temp to (- n 1))
         (cons (new-move from to)
               (h temp to from (- n 1))))))

(defun Hanoi (n)
  (h 'A 'B 'C n))
```

Figure 3.7: Modeling the Hanoi tower moves planner

```
(defthm hanoi-correct
  (equal (play (Hanoi n) (init-state n))
         (final-state n)))
```

Figure 3.8: Proving functional correctness of Hanoi tower mover

`1))`, a single move `(new-move from to)`, and `(h temp to from (- n 1))`. If we assume that by following first segment the mover will move top most `n - 1` disks of peg `from` to peg `temp`, we may convince ourselves that by following these three steps, the mover will successfully move the tower of size `n` to its destination. This leads to a proof by induction on the structure of the sequence of moves.

```
(defthmd h-lemma
  (implies (and (natp n)
                (pegp from)
                (pegp to)
                (pegp temp)
                (not (equal from to))
                (not (equal from temp))
                (not (equal to temp)))
           (equal (play (h from to temp n)
                        (s from (app (tower n)
                                     (g from st)) st))
                  (s to (app (tower n)
                             (g to st)) st)))
  :hints (("Goal" :do-not '(generalize fertilize)
           :in-theory (enable new-move)
           :induct (induction-hint from to temp n st))))
```

We tell the ACL2 theorem prover to induct according the pattern encoded in the function definition of `(induction-hint from to temp n s)` by specifying an induction `hint` — `:induct (induction-hint from to temp n st)`.

```
(defun induction-hint (from to temp n st)
  (if (zp n)
      (list from to temp n st)
    (list (induction-hint from temp to (- n 1)
                          (s from (push n (g from st)) st))
          (induction-hint temp to from (- n 1)
                          (s to (push n (g to st)) st)))))
```

Figure 3.9: Encoding induction hints in a function definition

With the induction hint (figure 3.9), The ACL2 theorem prover will create two subgoals corresponding to whether `(zp n)` holds.

When `n` is not an integer or is no more than zero, the prover will prove the result directly by resorting to the definitions of `h`, `play`, `tower`, as well as rules about operations `s` and `g` — imported by including the "records" book that exports these rules about `s` and `g`.

When `n` is a positive integer, the ACL2 theorem prover will follow the induction hints by "assuming"

- Starting from a state

  ```
  (s from (app (tower (- n 1))
               (push n (g from st))) st)
  ```

  and by following the `(h from temp to (- n 1))` moves, the mover `play` will move the top `n - 1` disks to from peg `from` to peg `temp`.

- Starting from a state

  ```
  (s temp (app (tower (- n 1))
               (g temp (s to (push n (g to st)) st)))
          (s to (push n (g to st)) st))
  ```

and by following the `(h temp to from (- n 1))` moves, the mover `play` will move the top `n - 1` disks from peg `temp` to peg `to`.

With this induction hint, together the rule `play-app` and rules about `s` and `g` operations, the ACL2 theorem prover can prove the `h-lemma`.

Once we prove the `h-lemma` lemma, we can instruct the ACL2 theorem prover to prove the functional correctness of the composition of the mover and move planner by giving an explicit `:use` hint.

```
(defthm hanoi-correct
  (equal (play (Hanoi n) (init-state n))
         (final-state n))
  :hints (("Goal"
           :use (:instance h-lemma
                           (from  'A)
                           (to    'B)
                           (temp  'C)
                           (st     nil)))))
```

Execution safety

We prove that the mover's execution following the generated plan will not lead to a "bad" state. We prove that all planned moves are legal. [10]

Recall our "good state" definition asserts (1) that the set of disks in the system must remain a permutation of the original set at all times — no new

[10]We have not defined guards for all operations in this Towers of Hanoi example. So we are not able to show that all operations are guard verified. To guard-verify an entire application is often a much bigger project; and demands that we identify strong preconditions on each operation and prove these preconditions to be consistent with each other.

disks are introduced, nor do disks disappear and (2) that disks on each peg are in the correct order.

It may appear that the first requirement can be trivially maintained — all the operations are just moving one disk to another disk, thus the total set of disks will remain unchanged.

However, in reality it is not easy. With a more careful inspection of our definition of the operation `do-move`, one may notice that both `(src m)` and `(dest m)` should be valid peg and they should not be the same. If they happen to be the same peg, one may verify that any time we execute such a move, the top-most disk from that peg will disappear.

```
(defun do-move (m st)
  (let* ((from (src m))
         (to   (dest m))
         (from-stk (get-peg from st))
         (to-stk   (get-peg to st)))
    (set-peg from
             (pop from-stk)
             (set-peg to
                      (push (top from-stk)
                            to-stk)
                      st))))
```

Another observation is that the `from-stk` should not be empty, since otherwise the disk `nil` will be erroneously added to the set of disks that we are manipulating. [11]

[11] Applying the `top` operation to an empty stack returns `nil` in our definition of `top`; applying `pop` operation to an empty stack returns the same empty stack.

These two observations help focus our attention on the definition of legal moves: good state status is not preserved when the mover is asked to execute an illegal operation, such as a move that specifies the same peg as both the source and the destination or a move that tries to get a non-existent disk and place the non-existent disk onto some peg.

We conjecture the following: legal moves preserve the "good-state". If we can prove that, then we can reduce the requirement that all encountered states are good states to (1) all planner-generated moves are legal moves (with respect the state reached by executing all the moves before the current move), and (2) the initial states are "good-states".

We formulated this conjecture and proved it.

```
(defthmd do-move-preserve-safe-state
    (implies (and (legal-movep m st)
                  (safe-state st n))
             (safe-state (do-move m st) n)))
```

Here, `safe-state`, `legal-movep` and related functions are defined in figure 3.10 on the next page.

After proving this lemma, we move on to show that the moves generated by the planner form a sequence of legal moves with respect to a certain initial state. We first need to define what a sequence of legal moves is with respect to some initial state. We have come up with the following definition: `safe-moves` as shown in figure 3.11

The `h-is-safe-lemma` (figure 3.11) is the key lemma that describes under what condition the sequence of moves generated by the planner function `h` are `safe-moves`.

```
(defun disks-perm (st l) ...)

(defun stack-inorder (stk) ...)

(defund disks-inorder (st)
  (and (stack-inorder (g 'A st))
       (stack-inorder (g 'B st))
       (stack-inorder (g 'C st))))

(defun safe-state (st n)
  (and (disks-perm st (tower n))
       (disks-inorder st)))

(defund wff-move (m)
   (and (movep m)
        (not (equal (src m)
                    (dest m)))))

(defund legal-movep1 (m s)
  (let ((from-stk (get-peg (src m) s)))
    (and (wff-move m)
         (has-morep from-stk))))

(defund legal-movep2 (m s)
  (let ((from-stk (get-peg  (src m) s))
        (to-stk (get-peg (dest m) s)))
    (and (wff-move m)
         (has-morep from-stk)
         (or (not (has-morep to-stk))
             (< (top from-stk)
                (top to-stk))))))

(defund legal-movep (m s)
  (and (legal-movep1 m s)
       (legal-movep2 m s)))
```

Figure 3.10: Expressing Hanoi tower mover safety

```
(defun safe-moves (moves s)
  (if (endp moves)
      t
    (and (legal-movep (car moves) s)
         (safe-moves (cdr moves)
                     (do-move (car moves) s)))))

(defthmd h-is-safe-lemma
   (implies (and (natp n)
                 (pegp from)
                 (pegp to)
                 (pegp temp)
                 (not (equal from to))
                 (not (equal from temp))
                 (not (equal to temp))
                 (big-tops from to temp s n))
            (safe-moves (h from to temp n)
                        (s from (app (tower n)
                                     (g from s)) s))))
```

Figure 3.11: Proving Hanoi tower mover safe: key lemma

Once ACL2 accepts `h-is-safe-lemma` as a theorem, we then can use ACL2 to prove that the sequence of moves is safe, i.e. all state transitions are legal.

```
(defthm safe-play
  (safe-moves (Hanoi n) (init-state n))
  :hints (("Goal" :use (....))))
```

In order to prove that all reachable states are "good states", we define

```
(defun collect-states (moves st)
  (if (endp moves)
      nil
    (let ((nx-state (do-move (car moves) st)))
      (cons nx-state (collect-states (cdr moves) nx-state)))))


(defun all-safe-states (stl n)
  (if (endp stl) t
    (and (safe-state (car stl) n)
         (all-safe-states (cdr stl) n))))
```

Here, `collect-states` serves as a "monitor". It collects all the intermediate states into a list. Operation `all-safe-states` iterates through the list and asserts that every state is a `safe-state`.

In order to prove that all the reachable states are safe,

```
(defthm safe-play-1
  (all-safe-states (collect-states (Hanoi n) (init-state n)) n)
  :hints (("Goal" :in-theory (e/d () (Hanoi init-state)))))
```

We prove the following lemma that relates `(safe-moves moves st)` with `(all-safe-states (collect-states moves st))`

```
(defthm safe-moves-implies-safe-states
  (implies (and (safe-moves moves st)
                (safe-state st n))
           (all-safe-states (collect-states moves st) n)))
```

By making use of the proven theorem `safe-play` and `safe-moves--implies-safe-states`, we can prove our safety specification `safe-play-1`, which asserts that all reachable states are safe.

Chapter 4

JVM Model

M6 is our formal executable JVM model in ACL2. It is implemented by following the official JVM specification [45].

In this chapter, we present how the JVM is modeled in M6 — its state representation and state transition functions. We also explain how the model can be used for proving properties of Java programs. A significant portion of the material in this chapter is presented in the first Interpreters, Virtual Machines and Emulators Workshop 2003 [23] and later published in *Science of Computer Programming* [15].

4.1 Introduction

To study the properties of the Java Virtual Machine (JVM) and Java programs, our research group has produced a series of JVM models written in the ACL2 programming language [19, 31]. In this chapter, we present our most complete JVM model from this series, namely, M6, which is derived from a careful study of the JVM Specification and the J2ME KVM [43] implementation.

On the one hand, our JVM model is a conventional machine emulator. M6 implements dynamic class loading, class initialization and synchronization via monitors. It executes most J2ME CLDC Java programs that do not use any I/O or floating point operations. Engineers may consider M6 an implementation of the JVM.

On the other hand, M6 is novel because it allows for analytical reasoning in addition to conventional testing. M6 is written in the ACL2 programming language. Properties of M6 and its bytecode programs can be expressed as formulas and proved as theorems in the ACL2 logic. Proofs are constructed interactively with the ACL2 theorem prover. Its concreteness, completeness, executability and mechanized reasoning support make our model unique among JVM models.

4.2 Approach

The JVM interpreter loop is modeled with an ACL2 function, `run`, which takes as its input a "schedule" and an ACL2 representation of the JVM state and returns the state obtained by executing individual threads as specified by the schedule. The semantics of each instruction is given by a corresponding state transition function. Primitives for class resolution, class loading, exception propagation as well as the Java native methods are also modeled with respective ACL2 functions. We did not accurately formalize the Java memory model. We assume that memory read and write primitives are atomic at the bytecode instruction level.

4.2.1 State Representation

Because our JVM simulator M6 is written in a functional programming language, it is quite different from most simulators written in C. All aspects of the machine state are encoded explicitly in one logical object denoted by a term.

A JVM state in M6 is a eight-tuple consisting of a global program counter, a current thread register, a heap, a thread table, an internal class table that records the runtime representations of the loaded classes, an environment that represents the source from which classes are to be loaded, an auxiliary field that records the pending exceptions to be handled, and a fatal error flag used by the interpreter to indicate an unrecoverable error.

The thread table is a table containing one entry per thread. Each entry has a slot for a saved copy of the global program counter, which points to the next instruction to be executed when this thread is scheduled next time. Among other things, the entry also records the method invocation stack (or "call stack") of the thread. The call stack is a stack of frames. Each frame specifies the method being executed, a return pc, a list of local variables, an operand stack, and possibly a reference to a Java object on which this invocation is synchronized.

The heap is a map from addresses to instance objects. The internal class table is a map from class names to descriptions of various aspects of each class, including its direct superclass, implemented interfaces, fields, methods, access flags, and the bytecode for each method.

All of this state information is represented as a single Lisp object composed of lists, symbols, strings, and numbers. Operations on state components, including the determination of the next instruction, object creation, and method resolution, are all defined as Lisp functions on these Lisp objects.

Below we describe the representations of selected state components in some detail.

Objects

Each Java object in the heap has three components:

- A *common-info* section that contains a unique integer for this object, a monitor for the JVM and Java programs to synchronize on, and the type of this object;
- A *specific-info* section to indicate whether this object is a regular object or an object that represents a class, and, if the latter, the class that it represents;
- A *java visible* section to record the information directly visible to a byte-code program, which can be accessed with instructions such as `GETFIELD` and `PUTFIELD`.

Figure 4.1 shows an object of type `java.lang.String`:

The unique integer associated with this object is 0. The "java visible" part is a list of immediate fields from `java.lang.String` and the immediate fields from its superclasses, in this case, `java.lang.Object`.

We need the information stored in these three components to implement primitives for `PUTFIELD`, `GETFIELD`, and monitor enter/exit operations. The information is also needed to implement the native methods such as the `getClass` method of `java.lang.Object`.

```
(OBJECT
  (COMMON-INFO 0 (MONITOR ...)
                "java.lang.String")
  (SPECIFIC-INFO STRING)
  (("java.lang.String" ("value" . 89)
                       ("offset" . 0)
                       ("count" . 4))
   ("java.lang.Object")))
```

Figure 4.1: Representing an object

Thread Table Entry

We have described the structure of the thread table informally at the beginning of this section. Each thread table entry has slots for recording a thread id, a pc, a call stack, a thread state, a reference to the monitor, the number of times the thread has entered the monitor, and a reference to the Java object representation of the thread in the heap.

As a concrete example, the following entry (figure 4.2) is taken from an actual thread table when we use our model to execute a multi-threaded program for computing factorial. A semicolon (;) begins a comment extending to the end of the line.

Method representation

We have developed a tool, *jvm2acl2*, which takes Java class files and converts them into a format to be used with our model. Each class file is converted into a Lisp constant. The environment component of a JVM state contains an external class table, which is composed of a list of such ACL2 constants. The class loader in our JVM model reads from the external class table and

```
(THREAD 0                  ; thread id is 0
 (SAVED-PC . 0)            ; slot for saved pc
 (CALL-STACK
  (FRAME (RETURN_PC . 7)   ; pc to return to
         (OPERAND-STACK)   ; empty operant stack
         (LOCALS 104)
         (METHOD-PTR "FactHelper" "<init>" ...)
         (SYNC-OBJ-REF . -1))
  (FRAME (RETURN_PC . 18)
         (OPERAND-STACK 104)
         (LOCALS 102)
         (METHOD-PTR "FactHelper" "compute"...)
         (SYNC-OBJ-REF . -1))
   ...)
 (STATUS THREAD_ACTIVE)     ; thread state
 (MONITOR . -1)             ; lock
 (MDEPTH . 0)               ; entering count
 (THREAD-OBJ . 55))         ; object rep in heap
```

Figure 4.2: Representing a thread

constructs a runtime representation of the class in the internal class table. The following (figure 4.3) is an example to illustrate how the Java method is represented in the internal class table of our state representation. The reader familiar with the class file format will recognize figure 4.3 as being just a Lisp representation of a method description normally found in a class file.

```
public static String valueOf(char data[]) {
       return new String(data);
}
```

is represented as

```
(METHOD
  "java.lang.String"
  "valueOf"
  (PARAMETERS (ARRAY CHAR))
  (RETURNTYPE . "java.lang.String")
  (ACCESSFLAGS *CLASS* *PUBLIC* *STATIC*)
  (CODE (MAX_STACK . 3)
        (MAX_LOCAL . 1)
        (CODE_LENGTH . 9)
        (PARSEDCODE
         (0 (NEW (CLASS "java.lang.String")))
         (3 (DUP))
         (4 (ALOAD_0))
         (5 (INVOKESPECIAL
              (METHODCP "<init>"
                        "java.lang.String"
                        ((ARRAY CHAR)) VOID)))
         (8 (ARETURN)
         (ENDOFCODE 9))
        (EXCEPTIONS)
        (STACKMAP)))
```

Figure 4.3: Representing a method

4.2.2 State Manipulation Primitives

Our JVM state is represented explicitly as an ACL2 object. We define a set of primitives to manipulate this ACL2 object. Many of those primitives correspond to the operations and procedures described in the JVM specification [45]. Some of them correspond to the native methods in the Java runtime library that are implementation dependent. Others are utilities that we introduce for implementing our simulator.

For example, to find the next instruction for execution in state `s`, we use the function `next-inst` (figure 4.4). It takes `s` as its only formal parameter. It then binds three local variables. The variable `ip` is bound to `(pc s)`. In ACL2, `(pc s)` denotes the value of the Lisp function `pc` when applied to `s`, i.e., $pc(s)$. Next, the variable `method-ptr` is bound to the method pointer in the top frame of the call stack of the current thread, as determined by the function `current-method-ptr`. Finally, the variable `method-rep` is bound to the method referred to by `method-ptr`; we use `deref-method` to look up the method in the instance class table. Having bound the three local variables, we then use `inst-by-offset` to determine the instruction at offset `ip` in the method `method-rep` and return the instruction.

```
(defun next-inst (s)
  (let* ((ip  (pc s))
         (method-ptr (current-method-ptr s))
         (method-rep
            (deref-method method-ptr
                          (instance-class-table s)))
       (inst-by-offset ip method-rep)))
```

Figure 4.4: Primitive: `next-inst`

```
(defun call_method_general
      (this-ref method s0 size)
  (let ((accessflags
         (method-accessflags method))
        (s1
         (state-set-pc (+ (pc s0) size) s0)))
    (cond
     ((mem '*native* accessflags)
      (invokeNativeFunction method s1))
     ((mem '*abstract* accessflags)
      (fatalError "abstract_method invoked" s0))
     (t
      (let ((s2 (pushFrameWithPop ... s1 ...)))
        (if (mem '*synchronized* accessflags)
            (mv-let
             (mstatus s3)
             (monitorEnter this-ref s2)
             (set-curframe-sync-obj this-ref s3))
          s2))))))
```

Figure 4.5: Primitive: invoke a method

As another example, the call_method_general function (figure 4.5) is used in the descriptions of the JVM instructions that invoke methods.

We read this roughly as follows. The function has four formals: `this-ref`, the current "self" reference; `method`, the method to invoke; `s0`, the current state; and `size`, the length in bytes of the invocation instruction. Let `accessflags` be the access flags of the method and let `s1` be a state like `s0` but with the program counter incremented by `size`. If `accessflags` indicates that the method is native or abstract, we handle the call with special-purpose functions. Otherwise, let `s2` be a state obtained from `s1` by popping the proper number of values off the current operand stack, and pushing a suitable call frame on top of the current call frame. We construct this state with `pushFrameWithPop`. If `accessflags` indicates the invoked method is synchronized, we call `monitorEnter` to obtain two results, namely, `mstatus` (which we ignore) and a state `s3`, in which the current thread holds the monitor on `this-ref`; we return `s3` after updating the new frame to indicate that the monitor on `this-ref` must be released upon exit from the method. If the invoked method is not synchronized, we return `s2`.

Notice, that the function `monitorEnter` mentioned above is just another state manipulating primitive that takes a reference and a state and returns a status flag and a state. There are two kinds of states that can be returned by `monitorEnter`. If `monitorEnter` succeeds, the state returned is representing a state in which the current thread is holding the monitor and the current thread is still active. If `monitorEnter` does not succeed, possibly because some other thread is holding the monitor, then the state returned represents a state in which the current-thread is suspended and moved to the waiting queue for the monitor.

As our last example in this section, in figure 4.6, we show the function implementing the native method `currentThread` of `java.lang.Thread`. It takes a state as its input and pushes on the operand stack of the current frame a reference to the Java object that represents the current thread.

4.2.3 State Transition Function

We model the semantics of the JVM instructions operationally, in terms of such primitives as discussed above. The meaning of executing a JVM instruction is given operationally by a state transition function on JVM states. Here is the state transition function for the `IDIV` instruction.

```
(defun execute-IDIV (inst s)
 (let ((v2 (topStack s))
       (v1 (secondStack s)))
  (if (equal v2 0)
      (raise-exception
          "java.lang.ArithmeticException" s)
   (ADVANCE-PC
     (pushStack (int-fix (truncate v1 v2))
                (popStack (popStack s)))))))
```

Here, `inst` is understood to be a parsed `IDIV` instruction. `ADVANCE-PC` is a Lisp macro to advance the global program counter by the size of the instruction. `PushStack` pushes a value on the operand stack of the *current frame* (the top call frame of the current thread) and returns the resulting state. When the item on the top of the operand stack of the current frame is zero, the output of `execute-IDIV` is the state obtained from `s` by raising an exception of type `java.lang.ArithmeticException`. If the top item is not zero, the resulting state is obtained by changing the operand stack in the current frame and advancing the program counter. The operand stack

```
(defun Java_java_lang_Thread_currentThread (s)
 (let*
  ((tid (current-thread s))
   (thread-rep
    (thread-by-id tid (thread-table s)))
   (thread-ref (thread-ref thread-rep)))
  (pushStack thread-ref s)))
```

Figure 4.6: Primitive: native method

```
(defun execute-invokestatic (inst s)
 (let* ((cp    (arg inst))
        (method-ptr (methodCP-to-method-ptr cp)))
  (mv-let
   (method new-s)
   (resolveMethodReference method-ptr t s)
   (if method
       (let*
        ((class
          (static-method-class-rep method new-s))
         (cname
          (classname class))
         (cref
          (class-ref class)))
        (cond
         ((class-rep-in-error-state? class)
          (fatalError "class in error state!"
                      new-s))
         ((not (class-initialized? cname  new-s))
          (initializeClass cname new-s))
         (t
          (call_static_method cref method new-s))))
     (fatalSlotError cp new-s)))))
```

Figure 4.7: Instruction semantics: invokestatic

is changed by pushing a certain value (described below) onto the result of popping two items off the initial operand stack. The value pushed is the twos-complement integer represented by the low-order 32-bits of the integer quotient of the second item on the initial operand stack divided by the first item on it.

As an example of a more complicated instruction, figure 4.7 shows our semantics for `invokestatic`. To invoke a static method, we first get the constant pool entry and obtain a symbolic `method-ptr`. Then we call the method resolution procedure `resolveMethodReference` to find the `method` to which `method-ptr` resolves. Since method resolution may cause class loading, the procedure returns a pair `(method new-s)` as declared in the `mv-let`. The fatal error flag is set when the method is `nil`, i.e., the method resolution fails. If the class to which the method belongs has not been initialized, we choose not to advance the program counter, instead returning a state prepared for starting the class initialization procedure using `(initializeClass cname new-s)`. Otherwise, we use `call_static_method` to return a state properly prepared for the interpreter to start executing the resolved method. The definition of `call_static_method` (not shown here) uses the previously shown `call_method_general` to construct that state.

4.3 M6 as a Simulator

Not only have we modeled the semantics of most JVM instructions, we have also provided implementations for native methods from the CLDC runtime library. As a result, we can use the formal model as a simulator to execute realistic JVM programs.

We have translated the entire Sun CLDC library implementation into our representation with 672 methods in 87 classes [43]. We provide implementations for 21 out of 41 native methods that appear in Sun's CLDC library. The remaining ones are mostly related to the I/O of the JVM, which we do not model in our current simulator.

We have written several test programs to run on this model to exercise various aspects of the simulator such as exception handling, synchronization, and class initialization. One of the test programs is a multi-threaded Java program that implements an impractical but illuminating parallel factorial algorithm.

The program takes a command line parameter represented in a `java.-lang.String`. It calls the method `parseInt` of `java.lang.Integer`, which in turn invokes a dozen Java functions to parse the string and return an integer. Then it spawns a specified number of `FactHelper` threads (5 in the current version). Those threads share an instance of `FactJob`, in which the intermediate result is stored. Those `FactHelpers` repeatedly compete for the monitor on the `FactJob`, compute one iteration and then release the monitor by executing `monitorexit`. The main thread prints the result and quits when it is awakened by a `notify` call from any of the `FactHelper`s indicating that the computation has terminated.

```
class FactJob {
    int value = 1;
    int n;
    ... };

class FactHelper implements Runnable {
    FactJob myJob;
    public void run() {
      ... wait for notifyAll on myJob.
      for (;;) {
        synchronized(myJob) {
         if (myJob.n<=0) {
             myJob.notify();
             return; // done
        } else {
            myJob.value = myJob.value*myJob.n;
            myJob.n = myJob.n - 1;
        }}}}
    ...};

public class Fact {
    static int HELPER_COUNT = 5;

    public static int fact(int n) {
        FactJob theJob = new FactJob(n);
        ... spawn HELPER_COUNTER FactHelpers.
        ... send notifyAll to FactHelpers
        try{
         synchronized (theJob){ theJob.wait();};
         //wait for at least one Helper to finish.
        } catch ...
       return theJob.value;};
}
```

The main method is:

```
public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);
    System.out.println(""+Fact.fact(n));
}
```

To compute factorial of 10, the simulator executes 1748 steps — most of which are spent in the class initialization, parsing an integer from a string, string copying, and printing the result character by character. In the process, 5 threads are spawned, 118 heap objects are created, and 18 classes are loaded and initialized.

In the rest of this section, we pick some aspects of our simulator and explain them in some more detail.

4.3.1 Interpreter Loop

As mentioned previously, our JVM model takes a "schedule" (a list of thread ids) and a state and repeatedly executes the next instruction from the thread as indicated in the schedule, until the schedule is exhausted.

```
(defun run (sched s)
 (cond ((endp sched) s)
       (t (let ((nid (car sched))
                (cid (current-thread s)))
            (cond ((equal cid nid)
                   (run (cdr sched) (step s)))
                  (t (run (cdr sched)
                          (loadExecutionEnvironment nid
                           (storeExecutionEnvironment s))...)
```

Figure 4.8: Interpreter loop

The scheduling policy is thus left unspecified. Any schedule can be simulated. However to use the model as a JVM simulator without providing a schedule explicitly, we have implemented some simple scheduling policies. One of them is a not very realistic round-robin scheduling algorithm, which does a rescheduling after executing each instruction.

4.3.2 Class Initialization

Class initialization is a very complicated aspect of the JVM. Multi-threading adds to the complexity. To initialize a class, the JVM needs to execute a special class initialization method as part of the process. Thus the initialization of a class cannot be made atomic. The process needs careful synchronization between threads that may try to initialize the same class at the same time. We explain this aspect of our simulator in detail.

Modeled after the KVM implementation [43], our simulator follows the 11-step algorithm described in the JVM specification Sec. 2.17.5. [45] and completes a class initialization in 6 stages. [1] Recall the state transition function that describes `invokestatic`. If the class to which the resolved method belongs has not been initialized the function does not advance the pc; instead it returns a state by calling `(initializeClass classname s)`. Roughly, the state returned is a special state recognized by the interpreter loop as indicating the start of the class initialization process. Each thread maintains a simple finite state machine to keep track of the class initialization stage that the thread is currently in. A series of functions, with names of the form `runCliniti`, implements the transitions of this machine. The details are in `jvm-static-initializer.lisp` of [22].

To initialize a class, the current thread first needs to acquire the monitor associated with the class. This is necessary because we need to cope with the situation that multiple threads may be trying to initialize the same class at the same time. If the acquisition attempt fails, i.e. the `mstatus` is not equal

[1]Because the class initialization process is not atomic, the interpreter loop is needed to executing a special method class initializer. A thread executing the class initializer may be interrupted. A finite state machine is used to keep track of the stage of the initialization process.

```
(defun runClinit1 (classname s)
 (let ((cid (current-thread s)))
  (mv-let (mstatus new-s)
          (classMonitorEnter classname s)
          (if (not (equal mstatus
                          'MonitorStatusOwn))
              (set-cinit-stage  cid 2 new-s)
             (runClinit2 classname new-s))))))
```

Figure 4.9: Class initializer: stage one

to `MonitorStatusOwn`, we return the following state to the interpreter loop: the original "current" thread is suspended and the class initialization process stage of the thread is set to `2` with `(set-cinit-stage cid 2 new-s)`. The `set-cinit-stage` call is needed so that when the thread is resumed after the JVM grants the monitor, the top level interpreter loop can resume from the second stage of the class initialization process. Otherwise, we directly advance to stage 2. In stage `2`, we first check whether some other thread is initializing the class. If some other thread is initializing the class, we return a state in which the current thread is suspended to wait for a notify signal from the monitor and the class initialization stage remains at `2`. If no other thread is initializing the class, we check whether the class has already been initialized. If it has not been initialized, we mark the state to indicate that the current thread is initializing the class and return the resulting state to the interpreter loop.

In stage `3` of class initialization, we first check whether the superclass has been initialized. If the superclass is not initialized yet, we use `initialize-Class` to prepare a state so that the interpreter loop can recognize and start the initialization process for the superclass. We also properly mark the state

```
(defun runClinit2 (classname s)
 (let*
  ((class-rep  (class-by-name classname ...))
   (initThread (init-thread-id class-rep))
   (cid        (current-thread s)))
  (cond
   ((and (not (equal initThread -1))
         (not (equal initThread
                     (current-thread s))))
    (let
      ((new-s (classMonitorWaitX classname s)))
     (set-cinit-stage  cid 2 new-s)))
   ((or (equal initThread (current-thread s))
        (class-initialized? classname s))
    ....)
   (t
    (let
     ((sinit
       (setClassInitialThread classname cid s)))
     (mv-let (mstatus exception-name s-new)
             (classMonitorExitX classname sinit)
             (runClinit3 classname s-new)))))))
```

Figure 4.10: Class initializer: stage two

```
(defun runClinit3 (classname s)
  (let ((class-rep (class-by-name classname ...))
        (cid (current-thread s)))
     (if (not (isInterface class-rep))
         (if (and (super-exists class-rep)
                  (not (class-initialized?
                         (super class-rep) s)))
             (initializeClass
                     (super class-rep)
                     (set-cinit-stage cid 4 s))
           (runClinit4 classname s))
       (runClinit4 classname s))))
```

Figure 4.11: Class initializer: stage three

so that, when class initialization for the superclass is completed, the process of initializing the current class enters stage 4.

```
(defun runClinit4 (classname s)
  (let* ((clinit-ptr (clinit-ptr classname))
         (thisMethod
            (getSpecialMethod clinit-ptr s))
         (cid (current-thread s)))
     (if thisMethod
         (pushFrame clinit-ptr nil
                  (set-cinit-stage cid 5 s))
       (runClinit5 classname s))))
```

Figure 4.12: Class initializer: stage four

Now we are in class initialization stage 4. To complete the class initialization, the interpreter must execute the class initialization method of the class. We achieve this by pushing a call frame for the class initialization method onto the call stack. We also mark the state so that when the interpreter is done with the invocation of the class initialization method, it can recognize that it needs to resume the class initialization process from stage `5`. In stage `5`, we try to acquire the monitor associated with the class. In stage `6`, the class initialization process is completed, and we send the `notifyAll` signal so that threads that are waiting in stage `2` can proceed.

4.3.3 Dynamic Class Loading

The Java virtual machine maintains a "method area" (see section 3.5.4 of the JVM specification [45]), which we call a class table in this paper. The class table contains various information about Java classes that exist in the JVM. Among others, type hierarchy information is encoded in the superclass and superinterface declarations in the class table.

When the interpreter encounters instructions that refer to a class by name, such as a `(NEW C)` instruction, the definition of the class `C` needs to be available to the interpreter. If class `C`'s definition is not already available, the class loader is used to load the definition and extend the class table at runtime.

The class loading mechanism in a full JVM is a delicate process, especially when user defined class loader are permitted. Being a model for CLDC JVM, M6 does not allow user defined class loaders. The class loader in M6 matches the description for the bootstrap class loader in the JVM Specification [45].

The simplified class loading process happens in two stages. In the first stage, given a name, the bootstrap class loader returns a byte array in a platform-dependent manner. The byte array is purported to represent the class of that name in the Java class file format. The JVM then checks the form and extracts relevant information to extend the class table in the second stage of class loading. We call this second stage the "defining" stage, because in the Sun Microsystems' JVM implementation, this stage is implemented with the `defineClass` method of `java.lang.ClassLoader`. For contrast, we also refer the first stage as the "loading" stage in its narrow sense, which corresponds to the `loadClass` method of the `ClassLoader` class or user defined subclasses. In the process of "defining" a class, the JVM may need to initiate class loading of the superclass and superinterfaces of the class being loaded, if they have not already been loaded. If any of the superclasses or the superinterfaces fails to load correctly, or the byte array does not represent a well formed class, the class loading fails.

Loading stage

In M6, we record all the available classes in the external class table (one of the components of the JVM state). Classes are stored "pre-parsed;" that is, instead of being in a byte array in class file format, we store the M6 form of the unloaded class. The "loading" stage is thus simply defined as a lookup operation in the table that maps class names to some ACL2 constants that represent class files. Figure 4.13 shows a fragment of the ACL2 constant that represent the class `java.lang.ArithmeticException`.

The external class table is a list of such constants. The "loading" stage is implemented with `class-by-name-s`, which iterates through the list of external class files `scl` as follows.

```
(defun class-by-name-s (name scl)
  (if (endp scl)             ; list exhausted?
(defconst *java.lang.ArithmeticException*
  '(class
     "java.lang.ArithmeticException" ; Class name
     "java.lang.RuntimeException"    ; Superclass name
      (constant_pool)                ;
      (fields)                       ; No instance fields
      (methods                       ; Method definitions
        (method "<init>"             ; One constructor
         (parameters )               ;
         (returntype void)           ;
         (accessflags  *class*  *public*)
        (code                        ;
           (max_stack 1) (max_locals 1) (code_length 5)
           (parsedcode               ;
              (0 (aload_0))          ; Actual instrs
              (1 (invokespecial      ;
                 (methodCP           ;
                   "<init>"          ;
                   "java.lang.RuntimeException" () void)))
              (4 (return))           ;
              (endofcode 5))         ;
           (Exceptions )             ; Exception handlers
           ....                      ;
         (method "<init>"            ; Other methods
             ....)))                 ;
      (interfaces)                   ; Interface decls
      (accessflags
           *class*  *public*  *super*  *synchronized*)
      (attributes ....))))           ; class attributes
```

Figure 4.13: Class loader: external class representation

```
      (mv nil nil)           ; search failed
    (if (equal (classname-s (car scl))  name)
        (mv t (car scl))     ; if the name matches.
      (class-by-name-s name (cdr scl)))))
```

Class-by-name-s returns a pair of values. The first slot indicates whether a class by that name has been found. If (equal (classname-s class) name) is true for some class in the scl, the class is returned in the second slot.

Later, load_class_internal (not shown here) takes the static representation returned by class-by-name-s. It creates a runtime representation of the class. Among other things, load_class_internal runs through the static constant pool and creates objects and values to populate a runtime constant pool in the runtime representation of the class. It adds the new definition into the class table and updates the heap by introducing an object of type java.lang.Class to represent the newly loaded class.

It is important that load_class_internal is only executed when the JVM state meets certain criteria to ensure correct execution of the Java programs. One requirement is that before a class is made visible for user programs, all its superclasses are correctly loaded.

Defining stage

In a real JVM loader, the "defining" stage takes the byte array returned by the "loading" stage, extends the class table, and creates a java.lang.Class instance to allow bytecode programs to refer to the class. For it to be correct, a set of constraints must be respected. Section 5.3.5 of the JVM specification provides a detailed description for this process of “Deriving a Class from a `class` File Representation”.

Since there is only one class loader in the CLDC JVM, the constraints on “deriving a class C from a `class` file representation” (section 5.3.5 of the JVM specification [45]) can be reduced to the following simplified form:

1. The purported representation of `C` needs to be in the predefined `class` format. Exceptions indicating format error may be thrown.

2. Symbolic reference from `C` to its superclass needs to be *resolved.* To resolve a symbolic reference, the JVM needs to “load” and “define” the class referred by the symbolic name, if a class of that name is not already loaded.

   If a class or interface named as the direct superclass of `C` is in fact an interface, loading throws an `IncompatibleClassChangeError`.

   If any of the superclasses of `C` is `C` itself, loading throws a `Class-CircularityError`

3. Symbolic references from `C` to its superinterfaces need to be `resolved` also, which involves initiating class loading for the superinterfaces.

   If any of the classes or interfaces named as direct superinterfaces of `C` is not in fact an interface, loading throws an `IncompatibleClass-ChangeError`.

   If any of the superinterfaces of `C` is `C` itself, loading throws a `Class-CircularityError`.

4. If any exception is thrown as a result of either class loading and resolution

or their recursive invocations, the class loading process fails to load the class.

The class loader in M6 follows these descriptions. The major difference is that M6 does not handle any exception generated from within the class loader. Instead of throwing an exception, a fatal error flag is set in the execution state and the top level interpreter is halted. Another difference is that instead of parsing a byte array and checking for the well-formedness of the representation, we rely on our external tool `jvm2acl2` to read the binary representation and convert the binary representation into a Lisp constant.

The "defining" stage of the class loader is implemented with four mutually recursive operations:[2]

- `load_class`

```
(defun load_class (classname s seen)
 (cond (class-loaded? classname s) s ; loaded? s
       ((mem classname seen)         ; otherwise
        (fatalError "ClassCircularityError" s))
       (t (if ...
           (let*
            ((s1 (load_super classname s seen))
             (s2 (load_interfaces ... s1 seen)))
            (if (not (no-fatal-error? s1)) s2
               (if (no-fatal-error? s2) ;
                 (load_class_internal ... ))..))))))
```

Figure 4.14: Class loader: `load_class`

[2]In the actual M6 implementation, the mutually recursive operations are defined to be different execution modes of a single recursive operation. The recursive operation takes a mode flag which identifies the operation being executed. The ACL2 code presented in this section is edited form of the less intuitive recursive definition.

To load a class, M6 first locates the purported representation for the class being using `class-by-name-s`. It then invokes `load_super` to make sure that the class referred is the superclass of the purported class. After the superclass is successfully loaded without error, it uses `load_interfaces` to make sure that all superinterfaces are loaded. Only after these steps complete without error, the JVM can build an internal representation of the class and insert it into the class table. This is achieved with `load_class_internal`.

```
(defun load_class_internal (classname s)
  (if ...            ; if not found
     (fatalError
        "java.lang.ClassNotFoundException" s)
   (let* (...
         (the-Class-Object ...))
         (new-heap (bind new-address
                          the-Class-object ...))
         (runtime-class-rep ...)
         (new-dcl (add-instance-class-entry
                        runtime-class-rep dcl)))
  (state-set-heap new-heap
    (state-set-class-table
       (make-class-table new-dcl
          (array-class-table new-state2))
                             new-state2)))))))
```

Figure 4.15: Class loader: `load_class_internal`

Next, we describe the clique of mutually recursive functions for implementing `load_class` in more detail. In order to detect loops in the superclass chain and superinterface chains, all four operations take an extra parameter `seen`, which is a list of class names that are being loaded because of the initiating class loading request.

- `load_super` To load a superclass, the JVM recursively invokes `load_class` after updating the `seen` parameter to note that the class of `classname` is being loaded.

  ```
  (defun load_super (classname s seen)
    (let ((supername ...))      ; just load the super
      (load_class supername s (cons classname seen))))
  ```

Figure 4.16: Class loader: `load_super`

- `load_interfaces` uses `load_interface_classes`, which in turn invokes `load_class` on each interfaces referred to by the class being loaded.

  ```
  (defun load_interfaces (classname s seen)
     (let ((interfaces ....))  ; load a list of classes
        (load_interface_classes interfaces s
                               (cons classname seen)))))
  ```

Figure 4.17: Class loader: `load_interfaces`

- `load_interface_classes` loads classes one at a time using `load_class` operation and checks that the classes are in fact representing interfaces.

This concludes the description of the top level operations for implementing the class loader. In order to implement `load_class_internal`, we have also defined many low level operations. For example, we defined operations to create `java.lang.String` objects that correspond to the string literals in the constant pool. We also defined operations to build instances of `java.lang.Class` for representing the new class being loaded. The figure 4.19 shows just one of such low level operations.

```
(defun load_interface_classes (interfaces s seen)
 (if (not (consp interfaces)) s
   ...
  (let*
    ((new-s (load_class (car interfaces) s seen))
     (class-rep (class-by-name (car interfaces)
                  (instance-class-table new-s))))
    ....
    (if (not (isInterface class-rep))
        (fatalError ...)  ; actual interface?
      (load_interface_classes
           (cdr interfaces) new-s seen))..)...))
```

Figure 4.18: Class loader: `load_interfaces_classes`

`Load_CP_entry` takes one argument, `cpentry-s` which represents a constant pool entry in a class file. It returns a pair. The first component of the pair is the runtime representation of the constant pool entry, the second is the updated state after loading the constant pool entry. If the entry represents a string literal, we first create a `java.lang.String` object `the-String-obj` with `ACL2-str-to-JavaString` to represent the string literal. We then create a runtime representation of the constantpool entry. Instead of referring to a string literal, the runtime constant pool entry contains a reference (in the form of a number) to the newly created `java.lang.String` object.

4.4 Formal Analysis

The focus of our research has been on developing a practical methodology for reasoning about complex hardware and software artifacts, including models of the complexity of the one described here. We have shown that our executable

```
(defun load_CP_entry (cpentry-s S)
 (if (equal (cpentry-type-s  cpentry-s) 'STRING)
   (let ((str  (string-value-cp-entry-s cpentry-s)))
     (mv-let (the-String-obj new-S)
             (ACL2-str-to-JavaString str S)
       (let* ((heap   (heap new-S))
              (new-addr (alloc heap))
              (new-heap (bind new-addr
                              the-String-obj heap)))
         (mv (make-string-cp-entry new-addr)
             (state-set-heap new-heap new-S)))))
    ...))
```

Figure 4.19: Class loader: `load_CP_entry`

JVM model includes enough detail to permit its use as a simulator. But it has dual use as a formal semantics of the JVM and as such permits us to reason about the JVM and its bytecode methods.

We study two broad categories of the properties using this accurate JVM model. We use our JVM model to study the properties of the JVM specification. We also use the formal model to study the properties of Java programs that run on "real" (practically efficient) JVMs. In the rest of the section, we present some work we have done in these two categories.

4.4.1 Properties of the Model

We are interested in studying dynamic class loading in the JVM. We have formally proved an invariant of dynamic class loading using our JVM model, M6. The invariant says that if class `A` is loaded and if a value of type class A is assignable to a slot of type class `B`, then, class `B` must have already been correctly loaded. As a relevant note, the concept of *assignable to* is rather complicated. For a value of type class A to be assignable to a slot of type class B, B must be a direct superclass of A, one of the direct superinterfaces of A, or there must be a class C, where values of type class C are assignable to a slot of type class B, and class C must be a direct superclass of class A or one of the direct superinterfaces of class A.

The JVM relies on this invariant about loaded classes to behave correctly in field resolution and method dispatching. In fact, JVM operations that involve examining the class hierarchy by searching through the superclass chain rely on this property to operate correctly. It is useful to be sure that this alleged "invariant" is in fact preserved by the class loader in the JVM.

We have proved that this invariant is preserved by the class loader in M6. Our proof has two main steps. First, we proved a different formulation of the invariant, which roughly says that all classes reachable from any loaded class through its superclass chain and superinterfaces chains are also loaded. The reformulated property is named `load-inv`. We first defined a Lisp function `collect-assignableToName` that crawls along the class hierarchy and collects all classes reachable from a given one. We use that concept to express the idea, formalized as `loader-inv-helper1`, that all classes reachable from a given one are correctly loaded.

```
(defun loader-inv-helper1
   (class-rep class-table env-class-table)
   (let* ((classname (classname class-rep))
          (supers (collect-assignableToName
                   classname env-class-table)))
```

```
     (all-correctly-loaded?
          supers class-table env-class-table)))
```

Finally, we defined the reformulated invariant conjecture, `loader-inv`, to check `loader-inv-helper1` for every class in the class table. To establish that `load-inv` is an invariant, we proved the following theorem.

```
(defthm loader-inv-is-inv-respect-to-loader
  (implies (loader-inv s)
           (loader-inv (load_class classname s))))
```

This theorem has one hypothesis and one conclusion. The hypothesis says that `s` satisfies the `loader-inv` property. The conclusion says that the state produced by `load_class` satisfies it.

The second step of our proof is to show that the above reformulated property implies our original one.

```
(defthm inv-and-isAssignableTo-env
  (implies
    (and (loader-inv s)
         (no-fatal-error? s)
         (isAssignableTo-env A B
                   (env-class-table (env s))))
     (implies (correctly-loaded? A
                   (instance-class-table s)
                   (env-class-table (env s)))
              (correctly-loaded? B
                   (instance-class-table s)
                   (env-class-table (env s))))))
```

4.4.2 Properties of Bytecode Programs

Besides reasoning about properties of the JVM model itself, we use the model to reason analytically about the programs that run on it. This gives us opportunities for finding program flaws beyond those likely to be uncovered by simulation of executions.

For example, we have proved a single-threaded program for computing factorial is correct on M6 [14]. The property is stated by the following ACL2 formula. The ACL2 system mechanically checked our proof of the theorem. As a note, our description below illustrates only one of many possible styles of proofs for reasoning about Java programs. This particular style uses an explicit clock function. This proof methodology works well for proving total correctness for single-threaded programs. However one is not limited to this particular style. Ray and Moore recently show that one can use different styles in verifying different parts of programs [30, 37, 18].

```
(defthm factorial-is-correct
 (implies
   (and (poised-to-execute-fact s)
        (integerp n)
        (<= 0 n)
        (intp n)
        (equal n (topStack s)))
   (equal (simple-run s (fact-clock n))
          (state-set-pc
           (+ 3 (pc s))
           (pushStack (int-fix (fact n))
(popStack s)))))
```

This theorem has five hypotheses. The first, `(poised-to- execute-fact s)`, is a rather complicated predicate whose definition is not shown here. It says that the state `s` is well formed and that the next instruction will invoke a particular bytecode factorial method produced by compiling with `javac` a recursive `int`-valued Java definition of factorial. The other hypotheses say that the value, `n`, on top of the operand stack is a non-negative `int`. The conclusion is an equality between two expressions denoting M6 states. The first expression denotes the state produced by executing the single thread in question a certain number of steps, namely, the number produced by `(fact-clock n)`, from state `s`. The second expression denotes the state obtained from `s` by incrementing the `pc` by `3` (the size of the invoke instruction), popping one item (namely `n`) from the operand stack, and pushing the `int` denoted by the low-order 32 bits of the mathematically precise `n`! (written in ACL2 as `(fact n)`).

In formal methods literature, "partial" correctness specifies that a method returns an acceptable answer *if* it terminates; "total" correctness adds to that — the proof requirement *that* the method terminates. The theorem above is a specification for total functional correctness of the corresponding bytecode factorial method. Indeed, we characterize, with `(fact-clock n)`, exactly how many bytecode instructions must be executed. This theorem was proved mechanically with the ACL2 theorem prover. Partial correctness results can also be proved [30]. An explicit clock function is not needed and sometimes could not be defined. ACL2 proofs that employ explicit *clock functions* and proofs that employ no such clock function constructs can be used together to reason about different parts of a same program [37].

4.5 Conclusion

In this chapter, we described several selected aspects of our realistic JVM model as a simulator. We presented our general approach for writing such a detailed JVM model in a functional programming language. The state is represented with a ACL2 object. The semantics of executing a bytecode instruction in a JVM state is defined by a state transition function. We show that this approach allows us to model the delicate internals of a realistic JVM, use it as a simulator, and prove theorems about the resulting behaviors.

This executable JVM model has been the basis for our ongoing inquiry into the correctness of the bytecode verifier specification.

Although a realistic JVM model is necessary to study the correctness of the bytecode verifier and the class loader, proving properties of concrete Java programs on this complicated JVM model may be difficult and unnecessary. This can be one of its limitations. After proving the properties of the bytecode verifier and the class loader, we may be able to reduce executions on this model to executions on an alternative simpler model via a proof. Porter has done a similar proof for relating a multi-threaded JVM model to a single-threaded model under necessary restrictions [33]. We will need to look into the proper thread reduction and heap abstraction techniques.

This model, although fairly complete, still contains certain omissions and simplifying assumptions. The following two might be of particular interest.

Our JVM simulator has assumed the simplest memory model. Any memory access in our simulator is always atomic at the instruction level. Although we expect that such omission will not affect our study on the correctness of bytecode verifier, it does prevents us from reasoning about certain behaviors of legal Java programs.

The other limitation is that our current model of the bytecode verifier does not interact with the class loader dynamically. The JVM specification allows a JVM implementation to delay the bytecode verification until link time. In our implementation, the bytecode verifier uses the class hierarchy information from "unloaded" classes to conduct the bytecode verification of a method. In fact, our bytecode verifier never causes dynamic class loading itself. We expect that one will be able to prove that the bytecode verification result is independent of the class table being used, as long as the runtime class table is correctly loaded from the "environment".

In summary, our approach of writing an executable JVM simulator in ACL2, a precise language with clean axiomatic semantics and a computer-aided deduction environment, provides an opportunity to analytically deduce the properties of the artifact being modeled. It has the benefits and opportunities of both simulation and machine-checked analytical reasoning. We are researching ways for making better use of such opportunities.

Chapter 5

Java Bytecode Verifier Model

The JSR139 (Java Specification Request No.139) expert group published the formal specification of the JVM bytecode verifier in 2004. The formal specification is given as a set of Prolog-style rules.

In this chapter, we present our ACL2 model of the bytecode verifier written in ACL2. It is constructed systematically from the Prolog-style rules. It is executable. One may think our model as an implementation of the bytecode verifier. One may also view our model as an alternative formal specification of the bytecode verification process.

Being operational, our model may serve as a better reference for guiding the Java bytecode verifier implementation; written in ACL2, it is amenable to rigorous reasoning. We present a simple property which we proved about our bytecode verifier. The model itself is a 4500-line program with 480 function definitions.

5.1 Introduction

As explained in chapter 2, *programs* can be understood as describing sequences of *operations* (algorithms) to be applied to *objects* (data structures) from a certain *universe*. *To execute a program on a machine* is to have the "machine" mechanically carry out the corresponding sequence of operations on the input objects.

It is often natural to describe a problem and design its solution as a program in a high level programming language like Java. However, we do not have a physical machine to execute Java programs. The physical machines that we have are (possibly) x86 computers. Programs that can be executed on x86 computers are assembled x86 programs, which only read and write machine registers and arrays of memory cells that holds 0s and 1s. To execute a Java bytecode program, we resort to representing abstract Java entities as collections of 0s and 1s, and emulating Java operations by manipulating these representations. These representations and manipulations are coded in programs on the low level machine (e.g., an X86).

A JVM implementation is a program that emulates a machine that can execute Java programs. It bridges the semantic gap between the Java universe and the 0s and 1s of the low level machine. It is desirable for a *good* JVM implementation to have the following properties:

- *Correctness (Accuracy)*

  A JVM implementation shall behave like the Platonic JVM on all bytecode programs;

- *Safety*

A JVM implementation shall always execute safely on the low level machine. No inputs (i.e. bytecode program + inputs to the bytecode program) can induce an execution that violates the safety constraints of the low level machine;

- *Efficiency*

  Additionally, a JVM implementation shall emulate the JVM with good efficiency for all bytecode programs.

In practice, it is difficult to build a JVM implementation that has all of the above properties on today's general purpose computers. For instance, the naive way to ensure that it is always safe to execute the JVM implementation with arbitrary inputs, is to always check before executing any step. If executing the next step would violate some safety constraints of the low level machine, the execution can just halt. This naive implementation will not be efficient for emulating the JVM. Furthermore, by refusing to take a step because executing it would violate the safety constraints on the low level machine, the naive implementation will likely to behave differently from the JVM being emulated.

Thus, we have the following scenario: on the one hand, people want a correct, safe, and efficient JVM implementation; on the other hand, people realize that it is difficult to build such an implementation on today's general purpose computers, because of the semantic gap between the universe of the JVM bytecode language and the universe of low level machine code language. A trade-off is necessary.

The trade-off that the JVM designers have made is to accept only a certain subset of bytecode programs as valid JVM bytecode programs. They intended to design a JVM that only executes valid bytecode programs so that a good JVM implementation is feasible. The intuition behind this trade-off is that although not all bytecode programs can be correctly, safely, and efficiently emulated, a large, useful subset can be. The suitable set can be efficiently identified with static program analysis algorithms.

The *bytecode verifier* is designed to recognize the subset. By design, it is the central piece to ensure that a JVM implementation may correctly, safely, and efficiently emulates the (Platonic) JVM for executing the *verified* bytecode programs [46, 29].

We are interested in the correctness of the bytecode verifier. Specifically, we want to show that executions of the verified bytecode programs can be correctly, safely and efficiently emulated.

The official JVM specification is carefully crafted to contain the following information:

- It describes a bytecode verification algorithm for recognizing the valid JVM bytecode programs.
- It defines the proper behaviors of the JVM operations, however, only in JVM states that meet certain constraints. Different operations put different constraints on states.

It is believed (*belief 1*) that these constraints will be met if the JVM executes a valid bytecode program. If belief 1 is in fact true, although the JVM specification only defines the behaviors of its operations partially, the JVM specification still completely defines the JVM behaviors for executing valid bytecode programs.

It is also believed (*belief 2*) that a correct, safe, and efficient JVM implementation is feasible on today's general purpose computers after we introduce the bytecode verifier to accept only valid bytecode programs.

Our project is to find out whether beliefs 1 and 2 are justified. If belief 1 is true, we want to present a mechanically checkable formal proof of it. After belief 1 has been proven to be true, we want to study the justification for belief 2, which asserts that it is feasible to represent the JVM states that meet the constraints, and more importantly, it is feasible to emulate the JVM operations on such states correctly, safely, and efficiently.

Towards this goal, we have built a model of JVM implementation, see Chapter 4. In this chapter, we explain how we modeled the JVM bytecode verifier. In Chapter 7, we explain the methodology for proving belief 1 is true: the overall approach and the supporting ACL2 libraries.

This chapter is organized as follows. We present the official CLDC bytecode verifier specification in section 5.2. We then describe our bytecode verifier model and explain how we derived it from the official CLDC specification. We present the formulation of one simple bytecode verifier property and highlight some steps in its mechanically checked proof.

5.2 Official BCV Specification

5.2.1 Bytecode Verification

Before the JVM executes any Java method for the first time, it invokes a bytecode verifier (BCV) to check the instructions in the method. The hope is that executing a verified method will not break the integrity of the JVM implementation. Nor will its execution lead to a JVM state in which the next state is undefined.

As a simple example, the bytecode verifier should ensure that whenever an `IADD` instruction is reached, the operand stack of the current activation record of the current thread has at least two values on it. The two values must be 32-bit integers. For a more complicated example, when an `INVOKEVIRTUAL` instruction is reached at runtime, the operand stack must contain a suitable number of values. The types of these values must be compatible with the types of the declared parameters. The object used to invoke the method should also be of a suitable type. Furthermore, if the resolved method is a protected method from a superclass of the current class and if the current class is in a different Java package than the class in which the resolved method is defined, the type of the object we used to invoke the method must be the current class or a subclass of the current class.

5.2.2 CLDC Bytecode Verifier

The new CLDC bytecode verifier specification is introduced to address the concerns of speed, memory requirement, and complexity of the original iterative bytecode verification algorithm. (The original iterative algorithm is described in Chapter 4 of the *Java Virtual Machine Specification* [45]). The CLDC bytecode verification algorithm is "lightweight" [9]. It is described in Appendix One of *CLDC 1.1 Specificiation* [42]. The new algorithm was originally designed for JVMs that execute on resource constrained devices such as cellphones. The next version of the regular JVM implementation, Mustang, will implement a similar lightweight bytecode verification algorithm.

The original bytecode verifier described in the regular JVM specification is "heavyweight". It uses an iterative type inference process to infer the most general type of runtime values that may appear when program execution reaches any given instruction in the method. It then checks the correctness of executing the instruction in that context. The iterative algorithm is expensive in both time and memory requirements.

Furthermore, the original algorithm is also more complex. It is difficult to ensure that the algorithm is correct. It is even more difficult to verify that an implementation of the algorithm is correct. Bugs have been found in the bytecode verifier implementation in the past [8, 38].

The new CLDC bytecode verifier implements a type checking algorithm. Different from the iterative type inference algorithm, the type checking algorithm effectively demands that the type information of the JVM state be given for each instruction before the bytecode verification starts.

With certain simplifications, the bytecode verifier can be considered to take an association list and a method as its inputs. The association list is understood to map each instruction of the method to an abstract state. The abstract state records type information for the operand stack and the locals of some JVM state as well the type hierarchy information. It is expected that when a JVM's execution reaches the given instruction, the state of the JVM will be "approximated" by the associated abstract state.
To check a method,

- The bytecode verifier constructs an initial abstract state from the method declaration.

  It checks that the initial abstract state is compatible with the abstract state recorded for the first instruction in the association list.

- The bytecode verifier checks for each instruction with respect to the association list, that

- it is safe to execute the instruction in the corresponding abstract state (as recorded in the association list).

  Typical checking includes whether the recorded abstract state has the right number of operands of suitable types for executing the current instruction, and whether it is allowed to access a method or a field of another class.

- it updates the abstract state to obtain the next abstract state according to the JVM semantics. It checks to confirm that the resulting abstract state is in fact compatible with the abstract state provided for executing the next instruction.

  For example, in cases of branching instructions, such as `ifeq` or `tableswitch`, the resulting abstract state must be compatible with given abstract states at all possible jump targets. If the instruction is protected by an exception handler, the state must be compatible with the given abstract state associated with the entry point of the exception handler.

The method is accepted if and only if the bytecode verifier can execute the entire method without an error, i.e. safety conditions for executing instructions are never violated and no incompatibility exists between the abstract state produced by an abstract step and the recorded abstract state for the next instruction.

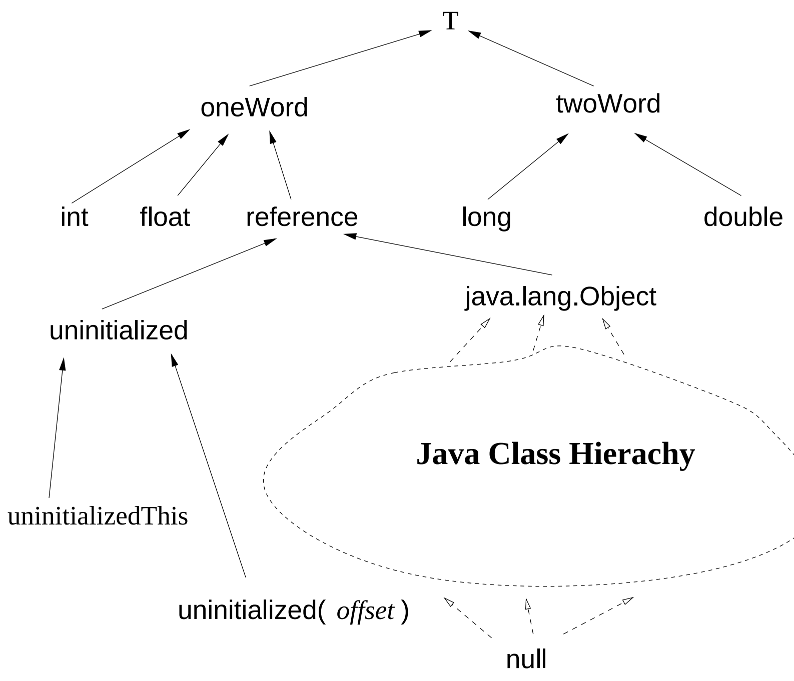


Figure 5.1: Bytecode verifier: type hierarchy

5.2.3 Type Hierarchy

Figure 5.1 shows types and the type hierarchy used by the CLDC bytecode verifier [11]. The upward arrow indicates the assignment compatible relation between types.

This picture is reproduced from the official bytecode verifier specification. It is used in the specification as a tool to help people to obtain an intuitive grasp of the relevant Prolog-style rules. The relevant rules encode type hierarchy information.

In our study of the formal specification, we noticed with surprise that the rules given in the formal bytecode verifier specification do not match with this intuitive picture — the actual `isAssignable` relation enforced by the official bytecode specification is *not* transitive. For example, `java.lang.Object` is assignable to any interface class and any array type is assignable to `java.-lang.Object`, however no array type is assignable to any interface type.

The following are the actual rules from the specification:

```
isJavaAssignable(class(_), class(To)):-
    loadedClass(To, ToClass),
    classIsInterface(toClass).


isJavaAssignable(arrayOf(_), class('java/lang/Object')).
```

The first rule asserts that any class is *Java assignable* to class `To`, if a class of name `To` exists and represents an interface type. The second rule asserts that any array type is *Java assignable* to `java.lang.Object`.

If the *Java assignable* relation were transitive, any array type would be assignable to any loaded interface class. However, the only other rule that can derive a `isJavaAssignable(arrayOf(_), class(_))` term is the following:

```
isJavaAssignable(arrayOf(_), class(X)):-
     isArrayInteface(X).
```

However, the specification notes that there is no rule to derive `isArrayInter-face`, because the array class from the CLDC JVM does not implement any interfaces. As a result, no array type is *Java assignable* to class `C`, if `C` is neither an array type nor a `java.lang.Object`.

This fact that `isJavaAssignable` is not transitive implies that while some assignments of an array reference into a slot of interface type will be detected at the bytecode verification time, some other assignments of an array reference into a slot of interface type will have to be detected at runtime and the JVM will throw an exception. [1]

5.2.4 Specification: AASTORE

The specification of the bytecode verifier is given in the form of Prolog-style declarative rules. As we have shown in the previous section, assignment compatible judgement is expressed by conditions showing whether a specific term is derivable from the set of rules. Similarly, whether it is safe to execute an instruction in an abstract state is also given by whether some specific term is derivable.

For example, the following is the rule for deciding whether it is safe to execute `AASTORE`. `AASTORE` stores an object reference into a given offset of an array.

```
instructionIsTypeSafe(aastore, _Environment, _Offset,
                      StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    canPop(StackFrame,
           [class('java/lang/Object'),
            int,
            arrayOf(class('java/lang/Object'))],
           NextStackFrame),
    exceptionStackFrame(StackFrame,
                        ExceptionStackFrame).
```

[1]This is because some array reference types can be generalized to `java.lang.Object` at bytecode verification time and thus be judged by the BCV as suitable for assigning into a slot for holding an interface value.

This rule literally means if one can find some substitution for variables `StackFrame`, `NextStackFrame`, and `ExceptionStackFrame` such that both `canPop(...)` and `exceptionStackFrame(...)` are derivable under that substitution, one can then derive the term `instructionIsTypeSafe(aastore, ....)` in which `StackFrame`, `NextStackFrame`, and `ExceptionStackFrame` are instantiated according to the original substitution, while `_Environment` and `_Offset` can be instantiated arbitrarily.
This rule can also be understood operationally as expressing:

- `AASTORE` instruction is checked in an abstract state. The abstract state is comprised of an `_Environment`, a program counter value `_Offset` and a `StackFrame`. The `StackFrame` records an abstract operand stack and an abstract local variable array as well as some other components (not shown here).

- In order for `AASTORE` to execute safely, one must be able to show that it is possible to pop three values from the abstract operand stack of the `StackFrame`. The three values must be assignment compatible to `java.lang.Object`, `int`, and array of `java.lang.Object`.

- If `AASTORE` can execute safely, the resulting abstract state is updated to `NextStackFrame`. The `ExceptionStackFrame` is obtained by popping every value off the operand stack of the original `StackFrame`.

We call this understanding an operational view of the rules. The difference between the operational view of the Prolog-style rules and the literal interpretation of them are the following:

- The operational interpretation obliges us to map the function symbol of a term into an operation.
- The operational interpretation obliges us to identify the input and output for each operation.
- The operational interpretation obliges us to clearly identify the condition under which an operation can be applied.
- The operational interpretation obliges us to clearly separate the state changing effects of applying one operation from the condition under which the operation can be applied.

We think that an operational interpretation of the rules is more helpful in guiding an implementation of an actual bytecode verifier. In the next section, we explain our implementation of the bytecode verifier and how we translate the original rules into their operational counterparts.

5.3 Executable BCV Model in ACL2

We derived the executable BCV model from the specification. The model in ACL2 has 480 function definitions in 4500 lines, which corresponds to over 200 rules described in the official (CLDC) bytecode verifier specification.

Our bytecode verifier model is executable. We use our `jvm2acl2` tool to translate a binary class file into a format readable by our bytecode verifier. We translated the entire CLDC class library (115 classes, 927 bytecode methods, 16723 instructions). We ran our bytecode verifier on the library.[2] The verifier

[2]We also ran our bytecode verifier on the translated JDK1.3.1's class library, which contains 5,273 classes, 34,177 methods, and 821,627 instructions.

verified all methods in a context where all classes are loaded. The running time is about 0.15 second on a Pentium M 1.6GHz laptop, achieving a bytecode verification speed of over 100K instruction per second.

5.3.1 Input Format

Class and methods

Figure 5.2 on the next page shows one class file from the CLDC library. Our bytecode verifier takes a list of classes in this format, and verifies every method.

Type annotations for the method

With a little simplification, the basic algorithm for verifying one method (as explained in the section 5.2.2) takes two inputs, the instructions and an association list that maps each instruction to an abstract state.

An abstract state (associated with instruction `X`) records the types of values that exist in the concrete execution state. An abstract state characterizes the expected execution state of the thread right before the that thread executes instruction `X`. The bytecode verification algorithm checks the instructions against the association list that maps instructions to abstract states. One may view the association list as type annotations for the methods.

The actual bytecode verifier algorithm does not expect the method to have type annotations for each instruction. The bytecode verification actually works on a list of instructions and a list of type annotations (*StackMaps*) for a subset of the instructions in the method.

Such type annotations (StackMaps) are recorded in our class representation. For example, the constructor method (figure 5.2) has the StackMaps

```
; A comment starts with a semicolon
; and extends to the end of the line.

'(CLASS  "com.sun.cldc.io.DateParser" ; class name
  NIL                          ; not an interface class
 "java.lang.Object"            ; super is java.lang.Object
 (CONSTANT_POOL (INT 1721426)  ; the constant pool entry
                (STRING "Jan") ;
                ....)
 (INTERFACES)                  ; implements no interfaces
 (FIELDS (FIELD "year" INT
                (ACCESSFLAGS *CLASS* *PROTECTED*) ...)
         ....)                 ; a protected field
 (METHODS                      ; list of method
  (METHOD
   "<init>"                    ; constructor for DateParser
   (PARAMETERS INT INT INT INT INT INT)
   (RETURNTYPE VOID)           ; return none
   (ACCESSFLAGS *CLASS*)
   (CODE
     (MAX_STACK 5)           ; use at most 5 slots for op stack
     (MAX_LOCALS 7)          ; use at most 7 slots for locals
     (CODE_LENGTH 329)       ; instructions are 329 byte long
     (PARSEDCODE
          (0 (ALOAD_0))
          (1 (INVOKESPECIAL  ; call constructor
                   (METHODCP "<init>" "java.lang.Object" ...)))
          ....))))))
```

Figure 5.2: Input to the bytecode verifier

in figure 5.3.

It is expected that when the execution of the init method reaches the instruction at offset 246, the machine state will have the following signature:

```
(246 (frame
      (locals (class "com.sun.cldc.io.DateParser")
              int int int int int int)
                ; locals contains values of specific types
      (stack )  ; the operand stack is empty
      nil))
```

That is, the operand stack is empty, and the local variable at slot 0 is a reference to a com.sun.cldc.io.DateParser object and the remaining 6 slots contain values of type int.

5.3.2 Algorithm

The bytecode verifier implements the operation for checking whether a class is safe as shown in figure 5.4
This operation takes two arguments. The `Class` is the class that needs to be checked. The `CL` is the class table that we want to check the `Class` against. `CL` encodes the Java type hierarchy information as shown in figure 5.1 on page 133.

First, the bytecode verifier checks that `Class` is a syntactically correct with (isClassTerm Class). Then it checks that the current class is either `java.lang.Object` or the superclass of the current class exists and is not marked with a `Final` flag — (classIsNotFinal superClass). It also checks that all methods from the class are well-typed with (checklist-methodIsTypeSafe Class Cl Methods).

```
(method "<init>"
         (parameters int int int int int int)
         (returntype void)
         (accessflags  *class* )
         (code
          (max_stack 5) (max_locals 7) (code_length 329)
          (parsedcode
           (0 (aload_0))
           (1 (invokespecial
                (methodCP "<init>" "java.lang.Object" () void)))
           ....)
          ...
          (StackMap
           (246 (frame
                 (locals (class "com.sun.cldc.io.DateParser")
                         int int int int int int)
                 (stack )
                 nil))
           (282 (frame
                 (locals (class "com.sun.cldc.io.DateParser")
                         int int int int int int)
                 (stack )
                 nil))
           (290 (frame
                 (locals (class "com.sun.cldc.io.DateParser")
                         int int int int int int)
                 (stack )
                 nil)))))
```

Figure 5.3: Type annotations on a method

```
(defun classIsTypeSafe (Class CL)
  (let ((Methods (classMethods Class))
        (Name    (classClassName class)))
    (prog2\$
     (acl2::debug-print  "Checking class ~p0 ~%"
                                         (classClassName class))
     (and (isClassTerm Class)
          (or
           (equal Name "java.lang.Object")
           (let*
               ((superClassName (classSuperClassName Class))
                (superClass (class-by-name superClassName CL)))
             (classIsNotFinal superClass)))
          (checklist-methodIsTypeSafe Class Cl Methods)))))
```

Figure 5.4: Bytecode verifier: `classIsTypeSafe`

The core of the bytecode verifier is embodied in the methodWithCodeIsTypeSafe, which describes how it checks for a regular method with instructions. The bytecode verifier first collects the relevant information out of the method, (such as `FrameSize`, `MaxStack`) and creates a data structure called `Environment` to hold them. In particular, at this information collection stage, the instructions `ParsedCode` and the their corresponding type annotations `StackMaps` are merged together into a single list `MergedCode`. The bytecode verifier invokes `mergedCodeIsTypeSafe` to check whether the resulting merged code is type safe in the context of the newly constructed `Environment`.

As a concrete example, figure 5.6 shows a merged sequence of instructions and type annotations from the `byteToCharArray` method of the `com.sun.cldc.i18n.Helper` class.

To check whether the merged code (instructions with type annotations mixed in) is safe to execute on the JVM, the bytecode verifier uses the following

```
(defun methodWithCodeIsTypeSafe (Class Method CL)
  (and (isWellFormedCodeAttribute Class Method)
       (let* ((FrameSize  (FrameSize method Class))
              (MaxStack   (MaxStack   Method Class))
              (ParsedCode (ParsedCode Method Class))
              (Handlers   (Handlers    Method Class))
              (StackMaps  (StackMap   Method Class))
              (MergedCode (mergeStackMapAndCode
                                StackMaps ParsedCode))
              (StackFrame (methodInitialStackFrame
                                Class Method
                                FrameSize))
              (ReturnType  (methodReturnType Method))
              (Environment (makeEnvironment Class Method
                                  ReturnType
                                  MergedCode
                                  MaxStack
                                  Handlers
                                  CL)))
         (and (handlersAreLegal Environment)
              (mergedCodeIsTypeSafe Environment
                                    MergedCode
                                    StackFrame)))))
```

Figure 5.5: Bytecode verifier: `methodWithCodeIsTypeSafe`

```
(8 (ILOAD_1))
(9 (INVOKESTATIC (METHODCP "toString"
                           "java.lang.Integer" (INT)
                           (CLASS "java.lang.String"))))
(12 (INVOKESPECIAL (METHODCP
                     "<init>"
                     "java.lang.IndexOutOfBoundsException"
                     ((CLASS "java.lang.String"))
                     'VOID)))
(15 (ATHROW))
(STACK_MAP (16 (FRAME (LOCALS (ARRAY BYTE)
                              INT INT
                              (CLASS "java.lang.String")
                              TOPX TOPX TOPX)
                      (STACK)
                      NIL)))
(16 (ILOAD_2))
(17 (IFGE 32))
```

Figure 5.6: Bytecode verifier: a fragment of the merged code

one-pass algorithm. The algorithm maintains an abstract state during its execution.

- If the current abstract state is the special state `aftergoto`,
  - If the head of the merged code indicates that it is the end of the instructions, the method is considered verified.
  - If the head of the merged code is a type annotation, the verifier uses the type annotation as its new current abstract state and continues to verify the tail of the merged code.
  - If the head of the merged code is a regular instruction, the verification fails.
- If the head of the merged code is a type annotation, the bytecode verifier checks whether the current abstract state is compatible with the type annotation.
  - If it is compatible, the bytecode verifier uses the type annotation as its current abstract state and continues.
  - If the type annotation is not compatible with the current abstract state, the method verification fails.
- If the head of the merged code is a regular instruction, the bytecode verifier checks whether it is safe to execute the instruction in the current abstract state.
  - If it is safe to execute the instruction, the bytecode verifier updates its current abstract state to the result of executing the instruction symbolically and continues the method verification process.

```
(defun mergedCodeIsTypeSafe(Environment MergedCode StackFrame)
   ....
  (let ((cur (first MergedCode))
        (rest (rest MergedCode)))
  (cond
   ((isStackMap cur)
    (let ((MapFrame (mapFrame (getMap cur))))
     (and (frameIsAssignable StackFrame MapFrame Environment)
          (mergedCodeIsTypeSafe Environment rest MapFrame))))
   ((isInstruction cur)
    (let ((offset     (instrOffset cur))
          (instr      cur))
      (if (instructionIsTypeSafe instr Environment StackFrame)
           (mv-let (NextStackFrame ExceptionStackFrame)
            (sig-do-inst instr Environment StackFrame)
            (and (instructionSatisfiesHandlers
                     Environment offset ExceptionStackFrame)
                 (mergedCodeIsTypeSafe Environment
                                       rest
                                       NextStackFrame)))
          nil))))))
```

Figure 5.7: Bytecode verifier: key algorithm

  – If it is not safe to execute the instruction, the verification process fails.

As one may observe, the bytecode verification algorithm is described in a very peculiar way. One may be wondering why we need first collect pieces of information into an `Environment`, why we need to first merge the instructions with their type annotations into a single list, and why we have to define `mergedCodeIsTypeSafe` to check the type safety against the merged code.

The explanation is that our ACL2 implementation of the bytecode verifier is systematically derived from the official CLDC bytecode verification specification. The official specification is formulated in this way.

The official bytecode verification formalizes the bytecode verification as a set of Prolog-style rules. In order to make the official specification executable, the declarative rules are over-engineered to express ideas that could be more naturally expressed with a conventional language. As a result, some of the rules from the specification are themselves convoluted. To keep our bytecode verifier model faithful to the official specification, our bytecode verifier also contains similar peculiarities in its implementation of the intuitively simple bytecode verification algorithm.

This peculiar way of specifying the bytecode verifier adds to the difficulty in proving that the bytecode verification algorithm is effective. One of the contributions of this dissertation is that we designed a simplified version of the bytecode verifier. We used the simplified bytecode verifier as an intermediate step for proving bytecode verification effective. To prove that the bytecode verification algorithm (as specified in the official CLDC document) is effective, we have first proved a *reduction* theorem that if the official bytecode verifier verifies a program, our simplified version will also verify it. We also have an incomplete proof that suggest that all verified programs (accepted by the simple bytecode verifier) will execute safely — together with an approach and a support lemma library for completing the proof.

5.3.3 Connection with the CLDC Specification

We derived our bytecode verifier from the Prolog style rules by collecting the distinct function symbols of all the *head terms* of Prolog rules and defining one ACL2 function for each such function symbol.

For example, the function symbol `classIsTypeSafe` appears and only appears in the head term of the following rule.

```
classIsTypeSafe(Class) :-
    ...
    classMethods(Class, Methods),              ; extract methods
    checklist(methodIsTypeSafe(Class), Methods).; verify all methods
```

We introduce a corresponding ACL2 definition of the same name,

```
(defun classIsTypeSafe (Class CL)
  (let ((Methods (classMethods Class)))
      ...
   (checklist-methodIsTypeSafe Class Cl Methods)))
```

The actual definitions of the corresponding ACL2 functions for Prolog clauses depend on the form of the clauses. We need to identify the inputs and outputs of an operation by studying the form of related clauses. We need to identify the relations between arguments of the head term. For each clause, we start by trying to determine whether bindings for some subset of the arguments uniquely determine the bindings for the remaining arguments. Consider the Prolog clause with head term `classClassName(Class, ClassName)`, in which it is obvious that once `Class` is bound to some concrete object that represents a class, the body of the clause uniquely determines the binding for `ClassName`. As a result, the corresponding ACL2 function takes a `Class` as its argument and returns its class name.

We introduced three kinds of ACL2 functions. Each kind corresponds to a specific pattern in the official Prolog-style rules.

- Accessor operations take one compound object and return one or more components of the compound object.

  One example is `classClassName(Class, ClassName)`. We translated it into an accessor function `classClassName(Class)`, which takes one input that presents the class and returns the class name component.

- Boolean predicates, like `classIsTypeSafe`. They characterize properties of the inputs. We effectively translated the derivability of a Prolog term into the truth value returned by the corresponding ACL2 function.

- Guarded operations, like `instructionIsTypeSafe`. They not only decide whether a term is derivable, they also give the binding for some non-input arguments.

  To be more concrete, take the clause for deriving `instructionIsTypeSafe(aastore, ...)` as an example. We translated the clause into a pair of ACL2 functions:

  – A guard function that defines the condition under which the operation is legal:

    ```
    (defun check-aastore (inst env StackFrame)
      (declare (ignore inst))
      (canPop Env StackFrame
              '((class "java.lang.Object")
                int (array (class "java.lang.Object")))))
    ```

  – An effect function that defines the effects of applying the operation:

    ```
    (defun execute-aastore (inst env StackFrame)
      (declare (ignore inst))
    ```

```
(mv (TypeTransition env
                   '((class "java.lang.Object")
                     int
                     (array (class "java.lang.Object")))
                   'void
                   StackFrame)
    (exceptionStackFrame StackFrame)))
```

We invite the reader to compare the supporting materials for the actual definition of our bytecode verifier [22] with the Prolog rules from the official bytecode verifier specification [11]. Our bytecode verifier definitions should be readily readable as a Common Lisp program. It follows closely with the Prolog-style rules.

5.4 A Simple Property Proved

We proved one property of our JVM bytecode verifier model. The property asserts that the result of bytecode verifying one method is independent of the code from other methods. The proof is mechanically checked with ACL2.

We were prompted to prove this intuitive property because of the following practical problem. We translated the entire class library of the J2SE 1.3.1 into our `jvm2acl2` format. This particular library contains 5,273 classes and over 34,000 methods. The entire class table in our format is a 190MB text file. Our Common Lisp environment (GNU GCL) cannot load such a big Lisp constant. To test our bytecode verifier on the entire class table, we needed to reduce the memory requirement. We abstracted each class by removing the actual code of each method. The resulting abstract class table can be loaded

```
(defthm classIsTypeSafe-judgment-is-unchanged
  (implies (isClassTerms cl)
     (iff (classIsTypeSafe C cl)
          (classIsTypeSafe C (abstract-class-table cl)))))
```

Figure 5.8: One simple bytecode verifier property verified

into the memory by GCL. We ran our bytecode verifier to verify classes one at a time with respect to the abstract class table that contains all classes modulo the actual method bodies.

We proved that verifying the bytecode verifier produces the same answer whether we use the actual class files or the abstract ones. Figure 5.8 shows the actual ACL2 theorem that we proved.

The name of the theorem is `classIsTypeSafe-judgment-is-unchanged`. It asserts that if `cl` is a list of classes, then for any class `C`, `(classIsTypeSafe C cl)` judgment is always the same with `(classIsTypeSafe C (abstract-class-table cl))`, where `(abstract-class-table cl)` strips off the code array from all methods contained in the `cl`.

This property is very intuitive. The bytecode verifier essentially executes a type checking algorithm. The necessary pieces of information are the type hierarchy information, the access permission information, and the instruction sequences being verified. Thus, stripping the actual code from all methods will not affect any judgment made on some known instruction sequence.

The proof is not as trivial as it may seem to be. We are proving a property of a 4500-line Common Lisp program. JVM types are no longer abstract entities, but instead are mapped to ACL2 objects manipulated by the concrete ACL2 functions. Our ACL2 proof script is about 500 lines.

To prove the theorem, we first formalize the intuitive reason why stripping off the code array does not matter. We identify an equivalence relation on the possible class tables. We show that (1) judgment about relations between types only depends on the equivalence class that the class table belongs to, and (2) the abstraction step produces a class table equivalent to the original one.

For example, we define the following predicate with two inputs. We use it to characterize a relation on the class table.

```
(defun class-table-type-equiv (ct1 ct2)
  (if (endp ct1)
      (endp ct2)
    (if (endp ct2)
        nil
    (and (class-def-type-equiv (car ct1) (car ct2))
         (class-table-type-equiv (cdr ct1) (cdr ct2))))))
```

This relation asserts that two class tables `ct1` and `ct2` are equivalent if and only if the class representations contained in them are pairwise `class--def-type-equiv`. We need to prove this relation is an equivalence relation, i.e. it is reflexive, transitive and symmetric. ACL2 proves the `class-table--type-equiv` is an equivalence relation automatically after we first ask ACL2 to show that `class-def-type-equiv` defines an equivalence relation. Two classes are `class-def-type-equiv` if the only differences between them are in method codes.

We then prove `isJavaSubClassOf` admits `class-table-type-equiv` as a congruence relation, so that ACL2 knows it is safe to replace a class table with some equivalent one for making judgments on whether one class is a subclass of another class.

5.5 Conclusion

We presented the official lightweight bytecode verifier from the latest CLDC specification. We described the type hierarchy used by the bytecode verifier. We showed examples of the Prolog style rules. We discussed the need for providing an operational interpretation of the formal Prolog-style rules used to specify the bytecode verifier.

We presented our formal executable model of the BCV. The model is derived from the Prolog rules. The model closely follows the BCV specification and executes efficiently, achieving a bytecode verification speed of 100K instruction per second.

The model is amendable to formal reasoning. We proved one intuitive theorem that asserts that bytecode verification depends only on the class hierarchy and the code being analyzed.

This work on modeling a realistic BCV is one step in our project of formalizing and verifying the JVM safety.

Chapter 6

JVM Safety Specification

6.1 Two Kinds of Safety

When people talk about Java being *safe*, we think that they expect two kinds of safety guarantees: (1) all reachable JVM states are “good” and all state transitions meet suitable preconditions and (2) executing arbitrary Java programs on a JVM implementation will not induce unexpected behavior of the JVM implementation.

The first kind of guarantee provides the comforting assurance for Java programmers that their program will be executed in a way consistent with their understanding of the JVM — operations will be applied to operands of correct type, all private data and functionalities will be protected against unauthorized access, and the reachable JVM states remain sensible.

The second kind of safety guarantee provides the assurance for JVM users that executing an arbitrary Java program via a correct JVM implementation will not produce unexpected effects in the environments in which they execute the JVM implementation. We may view a JVM implementation as providing a virtual universe of Java objects. Executing a Java program is to carry out the operations on objects from this virtual world. We expect that Java programs that execute inside the virtual world will not unexpectedly affect the outside environment. [1]

These two kinds of safety guarantees are in fact closely related. The first kind asserts that all reachable states are "good" JVM states and all operations are executed with their preconditions met. These are two useful facts to JVM implementors. As long as they can find a low level representation that can represent all the "good" JVM states and can emulate all the operationally defined JVM primitives, we can be sure that such a JVM implementation will provide the second kind of safety guarantee — the JVM implementation provides a confined sand box in which to execute Java programs.

As a concrete example, if we can prove during arbitrary execution of the JVM that the size of the operand stack is bounded by a fixed number, a JVM implementor can then chose to represent a JVM operand stack with a fixed sized memory section, knowing that such a representation always suffices. Similarly, if we can show that the `POP` operations are always executed with at least one operand on the stack, the JVM implementation can execute the low level operations that emulates the `POP` operations — knowing that the low level representation of the stack will always represent a stack with at least one operand.

For the JVM implementors, the more they know about constraints on a "good" JVM state, the more likely they can find an efficient low level repre-

[1]Some operations in this virtual world, e.g., printing, may have expected side effects on the outside environment, e.g., changing the pixels on the screen.

sentation to represent all the “good” JVM states. The more they know about the preconditions for executing a JVM operation, the more likely they can find an efficient way to emulate the JVM operation effectively.

JVM implementors can also benefit from a more direct formulation of “good” state. In order to write a correct JVM implementation, JVM implementors are obliged to check — for each JVM operation that they implement — whether executing their implementation of the operation on the low level state representation of a “good” state will result in a suitable low representation that also represents the next state and whether that next state is a “good” state.

One way to define a “good state” is as follows: (1) The state has property `G` and (2) all reachable states from the state also have property `G`.

However, a “good state” defined in this fashion is not directly useful to JVM implementors. To check whether a state is a “good state”, they need to reason about the JVM *in addition to* the state itself. It is better to have a more direct definition of a “good state” such that, to know whether a state is a “good” state or not, the JVM implementors do not need to reason about the JVM as a whole and only focus on the inherent property of the state itself.

In this dissertation, we formalize the first type of safety guarantee that the JVM provides. We identify a stronger version of “good” state (stronger than what is declaratively specified in the JVM specification). Whether a JVM state is a “good” state can be checked directly against the JVM state — without the need to check any requirements on all reachable states by the JVM from the state. We also identify a stronger version of preconditions for executing JVM operations. [2]

[2]We may also observe that our decision to identify stronger versions of the “good” state

6.2 Defensive JVM

6.2.1 Why Another JVM Model?

We formalize the first kind of safety guarantee. We want to show that program executions according to the operational specification of the JVM will not result in a "bad" state where the next state is not defined. Our JVM model, M6, embodies the specification that how JVM operations would execute operationally (see Chapter 4). We still need to capture the constraints under which, these operations can be executed. The existing M6 state does not maintain enough information to describe these constraints. This leads to the development of a new JVM model that we will describe in this section.

The JVM specification describes the operational semantics of the JVM instructions only when a certain set of constraints are met. Otherwise the semantics of executing the instruction is left undefined.

For example, `AALOAD` is an instruction that loads a value from the `index` slot of an array in the heap. The array object is expected to be holding an array of references to other objects. Let the reference to the array object be `arrayref`, the JVM specification stipulates the following constraints:

> the `arrayref` *must* be of type reference and *must* refer to an array whose components are of type `reference`. The `index` *must* be of

and preconditions is not only motivated by trying to create a more useful safety specification to the JVM implementors, it is also a necessary step for us to prove that the JVM as originally specified can provide the original (weaker) form of the safety guarantee. For example, suppose originally, the only requirement for a state to be a "good" state is that its operand stack size is within a fixed bound. However, as we will see in section 7.1 of this dissertation, to prove that all reachable states — while executing verified programs — have bounded operand stacks, induction requires that we strengthen the definition of the "good" states by demanding an additional *on-track* requirement.

type `int`. Both `arrayref` and `index` are popped from the operand stack. The reference value in the component of the array at index is retrieved and pushed onto the operand stack.

We need to write an ACL2 predicate that capture these constraints. To prove that the JVM is safe, we need to show when the JVM is about to execute an `AALOAD` instruction, such constraints are always met.

In order to capture the *must* requirements, we need to maintain type information for the values on the operand stack and the local variables. However, M6 only follows the operational part of the JVM specification and does not maintain the necessary information to express these requirements.

We define a "defensive" machine in parallel with M6. [3] The defensive JVM (DJVM) maintains the extra type information during the execution, before executing an instruction, the DJVM "defensively" looks for any potential causes for "bad" events and only executes a step if the operational semantics is defined according to the official JVM specification (JVMSpec) [45].

To prove that the JVM specification provides the first kind of safety guarantee (as asserted by JVMSpec) is to prove that executing DJVM does not get stuck and that the execution matches the corresponding M6 execution.

[3]We have not defined a complete DJVM that can execute realistic Java bytecode program like M6 can. Our DJVM model is still only a set of ACL2 definitions for eight kinds of bytecode instructions (`AALOAD`, `AASTORE`, `ALOAD`, `ASTORE`, `ANEWARRAY`, `ACONST_NULL`, `GETFIELD` and `IFEQ`) and primitives for implementing these instruction. These primitives include simple operations to manipulate the operand stacks. These primitives also include more complicated ones that load a class definition.

6.2.2 State Representation

In a regular JVM, the following information needs to be maintained to implement necessary runtime checks and JVM functionalities:

- class hierarchy information,
- access permissions to fields, methods, and classes
- type information for objects that exist in the heap, and
- the context of execution, including the current program counter, the current executing method, the operand stack, and the locals are being updated.

A defensive JVM needs to maintain additional information in its execution state:

- type information for values that reside in activation records,
- resource limits on various operand stacks and local variable arrays, and
- the initialization status of every Java object in the heap.

Using the `AALOAD` example from the previous section, one can verify that the following constraints can in fact be checked with the additional information maintained by the defensive machine.

- Current thread exists.
- Current call frame exists.
- Top element of the operand stack exists.

- Top element of the operand stack is of type `int`.
- Second element of the operand stack exists.
- Second element of the operand stack is a `reference` type.
- One of the following two is true:
  - the second element of the operand stack is a `null` reference,
  - the conjunction of
    - the object pointed by the reference at the second element is of a valid array type, and
    - the component type of the array is not a primitive type.
- Executing the instruction will not overflow the operand stack.

Like an M6 state, a DJVM state is an eight-tuple consisting of a program counter, a current thread register, a heap, a thread table, an internal class table, an environment, an auxiliary field, and an error flag.

The internal class table records the loaded classes; the environment records the sources from which a class definition is to be loaded; the heap maps addresses to Java instance objects.

The one obvious difference between an M6 state and a DJVM state is that values in the operand stacks and the local variable arrays are tagged with their types. For example, the corresponding DJVM thread for the M6 thread (figure 4.2) is depicted in figure 6.1 on the next page.

Another difference between an M6 state and a DJVM state is that the auxiliary field of a DJVM state not only records the pending exception that

```
(THREAD 0
 (SAVED-PC . 0)
 (CALL-STACK
  (FRAME (RETURN_PC . 7)   ; this frame represents the call
         (OPERAND-STACK)   ; to FactHelper's constructor
         (LOCALS (REF . 104))  ; a reference type
         (METHOD-PTR "FactHelper" "<init>" ...)
         (SYNC-OBJ-REF . -1))
  (FRAME (RETURN_PC . 18) ;
         (OPERAND-STACK (REF . 104))
         (LOCALS (REF . 102)) ; a reference type
         (METHOD-PTR "FactHelper" "compute"...)
         (SYNC-OBJ-REF . -1))
   ...)
 (STATUS THREAD_ACTIVE)
 (MONITOR . -1)
 (MDEPTH . 0)
 (THREAD-OBJ . 55))
```

Figure 6.1: Representing a DJVM thread

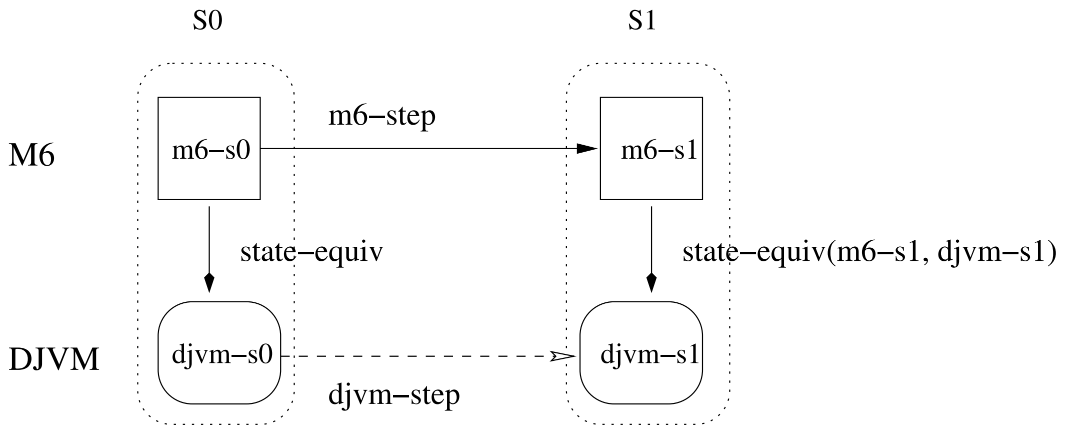


Figure 6.2: Relating the DJVM and the M6

needs to be handled, it also records the object initialization status for every object present in the heap.

Intuitively, as pictured in figure 6.2, some M6 state and some DJVM state might represent the *same* state, and if an M6 state `m6-s0` and a DJVM state `djvm-s0` do represent the same state `S0`, we expect that, by executing M6 one step from `m6-s0` and executing DJVM one step from `djvm-s0`, the two resulting states will still represent the same state `S1`. We formalize the connection between such an M6 state and its corresponding DJVM state in the `state-equiv` (see Figure 6.3).

A M6 state `m6-s` is `state-equiv` with a DJVM state `djvm-s` if and only if after stripping the types from the `djvm-s` state, the `(untag-state djvm-s)` is only different from the `m6-s` by having a `thread-table-equiv` thread-table. [4]

[4]This definition of `state-equiv` is not correct. Because a DJVM state also keeps track of the object initialization status for each objects. The DJVM maintains such information during its execution, however M6 will not keep such information. For a cut down version of the M6, M6', we have not defined the DJVM instruction `NEW` nor `INVOKESPECIAL` that updates the object initialization status. The current definition of `state-equiv` suffices. We

```
(defun state-equiv (m6-s djvm-s)
  (and (thread-table-equiv (thread-table m6-s)
                           (thread-table djvm-s))
       (equal (state-set-thread-table
                (thread-table m6-s)
                (untag-state djvm-s))
              m6-s)))
```

Figure 6.3: Relating DJVM and M6 states: `state-equiv`

It is necessary for us to define `state-equiv` in this particular way. The apparent (however, incorrect) alternative is to define `state-equiv` by asserting that the DJVM state with type annotation stripped is equal to the M6 state that it relates to.

```
(defun state-equiv (m6-s djvm-s)
   (equal (untag-state djvm-s)
          m6-s))
```

Figure 6.4: Incorrect `state-equiv` definition

This alternative definition of `state-equiv` is incorrect. This definition dictates that for each DJVM state `djvm-s`, there is one and only one M6 state `(untag-state djvm-s)` that is `state-equiv` with the original `djvm-s`. This is not the case. In any M6 state, slots in local variable arrays may contain values that will never be used before being overwritten or discarded. These memory slots are *irrelevant slots*. That is, information stored in them is irrelevant to the program execution. We want the set of M6 states to correspond to a same DJVM state, if the only difference among the M6 states is that they contains different values in these irrelevant slots.

can prove that M6', a cut-down version of M6, executes verified program safely.

```
(defun resolveMethodReference (method-ptr needStatic? s)
  (mylet* ((classname (method-ptr-classname method-ptr))
           (new-s (resolveClassReference classname s))
           (thisMethod (lookupMethod method-ptr new-s)))
    (if (pending-exception new-s)
        (mv nil new-s)
      (if thisMethod
              ....
              (mv thisMethod new-s)
            (mv nil (state-set-pending-exception-safe
                     "java.lang.IllegalAccessException" new-s))
              ....
        (mv nil (state-set-pending-exception-safe
                 "java.lang.NoSuchMethodError" new-s))))))
```

Figure 6.5: Shared JVM operations: `resolveMethodReference`

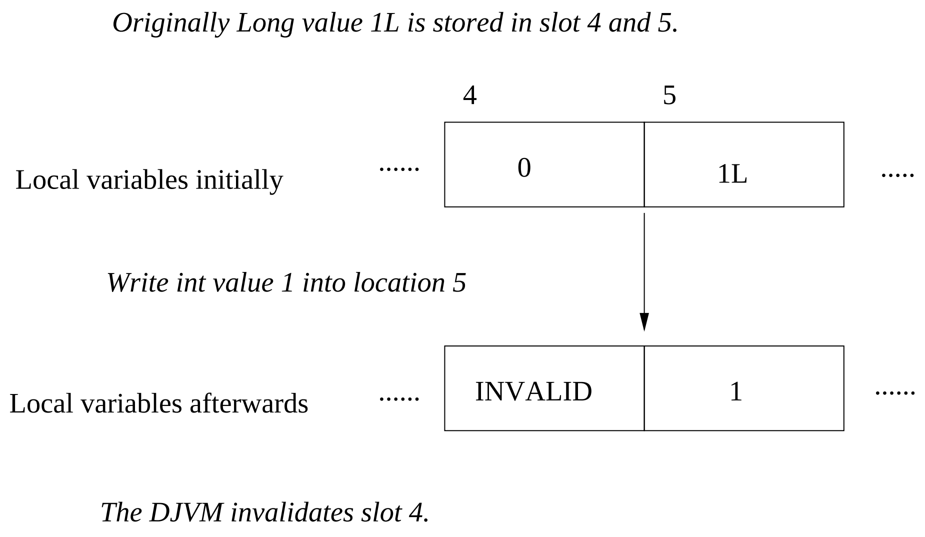
6.2.3 State Manipulation Operations

DJVM and M6 have very similar state representations. DJVM reuses many M6 operations to access and update the DJVM state components that share the same representation with the corresponding M6 state components.

For example, in both the DJVM and M6, the representation of the class table, heap, and environment are the same. We use the same operation `resolveMethodReference` (figure 6.5) to resolve a symbolic reference `method-ptr` to a method signature.

We also defined operations unique to DJVM. Some of these operations are used to access and update the operand stacks and local variables arrays. The operation `invalidate-category2-value` (figure 6.7) is one example.

In a JVM state, a `long` type value will occupy two adjacent slots when it is stored in a local variable array. The JVM specification demands that a Java program must never attempt to read a long value from these two slots,



Originally Long value 1L is stored in slot 4 and 5.
4
5
Local variables initially
......
0
1L
.....
Write int value 1 into location 5
Local variables afterwards
......
INVALID
1
......
The DJVM invalidates slot 4.

Figure 6.6: Corrupting a size 2 value

```
(defun invalidate-category2-value (index s)
  (if (< index 0)
      s
    (if (equal (type-size (tag-of (local-at index s)))
               1)
        s
      (state-set-local-at index '(topx . topx) s))))
```

Figure 6.7: Updating types for local variables

if one of them is reused to store some other value.

In order to detect whether a Java program is about to read a corrupted long value, we need to mark a long value as `invalid` after one of the slots for storing the long value has been written into. The `invalid-category2-value` operation marks a value contained in a specific offset of the local variable array as corrupted and not to be read from.

We also define a third kind of DJVM operation, illustrated by `safe-`

```
; DJVM operation: safe-pushStack
(defun safe-pushStack (value s)
 (declare (xargs :guard
     (and (consistent-state s)
          (not (mem '*native*
                    (method-accessflags
                     (deref-method
                          (method-ptr (current-frame s))
                          (instance-class-table s)))))
          (<= (+ 1 (len (operand-stack (current-frame s))))
              (max-stack s)))))
  (mylet* ((curthread-id (current-thread s))
           ....)
    (state-set-thread-table new-thread-table s)))

; M6 operation:  pushStack
(defun pushStack (value s)
  (declare (xargs :guard
       (and (current-frame-guard s)
            (wff-call-frame (current-frame s)))))
  (mylet* ((curthread-id (current-thread s))
           ....)
          (state-set-thread-table new-thread-table s)))
```

Figure 6.8: DJVM operation: `safe-pushStack` vs. `generic pushStack`

`-pushStack` (see Figure 6.8). The definition is essentially the same as its M6 counterpart `pushStack` — except that we have attached a stronger guard to the operation. The stronger guard of `safe-pushStack` asserts not only that the operand stack exists, it also asserts that the stack is a valid stack from a consistent DJVM state and pushing the value will not overflow the operand stack.

6.2.4 State Transition Function

For each DJVM instruction, we define a pair of operations. Consider `ALOAD`, for example. We define `check-ALOAD` and `execute-ALOAD`.

Before DJVM uses the `execute-ALOAD` to make a state transition, DJVM invokes the `check-ALOAD` operations to check whether it is safe to execute `ALOAD`. The pair of functions are composed by following the descriptions from the JVM specification. The JVM specification for `ALOAD` is shown in figure 6.9

According to this specification, the effects of executing `ALOAD` is only defined, when the index is a *valid index* into the local variable array and the value at the offset is of type *reference*.

We define the `check-ALOAD` operation (figure 6.10). The `wff-aload` asserts that the instruction is of proper form. The `valid-local-index` captures that index needs to be a valid offset into the local variable array of the current frame. The `REFp` asserts that the value at that specific offset is a *reference*.

We define the `(valid-local-index index locals)` to check whether the value at offset `index` is a valid value. A local variable array can hold values that each occupies either one or two slots. It can also contain uninitialized slots. For a certain `index` to be valid, the index cannot point to some uninitialized slots, nor can it point to the middle of a value that occupies two slots.

To check whether a value is of type *reference*, we define `(REFp v hp)`, we check that the value `v` is either a NULL value or that it points to some object in the heap `hp`. [5]

[5]Notice this is different from what the bytecode verifier checks when its symbolic execution reaches an `ALOAD` instruction. The bytecode verification process does not maintain a

Operation

Load *reference* from local variable

Format

`aload` index

Forms

`aload` = 25 (0x19)

Operand Stack

... ⇒ ..., `objectref`

Description

The index is an unsigned byte that must be an index into the local variable array of the current frame (§3.6). The local variable at index must contain a reference. The objectref in the local variable at index is pushed onto the operand stack.

Notes

The *aload* instruction cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack.

Figure 6.9: Official specification for `ALOAD`

```
(defun check-ALOAD (inst s)
  (declare (xargs :guard (consistent-state s)))
  (mylet* ((cframe  (current-frame s))
           (locals  (locals cframe))
           (opstack (operand-stack cframe))
           (index   (arg inst))
           (value   (local-at index s)))
          (and ....
              (wff-aload inst)
              ....
              (valid-local-index index locals)
              (REFp value (heap s))
              ....)))
```

Figure 6.10: ACL2 specification for `check-ALOAD`

In addition to what is specified in the official specification (figure 6.9) for `ALOAD`, we also need to capture the general declarative assertions that the official JVM specification made about the JVM executions. For example, the section 4.8.2 *Structural Constraints* of the JVMSpec [45], explicitly assert that:

> At no point during execution can the operand stack grow to a depth (§3.6.2) greater than that implied by the `max_stack` item.

Although not shown in figure 6.10, our definition of `check-ALOAD` in fact contains assertions that one plus the size of operand stack is still no more than `max_stack`.

heap nor keep track of concrete values stored in the locals. The bytecode verifier only keeps track of the type of the value that will be stored in that slot. The bytecode verifier will check whether the type is a reference type.

6.3 What is a Good Safety Specification

The obvious way to specify JVM safety is to assert that the `check-XXX` style operations will always succeed — DJVM will not get stuck when executing verified programs. This is in fact how JVMSpec declaratively specifies the safety of the JVM.

Although this *DJVM does not get stuck* specification is good for the JVM user and programmers who write Java programs, for the JVM implementors, this safety specification is not good enough. For a JVM implementor, who build a JVM by following the operational specification of the JVM, there is always the worry that such an implementation may still get stuck.

What the JVM implementors have is an implicit guarantee from the JVM designer that as long as their program implements operationally specified JVM operations (including class loading and class verification) correctly, the execution will not get stuck.

We are interested in proving that the JVM specification in fact fulfills the promise. In the process towards proving the that operationally specified JVM is safe (i.e. execution will not get stuck), we realize that we need to strengthen the concept of the JVM execution safety before we can ever expect to prove by mathematical induction that the JVM is safe.

We need to identify a stronger safety assertion that (1) implies the original weaker form of safety assertions (`check-XXX` always succeeds and the DJVM does not get stuck), (2) is *inductively preserved* over machine executions. [6], and (3) holds on intended initial states.

In rest of this section, we present our strengthened version of the safety

[6]For a property `P` to be *inductively preserved* over machine executions, we mean that for any state `s`, if `P` holds on state `s`, i.e. `(P s)`, then, `P` holds on the next state, `(P (step s))`

specification. Although the strengthened safety specification presented here was initially conceived as a stepping stone for proving that the JVM execution will not violate the original constraints specified in the JVM specification, our strengthened version is a good safety specification in its own right.

6.4 Global Inductive Invariant

In our JVM safety specification, we first demand that a safe JVM maintains a global invariant on its execution state.

Not only do we want to define a requirement (useful to application programmers) that a safe JVM shall preserve, we aim to define a stronger property that, once proved about the JVM specification, will be useful to the JVM implementors. In order for an invariant to be useful to the JVM implementors, we think such an invariant needs to be *direct* and *sufficiently strong*, but not *overly restrictive*.

- A: The requirement shall be expressible in term of the state itself without referring to some other requirements on the reachable states by the JVM.
- B: The requirement shall be sufficiently strong such that we will know enough about a JVM state to be able to judge with ease:
  - (1) whether the state is *sensible*,
  - (2) whether executing any operation in such a state will be safe — as specified in the official JVM specification, i.e. `check-XXX` succeeds,
  - (3) whether executing any operation in such a state will produce a resulting state that also meets this requirement.

- C: The invariant shall not be overly restrictive. The initial states for executing verified programs shall satisfy the invariant.

First, we need the invariant to be directly expressible without referring to other requirements on some other states. We intend to prove that the requirement that we identify is an invariant of the JVM execution. Our strategy for proving that is to prove that the property is preserved by every JVM operations individually. This way, we do not have to reason about the complicated JVM as a whole. The difficulty of the proofs is thus reduced. If the property itself were defined in terms of other properties on the reachable states by the JVM, our strategy would be defeated, because to show that any JVM operation preserves such a property, we would need to reason about the complicated JVM as whole.

We identify the following three kinds of requirements on a "good state" to "answer" the questions posed in B.1 and B.2: Objects must be well formed, the class table must represent classes that form a valid class hierarchy, and runtime states of threads must be sensible.

The difficulty lies in identifying a requirement on the JVM state so that we can judge whether B.3 holds for a state that satisfies the requirement.

Our key requirement is that the JVM state be *on track* with some abstract state observed by the bytecode verifier in a successful bytecode verification run.

6.4.1 Consistent Objects

Our first requirement is that a sensible JVM state shall not have ill-formed objects.

For a non-array object to be well formed, we expect that,

- the object is of a known class in the JVM state,
- the values recorded in the fields of the object are well-formed values, and,
- the types of these values are compatible with their declared field type

We define the ACL2 predicate `consistent-object` (figure 6.11) to assert that a non-array object `obj` is *consistent* with respect to the set of known objects `hp` and known class definitions `cl`.

```
(defun consistent-object (obj hp cl)
  (and (wff-obj-strong obj)             ; syntatically correct
       ...                              ; some assertions elided.
       (if (isArrayType (obj-type obj))
           t  ; we check the array object in a separate function
         (and
            (class-exists? (obj-type obj) cl)
            (consistent-jvp (obj-type obj)
                            (java-visible-portion obj)
                            cl hp)))))
```

Figure 6.11: DJVM: `consistent-object`

The function `consistent-jvp` asserts:

- all superclasses of `(obj-type obj)` exist in the class table,
- all fields declared by the class of `(obj-type obj)` and its superclasses exist in `(java-visible-portion obj)`,
- all field values are well-formed values, and,

- all field values are compatible with the declared types for the fields.

For a value to well-formed value, if the tag is a primitive type, we require that the value be within the range of that primitive type. If a tagged value is of reference type, we require that the object being referenced exist in the heap. [7]

For each field, the `consistent-jvp` operations needs to look up the actual type `rtype` of field value and compare it with the declared type `type` of the field using `AssignmentCompatible` operation (see Figure 6.13).

6.4.2 Consistent Class Table

For a JVM state to be sensible, we identify the following constraints on its class table:

- Classes recorded in the class table are well formed.
- If a class is represented in the class table, its superclasses are represented in the class table.
- Similarly, if a class is represented in the class table, its superinterfaces are represented in the class table.
- There are no loops in the class hierarchy.
- The class `java lang.Object` is a superclass of every Java class.
- Classes are correctly loaded from their static description.

[7]We could recursively demand that the referenced object is a `consistent-object`. However, because the objects in the heaps may form data structures with loops, to avoid nonterminating recursive assertions we chose only to demand that the referenced object exists and all objects in the heap are `consistent-object`.

```
(defun consistent-value (tagged-value type cl hp)
  (if (not (wff-tagged-value tagged-value)) nil
    (let ((vtype (tag-of  tagged-value))
          (value (value-of tagged-value)))
      (cond ((primitive-type? type)
             (and (equal vtype type)
                  (cond ((equal type 'INT)  (INT32p value))
                        ((equal type 'ADDR) (ADDRp value))
                        ((equal type 'CHAR) (CHARp value))
                        ((equal type 'BOOLEAN)
                         (jvmBOOLEANp value))
                        ((equal type 'SHORT)   (SHORTp value))
                        ((equal type 'BYTE)    (BYTEp value))
                        ((equal type 'FLOAT)
                         (jvmFLOATp value))
                        ((equal type 'DOUBLE)  (DOUBLEp value))

                        ((equal type 'LONG) (INT64p value))
                        (t nil))))
            ((NULLp tagged-value) t)
            ((REFp tagged-value hp)
             (let* ((ref tagged-value)
                    (obj (deref2 ref hp))
                    (rtype (obj-type obj)))
               (assignmentCompatible rtype type cl)))
            (t nil)))))
```

Figure 6.12: DJVM: `consistent-value`

```
(defun isJavaAssignmentCompatible (rtype type cl)
  (declare (xargs :guard (consistent-class-hierachy cl)))
  (cond ((primitive-type rtype)   ...)

         ((equal rtype 'NULL)     ...)

         ((isClassType rtype)
          (and (isClassType type)
               (class-exists? (classname-classtype rtype) cl)
               (class-exists? (classname-classtype type) cl)
               (isJavaClassAssignmentCompatible
                      (classname-classtype rtype)
                      (classname-classtype type) cl)))

         ((isArrayType rtype)
          (cond ((isClassType type)
                  (or ...

                     (isJavaLangObject type)))
                (t (and (isArrayType type)
                        (let ((x (component-type rtype))
                              (y (component-type type)))
                          (or ....
                              (and (compound x)
                                   (compound y)
                                   (isJavaAssignmentCompatible
                                                    x y cl)..))

(defun assignmentCompatible (rtype type cl)
  (and (or (primitive-type rtype)
           (reference-type-s rtype cl))
       (or (primitive-type type)
           (reference-type-s type cl))
       (isJavaAssignmentCompatible rtype type cl)))
```

Figure 6.13: DJVM: `assignmentCompatible`

```
(defun consistent-class-table (cl scl hp)
  (and (wff-instance-class-table-strong cl)
       (consistent-class-decls cl cl hp)
       ;; well formed.

       (consistent-class-hierachy cl)
       ;; super and interfaces exist
       ;; no loops in supers, and interfaces.

       (class-table-is-loaded-from cl scl)))
       ;; loaded correctly
```

Figure 6.14: DJVM: `consistent-class-table`

Our `consistent-class-table` (figure 6.14) captures these requirements.

The `consistent-class-hierarchy` requires that (1) for any class in the class table its superclass is defined in the class table, unless the class is `java.lang.Object`, (2) similarly, all the interfaces that the class is declared to implement are also defined in the class table, and (3) for any class in the class table, the superclass chain from the class contains no loops and the superinterface chains contain no loops.

Among other requirements, a good class table also requires that the basic classes such as "java.lang.Object", "java.lang.String", and "java.lang.Class" are present.

6.4.3 Sensible Thread Runtime State

In addition to the constraints that we described about objects and class tables, we also assert that execution state of the threads in a JVM state is sensible.

We identify the following desirable properties of a thread runtime state:

```
(defun class-hierachy-consistent1-class-n (n cl)
  ;;
  ;; 1. super ends with "java.lang.Object"
  ;; 2. interfaces all bounded and are in fact interfaces.
  ;;
  (and (class-exists? n cl)
       (if (equal n "java.lang.Object")
           (let ((class-rep (class-by-name n cl)))
             (and (not (class-exists? (super class-rep) cl))
                  (all-interfaces-bounded?
                                 (interfaces class-rep) cl)))
         (let ((class-rep (class-by-name n cl)))
           (and (class-exists? (super class-rep) cl)
                (all-interfaces-bounded?
                              (interfaces class-rep) cl))))))


(defun superclass-chain-no-loop-class-n (n1 cl seen)
  (if (not (wff-instance-class-table cl)) nil
    (if (not (class-exists? n1 cl)) t
      (if (mem n1 seen) nil
        (let ((n2 (super (class-by-name n1 cl))))
          (superclass-chain-no-loop-class-n
                      n2 cl (cons n1 seen)))))))
```

Figure 6.15: DJVM: `consistent-class-hierarchy`

- Call frames represent executions of non-abstract methods.
- Adjacent frames on the call stack form a valid caller-callee relation — with the lower call frame representing a paused caller execution and the upper call frame representing the callee's execution.
- The operand stack and local variable array in any call frame hold well-formed values.
- The size of the operand stack is less than the declared operand stack size limit of the method being executed.
- The uninitialized objects *accessible* in a frame are created in the correct call frames. An uninitialized object is accessible in a frame if and only if a reference to the object is stored somewhere on the operand stack or in the local variable array of the frame.

We formalize the first four requirements by asserting `consistent--thread-entry` (figure 6.16) for each thread in a safe JVM state. For a thread to be a `consistent-thread-entry`, we assert that every frame is a `consistent-frame` (figure 6.17) via `consistent-call-stack`. We demand that the caller-callee relation holds between adjacent call frames via `consistent-call-stack-linkage`.

For a call frame to be a `consistent-frame` (figure 6.17), among other things, its operand stack must contain only consistent values — `consistent--opstack`. Its local variable array must also contain consistent values — `consistent-locals`. [8]

[8]`Consistent-locals` is more complicated than `consistent-opstack`, because the local variable array may have slots with uninitialized values in addition to consistent values, while a consistent operand stack must record consistent values only.

```
(defun consistent-thread-entry (th cl hp)
  (declare (xargs :guard  (and (consistent-class-hierachy cl)
                               (wff-heap-strong hp))))
  (and (wff-thread th)
       (Valid-REFp (tag-REF (thread-ref th)) hp)

       (consp (thread-call-stack th)) ;; at least one frame

       (consistent-call-stack (thread-call-stack th) cl hp)
       ; each call frame are consistent-frame

       (consistent-call-stack-linkage (thread-call-stack th) cl)
       ; caller-callee relation holds between adjacenent frames.

       (let* ((obj (deref2 (tag-REF (thread-ref th)) hp))
              (rtype (obj-type obj)))
         (or (assignmentCompatible rtype "java.lang.Thread" cl)
             ; either a subclass of java.lang.Thread

             (classImplementInterface rtype
                                      "java.lang.Runnable" cl)))
             ; or implement the java.lang.Runnable interface.

       (or (equal (thread-mref th) -1)
           ; either does not hold a monitor on some object

           (bound? (thread-mref th) hp))))
           ; or the holding some monitor on some valid object
```

Figure 6.16: DJVM: `consistent-thread-entry`

```
(defun consistent-frame (frame cl hp)
  (mylet* ((method (deref-method (method-ptr frame) cl)))
          (and (wff-call-frame frame)
               (consistent-opstack (operand-stack frame) cl hp)
               (consistent-locals  (locals frame) cl hp)
               (consistent-frame-max-local frame cl)

               (wff-method-ptr (method-ptr frame))
               (valid-method-ptr (method-ptr frame) cl)
               (valid-sync-obj   (sync-obj-ref frame) hp)

               (wff-method-decl method)
               (not (mem '*abstract*
                            (method-accessflags method)))
               (or (mem '*native* (method-accessflags method))
                   ;; either it is a native method
                   ;;
                   ;; else it has valid code and max stack
                   (and (wff-code (method-code method))
                        (integerp (method-maxlocals method))
                        (integerp (method-maxstack method))
                        (<= (len (operand-stack frame))
                            (method-maxstack method)))))))
```

Figure 6.17: DJVM: `consistent-frame`

```
(defun consistent-adjacent-frame (caller callee cl)
  (and (equal (return-pc callee)
              (resume-pc caller))
       (valid-offset-into (return-pc callee)
                          (method-code
                            (deref-method
                                       (method-ptr caller) cl)))
       (<= (+ (len (operand-stack caller))
              (type-size
                   (method-ptr-returntype (method-ptr callee))))
           (method-maxstack
                   (deref-method (method-ptr caller) cl)))))
```

Figure 6.18: DJVM: `consistent-adjacent-frame`

The `consistent-adjacent-frame` (figure 6.18) records what needs to hold for the two call frames to represent a valid caller-callee relationship.

Concerning the accessible uninitialized objects from a frame, we require that:

- If the current method is a regular method, the only uninitialized objects *accessible* from the current frame must have been created in the current frame.
- If the current method is a constructor method, we can have references to at most one uninitialized object that is *not* created in the current frame and is passed in as the `this` pointer by the caller of the constructor.

  Any other uninitialized objects accessible to the current frame have to have beeen created in the current frame.

6.4.4 JVM Execution On Track

The assertions presented so far are necessary requirements for a sensible JVM state. However they are not sufficient to guarantee that the next state of a sensible state will also be a sensible state.

To see why the conjunction of assertions presented so far is not sufficient, we can look at our requirements for a sensible thread runtime state. Our requirements only assert that the operand stack contains consistent values and the size is within the declared limit; there are no requirements preventing the next JVM operation from overflowing the operand stack.

On the other hand, if we know that the program that we are executing has been accepted by the bytecode verifier, i.e., the abstract execution of the program finished successfully on the bytecode verifier, we then know that the corresponding operation during the abstract execution did not overflow the operand stack. If we can use such a fact about the abstract execution state to predict the result during actual execution, we can show that executing the next operation will not overflow the operand stack of the runtime state.

We thus identify the *on-track* requirement.

A JVM state records a set of threads in execution. Each thread maintains a call stack. The top most call frame of each call stack represents an "on-going" method execution. Other call frames in the call stacks represent "paused" (suspended) executions of a "caller" method, waiting its "callee" to return a value to its operand stack stack. Each call frame effectively records a `pc` that points to the next instruction for execution.

For a JVM execution to be *on-track* (with the bytecode verifier's abstract execution), we require that each call frame in the JVM state is *approxi-*

```
(defun frame-sig (frame cl hp hp-init)
  (declare (xargs :guard ...))
  (let* ((locals-sig (locals-sig (locals frame)
                                 cl hp hp-init
                                 (method-ptr frame))
         (stack-sig  (opstack-sig (operand-stack frame)
                                  cl hp hp-init
                                  (method-ptr frame)))
         (flags (gen-frame-flags frame hp-init))))
    (make-sig-frame  locals-sig stack-sig flags)))
```

Figure 6.19: `On-track` definition: `frame-sig`

mated by a specific abstract state — the one observed by the bytecode verifier when the abstract execution of the method (associated with the call frame) reaches the specific `pc`. An abstract state `bcv-state` is said to *approximate* a concrete state `djvm-state`, if the extracted type signature (obtained by applying `frame-sig` (figure 6.19) to the concrete state), is `frameIsAssignable` (figure 6.20) to the `bcv-state`.

The full definition of the `on-track` requirement on a "good" state is encoded in `consistent-state-bcv-on-track` (figure 6.21) from `DJVM/consistent-state-bcv-on-track.lisp` [22].

We expect with this additional *on-track* requirement, we can prove that our conjectured global inductive invariant is in fact an inductive invariant. The two major proof steps involved are

- we need to prove that when the top frame of the current thread is approximated by an abstract state and the bytecode verifier's check succeeds on the abstract state, then the runtime check by DJVM `check-XXX` will also succeed.

```
(defun frameIsAssignable (Frame1 Frame2 env)
  (let ((Locals1 (frameLocals Frame1))
        (Locals2 (frameLocals Frame2))
        (StackMap1 (frameStack Frame1))
        (StackMap2 (frameStack Frame2))
        (Flags1  (frameFlags Frame1))
        (Flags2  (frameFlags Frame2)))
    (and (equal (len StackMap1)
                (len StackMap2))
         (typeListAssignable Locals1 Locals2 env)
         (typeListAssignable StackMap1 StackMap2 env)
         (subset Flags1 Flags2))))
```

Figure 6.20: `On-track` definition: `frameIsAssignable`

```
(defun consistent-state-bcv-on-track (s)
  (and (bcv::good-scl-strong (env-class-table (env s)))
       (consistent-thread-table-bcv
        (thread-table s)
        (pc s)
        (current-thread s)
        (heap s)
        (heap-init-map (aux s))
        (instance-class-table s)
        (env-class-table (env s)))))
```

Figure 6.21: `On-track` definition

- we need to prove that when `check-XXX` succeeds and the top frame of the current thread is approximated by the abstract state observed by the bytecode verifier at the current program counter, the updated top frame of the resulting state of executing the operation `execute-XXX` will be approximated by the abstract state observed by the bytecode verifier at the next program counter. For operations that modify the call frames stack (such as `INVOKE`-family and `RETURN`-family operation), we need to prove that after those updates to the call stack, the DJVM state is still on-track.

6.5 Local Guard Assertions

As explained in the *Using ACL2* chapter, section 3.2.3, we can write assertions for an ACL2 program as *guards*. Similar to type annotations in Java programs, guards for an ACL2 program can be viewed as a systematic way of expressing expectations about the domain that a program will operate in. Different from a typical type system, we do not have a fully automatic decision procedure to "type check" whether the set of guards is compatible with each other. Instead, we need to interact with the ACL2 theorem prover to prove theorems that relate the guards for the set of ACL2 functions.

We interact with the ACL2 theorem prover to identify strong guards about JVM operations. We aim to identify guards for operations that can be *guard verified*. That is the guard for an operation is strong enough so that for all sub-operations it invokes, these sub-operations will be invoked on inputs that meet the corresponding guards — assuming the top level operation is invoked with its guard met.

To identify *verifiable guards* for JVM operations, we often start with what is obviously necessary to hold. We then attempt to guard-verify it. By studying the failed guard verification attempts, we obtain more insight into what other necessary properties the inputs must have. We use the new insights to strengthen our guard definition for JVM operations.

The process of identifying *verifiable guards* is non-trivial. It often involves several iterations of strengthening of the guard conjecture and we are obliged to prove non-trivial theorems.

The value of having strong verifiable guards on JVM operations is that by knowing that the lower level sub-operations will be invoked with their guards met, a JVM implementor have more ways to optimize the implementations of these JVM operations — choosing how to represent the inputs and deciding on how to emulate the JVM operations. For the JVM designers, having a set of verifiable guards, is also a good indication that the JVM specification itself is internally consistent and can be implemented.

We use the identified guards as part of our JVM safety specification: (1) JVM operation must pass guard verification using this set of guards and (2) the JVM initial state must meet the guard for executing the very first JVM operation used for starting the JVM's execution.

In the following sections, we present guards that we have identified for a few JVM operations.

6.5.1 Simple Primitives

Before we can guard-verify a complicate JVM operation, we need to guard-verify all the operations that may be invoked by this operation.

ACL2 primitives such as `cons`, `consp` are guard-verified with their built-in guard definition. For every user defined primitive operations, no matter how simple it is, we need to guard-verify the operation, before we can attempt to guard-verify any operations that use them.

The following are a few examples of such simple primitives that we have defined guards for.

```
(defun cpentry-type (cpentry)
  (declare (xargs :guard (wff-constant-pool-entry cpentry)))
  (car cpentry))
```

Figure 6.22: Guards: `cpentry-type`

The `cpentry-type` is an accessor function. It takes one input argument `cpentry`, which is expected to be a `wff-constant-pool-entry`. It returns the first component of the entry.

To guard-verify the operation `cpentry-type`, the ACL2 theorem prover generates the proof obligation that [9]

```
(implies (wff-constant-pool-entry cpentry)
         (consp cpentry))
```

This is because the operation `car` expects its operand to be `consp`.

For another example, `class-by-name` (figure 6.23), is an operation that looks up a class definition of a given name. It is defined as a recursive function.

Before the ACL2 theorem prover can guard-verify the operation, we first need to guard-verify `classname` (`consp`, `car`, `equal`, `if` and `cdr` as well), whose guard is `wff-class-rep`.

[9]ACL2 also demands that the `wff-constant-pool-entry` is also guard verified to have a guard `t`.

```
(defun class-by-name (class-name dcl)
  (declare (xargs :guard (wff-instance-class-table dcl)))
  (if (not (consp dcl))
      nil
   (if (equal (classname (car dcl)) class-name)
      (car dcl)
      (class-by-name class-name (cdr dcl)))))
```

Figure 6.23: Guards: `class-by-name`

To guard-verify `class-by-name`, we need to prove theorems showing that when various operations are invoked, their guards are met. One of the properties that we need to prove when the recursive call of `(class-by-name class-name (cdr dcl))` is reached is that its guard, `(wff-instance-class--table (cdr dcl))`, is met.

```
(implies (and (wff-instance-class-table dcl)
              (consp dcl)
              (not (equal (classname (car dcl)) class-name)))
         (wff-instance-class-table (cdr dcl)))
```

6.5.2 Class Loading Operations

Class loading operations are complicated. Recall our description of the JVM bootstrap class loader in the section 4.3.3, loading a class involves parsing the static class description, creating new objects in the heap, and recursively loading the superclasses and superinterfaces. A class loader needs to do all the above — in addition to creating a representation of the class and adding it to the internal class table.

To invoke these class loading operations, delicate conditions must be

```
(defun build-a-java-visible-instance-guard (classname S)
  (and (wff-state s)
       (wff-class-table (class-table s))
       (wff-env (env s))
       (wff-instance-class-table (instance-class-table s))
       (wff-static-class-table (external-class-table s))
       (equal (collect-superclass-list classname
                                       (instance-class-table s))
              (collect-superclassname classname
                                      (external-class-table s)))
       (build-a-java-visible-instance-data-guard
        (collect-superclass-list classname
                                 (instance-class-table s)) s)))
```

Figure 6.24: Guards: `build-a-java-visible-instance-data-guard`

met. We formulate these conditions as guards.

For example, to add a new object to the heap with the `bind` operation, we need the heap to be an `alistp`.

```
(defun bind (x y alist)
  (declare (xargs :guard (alistp alist)))
  (cond ((endp alist) (list (cons x y)))
        ((equal x (car (car alist)))
         (cons (cons x y) (cdr alist)))
        (t (cons (car alist) (bind x y (cdr alist))))))
```

To create an object of type `classname`, we expect that the precondition shown in figure 6.24 to be met.
Specifically, to build an object of class `classname` in the heap

- we first expect that the components of the JVM state to be well formed

```
(and (wff-state s)
     (wff-class-table (class-table s))
     (wff-env (env s))
     (wff-instance-class-table (instance-class-table s))
     (wff-static-class-table (external-class-table s)))
```

- We also expect that all superclasses of the class `classname` to be loaded

```
(equal (collect-superclass-list classname
                                (instance-class-table s))
       (collect-superclassname classname
                               (external-class-table s)))
```

- It also needs to be safe for the JVM to create default data values for all declared fields

```
(build-a-java-visible-instance-data-guard
    (collect-superclass-list classname
                             (instance-class-table s)) s)
```

As our last example, the safety constraint that we identified for `load-_class_internal` is formalized in `load_class_internal_guard` (figure 6.25). To safely execute the `load_class_internal` to load a new class definition into the class table, we need to check

- The state is well formed

```
    (and (wff-state s)
         (wff-env env)
```

```
(defun load_class_internal_guard (classname s)
  (mylet* ((env (env s))
           (class-table (class-table s))
           (scl (env-class-table env)))
    (and (wff-state s)
         (wff-env env)
         (wff-heap (heap s))
         (wff-class-table class-table)
         (loader-inv s)
         (all-correctly-loaded?
          (cdr (collect-superclassname
                   classname (external-class-table s)))
          (instance-class-table s)
          (external-class-table s))
         (all-correctly-loaded?
          (cdr (collect-superinterface
                   classname (external-class-table s)))
          (instance-class-table s)
          (external-class-table s))
         (wff-static-class-table scl)
         (mv-let (found class-desc)
                 (class-by-name-s classname scl)
                 (mylet* ((static-cp
                                (constantpool-s class-desc))
                          (static-field-table
                                (fields-s class-desc))
                          (static-method-table
                                (methods-s class-desc)))
                 (or (not found)
                     (and (wff-class-rep-static class-desc)
                          (load_CP_entries_guard static-cp s)
                          (build-a-java-visible-instance-guard
                               "java.lang.Class" s)
                          (wff-fields-s static-field-table)
                          (runtime-method-rep-guards
                               static-method-table))))))))
```

Figure 6.25: Guards: load_class_interna_guard

```
          (wff-heap (heap s))
          (wff-class-table class-table))
```

- The current state satisfies an invariant `loader-inv`

  ```
      (loader-inv s)
  ```

  which asserts that either the execution is in a fatal error state or any loaded class are in fact *correctly loaded* — their superclasses are all loaded and the internal representation matches the external description.

- all superclasses and superinterfaces of the class `classname` are loaded.

  ```
  (and
      (all-correctly-loaded?
       (cdr (collect-superclassname
                 classname (external-class-table s)))
       (instance-class-table s)
       (external-class-table s))

      (all-correctly-loaded?
       (cdr (collect-superinterface
                 classname (external-class-table s)))
       (instance-class-table s)
       (external-class-table s)))
  ```

- Either the external class descriptions do no have an entry for the class `classname` or the corresponding external description must be well-formed `wff-class-rep-static` and it must be safe to use `load_CP_entries` and other operations to create a runtime representation of the class.

```
(and
 (wff-static-class-table scl)
 (mv-let (found class-desc)
         (class-by-name-s classname scl)
         (mylet* ((static-cp  (constantpool-s class-desc))
                  (static-field-table
                             (fields-s class-desc))
                  (static-method-table
                             (methods-s class-desc)))
         (or (not found)
             (and (wff-class-rep-static class-desc)
                  (load_CP_entries_guard static-cp s)
                  (build-a-java-visible-instance-guard
                       "java.lang.Class" s)
                  (wff-fields-s static-field-table)
                  (runtime-method-rep-guards
                       static-method-table))))))
```

6.5.3 JVM Instructions

For the JVM operation `execute-GETFIELD` (figure 6.26), we specify the local safety condition for executing it as `GETFIELD-guard` (figure 6.27).

```
(defun execute-GETFIELD (inst s)
  (declare (xargs :guard (GETFIELD-guard inst s)))
   (mylet* ((fieldCP (arg inst)))
    (mv-let (field-rep new-s)
     (resolveFieldReference fieldCP s)
     (if (not (no-fatal-error? new-s)) new-s
       (if (pending-exception s)
           (raise-exception (pending-exception s) s)
         (if field-rep
             (let ((new-s2 (execute-getfield1 field-rep new-s)))
               (if (pending-exception new-s2)
                   (raise-exception
                           (pending-exception new-s2) new-s2)
                 (ADVANCE-PC new-s2)))
           (fatalSlotError fieldCP new-s)))))))
```

Figure 6.26: Guards: `execute-GETFIELD`

6.5.4 Guard as Specification

A guard defines the set of expected inputs that the operation will execute with.

We would like to note that not all guard definitions are useful as good local safety specifications. A good guard definition should *interestingly* capture what are the valid inputs for executing the operation. The process for checking whether a input satisfies the guard should be direct and should not depend on how the operation is implemented. These statements may be more clear after we look at the following fact.

There is a systematic but uninteresting method to define a *verifiable* guard for every ACL2 function.

For a simple non-recursive operation, `simple-op`, we can define the guard `simple-op-guard` (figure 6.28) — assuming that the `test`, `f`, and `g`

```
(defun GETFIELD-guard (inst s)
  (mylet* ((obj-ref (safe-topStack s))
           (fieldCP (arg inst)))
  (and
   (wff-getfield inst)           ; safe to access the fieldCP
   (wff-fieldCP fieldCP)         ; fieldCP syntatically correct
   (consistent-state-strong s)   ; additional safety requirement
   (topStack-guard-strong s)     ; safe to access the topStack
   ...
   (resolveClassReference-guard s) ; can invoke class loader
   (protectedAccessCheck inst s)   ;
   (or (mv-let (field-rep new-s)
             (resolveFieldReference (arg inst) s)
             (declare (ignore new-s))
             (not field-rep))    ; either field doesnot exists
       (or (CHECK-NULL obj-ref)    ; or either obj-ref is NULL
           (and                    ;    or
            (REFp obj-ref (heap s));           valid pointer

            (mv-let (assignable new-s)
              (isAssignableTo
                      (obj-type (deref2 obj-ref (heap s)))
                      (fieldCP-classname fieldCP) s)
              (declare (ignore new-s))
              assignable)          ;          suitable type
            (mv-let (field-rep new-s)
              (resolveFieldReference (arg inst) s)
              (declare (ignore new-s))
              (or (and (equal (field-size field-rep) 2)
                 (<= (+ 1 (len (operand-stack
                                (current-frame s))))
                     (max-stack s)))  ; operand stack size ok
                  (and (equal (field-size field-rep) 1)
                 (<= (len (operand-stack (current-frame s)))
                     (max-stack s))))))))))
```

Figure 6.27: Guards: GETFIELD-guard

```
(defun simple-op (a b)
  (if (test a b)
       (f a)
    (h a b)))

(defun simple-op-guard (a b)
  (and (test-guard a b)
       (if (test a b)
           (f-guard a)
         (h-guard a b))))
```

Figure 6.28: Guards: `simple-op-guard`

already have their verifiable guard defined.

To understand how we arrive at this guard definition, one can imagine how the operation *simple-op* is evaluated. Because the first operation that we will do is the `(test a b)`, thus we first assert `(test-guard a b)` to hold. Then depending on whether `(test a b)` is true or not, we may evaluate `(f a)` or `(h a b)` as necessary. So subsequently, we assert the condition `(if (test a b) (f-guard a) (h-guard a b))`

Using the same method, we can define for a recursively defined operation `recursive-op` the following guard in figure 6.29.

However, guards defined this way are not particularly interesting. The definition of the guard closely depends on the definition of the operation itself. They are of a *dynamic* flavor. To check whether `recursive-op-guard` holds, we need to essentially *execute* `recursive-op` — because we have to check the following assertion

```
(op3-guard (recursive-op a (op2 b)))
```

which depends on the return value of `(recursive-op a (op2 b))`

```
(defun recursive-op (a b)
  (if (test (f a) b)
       (simple-op (op1 a) b)
    (op3 (recursive-op a (op2 b)))))

(defun recursive-op-guard (a b)
  (and (f-guard a)
       (test-guard (f a) b)
       (if (test (f a) b)
           (and (op1-guard a)
                (simple-op-guard (op1 a) b))
         (and (op2-guard b)
              (recursive-op-guard a (op2 b))
              (op3-guard (recursive-op a (op2 b)))))))
```

Figure 6.29: Guards: uninteresting `recursive-op-guard`

For a guard to be useful as a local safety specification, we demand the guard be *direct* about what the set of valid inputs is. Given an input, there should be a simple way to tell whether the input is among the valid inputs. Specifically, we do not expect that checking whether an input satisfies the guard for `recursive-op` will involve invoking `recursive-op` itself.

In our work, we have strived to define good guards, ones useful as local safety specifications. For example, we define the guard for `load_CP_entries` (figure 6.30).

An uninteresting guard for `load_CP_entries` would be of the following form (figure 6.31).

The uninteresting version of `load_CP_entries-guard` is easy to guard verify. However, it invokes the complicated operation `(load_CP_entry (car cpentries) s)` to produce a `new-state` and asserts the additional requirement `(load_CP_entries-guard (cdr cpentries) new-state)` about the

```
(defun load_CP_entries-guard (cpentries s)
  (if (not (consp cpentries)) t
    (and (load_CP_entry-guard (car cpentries) s)
         (load_CP_entries-guard (cdr cpentries) s))))

(defun load_CP_entries (cps s)
  (declare (xargs :guard (load_CP_entries-guard cps s)))
  (if (not (consp cps))
      (mv nil s)
    (mv-let (new-ent new-state)
            (load_CP_entry   (car cps) s)
            (mv-let (new-ents final-state)
                    (load_CP_entries (cdr cps) new-state)
                    (mv (cons new-ent new-ents) final-state)))))
```

Figure 6.30: Guards: `load_CP_entries-guard`

```
(defun load_CP_entries-guard (cpentries s)
  (if (not (consp cpentries)) t
    (and (load_CP_entry-guard (car cpentries) s)
         (mv-let (new-ent new-state)
                 (load_CP_entry (car cpentries) s)
                 (ignore new-ent)
             (load_CP_entries-guard
                 (cdr cpentries) new-state)))))
```

Figure 6.31: Guards: uninteresting `load_CP_entries-guard`

```
 ....
(defthm create-string-guard_after_load_cp_entry
  (implies (create-string-guard str s)
           (create-string-guard str
                   (mv-nth 1 (load_CP_entry any s)))))

(defthm load_CP_entry_guard-load_CP_entry_guard
  (implies (load_CP_entry-guard x s)
           (load_CP_entry-guard x
                   (mv-nth 1 (load_CP_entry any s)))))


(defthm load_CP_entries-guard-load_CP_entry-guard
  (implies (load_CP_entries-guard cps s)
           (load_CP_entries-guard cps
                   (mv-nth 1 (load_CP_entry any s)))))
```

Figure 6.32: Lemmas for guard-verify `load_CP_entries`

new state. Checking such a guard is of *dynamic* flavor and obscures the essential reason it is safe to execute `load_CP_entries`.

Our definition of the guard for `load_CP_entries` is more difficult to guard-verify. But it is a more direct specification. Given an input, we do not need to invoke an operation to modify the state. We essentially assert for all entries in the `cpentries`, `(load_CP_entry-guard entry s)` holds. Checking such a guard is of a *static* flavor.

To guard-verify `load_CP_entries` with our definition of the guard, we need to prove the additional lemmas in figure 6.32. The `load_CP_entry-_guard-load_CP_entry_guard` lemma says that if it is safe to create a runtime time constant pool entry `x` in the current state `s`, it will still be safe to do so after we create a constant pool entry `any` first.

6.6 Conclusion

The official JVM specification specifies the semantics of executing JVM operations only when certain constraints on the JVM state and inputs to the operations are satisfied. The official JVM specification also declaratively asserts that these constraints will always be met during a JVM execution. The safety guarantee of a Platonic JVM is that it will not get *stuck* because some of the constraints are violated.

To formalize this concept that *the JVM will not get stuck*, we introduce the defensive JVM model (DJVM) that maintains additional type information and checks the stipulated constraints at runtime. The safety of the JVM is then formalized by requiring that the execution bytecode verified programs will not fail any runtime check (`check-XXX`) conducted by the DJVM.

Since our ultimate project is to study the effectiveness of the bytecode verification process, we want to prove that a JVM is in fact safe while executing verified programs. In the process, we realized that safety as specified can not be proved inductively. We strengthened the safety specification to prove it inductively. We also think that safety as specified as *the JVM will not get stuck* is not very useful to JVM implementors. They will benefit by having a direct way to check whether a given JVM state is a "good" state or not — so that they can easily check whether their choice of the state representation is adequate and whether their implementation of the JVM operations preserve the "good" state property.

These reasons motivated us to search for a *better* safety specification — both as a stepping stone for proving the the bytecode verifier is effective and to make the safety specification more useful to JVM implementors.

We formalized the safety requirement for a JVM as

- it maintains a `consistent-state`, which we hopes to prove to be an inductive invariant of the JVM execution.

  The key observation in defining this `consistent-state` is that we need to assert that the JVM executions being *on-track* with a collection of the abstract executions of the bytecode verifier.

- the JVM operations have strong guards that can be guard verified.

Chapter 7

Framework and Relevant Proofs

7.1 A Complete Proof For A Simple System

We modeled a simple virtual machine and its static checker. We call the virtual machine `Small`. The `Small` machine maintains its execution state in a stack of call frames. Each call frame has an operand stack and a local variable array. The static checker takes a program as its input and returns a *yes* or *no* answer. If the static checker returns a *yes* answer on a specific program, the program is *verified*.

We formalized a safety requirement that executing verified programs never overflows the operand stack in any call frame. We proved that the simple virtual machine is in fact safe.

The machine model (the interpreter and the static checker) is 913 lines with 92 function definitions. The mechanically checked ACL2 proof input is 11,360 lines in 47 files, which includes 175 function definitions, 691 lemmas, requiring 196 inductive proofs.

The work described in this section is a good miniature of our project for specifying and verifying the Java Virtual Machine and its bytecode verifier. Formalizing and verifying the `Small` machine provides a road map for proving that the JVM is safe and the bytecode verifier is effective. It presents a useful big picture for understanding the apparently disconnected pieces of proofs that we present in the later sections of this chapter.

We have gained a few keys insights for how to formalize and verify that a virtual machine is safe and its static checker is effective.

- We need to strengthen the safety requirement (*never overflow operand stack*) into a "good-state" predicate on the machine state so that it can be proved as an inductive invariant of program executions.

- In order for a "good-state" predicate to be strong enough, we need to assert that the runtime state is not only well formed (*operand stack has suitable size*) but also *on-track* with some specific abstract execution of the static checker.

  This *on-track* requirement is necessary for allowing the execution of the actual machine to be predicted by the fact that the static checker succeeded in verifying the programs.

- It is of great help to first define an intermediate static checker and prove that programs verified by this simpler static checker are safe to execute.

  We then need to prove a *reduction* lemma showing that any program verified by the original static checker can also be verified by the simpler static checker.

7.1.1 Simple Virtual Machine

State representation

The virtual machine state has two components: a *call stack* and a *method table* (figure 7.1). The method table is the "static" component of the state that maps method names to their definitions. The method table is not dynamically extended during program execution. The call stack is the "dynamic" component of the machine state, which is updated during each step of program execution to capture effects of the program execution. [1]

The call stack is a stack of *call frames.* Each call frame records a program counter *pc*, an *operand stack*, a *local variable array*, and a *method-name.* Using the pc and method-name from a call frame, together with the method table, one can locate the next instruction to execute. The top-most call frame corresponds the current method being executed. Call frames below it correspond to paused executions of callers.

The method table is a collection of method definitions. Each method definition of this simple machine contains a *max-stack* field — in addition to the fields for describing the method name, number of formals, and the sequence of bytecode instructions.

It is significant that method definitions to have this *max-stack* field. Such a method definition not only encodes the operational behavior of the program with sequences of instructions, but also encodes (implicitly) a declarative "claim" that this method will never use more than max-stack number of

[1]This is a great simplification from a JVM state, which has multiple call stacks (one for each thread), a heap, and a class table that can be dynamically extended during program execution. In addition, a JVM method of a given name may have different definitions depending on the type of the object that is used to invoked it.

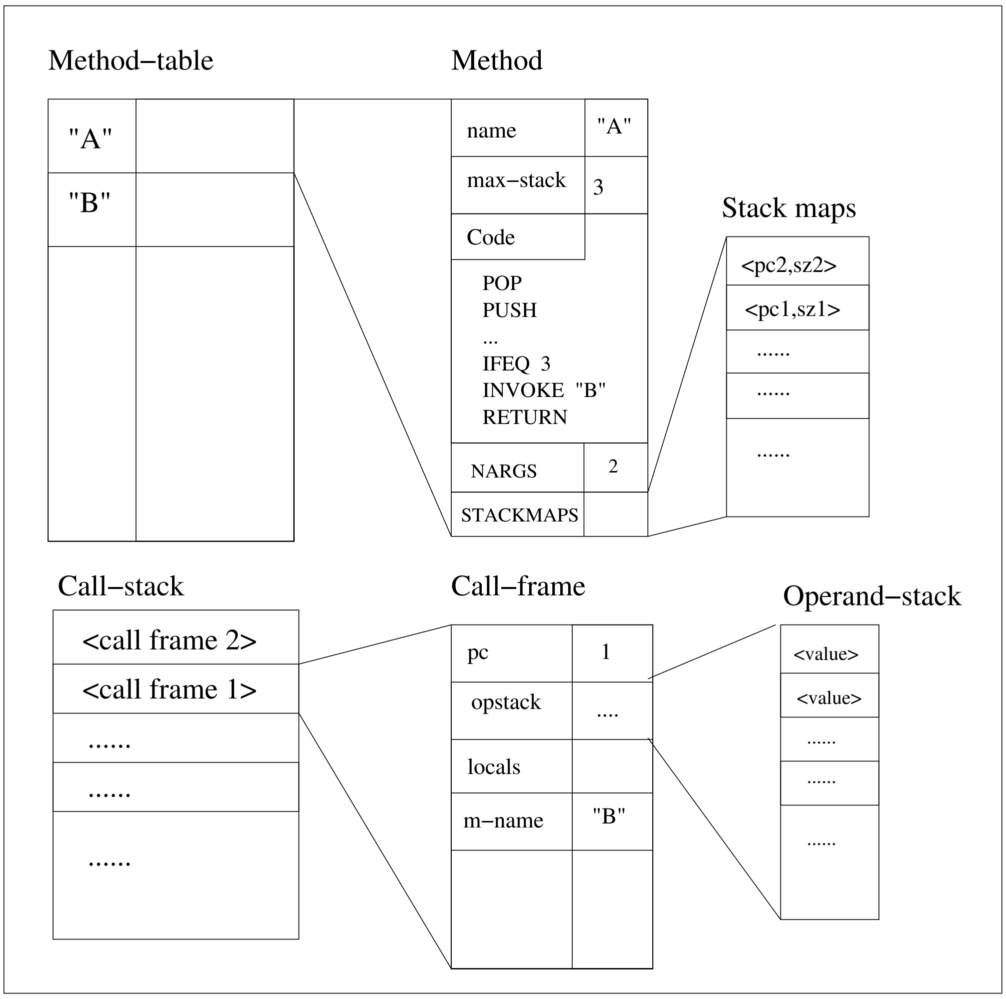


Figure 7.1: Small machine: state representation

slots in the operand stack during its execution. This distinguishes the method definition from a conventional assembly subroutine for an X86, for example, which only specifies the operational behavior of the code. The `Small` machine will use the static checker to check the "claim" of a method before it executes the method. The expectation is that the static checker is effective in detecting all possible violations of the "claim" without actually executing the method.

Semantics of Instructions

The simple virtual machine only recognizes six simple instructions `PUSH`, `POP`, `INVOKE`, `IFEQ`, `RETURN`, and `HALT`.

- (PUSH `v`) instruction takes one argument, and pushes the argument `v` onto the *current operand stack* — the operand stack of the top most call frame (*current frame*) in the call stack. It also advances the program counter in the current frame by one.

  Effects are not defined if the push operation would overflow the operand stack.

- (POP) instruction pops one value off the current operand stack and advances the program counter by one.

  Effects are not defined if there are no values to be popped off.

- (INVOKE `method-name`) instruction looks up the method and pops the correct number of values off the operand stacks. The virtual machine then creates and initializes a new call frame for executing the callee. It also adjusts the program counter to the next instruction after the INVOKE instruction.

Effects are not defined if the method does not exist. Effects are also not defined if there are not enough values on the operand stack.

- (IFEQ `target`) instruction pops a value off the current operand stack and updates the program counter in the current frame to either `target` or the current program counter plus one — depending on whether the popped value is zero or not.

  Effects are not defined if the operand stack is empty. Effects are also not defined when the `target` is not a valid pc in the current method being executed.

- (RETURN) instruction pops a value off the current operand stack, pushes the value onto the caller's operand stack, and pops the callee's call frame off the call stack.

  Effects are not defined if the current operand stack is empty or if pushing one extra value on the caller's operand stack will overflow the caller's max stack limit.

- (HALT) instruction does not change the state – thus halting any execution whose pc points to a HALT.

What is expected of a specification

If the machine specification had only described the above semantics for its instructions, it would have been *incomplete* in specifying the behavior of certain programs — programs whose executions violate the preconditions for instructions.

However, application programmers, who write programs for executing on the virtual machine, expect a complete specification.

A specification can be made complete either by specifying the effects of executing an instruction in all states, or declaratively demanding that the virtual machine will never attempt to execute an instruction in states where its effects are not defined.

It is often more desirable to adopt the second approach to complete a machine specification. In this case, the declarative requirements become a safety guarantee, asserting all reachable state will not get "stuck". This is useful to an application programmer and user of the virtual machine.

On the other hand, virtual machine implementors prefer to avoid any declaratively specified requirements. They write programs that execute on physical machines to emulate the ideal virtual machine. It is easy to follow an operationally described specification, which is explicit about what operations needs to be emulated and how they are going to be emulated. It is more difficult to know when declarative specifications are met by their implementation.

To address these requirements from both application programmers and virtual machine implementors, the official JVM specification describes a bytecode verifier operationally. The promise is that programs verified by the operationally described bytecode verifier will never attempt to execute an instruction whose effects are not defined.

Following this pattern, the specification for our simple virtual machine `Small` also defines a bytecode verifier style static checker. We need to prove that the static checker is effective to guarantee that the behavior of each verified program is well defined.

Static checker

The objective for the static checker is to detect potentially *unsafe* programs. Verified programs will not get stuck.

One may take the following high level view of a CLDC bytecode verifier style static checker:

- The specification demands each method carry both the code and its "proof" that it is safe to execute.
- The checker checks the "proof" against the code
- If the checker accepts the "proof", the method is allowed to execute on our simple virtual machine.

In our simple static checker for the `Small` machine, a safety "proof" appears in the form of a list of pairs. The first number of a pair is interpreted to be a pc into the method. The second number of the pair is interpreted to be the size of the operand stack when the program is about to execute the instruction at the pc.[2]

To check a "proof" against a method, the static checker maintains an abstract state also in form of a pair, `(pc . opsize)`.

- Initially, the abstract state is set to `(0 . 0)` to represent the initial operand stack is empty where the next instruction to execute is at pc 0.

[2]Note that this notion of proof precludes some "correct" programs from being verifiable. For example, no verified program can contain an instruction that can be reached by two paths, each of which produces a different stack size upon arrival at the instruction. But one can write programs that never overflows their declared stack size while using such paths. The JVM bytecode verifier suffers this same limitation.

- Before the static checker symbolically executes any instruction, it first checks if the abstract state is *compatible* with what the "proof" claims the abstract state would be. [3].
- Then the static checker checks whether it is safe to execute the next instruction in the abstract state. It checks that
  - the modification to the operand stack is valid. For example, to check a `PUSH` operation, it checks whether, after the push, the operand stack size is within the size limit stipulated by the current method,
  - the updated program counter is a valid pc in the method (and in the case of IFEQ both possible updated program counters are valid)
  - for instruction `INVOKE`, it checks that the method to be invoked exists and the current operand stack has enough values for invoking the method with, and
  - for instruction `IFEQ`, it checks that, at the target of branch, the "proof" records an abstract state that is compatible with popping one value off the current abstract state and setting the program counter to the target.
- If the checks succeed, the static checker updates its abstract state and moves on the next instruction in the sequence of the code.
  - For `INVOKE`, it assumes the callee returns immediately.
  - For `IFEQ`, it assumes the true branch is never taken.

[3]For an abstract state to be compatible with the proof, the stack size of the abstract state must be equal to that recorded in the proof at the pc of the abstract state. In the world of the CLDC JVM bytecode verifier, the definition for *compatibility* is much more complex and involves assignment compatibility (figure 6.20 and figure 6.13)

- If the static checker successfully reaches the end of the method, the method is considered verified.

7.1.2 Formal Machine Model and Its Static Checker

State representation

We use the *dictionary* data structured defined in the ACL2 system book `misc/records` extensively to represent various state components. [4]

The machine state is represented as a dictionary with two keys `call-stack` and `method-table`. The operation `(g 'call-stack s)` gets the call stack component of state `s`, and `(s 'call-stack new-cs s)` sets the call stack component to `new-cs`.

A call frame is also represented as a dictionary. The operation `(g 'pc frame)` returns the program counter of a frame. Similarly, `(g 'op-stack frame)` returns the operand stack.

State Manipulation Primitives

The concept of the *current method* is defined as follows:

```
(defun current-method (st)
  (let* ((method-name (g 'method-name
                         (topx (g 'call-stack st))))
        (method-table (g 'method-table st)))
    (binding method-name method-table)))
```

[4]It is also called the *record* data structure. A dictionary defines a "get" operation `g` and a "set" `s`. The ACL2 book not only describes how `g` and `s` are implemented, it also provides lemmas that configure the ACL2 theorem prover for reasoning about compositions of `g` and `s` operations.

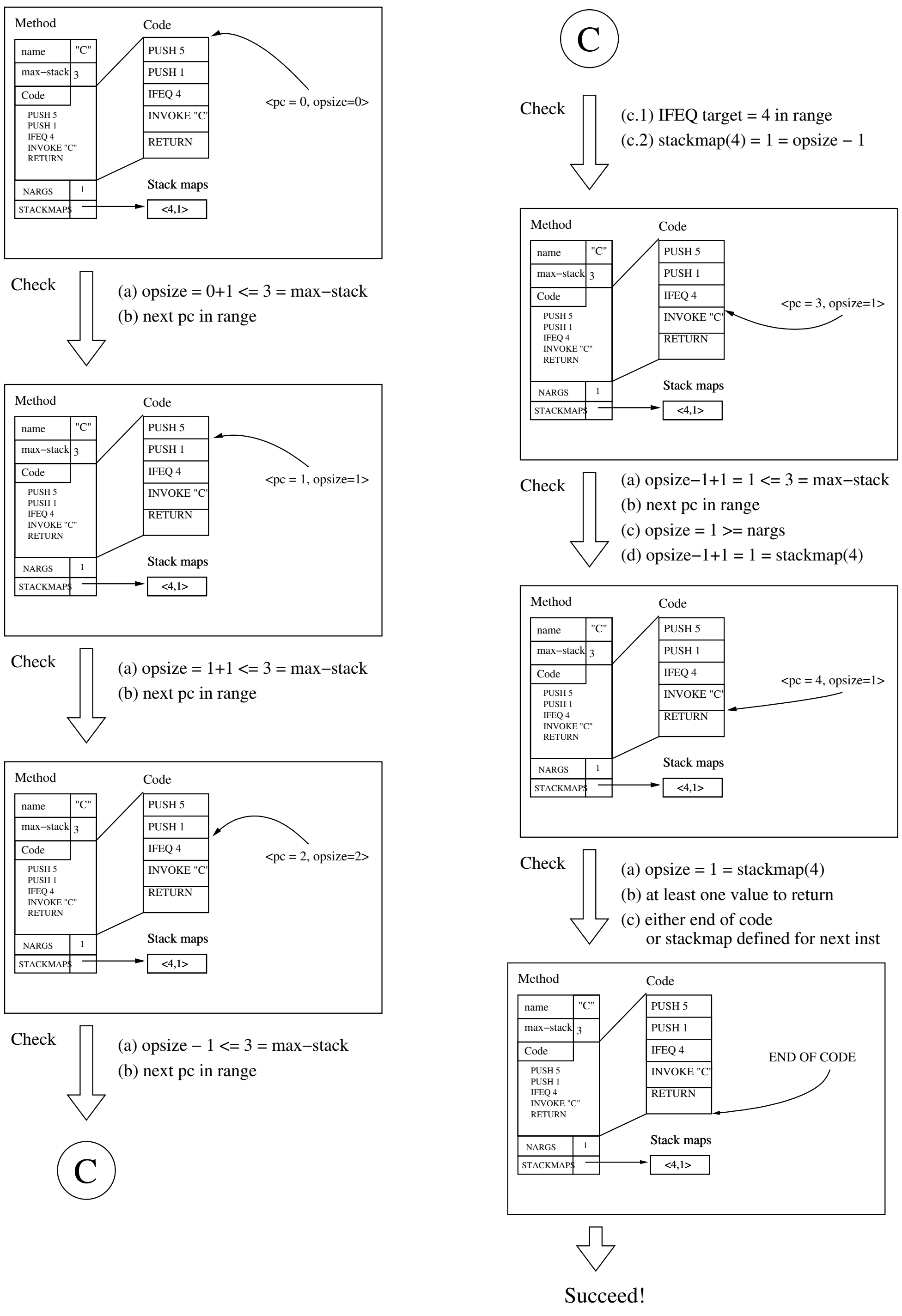


Figure 7.2: Small machine: static checker algorithm by example

That is, let `method-name` be the value of the `method-name` slot in the top frame of the `call-stack` slot of the state and let `method-table` be the method table of the state. Then return the binding of `method-name` in `method-table`.

The operation `popStack` pops one value of the current operand stack of the top most call frame in a state and then updates the state with the new call stack.

```
(defun popStack (st)
  (let* ((call-stack (g 'call-stack st))
         (top-frame  (topx call-stack))
         (op-stack   (g 'op-stack top-frame)))
    (s 'call-stack
       (pushx (s 'op-stack
                 (popx op-stack)
                 top-frame)
              (popx call-stack))
       st)))
```

State Transition Functions

The semantics for executing an instruction is encoded in a corresponding ACL2 function that maps a state to a state. For example, the ACL2 function `execute-INVOKE` encodes the semantics for executing an `INVOKE` instruction.

```
(defun execute-INVOKE (inst st)
  (let* ((method-name  (arg inst))
         (method-table (g 'method-table st))
         (method   (binding method-name method-table))
```

```
          (nargs    (g 'nargs method)))
     (pushInitFrame
      method-name
      (init-locals (op-stack st) nargs)
      (set-pc (+ 1 (get-pc st))
              (popStack-n st nargs)))))
```

To execute an `INVOKE` operation,

- we first look up the method, using the information encoded in the instruction and the method table.

  `(binding method-name method-table)`.

- We find out how many formals the method takes.

  `(g 'nargs method)` where `method` is `(binding method-name method--table)`.

- We pop off that number of values from the operand stack.

  `(popStack-n st nargs)`

- We then use these values to initialize a new call frame and push that call frame onto the call stack

  `(pushInitFrame (init-locals (op-stack st) nargs) ...)`

CLDC-style static checker

We implemented a static checker algorithm for the simple machine (section 7.1.1) in the same way we implement the CLDC bytecode verifier (figure 5.7) on page 146.

```
(defun bcv-method (method method-table)
  (let* ((code (g 'code method))
         (maps (g 'stackmaps method)))
    (and (wff-code (parsecode code))
         (wff-maps maps)
         (merged-code-safe
           (mergeStackMapAndCode
               maps
              (parsecode code)
              (g 'method-name method)
              method-table)
           (sig-method-init-frame method
                                  method-table)))))
```

Figure 7.3: Small machine: static checker

The maps here (figure 7.3) is a list of pairs. Conceptually, the first element of a pair from the list is interpreted as a pc in the method, the second element of the pair records an expected abstract state when any execution of the method reaches the pc. It is expected that the list is sorted in the non-decreasing order using the first element as the key.

To verify a method using `bcv-method`, we first extract the code `code` and the "proof" — `maps`. We then use `mergeStackMapAndCode` to merge the `maps` with the list of instructions — `(parsecode code)` into one list. We then set up an initial state `(sig-method-init-frame method method-table)` and symbolically execute the merged code with `merged-code-safe`.

The definition of `merged-code-safe` is shown in figure 7.4.

- When `mergedcode` is exhausted, if either the last element is not a special flag `END_OF_CODE`, or the current `sig-frame` is not a special frame `aftergoto`, the merged code fails verification.

```
(defun merged-code-safe (mergedcode sig-frame)
  (if (endp mergedcode) nil
    (if (endp (cdr mergedcode))
        (and (equal (car mergedcode) 'END_OF_CODE)
             (equal sig-frame 'aftergoto))
      (if (equal sig-frame 'aftergoto)
          (and (stack-map? (car mergedcode))
               (equal (g 'pc (car mergedcode))
                      (g 'pc (cadr mergedcode)))
               (merged-code-safe (cdr mergedcode)
                                 (car mergedcode)))
        (cond ((stack-map? (car mergedcode))
               (and (sig-frame-compatible sig-frame
                                          (car mergedcode))
                    (merged-code-safe (cdr mergedcode)
                                      (car mergedcode))))
              ((inst? (car mergedcode))
               (and (equal (g 'pc sig-frame)
                           (g 'pc (car mergedcode)))
                    (bcv-check-step-pre (car mergedcode)
                                        sig-frame)
                    (merged-code-safe
                     (cdr mergedcode)
                     (bcv-execute-step (car mergedcode)
                                       sig-frame))))
              (t nil))))))
```

Figure 7.4: Small machine: `merged-code-safe`

- If the current frame `sig-frame` is an `aftergoto`, we demand that the head element in the list `mergedcode` satisfy `stack-map?`. The algorithm also demands that the recorded stack map must `(car mergedcode)` have the same program counter as the next item `(cadr mergedcode)` (which can be either a stack map frame or an actual instruction).

- Otherwise, if the head element satisfies `stack-maps?`, we demand that the current frame is compatible with the head element

  ```
  (sig-frame-compatible sig-frame (car mergedcode))
  ```

- If the first element of the list is an instruction, we expect it is safe to execute the instruction in the context of the current frame.

  ```
  (bcv-check-step-pre (car mergedcode) sig-frame)
  ```

  We then update the current frame by executing the current instruction and assert `merged-code-safe` on the rest of the code with respect to updated current frame.

  ```
  (merged-code-safe
         (cdr mergedcode)
         (bcv-execute-step (car mergedcode) sig-frame))
  ```

The definition of `merged-code-safe` refers to the function `bcv-check--step-pre` and `bcv-execute-step`. We show their definitions in figure 7.5 to invoke the appropriate `bcv-check-XXX` function for each opcode.

```
(defun bcv-check-step-pre (inst sig-frame)
  (let* ((opcode (bcv-op-code inst)))
    (cond ((equal opcode 'INVOKE)
           (bcv-check-INVOKE inst sig-frame))
          ((equal opcode 'PUSH)
           (bcv-check-PUSH inst sig-frame))
          ((equal opcode 'IFEQ)
           (bcv-check-IFEQ inst sig-frame))
          ((equal opcode 'HALT)
           (bcv-check-HALT inst sig-frame))
          ((equal opcode 'POP)
           (bcv-check-POP inst sig-frame))
          ((equal opcode 'RETURN)
           (bcv-check-RETURN inst sig-frame))
          (t nil))))

(defun bcv-execute-step (inst sig-frame)
  (let* ((opcode (bcv-op-code inst)))
    (cond ((equal opcode 'INVOKE)
           (bcv-execute-INVOKE inst sig-frame))
          ((equal opcode 'PUSH)
           (bcv-execute-PUSH inst sig-frame))
          ((equal opcode 'IFEQ)
           (bcv-execute-IFEQ inst sig-frame))
          ((equal opcode 'HALT)
           (bcv-execute-HALT inst sig-frame))
          ((equal opcode 'POP)
           (bcv-execute-POP inst sig-frame))
          ((equal opcode 'RETURN)
           (bcv-execute-RETURN inst sig-frame))
          (t nil))))
```

Figure 7.5: Small machine: static checker primitives

```
(defun bcv-check-INVOKE (inst sig-frame)
  (let* ((method-name (bcv-arg inst))
         (method-table (g 'method-table sig-frame))
         (method      (binding method-name method-table))
         (nargs       (g 'nargs method)))
    (and (bound? method-name method-table)
         ;; method defined

         (integerp nargs)
         (<= 0 nargs)
         (<= nargs (g 'op-stack sig-frame))

         ;; enough operand.
         ;; note the op-stack field of sig-frame
         ;; is an number. c.f. with the concrete state
         ;; where the op-stack is a list of values.

         (<= (+ 1 (- (g 'op-stack sig-frame)
                     nargs))
             (max-stack sig-frame))

         ;; assume the method returns,
         ;; pushing the return value will not overflow
         ;; the operand stack
```

Figure 7.6: Small machine: `bcv-check-INVOKE`

Bcv-check-XXX style functions are defined to check whether it is safe to execute a specific instruction in a given abstract state. Bcv-execute-XXX style functions are defined to update the abstract state.

The Sig-frame-compatible function is defined to check whether one abstract state is compatible with another abstract state.

7.1.3 The Static Checker Is Effective

Defensive version of the machine

To specify the safety of program execution, we introduce a defensive version of the virtual machine. The defensive machine always checks for safety constraints before executing an instruction. If the safety constraints are violated, the instruction is effectively turned into a HALT instruction and execution loops on the same state. [5]

For each instruction, we define the following djvm-check-XXX style function (figure 7.7) to capture the constraints under which the effects of executing the instruction are defined.

A djvm-check-XXX style constraint checks the following three types of requirement:

- The current state is a good state consistent-state;
- The operand stack has the proper number of operands on it for the operation in question;

[5]This defensive machine is much simpler than the DJVM that we have defined for our JVM model M6. The defensive machine can use the same state representation of the original Small machine. This is feasible because the safety requirements in the world of Small can be readily expressed with existing components in the state of Small machine. In case of DJVM and M6, DJVM needs to augment M6 state representation to include type information.

```
(defun djvm-check-INVOKE (inst st)
  (let* ((method-name  (arg inst))
         (method-table (g 'method-table st))
         (method   (binding method-name method-table))
         (nargs    (g 'nargs method)))
    (and (consistent-state st)
         (bound? method-name method-table)
         (<= 0 (g 'max-stack
                  (binding method-name method-table)))
         (integerp nargs)
         (<= 0 nargs)
         (<= nargs (len (op-stack st)))
         (<= (+ 1 (- (len (op-stack st))
                     nargs))
             (g 'max-stack (topx (g 'call-stack st))))
         (pc-in-range (set-pc (+ 1 (get-pc st))
                              st)))))
```

Figure 7.7: Small machine: djvm-check-INVOKE

```
(defthm verified-program-never-overflow-operand-stack-in-m
  (implies
    (and (consistent-state stx)
         (state-equiv st stx))
    (<= (len (g 'op-stack (topx (g 'call-stack (m-run st n)))))
        (max-stack
             (binding (g 'method-name
                         (topx (g 'call-stack (m-run st n))))
                      (g 'method-table st))))))
```

Figure 7.8: Verified programs never overflow operand stack

- The next program counter is well defined (except for `djvm-check--RETURN`).

Verified programs are safe and the static checker is effective

We formulated and proved that verified programs execute safely. Specifically, we proved that executing verified programs never overflows the operand stacks in any call frame (figure 7.8).

This theorem (figure 7.8) can be read as follows: starting from a machine state `st` and executing some arbitrary number `n` steps without checking `djvm-check-XXX`-type of conditions — `(m-run st n)` — the size of the operand stack in the topmost frame is no greater than the operand stack size limit of the method being executed — `(max-stack (binding ...))`, as long as if there exists a *good* defensive machine state `stx` that matches the initial `st` — `(state-equiv st stx)`.

To understand this particular formulation, one may view

- `st` as an untyped term — a program without a safety proof;
- `stx` as a typed term — a program with its safety proof;

```
(defthm djvm-run-preserve-consistent-state
  (implies (consistent-state st)
           (consistent-state (djvm-run st any))))

(defthm verified-program-executes-safely
  (implies (and (consistent-state stx)
                (state-equiv st stx))
           (state-equiv (m-run st n)
                        (djvm-run stx n))))
```

Figure 7.9: Efficient machine is safe

- `(consistent-state stx)` asserts that `stx` is well typed — the proof is accepted by the static checker as a valid proof that the program will not overflow its operand stacks;
- `(state-equiv st stx)` asserts that erasing type annotations from `stx` results in `st` — `stx` and `st` are essentially the same program to the `Small` machine.

In the jargon of programming language research, this theorem roughly asserts that if a term (with no type annotation) can be augmented into an well-typed term, we then know that during evaluation of the term, the size of its operand stack component is always within bound.

If the above theorem is a kind of `progress` property in the *type safety* jargon, the following theorems may be considered a `preservation` property (figure 7.9).

Viewed from a different perspective (other than type safety), these two theorems assert that it is safe to execute a program on a more efficient machine (using `m-run` rather than the defensive `djvm-run`) to obtain *equivalent* effects on states, without violating any constraints — as long as the initial state is a good state.

These two theorems are what we consider as expressing that *the static checker is effective.*

Intuition for why static checking works

To prove that the verified programs execute safely (executions do not overflow operand stacks) and the static checker is effective (the efficient machine produces an equivalent final state as the defensive machine does), the key is to show `djvm-check-XXX` style checks always succeeds. This is because (1) `djvm--check-XXX` explicitly asserts that the operand stacks do not overflow and (2) the only difference between the original machine `m-run` and the defensive version `djvm-run` is that `djvm-run` may get stuck because some `djvm-check-XXX` checks fails.

As explained earlier, a `djvm-check-XXX` style constraint demands the following three types of properties:

- The current state is a good state `consistent-state`;
- The operand stack has a proper number of operands;
- The next instruction is well defined (except for `djvm-check-RETURN`).

Consequently, to show that the `djvm-check-XXX` style constraints are always met, we need to prove that the three types of properties asserted in `djvm-check-XXX` always hold during executions of verified programs.

First we observe that there is a direct connection between the checks encoded in `bcv-check-XXX` and the second type of check from `djvm-check--XXX`. For example, `djvm-check-PUSH` checks

```
(<= (+ 1 (len (op-stack st)))
    (g 'max-stack (topx (g 'call-stack st))))
```

while `bcv-check-PUSH` checks

```
(<= (+ 1 (g 'op-stack sig-frame))
    (max-stack sig-frame))
```

We can show that the second type of check will succeed if

```
(1)  (len (op-stack st))
  == (g 'op-stack sig-frame)
```

and

```
(2)  (g 'max-stack (topx (g 'call-stack st)))
  == (max-stack sig-frame)
```

and the method being executed is verified, i.e. the `bcv-check-PUSH` style checks succeed.

Conditions (1) and (2) above encode the fact that the execution state `st` is *on-track/approximated by* with the abstract execution state seen by the static checker at the corresponding program counter — `sig-frame`.

We add this on-track requirement directly into the definition of `consistent-state`. The remaining task is to show this `consistent-state` is preserved over steps for executing verified programs.

When we are trying to prove the `consistent-state` is an inductive invariant, we can relieve the first type of property (asserted in the `djvm-check-XXX`) by resorting to the inductive hypothesis.

As for the third type of assertion — that verified programs will never branch into an undefined pc nor will they *fall through* the end of their code —

we observe that this type of property can also be derived from the fact that the program has been successfully verified. Although the `bcv-check-XXX` does not explicitly check that the next instruction is defined, such requirement is implicitly checked in the algorithmic details of `merged-code-safe` (figure 7.4) and `mergeStackMapAndCode`.

We fold requirement on the program counter into the `consistent--state` definition as well. When we try to prove that `consistent-state` is an inductive invariant, we assume that the current program counter is within range. We can then resort to the facts about the static checking at the pc to show that the program counter after taking a step is also within range.

The proof for our goal theorems hinges on identifying a strong `consistent-state` property on the machine state and proving that it is preserved over machine execution.

One may better appreciate the necessity for identifying a good `consistent-state` with the following figure 7.10.

We start with an initial conjecture about what kind of state is a good state. The state space enclosed by the dotted line represents our conjecture. For example, initially, we might postulate that states that have, (1) a valid program counter, (2) a sufficiently small operand stack size, and (3) a defined next state are good states.

The states outside the boundary are considered "bad" states. We can define the "safe" states as being those states from which arbitrary execution can never reach a "bad" state. This subset corresponds to the space that has a curvy boundary in the picture.

However identifying the exact boundary is difficult and is often not necessary. People are more interested in finding an efficient way to confirm that

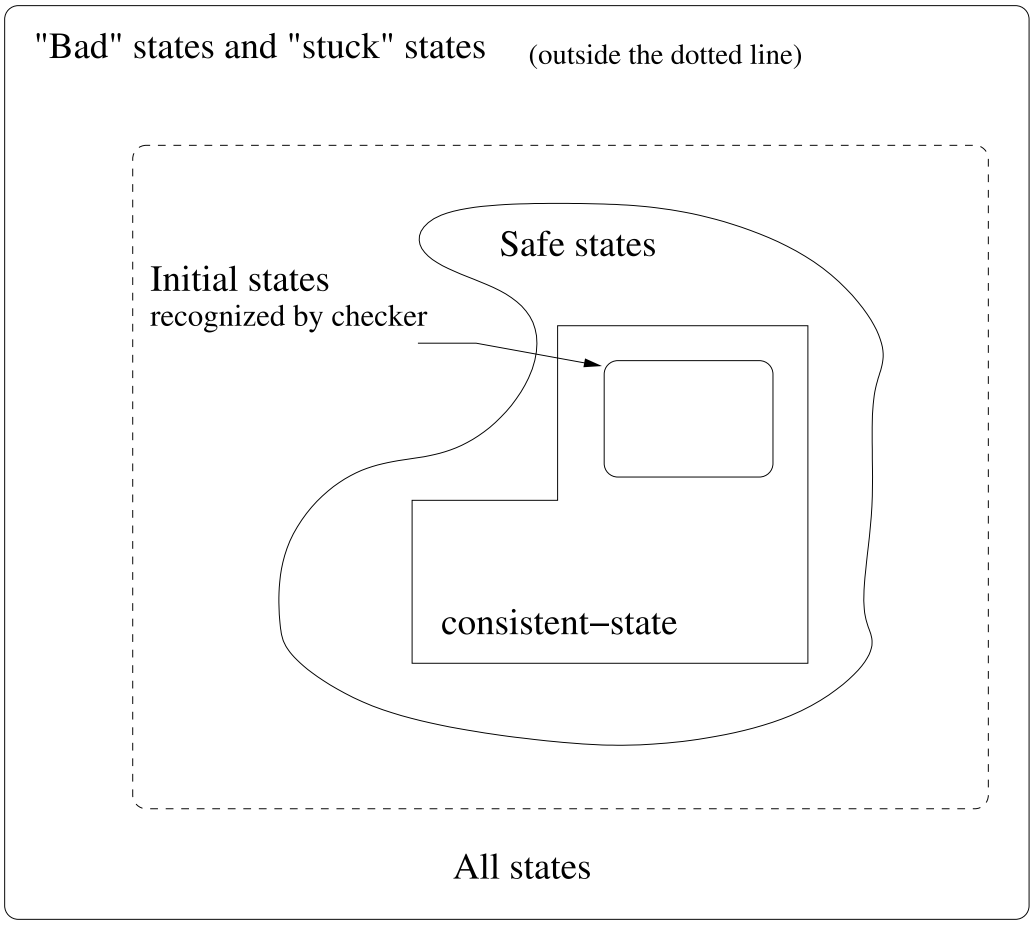


Figure 7.10: Categorizing machine states

it is safe for a program to execute — people are willing to reject some programs (that are actually safe) when recognizing them would be too inefficient.

The static checking algorithm is such an algorithm; it can can recognize a subset of the safe states among the initial states — the inner-most rectangle.

If we can define a `consistent-state` and prove that,

- it does not admit any obviously "bad" states,
- it is an inductive invariant of machine execution — that it is a transitive closure of itself (a fix point under machine steps), and,
- it admits all the initial states that the bytecode verifier admits,

we will have effectively proved that (1) executing verified programs is safe and (2) it is not necessary to conduct the runtime checks for the verified programs — the static checker is effective.

Approach

The first challenge is to identify a suitable `consistent-state`.

We start out with some inherent requirements for a state to be considered as a `consistent-state`:

- current program counter is within range;
- the operand stack size is within bounds;
- there is a well defined next state.

We attempt to prove that this definition is an inductive invariant of machine execution. We study the failed attempts and decide why each attempt failed: is it because what we want to prove is not true — the "inductive invariant" that we identified is not strong enough – or is it because the non-essential peculiarities in the definition of `consistent-state` and `static checker` have prevented the ACL2 theorem prover (and us) from proving it?

The process for identifying a good `consistent-state` is an iterative process that is interleaved with strengthening the invariants and simplifying the definition by removing non-essential aspects.

- We strengthen the `consistent-state` definition to capture the idea that the executions of verified programs are in some sense *on-track* with the *accepting run* of the static checker — the successfully abstract execution that leads the static checker to accept the program.

- We separate the procedural aspects of the static checker from the essential checks that it conducts. We introduce an alternative version of the static checker so that (1) we can explicitly talk about the abstract state observed by the static checker during its abstract execution of a program and (2) we can directly refer to the checks done by the static checker in the abstract state.

To formalize the *on-track* concept that relates a concrete state with a set of abstract states observed by the static checker during its verification process, we need to be able to more directly refer to the intermediate state observed by the static checker.

However, the existing definition of `bcv-method` is one function that only returns a "yes" or "no" answer without exposing the intermediate abstract state observed by the static checker.

We augment the static checker `bcv-method` so that when the checker returns "yes" it also returns a list of abstract states, one abstract state for each pc into the method. The list records the specific abstract states observed by the static checker when the static checker's execution reached that location. The augmented version is `collect-witness-bcv-method`.

To state that a concrete state is on-track, we assert that for each frame, after abstracting away the concrete value in the operand stack with `extract--sig-frame` — converting the concrete frame into the size of its operand stack — the resulting abstract state shall be compatible with the recorded abstracted state at the location returned by `collect-witness-bcv-method`. The function `consistent-caller-frame` (figure 7.11) defines whether a caller frame `caller-frame` is representing an execution that is on-track with the abstract execution of the method by the static checker.

After identifying the `consistent-state`, one useful lemma about it is that that the `consistent-state` is an inductive invariant of the executions on the defensive machine (figure 7.12). [6]

The proof of `djvm-run-preserve-consistent-state` is reduced to proving the leaf lemma shown in figure 7.13.

The mechanically checked ACL2 proof input is presented in `small/djvm--is-safe.lisp` [22]. We produced a set of ACL2 proof library books to prove

[6]In the case of `Small` machine, we can prove that it is an inductive invariant of the executions on the original machine — `m-run`. A simple proof sketch is as follows, we first prove `verified-program-executes-safely` (figure 7.9) as we have done in this work, we then prove `(state-equiv st st)`, combining these two results, we can conclude the `consistent-state` is preserved by `m-run` also.

However, in general, such as in the case of the JVM, such a result is not meaningful. This is because, `m-run` and `djvm-run` will operate on states that use different representations. The `consistent-state` never holds on any of the regular state that `m-run` may operates on. The `consistent-state` will be a trivial inductive invariant, as none of the regular machine state satisfies it.

```
(defun consistent-caller-frame
       (caller-frame callee-frame method-table)
  (let*  ((caller-name (g 'method-name caller-frame))
          (callee-name (g 'method-name callee-frame))
          (caller (binding caller-name method-table))
          (callee (binding callee-name method-table))
          (sig-frame (extract-sig-frame caller-frame
                                        method-table))
          (pc (g 'pc sig-frame))
          (record-frame-sig
              (binding pc
                   (collect-witness-bcv-method
                                caller method-table))))
    (and (bcv-method caller method-table)
         (<= (+ 1 (len (g 'op-stack caller-frame)))
             (g 'max-stack caller-frame))
         ;; enough space for return value

         (equal (g 'method-name caller)
                caller-name)
         ;; correct method-name

         (integerp (g 'pc caller-frame))
         (<= 1 (g 'pc caller-frame))
         (< (g 'pc caller-frame) (len (g 'code caller)))
         ;; pc with in range

         (sig-frame-compatible
          (sig-frame-push-value (g 'ret callee) sig-frame)
          record-frame-sig))))
         ;; runtime state's signature matches the
         ;; abstract state that bcv-method recorded.
```

Figure 7.11: Concrete execution on track with abstract execution

```
(defthm djvm-run-preserve-consistent-state
  (implies (consistent-state st)
           (consistent-state (djvm-run st any))))
```

Figure 7.12: Inductive invariant `consistent-state`

```
(encapsulate ()
 (local (include-book "djvm-INVOKE"))
 (defthm consistent-state-preserved-by-DJVM-INVOKE
   (implies (and (consistent-state st)
                 (equal (next-inst st) inst)
                 (equal (op-code inst) 'INVOKE)
                 (djvm-check-INVOKE (next-inst st) st))
            (consistent-state
             (djvm-execute-INVOKE inst st)))))

(encapsulate ()
  (local (include-book "djvm-PUSH"))
  (defthm consistent-state-preserved-by-DJVM-PUSH
    (implies (and (consistent-state st)
                  (equal (next-inst st) inst)
                  (equal (op-code inst) 'PUSH)
                  (djvm-check-PUSH inst st))
             (consistent-state
              (djvm-execute-PUSH inst st)))))

(encapsulate ()
 (encapsulate ()
  (local (include-book "djvm-IFEQ"))
  (defthm consistent-state-preserved-by-DJVM-IFEQ
   (implies (and (consistent-state st)
                 (equal (next-inst st) inst)
                 (equal (op-code inst) 'IFEQ)
                 (djvm-check-IFEQ inst st))
            (consistent-state
             (djvm-execute-IFEQ inst st))))))
```

Figure 7.13: Leaf lemma: `djvm-is-safe`

```
(defun bcv-simple-method1 (pc code sig-vector)
  (declare (xargs :measure (len code)))
  (if (endp code) t
    (let* ((inst (car code)))
      (and (bcv-simple-inst pc inst sig-vector)
           (bcv-simple-method1 (+ 1 pc)
                               (cdr code)
                               sig-vector)))))
```

Figure 7.14: Alternative version of `bcv-method`

this type of leaf lemma for the `Small` machine. [7]

The third step is to relate the checking done by the static checker to the checking done by the defensive version of the machine. The facts that we have access to are (1) the concrete state is related to some abstract state by `consistent-state` and (2) the `bcv-method` succeeded.

The difficulty lies in the apparent mismatch between the step-by-step execution of the defensive machine and the one-sweep style operation of the `bcv-method`. We need a more direct way to obtain `bcv-check-XXX` type results from the fact that `bcv-method` succeeded.

We define a new and simple step-by-step static checking algorithm (figure 7.14) in `small/bcv-simple-model.lisp`. In this alternative version of the static checker, what is checked at each program counter is very explicit.

We then prove a second set of leaf-level theorems about each instruction, relating the results of checks done by this alternative static checker to the result of runtime checks done by the defensive machine.

The last big step is to prove if a method is verified by the original static

[7] See `small/consistent-state-properties.lisp`, `small/generic.lisp`, and `small/consistent-state-step.lisp`.

```
(defthm bcv-simple-check-implies-djvm-check
  (implies (and (consistent-state djvm-s)
                (bcv-simple-check-step-pre
                   (next-inst djvm-s)
                   (extract-sig-frame
                      (topx (g 'call-stack djvm-s))
                      (g 'method-table djvm-s)))
           (djvm-check-step djvm-s))))
```

Figure 7.15: Relating bcv-simple-inst with djvm-check-step

```
(encapsulate ()
 (local (include-book "INVOKE"))
 (defthm bcv-simple-check-invoke-implies-djvm-check
   (implies
    (and (consistent-state djvm-s)
         (equal (op-code (next-inst djvm-s)) 'INVOKE)
         (bcv-simple-check-invoke
            (next-inst djvm-s)
            (extract-sig-frame (topx (g 'call-stack djvm-s))
                               (g 'method-table djvm-s))))
    (djvm-check-invoke (next-inst djvm-s) djvm-s))))

(encapsulate ()
  (local (include-book "PUSH"))
  (defthm bcv-simple-check-push-implies-djvm-check
    (implies
      (and (consistent-state djvm-s)
           (equal (op-code (next-inst djvm-s)) 'PUSH)
           (bcv-simple-check-push
              (next-inst djvm-s)
              (extract-sig-frame (topx (g 'call-stack djvm-s))
                                 (g 'method-table djvm-s)))
      (djvm-check-push (next-inst djvm-s) djvm-s))))
```

Figure 7.16: Leaf lemma: bcv-simple-check-implies-runtime-check

```
(defthm |method-verified-implies
         -bcv-simple-check-step-pre-on-recorded-signature|
  (implies
    (and (bcv-method method method-table)
         (equal method
                (binding (g 'method-name method) method-table))
         (bcv-verified-method-table method-table)
         (integerp pc)
         (<= 0 pc)
         (< pc (len (g 'code method)))
         (equal inst (nth pc (g 'code method)))
         (member inst (g 'code method)))
   (bcv-simple-check-step-pre
         inst
         (binding pc
                 (collect-witness-bcv-method
                              method method-table)))))
```

Figure 7.17: Relating `bcv-method` with `bcv-simple-inst`

checker algorithm, the particular checks conducted by the `bcv-simple-method` at each instruction are satisfied (figure 7.17).

An important lemma for proving this result is to show that all methods accepted by the original `bcv-method` will be also be accepted by the `bcv--simple-method`.

After strengthening the definition of `consistent-state` and simplifying the definition of the static checker, we can prove that `consistent-state` is preserved by execution according to `m-run`.

```
(defthm djvm-run-preserve-consistent-state
  (implies (consistent-state st)
           (consistent-state (djvm-run st any))))
```

```
(encapsulate ()
  (local
    (include-book "bcv-succeed-implies-bcv-simple-succeed"))
  (defthm bcv-succeed-implies-bcv-simple-succeed
    (implies (and (bcv-method method method-table)
                  (equal method (binding (g 'method-name method)
                                         method-table))
                  (bcv-verified-method-table method-table))
             (bcv-simple-method
              (s 'sig-vector
                 (collect-witness-bcv-method
                  method method-table)
                 method)
              method-table))))
```

Figure 7.18: Relating `bcv-method` with `bcv-simple-method`

```
(defthm verified-program-executes-safely
  (implies (and (consistent-state djvm-s)
                (state-equiv jvm-s djvm-s))
           (state-equiv (m-run jvm-s n)
                        (djvm-run djvm-s n))))
```

Making use of the above results and the definition of `consistent-state` and `state-equiv`, we can prove

```
(defthm verified-program-never-overflow-operand-stack-in-m
  (implies
    (and (consistent-state stx)
         (state-equiv st stx))
    (<= (len (g 'op-stack (topx (g 'call-stack (m-run st n)))))
        (max-stack
             (binding (g 'method-name
                         (topx (g 'call-stack (m-run st n))))
                      (g 'method-table st))))))
```

7.1.4 Summary

The simple safety requirement for the `Small` machine is that executing verified program does not overflow operand stacks. However this simple property can not be proved directly. We identified a stronger property `consistent-state`. We proved that executing verified program preserves this property. We then proved that operand stacks from various call frames in a `consistent-state` are never too big. To prove that verified programs preserve the `consistent-state`, we introduced an alternative definition of the static checker. We appealed to the fact that programs are *verified* — that they are accepted by the abstract executions of the new static checker to prove that `consistent-state` is preserved. We then showed that programs verified by the original static checker will also be verified by the new static checker. We completed the definitions and proofs in formalizing and verifying the simple machine and its static checker. The mechanically checked ACL2 proof input is of 11,360 lines in 47 files [22]. The proof input describes 691 lemmas, requiring 196 inductive proofs.

Our (incomplete) proof for showing that the JVM is safe follows the same approach. The JVM and its CLDC bytecode verifier are much more complicated than the `Small` counterparts. (1) The JVM has a class loading mechanism that can introduce new classes and methods during program execution and (2) the abstract states explored by the bytecode verifier is different from the concrete states traversed by a regular JVM — it is more difficult to show that `bcv-check-XXX` style checks can be used to predict the outcome of `djvm-check-XXX` (See *Divergence 1*, page 1.2.4).

In the rest of this chapter, we present a few results that we have proved about the JVM and its bytecode verifier. These results are strategic steps for constructing a final safety proof for the JVM and its bytecode verifier.

7.2 Bootstrap Class Loader Verified

The bootstrap class loader updates a JVM state. It modifies the class table to reflect the facts that new classes have been loaded. It also introduces new objects into the heap to represent the runtime constant pool and the class itself. It does not change the thread's execution state.

A safe class loader shall first maintain the consistency in a JVM state. For example, each loaded class has a corresponding object of type `java.lang.Class` in the heap, superclasses of all loaded classes need to be loaded, all newly created objects in the heap must be *consistent objects* — their classes are loaded, and they have all the declared fields, and these fields are initialized to values of suitable type.

A safe class loader shall never invoke operations with inputs that violates the *guards* of these low level operations.

In this section, we describe the ACL2 proofs that shows that the class loader preserves the `consistent-state`, as well as proofs necessary to *guard-verify* the class loader.

```
(defun resolveClassReference1 (classname s)
  (declare (xargs :guard (and (load_array_class_guard s)
                              (load_class-guard s))))

  (if (array-type? classname)
      (load_array_class (array-base-type classname) s)
    (if (class-loaded? classname s)
        s
      (load_class classname s))))

...

(defun resolveClassReference (classname s)
  (declare (xargs :guard (resolveClassReference-guard s)))
  (let ((new-s (resolveClassReference1 classname s)))
    (if (not (hasAccessToClass (current-class s) classname s))
        (state-set-pending-exception-safe
              "java.lang.IllegalAccessException" s)
      new-s)))
...

(defthm resolveClassReference-preserve-consistency
  (implies (consistent-state s)
           (consistent-state (resolveClassReference any s)))
```

Figure 7.19: Class loading preserves consistent state

7.2.1 Consistent State Preserved

What we proved

We proved that the bootstrap class loader of the M6 preserves `consistent--state` (figure 7.19). [8]

To prove that an update operation preserves the consistent state predicate, we prove two kinds of lemmas. The first kind of lemma asserts that

[8]We proved the above results with some ACL2 *skip proofs*.

```
(defun consistent-state (s)
  (and
    ...
    (consistent-class-hierachy
                     (instance-class-table s))
    (consistent-heap (heap s)
                     (instance-class-table s)
                     (array-class-table s))
    (consistent-heap-init-state (heap s)
                                (instance-class-table s)
                                (heap-init-map (aux s)))
    (consistent-heap-array-init-state (heap s)
                                      (instance-class-table s)
                                      (array-class-table s)
                                      (heap-init-map (aux s)))

    (consistent-class-table (instance-class-table s)
                            (external-class-table s) (heap s))

    (consistent-thread-table (thread-table s)
                             (instance-class-table s)
                             (heap s))
    ...
    (instance-class-table-inv s)
    (array-class-table-inv s)
    (boot-strap-class-okp s)
    ...
    ))
```

Figure 7.20: Class loader: `consistent-state`

unchanged components are consistent in the new state. The second kind of lemma asserts that the updated components are consistent with the new state.

Consider a simple operation that creates a new object and adds it to the heap. We prove that the simple operation preserves the consistency of the heap. The first kind of lemma asserts that any object that exists in the heap before inserting the new object, will remain a good object in the new state. [9] A second kind of lemma asserts that the newly created objects are good objects with respect to the newly updated state. That is, each newly created object has all the fields declared by its class and its superclasses. Each field is initialized either by `NULL` or a valid reference to some object in the heap, and the referenced object is of suitable type.

What the challenges are

The complexity for proving that the class loader preserves the consistent state property (figure 7.20) results from two facts:

- The definition of consistency is complicated.

  There are many consistency requirements between different state components. Whether a value is valid not only depends on the type of the value and the heap, but also depends on the class table. Updates to any of the three obligate us to prove that a previously valid value remains valid — even when the value itself has not changed. Similarly, whether a class declaration is valid depends on the class table, the heap, and the external class table. Whether an object is a valid object depends on the

[9]To prove this is not as trivial as it may appear. The judgement about whether an object is good depends directly on what other objects exist in the heap. We need to know that the operation does not remove an existing object nor change the type of any existing object.

heap, the class table, and the values in every field.

- The class loading operation is complicated.

  The class loader invokes multiple update operations to load a class. To show the class loading operation as a whole preserves the consistent state predicate, we need to show that each individual low level operation preserves the predicate.

These two factors together make it necessary to prove n^2 number of properties (in the worst case) — assuming the number of consistency constraints is on the order of n and the number of different operations is on the order of n.

The following is an example showing the lemmas that are necessary to prove that one simple update operation preserves one specific consistency property on the JVM state.

The simple state update operation we want show is a `cons` operation — the operation adds a new runtime representation of a class into the class table. It is one of the lowest level operations invoked by the class loader.

The specific consistency requirement in this example is that a good thread runtime state remains good. We want to prove `consistent-thread--entry-add-new-class` as shown in figure 7.21.

We know the class loader does not change the thread' execution state. However in order for one to conclude a thread's execution state is good, one needs to check that

- all values stored in the call frames are valid;
- the methods pointed by the `method-ptr` fields of call frames exists;

```
(defthm consistent-thread-entry-add-new-class
  (implies (and (consistent-thread-entry thread cl hp)
                (consistent-class-hierachy (cons class-rep cl))
                (wff-class-rep class-rep)
                (isClassTerm class-rep)
                (not (isClassTerm
                        (class-by-name
                            (classname class-rep) cl))))
           (consistent-thread-entry thread
                                    (cons class-rep cl)
                                    hp)))
```

Figure 7.21: `Cons` Preserving `consistent-thread-entry`

- the object pointed to by the `thread-ref` field exists and the type of the object is `assignmentCompatible` to `java.lang.Thread` or implements `java.lang.Runnable` interface;

- the caller-callee relation holds between adjacent frames in the call stack.

We know that judgement about all these four type of consistency conditions depend on the class table — what is a valid value depends on the class table that we are checking the validity against, whether a method pointer points to anything also depends on the class table, and so on.

Although the simple operation only add a new runtime class representation to the class table, we need to prove many lemmas to show that what has been valid will remain valid. Specifically, figure 7.22 gives an incomplete list of lemma necessary to prove that `consistent-thread-entry` is preserved over this simple update operation.

Some of these lemmas are highly non-trivial.

```
(defthm consistent-value-x-add-new-class
  (implies (and (consistent-value-x v  cl hp)
                ...)
           (consistent-value-x v (cons class-rep cl) hp)))

(defthm deref-method-equal-add-new-class
  (implies (and (deref-method method-ptr cl)
                ...)
           (equal (deref-method method-ptr (cons class-rep cl))
                  (deref-method method-ptr cl))))

(defthm isJavaSubclass1-cons-new-class
  (implies (and (isJavaSubclassOf1 new1 new2 cl seen)
                ...)
           (isJavaSubclassOf1 new1 new2
                              (cons class-rep cl) seen)))

(defthm assignmentcompatible-cons-new-class
  (implies (and (assignmentCompatible new1 new2 cl)
                ...)
           (assignmentCompatible new1 new2
                                  (cons class-rep cl))))

(defthm consistent-call-frame-add-new-class
  (implies (and (consistent-frame frame cl hp)
                ...)
           (consistent-frame frame (cons class-rep cl) hp)))

(defthm consistent-thread-entry-add-new-class
  (implies  (and (consistent-thread-entry thread cl hp)
                 ...)
         (consistent-thread-entry thread
                                   (cons class-rep cl) hp)))
```

Figure 7.22: Lemmas needed to show `consistent-thread-entry` is preserved

```
(defthm assignmentcompatible-cons-new-class
  (implies
     (and (assignmentcompatible new1 new2 cl)
          (isClassTerm class-rep)
          (consistent-class-hierachy (cons class-rep cl))
          (not (isClassTerm
                  (class-by-name (classname class-rep) cl))))
     (assignmentcompatible new1 new2 (cons class-rep cl))))
```

Figure 7.23: `AssignmentCompatible` judgement not affected

For example, in order to prove the first lemma in the list, `consistent--value-x-add-new-class` (figure 7.22), we need to prove the `assignment-compatible-cons-new-class` lemma (figure 7.23).

This lemma roughly asserts that a *positive* `assignmentCompatible` judgement will never be changed to a *negative* judgement, after we introduce a new class that has not been loaded yet — assuming adding the new class into the class table, the resulting class table describes a consistent class hierachy.

To prove this lemma, we need to make good use of the third hypothesis, `consistent-class-hierarchy`, which asserts if a class is loaded, then all its superclasses and superinterfaces are already loaded. Adding some new class to the leaves of the class hierarchy tree does not affect a previously positive judgement.

One tips for proving complicated inductive invariants

Our definition of `consistent-state` is a conjunction of 23 conditions. We often face the challenge of proving that a given operation when executed under specific conditions will produce another state that satisfies the `consistent--state` predicate.

```
(implies (consistent-state s)
         ...  (op-condition-x value s) ...
         (consistent-state (op value s)))
```

If we let ACL2 to blindly expand both `consistent-state` terms, the goal theorem will have a very complicated form. The ACL2 theorem prover's built-in heuristics seem always to fail to pick a correct proof strategy. Furthermore, with such a complicate term, ACL2's human operator cannot identify helpful lemmas to guide the theorem prover.

To prove such a complicated theorem, we have used the following strategy — instead of proving the above theorem, we prove the following alternative version first

```
(implies (consistent-state s)
         ...  (condition-x value s) ...
         (consistent-state-step (op value s)))
```

Here, the definition of `consistent-state-step` exactly matches `consistent-state` but by giving it a different name we can control the expansion of the two definitiions separately.

We keep `consistent-state` *disabled* — we tell ACL2 never to use the definition of `consistent-state` directly. We keep `consistent-state-step` *enabled.* So the above intermediate form can be reduced by ACL2 theorem prover into 23 cases. In the very beginning, because the definition of `consistent-state` is disabled, the ACL2 theorem prover *knows nothing* about a state `s` that is a `consistent-state`. All the subgoals will fail. We then identify necessary properties of a `consistent-state` and encode them as lemmas. We prove these lemmas with `consistent-state` enabled. After we identified enough properties of a `consistent-state`, we can prove the intermediate form of the lemma.

Then we can prove a very simple and very specific lemma

```
(implies (consistent-state-step (op value s))
         (consistent-state (op value s)))
```

and then disable `consistent-state-step`.

We find this trick generally useful for proving that a complicated property on a data structure is preserved over an update operation.

7.2.2 Guard Assertions Verified

Guard and guard verification

We write *guards* for the JVM operations. We use them to express requirements on operands that we expect to invoke the operations with. For the simple operation `car`, which returns the first element of a pair, the guard asserts that the input shall be a pair `consp`.

Different from a typical assertion in a program, by writing down the guard assertions, we are also expressing an internal consistency requirement on the interactions between operations that invoke each other. Suppose operation A invokes operation B under certain conditions C. We consider the guard assertions for them as expressing the condition that when A is invoked with inputs satisfing its guard and the condition C holds, B is guaranteed to be invoked with inputs that satisfy B's guard.

To show that an operation is safe, we *guard-verify* the operation. We prove that when the input to the operation satisfies the guard, all subsequent

```
(defun load_CP_entry (cpentry-s S)
  (declare (xargs :guard (load_CP_entry_guard cpentry-s S)))
  (if (equal (cpentry-type-s  cpentry-s) 'STRING)
      (let ((str  (string-value-cp-entry-s cpentry-s)))
        (mv-let (the-String-obj new-S)
                (ACL2-str-to-JavaString str S)
                (let* ((heap   (heap new-S))
                       (new-addr (alloc heap))
                       (new-heap
                           (bind new-addr the-String-obj heap)))
                  (mv (make-string-cp-entry new-addr)
                      (update-trace
                           new-addr
                           (state-set-heap new-heap new-s))))))
    (mv cpentry-s s)))
```

Figure 7.24: Guard: `load_CP_entry`

low level operations are invoked with inputs that satisfy their guards. For example, when `load_CP_entry` (figure 7.24) is used to create a runtime representation with an input satisfing `load_CP_entry_guard` , the operation used to create a new `java.lang.String` object `ACL2-str-to-JavaString` is invoked with suitable inputs.

If (1) we can guard-verify the top-most level operation that starts the JVM interpreter and (2) we can show that all the initial states for executing verified programs satisfy the guard of the top-most operation, we can conclude with confidence that during executions of verified programs, none of the low level operation will be invoked with their guard violated. We can then inspect the guard definition of each operation and see whether it captures our *safety* requirement.

What we proved

We defined guards for the class loading operations. We defined the guards for executing a subset of the JVM instructions. We verified the guards of these operations.

It is worth noting that we do not explicitly write down the ACL2 formulas to state that a function is guard verified. Doing so would require writing down too many formulas. Instead, the ACL2 theorem prover automatically generates the proof obligations when we (implicitly) ask it to guard-verify a function by exhibiting a guard for the function.

What the challenges are

The first challenge is to identify meaningful and verifiable guards for JVM operations. As we explained in the section 6.5.4, there is an uninteresting way to define verifiable guards. If one adopts such a approach, evaluating a guard is just as complicated as executing the operations. We want the guard that we define to be simpler than the operation — it should capture the intrinsic safety requirements on the input for the operation but not peculiarities of a specific implementation of the operation. The intrinsic safety requirement is ideally be expressible directly without resorting to executing the operation and checking all the conditions along the way.

The second challenge is to configure the ACL2 theorem prover to verify the guards. This is a non-trivial task. It is often the case that in order to guard-verify an operation composed of a few simpler operations, we needs to prove theorems that capture the effects of executing these simpler operations.

Consider an operation `OP`, defined as `(B (A x))` — a composition of two simple operations `A` and `B`, where `A`'s outputs become inputs to the operation `B`. To guard-verify `OP`, not only do we need to prove theorems that relate the guard of `OP` with the guard of `A`, but also we need to prove theorems for characterizing the effects of executing the operation `A`. We need the latter kind of theorems before we can prove that the guard of `B` will be met when `OP` is invoked with inputs that satisfies its guard. When the operation `B` is a recursively defined operation, we also need to use induction to prove any property of executing `B`.

7.3 Proofs Relating Two Bytecode Verifiers

7.3.1 Introduction

In proving that the `Small` machine is safe and its CLDC-style static checker is effective (section 7.1), one key concept that we have identified is the *on-track* requirement, which asserts that a good concrete state needs to be in the "predictable" part of the state space by being on track with the abstract executions of the static checker.

To express the "on-track" property, we need to be able to refer to the bytecode verifier's abstract execution in a step by step way — what is the current abstract state at a specific program counter and what are checked against the abstract state at that step. We can then match the step-by-step concrete execution with the abstract execution.

However, a CLDC-style static checker is defined as a function that returns an "yes" or "no" answer. We need to define an alternative version of the static checker to expose the intermediate abstract states observed by the static checker.

We also need to prove that all programs verified by the CLDC-style static checker can be verified by the alternative version of the static checker.

For the `Small` machine, we defined the alternative static checker `bcv-simple-method` (figure 7.14). We proved a verified program by the alternative static checker preserves the good state predicate. We proved that any program verified by the original CLDC-style static checker can also be verified by the alternative version of the static checker.

In this section, we present the alternative definition of the JVM bytecode verifier (in the style of `bcv-simple-method` from the `Small` machine). We present the ACL2 proofs showing that any verified program by the original CLDC bytecode verifier are also verifiable with respect the simpler JVM bytecode verifier.

7.3.2 Alternative Bytecode Verifier

The official CLDC specification describes the bytecode verification process with over 100 Prolog-style derivation rules. Although the specification is declarative in form (as declarative rules), the verification process encoded is *procedural* in essence. The official specification describes how different steps shall be implemented and sequenced together — instead of describing directly what needs to be checked at each offset into the program.

To verify a method, the CLDC specification describes: (1) how to extract *stackmaps*, (2) how to merge the stackmaps with the instructions into a single mixed stream of *stackmaps* and instructions, and (3) how to symbolically execute the mixed stream of instructions and stackmaps. There is no way to know what is checked at a particular offset into the program — except by using the procedural bytecode verifier to symbolically execute the program from the program start until the execution reaches the program location, and then consulting the abstract state of the bytecode verifier to find out what is checked.

We defined an alternative bytecode verifier in BCV/typechecker--simple.lisp [22]. What is checked at each program location is made more explicit.

The core of the alternative bytecode verifier takes three inputs: a sequence of instructions — code, a mapping that maps pcs of instructions into the abstract states that the instruction will be executed in — stackmaps, and a data structure representing the environment in which the verification is conducted — env (figure 7.25. The env data structure records the class, the method, and the type hierarchy information.

For each instruction inst from the code, the new algorithm first locates the recorded abstract state from the stack-maps with

```
(searchStackFrame (instrOffset inst)
                  (stack-map-wrap stack-maps))
```

. Then, it checks whether it is safe to execute the instruction in such an abstract state by calling

```
(instructionIsTypeSafe inst env  ... the-abstract-state..)
```

. The algorithm also checks that the state resulting from executing the instruction is compatible with the next abstract state that is recorded in stack-maps with

```
 (frameIsAssignable  nextStackFrame
(defun bcv-simple-method1 (code stack-maps env)
  (if (endp code) t
    (let* ((inst (car code)))
      (and (instructionIsTypeSafe inst env
              (searchStackFrame (instrOffset inst)
                                (stack-map-wrap stack-maps)))
              ;; locate the abstract state for executing
              ;; the instruction in.
           ;; check whether it is safe to execute
           (mv-let (nextStackFrame exceptionStackFrame)
             (sig-do-inst inst env
                          (searchStackFrame
                               (instrOffset inst)
                               (stack-map-wrap stack-maps)))
             ;; compute next state
             ;; need to check
             ;;     (1) compatible with exception handler
             ;;     (2) compatible with recorded next state
             (and (instructionSatisfiesHandlers
                   env
                   (instrOffset inst)
                   exceptionStackFrame)
                  (or (equal nextStackFrame 'aftergoto)
                      (and (consp (cdr code))
                           (not (isEnd (cadr code)))
                           (frameIsAssignable
                                 nextStackFrame
                                 (searchStackFrame
                                  (instrOffset
                                           (cadr code))
                                  (stack-map-wrap
                                           stack-maps))
                                 env)))
                  (bcv-simple-method1 (cdr code)
                                      stack-maps env)))))))
                  ; remaining code are safe.
```

Figure 7.25: Alternative bytecode verifier

```
                    (searchStackFrame
                          (instrOffset (cadr code))
                          (stack-map-wrap stack-maps))
                    env)
```

The key difference between this definition and the original bytecode verifier definition (figure 5.7) is that at each instruction, the new bytecode verifier checks

```
(instructionIsTypeSafe
             inst env
             (searchStackFrame (instrOffset inst)
                               (stack-map-wrap stack-maps)))
```

, where `stack-maps` and `env` remain unmodified. The only variable is `(instOffset inst)`. On the other hand, the original bytecode verifier checks

```
(instructionIsTypeSafe instr Enviornment  StackFrame)
```

where the `StackFrame` is updated on the fly — during the bytecode verification process — to capture the accumulated effects of the symbolic execution of the merged stack frame and instructions up to the current instruction.

7.3.3 Bytecode Verifier Is Effective

In our efforts to showing that the CLDC bytecode verifier is effective and the JVM is safe for executing verified programs, we followed the approach that we had successfully used for proving the CLDC-style static checker verifier (section 7.1).

The top level proof step is to show that (1) the alternative bytecode verifier is effective and (2) programs verified by the alternative bytecode verifier are safe to execute.

To prove the alternative bytecode verifier is effective,

- We need to define a strong good state.

  We strengthen the inherent safety requirement — such as the operand stack sizes being within bound and fields in objects holding sensible values — by also asserting an *on-track* property.

  The on-track property asserts that the current JVM runtime state is approximated with some corresponding abstract state observed by the bytecode verifier.

- We relate the outcome of the runtime checking on a runtime state to the result of the static checking on the corresponding abstract state with which the runtime state is on-track.

  We make use of the on-track property between the concrete and the abstract state. We show that if the static checking succeeds on the abstract state — which we know it will because we can assume the program being executed has been verified — the concrete machine's runtime checks will also succeed.

- We prove that the next concrete state remains on-track with the next abstract state observed by our alternative bytecode verifier.

  We need to prove this third property to construct our inductive proof that executing verified programs from a good state will remain safe.

We call the second and third bullets above *leaf-level properties* that we need to prove for each simple JVM operation.

We have not finished the proof that our alternative bytecode is effective for M6. In the later section 7.4 *Proving Leaf-level Lemmas*, we explain the leaf-level lemmas that we have formulated for showing that the bytecode verifier is effective. We explain our support lemma library infrastructure for proving them.

7.3.4 Reduction Theorem Verified

Following our overall approach for proving that the CLDC bytecode verifier is effective, we need to prove a *reduction* theorem — in addition to the important theorem that asserts that the alternative version of the bytecode verifier is effective.

The reduction theorem states that any program verified by the CLDC bytecode verifier can be also verified by the alternative version of the bytecode verifier. Once we prove this reduction theorem, assuming that we proved that all programs verified by the alternative version of the bytecode are safe to execute, we can then show that the CLDC bytecode verifier is effective and program verified by the CLDC bytecode verifier execute safely.

The reduction theorem

The desirable reduction property is captured in this ACL2 formula shown in the figure 7.26 from the ACL2 book `BCV/bcv-succeed-implies-bcv-simple--succeed.lisp` in [22].

The first hypothesis `(mergedcodeIsTypesafe env ...)` asserts that

```
(defthm mergedCodeIsTypeSafe-implies-bcv-simple-method1
  (implies (and (mergedCodeIsTypesafe  env
                       (allinstructions env) init-frame)
                (good-frame init-frame env)
                (pc-wff-mergedcode1 0 (allinstructions env))
                (all-good-frames
                       (extract-frames (allinstructions env))
                       env)
                (good-env env))
           (bcv-simple-method1
                (extract-code (allinstructions env))
                (collect-sig-frame-vector env
                                          (allinstructions env)
                                          init-frame)
                 env)))
```

Figure 7.26: Alternative verifier accepts all CLDC verified programs

the `(allinstructions env)`, which is a mixed stream of instructions and stack maps, can be verified with respect to the initial abstract state `init-frame`.

`Good-frame` asserts that the initial frame represents a valid abstract state — all types mentioned in a *good-frame* are well defined in the context of the `env`. [10]

The third hypothesis `(pc-wff-mergedcode1 ...)` asserts that the program counters from the mixed stream of instructions and stack maps are well formed — the program counters are consecutive in a non-decreasing order. If the `n`-th element of the stream (with program counter `pc`) is an instruction,

[10]We need these types to be well defined, because in proving the reduction theorem, we often need to show that when a certain check succeeds on a more general abstract state, the check will succeed on a more specific abstract state. We need to show that the *more general* and *more specific* relations between abstract states are transitive. However, we can only prove the type assignment compatible property when the types involved are known types in a consistent type hierarchy.

```
(defun pc-wff-mergedcode1 (pc mergecode)
  (if (endp mergecode) nil
    (cond ((isEnd (car mergecode))
           (and (integerp pc)
                (< 0 pc)
                (equal (nth 1 (car mergecode)) pc)
                (endp (cdr mergecode))))
          ((isStackMap (car mergecode))
           (and (equal (mapOffset (getMap (car mergecode))) pc)
                (consp (cdr mergecode))
                (not (equal (mapFrame (getMap (car mergecode)))
                            'aftergoto))
                (not (isEnd (cadr mergecode)))
                (pc-wff-mergedcode1 pc (cdr mergecode))))
          ((isInstruction (car mergecode))
           (and (equal (instrOffset (car mergecode)) pc)
                (<= 1 (jvm::inst-size (car mergecode)))
                (pc-wff-mergedcode1
                    (+ (JVM::inst-size (car mergecode)) pc)
                    (cdr mergecode))))
          (t nil))))
```

Figure 7.27: Well-formed program counter: `pc-wff-mergedcode1`

the `n+1`-th element of the stream will have a program counter `pc + length of the instruction`. If the `n`-th element of the stream is an abstract state, the `n+1` element of the stream needs to have a program counter equal to `pc`.

The fourth hypothesis asserts if we pick out stack maps from the input to the `mergedCodeIsTypeSafe` and we build an abstract state out of them, every abstract state must represent a good abstract state with respect to `env`.

The last hypothesis `(good-env ...)` asserts that the class descriptions from the `env` represents a good class hierarchy — the superclass chain from any class does not have loops in it, that the superinterface chains do not have loops, and that the superclass and superinterfaces of a class are also represented, except for `java.lang.Object`'s superclass.

The conclusion asserts that we can then collect a vector of stack frames. We can verify the sequence of instructions against this vector of stack frames in the context of `env`. We call this vector of stack frames a *witness vector*.

```
(bcv-simple-method1
     (extract-code (allinstructions env))
     (collect-sig-frame-vector env
                               (allinstructions env)
                               init-frame)
     env)
```

The reduction theorem says if the CLDC bytecode verifier `mergedCode-IsTypeSafe` succeeds, we can construct a witness with `(collect-sig-frame--vector ...)` and our alternative bytecode verifier accepts this witness as a proof that the sequence of instructions is safe to execute on the JVM.

We construct the witness using the function `collect-sig-frame--vector` (figure 7.28). The function works much like the function `mergeCode-IsTypeSafe` (figure 5.7, page 146). `Collect-sig-frame-vector` symbolically executes the `mergecode` following the same core algorithm. In addition, when an instruction is known to be safe to execute, the operation records the abstract state in which the instruction is executed. When the symbolic execution of `mergecode` ends successfully, the collected stack frames are returned.

```
(defun collect-sig-frame-vector (env mergedcode stackmap)
  (if (endp mergedcode) nil
    (if (endp (cdr mergedcode)) nil
      (if (equal stackmap 'afterGoto)
          (if (isStackMap (car Mergedcode))
              (collect-sig-frame-vector
                   env (cdr mergedcode)
                   (mapFrame (getMap (car mergedcode))))
           nil)
    (cond ((isStackMap (car mergedcode))
           (and (frameIsAssignable stackmap
                     (mapFrame (getMap (car mergedcode)))
                     env)
                (collect-sig-frame-vector env
                    (cdr mergedcode)
                    (mapFrame (getMap (car mergedcode))))))
          ((isInstruction (car mergedcode))
           (let ((offset     (instrOffset (car MergedCode)))
                 (instr      (car MergedCode)))
             (and (instructionIsTypeSafe instr env stackmap)
                  (mv-let (NextStackFrame ExceptionStackFrame)
                          (sig-do-inst instr env stackmap)
                          (and (instructionSatisfiesHandlers
                                          env offset
                                          ExceptionStackFrame)
                               (mergedCodeIsTypeSafe
                                     env
                                     (cdr MergedCode)
                                     NextStackFrame)
                               (cons (list offset stackmap)
                                     (collect-sig-frame-vector
                                          env
                                          (cdr mergedcode)
                                          NextStackFrame)))))))
          (t nil))))))
```

Figure 7.28: Creating a “witness”: `collect-sig-frame-vector`

Sketch of the mechanically verified proof

Our goal is to prove that all programs accepted by the CLDC bytecode verifier will also be accepted by the alternative simpler bytecode verifier. We first observe that the alternative version of the bytecode verifier `bcv-simple-method1` (figure 7.25) does the two kinds of checks

- For instruction, it looks up its corresponding abstract state in the witness — vector of stack maps. It checks that the safety *pre-condition* for executing the instruction is met.
- It symbolically executes the instruction. It looks up the abstract state that corresponds to the resulting state after execution. It checks the safety *post-condition* that the state resulting from symbolic execution is compatible with the recorded state.

We need to prove that both pre-condition and post-condition checks against the recorded states from the witness vector `(collect-sig-frame--vector ...)` will succeed, when the original CLDC bytecode verifier `mergedCodeIsTypeSafe` accepts the program as verified.

We first prove that the first kind of checks succeeds (figure 7.29. That is, if a program is verified by `mergedCodeIsTypesafe` (the original bytecode verifier), the precondition for executing a given instruction is met in the abstract state witnessed by `collect-sig-frame-vector`.

We use induction to prove the reduction theorem. The natural choice appears to be inducting according to the recursive pattern of `mergedcodeIsTypesafe` or `bcv-simple-method1`. However, we observe that the induction patterns of `mergedcodeIsTypesafe` and `bcv-simple-method1`

```
(encapsulate ()
  (local (include-book "bcv-instructionIsTypeSafe-if-verified"))
  (defthm mergedcodeIsTypesafe-implies-instructionIsTypeSafe
    (implies (and (mergedcodeIsTypesafe env mergedcode
                                        stackframe)
                  (pc-wff-mergedcode1 0 (allinstructions env))
                  (is-suffix mergedcode (allinstructions env))
                  (member inst (extract-code mergedcode)))
             (instructionIsTypeSafe
                 inst
                 env
              (searchStackFrame
                   (instrOffset inst)
                   (stack-map-wrap
                   (collect-sig-frame-vector env
                                             mergedcode
                                             stackframe)))))))
```

Figure 7.29: Instruction safe to execute in recorded abstract state

do not match with each other — the CLDC bytecode verifier inducts on the `mergedCode`, while the `bcv-simple-method1` inducts on the actual sequence of instructions from the `mergedCode`. Between any two real instructions in the `mergedCode`, there can be an unknown number of stackmaps all associated with one specific program counter. A CLDC bytecode verifier `mergedcodeIsTypesafe` may have to execute unknown number of steps to reach the next instruction to check for the next `instructionIsTypeSafe`, while the simple bytecode verifier always takes one step.

We define the induction scheme `mergedCodeIsTypeSafe-induct` (figure 7.30).

We prove the following lower level lemma to relate two `mergedCodeIs-TypeSafe` terms that appear in the induction scheme. From the fact that

```
(mergedcodeIsTypesafe
     env mergedcode
     (mapFrame (getMap mergedcode1)))
```

we conclude that one can "skip ahead" to obtain the fact that

```
(mergedcodeIsTypesafe
     env
     (forward-to-next-inst (cons mergedcode1 mergedcode))
     (next-stackframe (cons mergedcode1 mergedcode)))
```

We also need to prove how the `next-stackframe` relates to the recorded stack maps at the next program counter — because the `bcv-simple-method1` (figure 7.25, page 254) is expressed as first *looking up* the recorded stack map at a program counter, then checking whether the next instruction is safe to

```
(defun mergedcodeIsTypeSafe-induct
       (env init-frame mergedcode stackmap)
  (if (endp mergedcode)
      (list env init-frame mergedcode stackmap)
    (if (endp (cdr mergedcode))
        (list (list env init-frame mergedcode stackmap))
      (cond
        ((isinstruction (car mergedcode))
         (cond
           ((isStackMap (cadr mergedcode))
            (mergedcodeIsTypeSafe-induct
             env init-frame
             (forward-to-next-inst (cdr mergedcode))
             (next-stackframe (cdr mergedcode))))
           ((isInstruction (cadr mergedcode))
            (mv-let
             (next-stack-frame exception-frame)
             (sig-do-inst (car mergedcode) env stackmap)
             (declare (ignore exception-frame))
             (mergedcodeIsTypeSafe-induct env init-frame
                                          (cdr mergedcode)
                                          next-stack-frame)))))
        ((isStackMap (car mergedcode))
         (mergedcodeIsTypeSafe-induct
              env init-frame
              (forward-to-next-inst mergedcode)
              (next-stackframe mergedcode)))))))
```

Figure 7.30: Induction scheme for proving reduction theorem

```
(defthm mergedcodeIsTypesafe-forward-to-next-inst-b
  (implies (and (mergedcodeIsTypesafe
                     env mergedcode
                    (mapFrame (getMap mergedcode1)))
                (isStackMap mergedcode1)
                (pc-wff-mergedcode1 (next-pc mergedcode)
                                    mergedcode))
           (mergedcodeIsTypesafe
                env
                (forward-to-next-inst
                          (cons mergedcode1 mergedcode))
                (next-stackframe
                          (cons mergedcode1 mergedcode)))))
```

Figure 7.31: Adapting the induction pattern of `mergedcodeIsTypesafe`

execute and whether the resulting state is compatible with the recorded stack map.

Different from `mergedCodeIsTypeSafe`, the `bcv-simple-method1` immediately checks the post safety conditions after symbolically executing one instruction (figure 7.25).

```
(frameIsAssignable
      nextStackFrame
      (searchStackFrame
           (instrOffset (cadr code))
           (stack-map-wrap stack-maps))
       env)
```

The `mergedCodeIsTypeSafe` does not check such a requirement (figure 5.7) directly. Instead, the symbolic execution engine (as encoded in `mergedCodeIsTypeSafe`) will take the `nextStackFrame` from state resulting

```
(encapsulate ()
  (local (include-book "bcv-searchStackFrame-reduce"))
  (defthm searchStackFrame-is-if-stack-map
    (implies (and (isStackMap (car mergedcode))
                  (equal (forward-to-next-inst mergedcode)
                         (forward-to-next-inst x))
                  (is-suffix mergedcode all-merged-code)
                  (pc-wff-mergedcode1 pc all-merged-code)
                  (mergedcodeIsTypeSafe env
                                        all-merged-code
                                        init-frame))
             (equal
              (searchStackFrame
               (instrOffset    (car (forward-to-next-inst x)))
               (stack-map-wrap (collect-sig-frame-vector
                                env
                                all-merged-code
                                init-frame)))
              (next-stackframe mergedcode)))))
```

Figure 7.32: Rewriting searchStackFrame: I

from `sig-do-inst` as its current state and continue. This safety post-condition is indirectly enforced by `mergedCodeIsTypeSafe` when a stack frame is encountered in the `mergedcode`. `mergedCodeIsTypeSafe` checks that the current abstract state is compatible with the next stack frame that symbolic execution will encounter.

```
  (cond
   ((isStackMap cur)
    (let ((MapFrame (mapFrame (getMap cur))))
     (and (frameIsAssignable StackFrame MapFrame Environment)
          ...))))
```

We prove the following theorems to relate the post-condition checks by the `bcv-simple-method1` with the checks done by the `mergedcode-IsTypeSafe`.

This theorem (figure 7.33) asserts that if the next item in the `mergecode` is an instruction — `(isInstruction (next-inst inst mergecode))`[11], then the recorded stack map at the next program counter is just the result of executing the current instruction —

```
(equal (car (sig-do-inst ...))
       (searchStackFrame ...))
```

.

With this lemma, we know the additional postcondition check by the `bcv-simple-method1` is degenerate, because the recorded state at the next

[11] `(next-inst inst mergecode)` should have been named as `next-item`; it returns the item that appears right after `inst` in the list `mergecode`.

```
(encapsulate ()
  (local (include-book "bcv-searchStackFrame-reduce"))
   (defthm |mergecodeIsTypeSafe-
                    implies-collect-sig-vector-compatible-1|
    (implies
       (and (mergedcodeIsTypeSafe env mergecode stackframe)
            (member inst mergecode)
            (isInstruction inst)
            (isInstruction (next-inst inst mergecode))
            (pc-wff-mergedcode1 (next-pc mergecode) mergecode))
       (equal (car (sig-do-inst
                      inst
                      env
                      (searchStackFrame
                           (instroffset inst)
                           (stack-map-wrap
                            (collect-sig-frame-vector
                                 env mergecode
                                 stackframe)))))
              (searchStackFrame
                   (instroffset (next-inst inst mergecode))
                   (stack-map-wrap (collect-sig-frame-vector
                                     env mergecode
                                     stackframe)))))))
```

Figure 7.33: Rewriting searchStackFrame: II

program counter is the state resulting from executing the current instruction itself. And it is easy to prove

```
(frameIsAssignable  nextStackFrame
                    nextStackFrame
                    env)
```

.

Consequently, we can conclude that it is fine for the `mergecodeIs-TypeSafe` to skip this degenerate of check in this context — when the next item in the list is an instruction.

When the next item in the list is a stack map, (instead of an instruction), we prove the `frameIsAssignable-transitive-specific` lemma to assert that the postcondition check done by the `bcv-simple-method1` is not skipped and is enforced by `mergecodeIsTypeSafe`, although indirectly. The lemma is proved in `BCV/bcv-sig-do-produce-compatible-next-state.lisp` [22]. The form of lemma is complicated (71 lines with 3775 characters) and is omitted here.

With our induction scheme and supporting lemmas, we have been able to prove the reduction theorem (figure 7.26).

7.3.5 Summary

We introduced the alternative version of the bytecode verifier to separate the procedural aspects of the CLDC bytecode verifier specification from the essential aspects of an effective bytecode verifier.

Defining the alternative bytecode verifier is one part of our overall strategy for proving the JVM is safe for executing verified programs. The overall strategy has been used for proving that a simpler machine — `Small` — does not overflow its operand stacks, when the program being executed has been verified by a CLDC-like static checker.

Following the strategy, we need to prove a `reduction theorem`. The reduction theorem asserts that all programs verified by the procedural CLDC bytecode verifier can also be accepted by the simpler bytecode verifier. We completed the proof of this theorem.

7.4 Leaf-Level Lemmas and Their Proofs

One key to prove that M6 is safe while executing verified programs is to relate the success of checks done by the bytecode verifier to the safe execution of M6.

We defined a defensive JVM — DJVM — to make explicit what we mean by *safe execution of M6* (see Chapter 6). A safe execution shall not get *stuck* because some runtime check conducted by the defensive JVM is violated and the next state is not well defined.

We defined the alternative bytecode verifier to expose the essential checks conducted during the bytecode verification, so that we can refer to what is checked at a specific program location directly. We proved the reduction theorem, which asserts that the original CLDC bytecode verifier at least enforces these checks — all programs verified with the CLDC bytecode verifier will be verified with the alternative bytecode verifier.

The remaining task is to prove — for each instruction:

Passing the safety checks on an abstract state will ensure that the defensive checks will succeed on some concrete DJVM state. Obviously, arbitrary safety checks of the bytecode verifier cannot be used to predict the outcome of unrelated, defensive checks. We identify the `consistent-state` that asserts that (1) the state representation is *internally coherent* that they present a sensible configuration of the machine state, and (2) the state is *externally consistent* — on-track — with the abstract state observed by the bytecode verifier.

We need to prove two broad kinds of lemmas for each instruction.

- Assuming that a concrete state is internally coherent and externally consistent, the success of the bytecode verifier checks imply the success of the defensive checks on the concrete state.
- Executing a step from the concrete state, the resulting concrete state remains both internally coherent and externally consistent (with the next corresponding abstract state).

We call these kinds of lemmas for each instruction as *leaf-level lemmas*. In this section, we describe a sufficient set of leaf-level lemmas that we have identified. [12] We also describe our supporting ACL2 library for proving specific types of leaf-level lemmas from this set.

We have used our supporting library to prove the leaf lemmas for a limited subset of instructions including `AALOAD`, `IFEQ`, and `GETFIELD`.

7.4.1 Leaf-level Lemmas

DJVM is safe

Because DJVM is a very complex program in itself, one may want to have more direct evidence that DJVM itself is in fact safe besides looking at the

[12]Since we have not proved the final *M6 is safe* theorem, the *sufficiency* of our set of leaf-level lemmas is only speculative.

definition of DJVM itself.

We identified the following leaf-level lemmas to prove:

- when DJVM is not `stuck`, executing each instruction preserves the `consistent-state` property. [13]

- the `execute-XXX` style operations are guard verified. [14]

Taking `AALOAD` instruction as an example, we identified and proved the following lemmas that assert the DJVM's `execute-AALOAD` operation is safe.

- `execute-AALOAD` is guard verified (figure 7.34)

- `execute-AALOAD` preserves the `consistent-state` when executed with its guard met.

```
(defthm |AALOAD-guard-implies-
                         execute-AALOAD-perserve-consistency|
  (implies (AALOAD-guard inst s)
           (consistent-state-strong
                                 (execute-AALOAD inst s))))
```

- In a consistent state, if the DJVM's runtime checking `check-AALOAD` succeeds, the preconditions for executing `AALOAD`, `AALOAD-guard`, will be met.

[13]Our current definition of `consistent-state-strong` only captures the consistency requirement for its internal components. It does not assert the external consistency requirement that the execution state is on track with with some abstract state seen by the bytecode verifier. The latter requirement is captured in the definition `consistent-state-bcv-on-track` (figure 6.21). We prove that DJVM operations also preserve the `consistent-state-bcv-on-track` with a separate leaf-level lemma.

[14]There is no formal statement in form of a theorem that asserts that an operation is guard verified. But we do need to prove a set of supporting lemmas that ACL2 can use to prove that invoking the top level operation in a state that satisfies its guard will result in the no guard of a lower level operation being violated.

```
     (defthm check-AALOAD-implies-guard-succeeds
       (implies (and (consistent-state-strong s)
                     (check-AALOAD inst s))
                (AALOAD-guard inst s)))
```

M6 behaves the same as the DJVM

After proving that DJVM's step is safe when its runtime check `check-AALOAD` succeeds, we prove the following to assert that if DJVM is taking a step without violating its guard, M6 behaves the same as DJVM.

```
(encapsulate ()
   (local (include-book "base-state-equiv"))
    (defthm equal-AALOAD-when-guard-succeeds
      (implies (and (AALOAD-guard inst djvm::djvm-s)
                    (state-equiv m6::m6-s djvm::djvm-s))
               (state-equiv (m6::execute-AALOAD inst m6::m6-s)
                            (djvm::execute-AALOAD
                                          inst
                                          djvm::djvm-s)))))
```

Bytecode verifier checks implies the DJVM checks

We prove the following leaf-level lemma to relate the checks done by the bytecode verifier to the runtime checks done by the DJVM.

- If the bytecode verifier asserts that it is safe to execute `AALOAD` in the type signature state of the current state, the DJVM's runtime checking `check-AALOAD` always succeeds.

```
(defun AALOAD-guard (inst s)
  (mylet*
   ((index (safe-topStack s))
    (array-ref (safe-secondStack s)))
   (and (consistent-state-strong s)
        (wff-inst inst)
        (topStack-guard-strong s)
        (secondStack-guard-strong s)
        (wff-REFp array-ref)
        (INT32p (value-of index))
        (<= (len (operand-stack (current-frame s)))
            (max-stack s))
        (or (CHECK-NULL array-ref)
            (and (CHECK-ARRAY-guard (rREF array-ref) (heap s))
                 (not (primitive-type?
                       (array-component-type
                        (obj-type (binding
                                   (rREF array-ref)
                                   (heap s)))))))))))

(defun execute-AALOAD (inst s)
  (declare (xargs :guard (AALOAD-guard inst s)))
  (let* ((index (safe-topStack s))
         (array-ref (safe-secondStack s)))
    (if (CHECK-NULL array-ref)
        (raise-exception "java.lang.NullPointerException" s)
      (if (check-array (rREF array-ref) (value-of index) s)
          (ADVANCE-PC
             (safe-pushStack
                 (tag-REF (element-at-array (value-of index)
                                            (rREF array-ref)
                                            s)
                 (safe-popStack (safe-popStack s)))))
        (raise-exception
             "java.lang.ArrayIndexOutOfBoundsException" s)))))
```

Figure 7.34: Guard verified: execute-AALOAD

```
(encapsulate ()
  (local (include-book "base-bcv"))
  (defthm bcv-check-aaload-ensures-djvm-check-aaload
    (implies (and (bcv::check-AALOAD
                            inst
                            (env-sig s)
                            (frame-sig
                               (current-frame s)
                               (instance-class-table s)
                               (heap s)
                               (heap-init-map (aux s))))
                  (wff-inst inst)
                  (not (mem '*native*
                            (method-accessflags
                                   (current-method s))))
                  (consistent-state s))
             (djvm::check-AALOAD inst s))))
```

- If it is safe for the bytecode verifier to execute `AALOAD` in some type signature state, it is safe to execute `AALOAD` in a more specific type signature state provided both type signature states are *good* with respect to some *consistent* type hierarchy `good-icl`.

```
(encapsulate ()
    (local (include-book "base-bcv-check-monotonic"))
    (defthm |sig-check-AALOAD-on-more-general-implies-
                                              more-specific|
      (implies (and
                 (good-icl icl)
                 (good-scl (classtableEnvironment env1))
                 (sig-frame-more-general gframe
                                         sframe env1)
                 (consistent-sig-stack (frameStack gframe)
                                       icl)
                 (consistent-sig-stack (frameStack sframe)
                                       icl)
                 (not (equal (nth1OperandStackIs 2 sframe)
                             'NULL))
                 (not (equal (nth1OperandStackIs 2 gframe)
                             'NULL))
                 (bcv::check-AALOAD inst env1 gframe)
                 (icl-scl-compatible
                             icl
                             (classtableEnvironment
                                                  env1)))
               (bcv::check-AALOAD inst env1 sframe))))
```

With these two types of leaf-level lemmas, together with the *on-track* requirement asserted in a *consistent-state*, we can conclude that if the bytecode verifier's check succeeds in the more general abstract state seen by the bytecode verifier, the corresponding runtime checks will succeed in the concrete state. This is because the on-track requirement will provide us the needed hypothesis that the type signature state of the concrete state is no more general than the type signature state observed by the bytecode verifier.

DJVM execution remains on track

Our existing definition of `consistent-state-strong` captures the internal consistency requirement for a DJVM state to represent a sensible machine state.

Another important requirement for a useful *consistent* state is that the executions from such a state need to remain on track with the abstract execution of the bytecode verifier. As explained in the previous section, we need the *on-track* property to relate the checks done by the bytecode verifier to the runtime check done by the DJVM.

To prove that this property is preserved over the DJVM execution of a verified program, we identified the following leaf-level lemma that one needs to prove about every instruction. We state the lemma for the case of the `AALOAD` instruction.

- If the DJVM is safe to execute `AALOAD` and the starting state is a `consistent-state`, the type signature of the state obtained by executing the `AALOAD` instruction is no more general than the state resulting from executing one step of the bytecode verifier from the type signature state of the starting state.

```
(encapsulate ()
  (local
   (include-book
          "base-next-state-more-specific"))
  (defthm next-state-no-more-general-aaload
    (mylet* ((oframe (frame-sig
                           (current-frame s)
                           (instance-class-table s)
                           (heap s)
                           (heap-init-map (aux s))))
             (ns   (djvm::execute-aaload inst s))
             (nframe (frame-sig
                           (current-frame ns)
                           (instance-class-table ns)
                           (heap ns)
                           (heap-init-map (aux ns)))))
            (implies (and (consistent-state s)
                          (bcv::check-aaload
                                      inst
                                      (env-sig s) oframe)
                          (djvm::check-aaload inst s)
                          (not (check-null
                                  (topStack (popStack s))))
                          (check-array
                             (RREF (topStack (popStack s)))
                             (value-of (topStack s)) s))
                     (bcv::sig-frame-more-general
                      (car (bcv::execute-aaload
                                         inst
                                         (env-sig s)
                                         oframe))
                      nframe
                      (env-sig s))))
```

- The execution of the bytecode verifier is monotonic. If two frames are in the `sig-frame-more-general` relation (and both (and both are suitable for executing an `AALOAD`), then the frames produced by executing `AALOAD` on each of the two frames are in the same relation.

```
(encapsulate ()
  (local (include-book "base-bcv-step-monotonic"))
  (defthm AALOAD-monotonicity
    (implies
      (and (sig-frame-more-general gframe
                                   sframe env1)
           (consistent-sig-stack (frameStack sframe)
                                 icl)
           (consistent-sig-stack (frameStack gframe)
                                 icl)
           (not (equal (nth1OperandStackIs 2 gframe)
                       'NULL))
           (not (equal (nth1OperandStackIs 2 sframe)
                       'NULL))
           (bcv::check-AALOAD inst env1 gframe)
           (bcv::check-AALOAD inst env1 sframe)
           (good-icl icl)
           (good-scl (classtableEnvironment env1))
           (icl-scl-compatible
                     icl
                     (classtableEnvironment env1)))
          (sig-frame-more-general
             (normal-frame (bcv::execute-AALOAD
                                             inst env gframe))
             (normal-frame (bcv::execute-AALOAD
                                             inst env sframe))
             env1))))
```

7.4.2 Libraries For Proving Leaf-level Lemmas

We have developed a set of *ACL2 books*. Each ACL2 book is a collection of theorems that can be accepted by the ACL2 theorem prover. More importantly, each book configures the ACL2 theorem prover with specific set of "rules" for proving the specific type of leaf lemma.

Methodology

We organized the leaf-level lemmas about one JVM instruction into one specific file in `DJVM/INST` directory [22]. For example, all leaf-lemmas about the `GETFIELD` instruction appear in `DJVM/INST/GETFIELD.lisp`.

A `GETFIELD.lisp`-style file can be viewed as having four sections. Each section corresponds to a specific type of the leaf-level lemmas. *DJVM is safe*, *M6 behaves the same as the DJVM*, *Bytecode verifier checks imply the DJVM checks when on track*, and *DJVM execution remains on track.* In each section, we import specific ACL2 books to configure the ACL2 theorem prover for proving the leaf-level lemmas in that section.

In the process of proving a leaf-level lemma, we may identify supporting lemmas. If the supporting lemma is consider generally useful for proving this type of leaf-level lemma for other instructions, we move the supporting lemma into the books that we have included in this section.

We build and expand our supporting books in this fashion. The process of proving leaf-level lemmas for `AALOAD`, `AASTORE`, `ANEWARRAY`, `ALOAD`, `ASTORE`, `IFEQ`, and `GETFIELD`, has guided us to build a set of useful supporting books.

These supporting ACL2 books are relatively complete and robust in helping us proving leaf-level lemmas about new instructions. We expect that one can follow our methodology and build on this set of books to eventually prove all the leaf-level theorems for all M6 instructions.

The known limitation of the existing library include that:

- We have *skip-proved* the necessary lemma that reason about raising and handling an exception. That means these lemmas have been assumed rather than proved.
- We have not formulated and proved the lemma necessary for reasoning about operations that updating the call stack in the implementation of the `INVOKEVIRTUAL` operation.
- We have not defined verifiable guards for some low level operations.
- Some other lemmas in the supporting library have been "skip-proved" and are yet to be proved by ACL2.

DJVM is safe

For each instruction, we always first load a fixed set of books

```
(defthm pc-pushStack-unchanged
  (equal (pc (pushStack v s))
         (pc s)))
```

Figure 7.35: Selected lemma: `pc-pushStack-unchanged`

```
(include-book "base")

(include-book "base-consistent-state")

(include-book "base-extra")
```

These ACL2 books *export* useful lemmas/rules for proving that DJVM is safe.

Exported properties can be as simple as asserting that pushing a value on the current operand stack `(pushStack v s)` does not change the program counter value (figure 7.35).

Among the more interesting lemmas exported by these books, the following theorem `REFp-not-NULLp-consistent-object-alternative` (figure 7.36) asserts that if a value `v` is `REFp` with respect to `(heap s)` and `v` is not a NULL value, and `s` is a `consistent-state`, then the object pointed to by `v` — `(deref2 v (heap s))` — is a `consistent-object` with respect to `(heap s)` and a `cl`, as long as the `cl` is equal to `(instance-class-table s)`.

These books also contain proofs to guard-verify all the common operations that are used to define `execute-XXX` operations. For example, because the operation `execute-GETFIELD` may invoke the class loading operation `resolveClassReference`, these ACL2 books contain the proofs that help to guard-verify the `resolveClassReference` operation.

```
(defthm REFp-not-NULLp-consistent-object-alternative
  (implies (and (consistent-state s)
                (REFp v (heap s))
                (not (check-NULL v))
                (equal (instance-class-table s) cl))
           (consistent-object (deref2 v (heap s))
                              (heap s)
                              cl))))
```

Figure 7.36: Non-`null` pointer points to `consistent-object`

```
(defun m6-getfield (classname fieldname obj-ref s)
  (declare (xargs :guard
           (field-access-guard classname fieldname obj-ref s)))
  (binding fieldname
          (binding classname
           (java-visible-portion (deref obj-ref (heap s))))))
```

Figure 7.37: Operation: `m6-getfield`

For some more complicated instructions, such `GETFIELD`, we also include ACL2 books specific for proving properties about them. We include two extra books to prove that the *DJVM is safe* style lemma for `GETFIELD`: `base-consistent-state-load-class` and `base-consistent-state-lookup-field`

One of the operations that `execute-GETFIELD` invokes is `m6-getfield`.

Operation `m6-getfield` defines a guard `field-access-guard` (figure 7.38), the core of which is the assertion that some particular field exists in the object that we are trying to access — `jvp-access-field-guard` (figure 7.39).

One key lemma that we have proved for guard-verifying the operation `execute-GETFIELD` `base-consistent-state-lookupfield` is presented

```
(defun field-access-guard (classname fieldname obj-ref s)
  (and (wff-state s)
       (wff-heap (heap s))
       (bound? obj-ref (heap s))
       (wff-obj (binding obj-ref (heap s)))
       (jvp-access-field-guard
            classname fieldname
            (java-visible-portion (binding obj-ref (heap s))))))
```

Figure 7.38: Guard for invoking `field-access-guard`

```
(defun jvp-access-field-guard (classname fieldname obj)
  (and (alistp obj)
       (bound? classname obj)
       (alistp (binding classname obj))
       (bound? fieldname (binding classname obj))))
```

Figure 7.39: Guard for accessing Java visible portion

in figure 7.40.

The theorem says that if the object `obj` that we used to access the field is a subclass of the class `(fieldCP-classname fieldCP)`, and the field resolution operation `lookupField` confirms that there is such a field, then the guard for accessing the *java visible portion* of the object will be met.

Another interesting lemma (figure 7.41) asserts if that we already have an object that is *assignable to* of `typ2`, loading the class `typ2` produces no effect when the starting state `s` is a `consistent-state`.

We observe that the existing supporting library for proving *DJVM is safe* style leaf-level is relatively complete. Although we have only proved the *DJVM is safe* style lemma for a limited number of JVM instructions[15], we

[15] `AALOAD`, `AASTORE`, `ANEWARRAY`, `ALOAD`, `ASTORE`, `IFEQ`, and `GETFIELD`

```
(defthm  |consistent-object-and-field-found-in-lookup
                           -implies-jvm-field-access-guard|
  (implies (and (consistent-state s)
                (consistent-object obj
                                  (heap s)
                                  (instance-class-table s))
                (car (isAssignableTo
                                  (obj-type obj)
                                  (fieldCP-classname fieldcp)
                                  s))
                (lookupField (fieldcp-to-field-ptr fieldCP) s))
           (jvm::jvp-access-field-guard
                 (field-classname
                       (lookupField (fieldcp-to-field-ptr
                                                 fieldCP) s))
                       (fieldcp-fieldname fieldcp)
                       (java-visible-portion obj))))
```

Figure 7.40: Safe to access fields from superclasses

```
(defthm |resolveClassReference-no-change-
                          if-already-loaded-if-not-array-Object|
  (implies (and (consistent-object obj (heap s)
                                   (instance-class-table s))
                (case-split (not (isArrayType (obj-type obj))))
                (car (isAssignableTo (obj-type obj) typ2 s))
                (consistent-state s))
         (equal (resolveClassReference typ2 s) s)))
```

Figure 7.41: Loading superclass of existing objects produces no effect

expect that one does not need to add significantly more new lemmas to proving this type of lemmas.

Consider the `INVOKEVIRTUAL` instruction, for which, we have not yet proved the *DJVM is safe* style lemma. The `INVOKEVIRTUAL` operation will first lookup the method, it then will pop some values of the operand stack, create a new frame and initialize the values in the new call frame. Among these low level operations, only frame creation is new. In proving the leaf-level lemma for `AALOAD`, we proved that popping an operand off the stack will preserve the *consistent state*. In proving the leaf lemma for `ASTORE`, we proved lemmas for reasoning about operations that write the local variables in a frame. The same lemmas can be used for reasoning about initializing the new call frame. The method resolution operation invoked by `INVOKEVIRTUAL` is very similar to the field resolution operation invoked by `GETFIELD`, which has been verified. It is reasonable to expect our *load superclass produces no effect* lemma (figure 7.41) to be directly useful. For guard-verifying the method resolution operation, we can benefit from a similar one for guard-verifying field access (figure 7.40).

M6 behaves the same as DJVM

Our DJVM model maintains additional type information for the values in the operand stack and locals. Before DJVM executes an instruction with `execute-XXX` style operations, DJVM also checks the preconditions (that are explicitly stated in the official JVM specification for the instruction) by executing `check-XXX` style operations — if such checks return false, the bytecode interpreter keeps DJVM in the same (stuck) state.

Our regular JVM model, M6, executes like DJVM with two exceptions. M6 does not maintain type information for values stored on the operand stack

```
(encapsulate ()
   (local (include-book "base-state-equiv"))
   (defthm equal-GETFIELD-when-guard-succeeds
      (implies (and (state-equiv M6::m6-s DJVM::djvm-s)
                    (GETFIELD-guard inst DJVM::djvm-s))
               (state-equiv (m6::execute-GETFIELD inst M6::m6-s)
                            (djvm::execute-GETFIELD
                                             inst
                                             DJVM::djvm-s)))
      :hints (("Goal" :do-not '(generalize fertilize)))))
```

Figure 7.42: M6 and DJVM behaves the same for executing `GETFIELD`

and local variable arrays. M6 does not check the precondition listed in the JVM specification before carrying out the operation described in the JVM specification.

While DJVM may stick when executing a program, M6 will continue — violating the preconditions – and may invoke an operation with inputs that do not meet its guards.

Our supporting library `DJVM/INST/base-state-equiv.lisp` contains effective supporting lemmas for proving that *M6 behaves the same as the DJVM* — when (1) the DJVM guard for the instruction is met and (2) the initial M6 state and the DJVM are essentially the *same* state.

For `GETFIELD`, we proved the leaf-level lemma `equal-GETFIELD-when--guard-succeeds` (figure 7.42) in `DJVM/INST/GETFIELD.lisp`

Combining this *M6 behaves the same as DJVM* type result with the *DJVM is safe* type result for `GETFIELD` (figure 7.43) — which asserts that in a consistent state, if `check-GETFIELD` succeeds, the guard for executing `GETFIELD` will succeed — we can conclude that when the precondition as specified in the JVM specification is met, executing `GETFIELD` on M6 and the `DJVM`

```
(defthm check-GETFIELD-implies-guard-succeeds
  (implies (and (consistent-state-strong s)
                (check-GETFIELD inst s))
           (GETFIELD-guard inst s)))
```

Figure 7.43: `Check-GETFIELD` Implies `GETFIELD-guard`

produce the “same” resulting states, if the initial states are the “same” and the DJVM state has is a *consistent-state*.

The supporting lemmas for proving lemmas such as `equal-GETFIELD-when--guard-succeeds` are simple and their proofs straightforward. We have chosen to “skip-proof” many simple support lemmas of this type in `DJVM/INST/base--state-equiv.lisp`.

Bytecode verifier checks implies the DJVM checks

The key for proving this type of leaf-level lemma is to make good use of the property that that a *consistent* state is approximated by some abstract state observed by the bytecode verifier.

We identify four *bytecode verifier checks implies the DJVM checks* type leaf-level lemmas.

- (1) If the bytecode verifier can safely execute the current instruction with respect to the *current type signature state*, the DJVM can safely execute the instruction.

- (2) If the bytecode verifier state can safely execute an instruction in a more general type signature state, the bytecode verifier can safely execute the instruction in a more specific signature state

- (3) If the current state is on-track with some corresponding abstract state seen by the bytecode verifier, then its type signature state is no more general than the abstract state observed by the bytecode verifier.

- (4) If the program is verified, for each instruction in that method, there exists an abstract state such that the bytecode verifier can safely execute that instruction in that abstract state.

We observe that only the first two of these are specific to individual JVM instructions. Lemma (3) depends on how we define the *on-track* requirement for a *consistent state*. Lemma (4) depends on how the bytecode verifier is formalized. Consider the fact that we have successfully reduced the procedural CLDC bytecode verifier into a much simpler bytecode verifier; then it is easy to show that lemma (4) holds.

To state lemma (1), we defined an operation `frame-sig` that extracts the current type signature state from a concrete DJVM state.

The `frame-sig` takes a call frame from the DJVM state and returns a type signature state that a bytecode verifier can use to check the whether it is safe to execute an instruction. One operation that `frame-sig` uses is `value-sig`. The operation `value-sig` (figure 7.44) takes a tagged value and other parameters corresponding to the context and returns the type of the value.

The definition of `value-sig` is complicated. When the reference value is a `NULLp` pointer, the return type is `null`. When the object pointed to by the reference value is already initialized — one of the constructors has been executed with the object as the input — the type returned by `value-sig` is the actual type of the object. When the object pointed to by the reference

```
(defun value-sig (v cl hp hp-init curMethodPtr)
 (declare (xargs :guard ....))
   (if (REFp v hp)
      (if (NULLp v)
          'null
        (let ((obj-init-tag (deref2-init v hp-init))
              (obj (deref2 v hp)))
          (if (not (consp obj-init-tag))
              ; if initialized, we return the actual type
              ; of the object pointed by the reference value
              (fix-sig (obj-type obj))

            (if (equal (cdr obj-init-tag) curMethodPtr)
                ; if the object is created in this method
                ; then translate into an uninitialized(Offset)
                (cons 'uninitialized (car obj-init-tag))

              ; else translate it into a 'uninitializedThis
              'uninitializedThis))))

              ; The following is an invariant: at any point
              ; of program execution, at the give frame,
              ; there is at most one uninitialized object
              ; not created from this frame.
              ;
              ; We have asserted this in the
              ;                      consistent-state-strong
     (djvm-translate-int-type (tag-of v))))
```

Figure 7.44: Extracting type: `value-sig`

```
(defthm isArrayType-bcv-isArrayType
  (implies (and (ISARRAYTYPE (OBJ-TYPE (DEREF2 v hp)))
                (not (deref2-init v hp-init))
                (REFp v hp)
                (not (NULLp v)))
           (bcv::isArrayType
                (value-sig v cl hp hp-init method-ptr))))
```

Figure 7.45: Relating `isArrayType` checks

value is not yet initialized, depending on whether the object is created in the current frame, the type returned can either be `'(uninitialized <Offset>` or `'(uninitializedThis)`.

We proved lemmas that relate the checks done by DJVM to the checks done by the bytecode verifier on the extracted type signature state, including the simple lemma `isArrayType-bcv-isArrayType` (figure 7.45) and the much more complicated lemma `bcv-isAssignable-value-sig-djvm--isAssignableTo` (figure 7.46).

Supporting lemmas are collected into the ACL2 book `base-bcv` (`DJVM-/INST/base-bcv.lisp` from [22]). We have used the book to prove that the `bcv-check-GETFIELD-ensures-djvm-check-GETFIELD` style leaf lemmas for the subset of DJVM instructions that we have defined — `AALOAD`, `AASTORE`, `ALOAD`, `ASTORE`, `ANEWARRAY`, `IFEQ`, and `GETFIELD`.

To prove lemma (2) of the *Bytecode verifier checks implies the DJVM checks* type leaf level lemmas, we have created the `base-bcv-check-monotonic` book (`DJVM/INST/base-bcv-check-monotonic.lisp` from [22]). The book contains lemma like `TypeListAssignable-isMatchType-prefix-class` (figure 7.48) which asserts if that a list of types `sl` is `TypeListAssignable` to a list `gl` and if we can pop a value of type `(prefix-class any)` from the list

```
(defthm bcv-isAssignable-value-sig-djvm-isAssignableTo
  (implies
   (and (isAssignable
         (value-sig v
                    (instance-class-table s)
                    (heap s)
                    (heap-init-map (aux s))
                    (method-ptr (current-frame s)))
         (prefix-class typ)
         (env-sig s))
        (not (NULLp v))
        (isClassTerm (class-by-name typ
                                    (instance-class-table s)))
        (no-fatal-error? s)
        (consistent-value-x v (instance-class-table s) (heap s))
        (consistent-state s)
        (class-by-name typ
                       (classtableEnvironment (env-sig s)))
        (not (classIsInterface
              (class-by-name
                  typ
                  (classtableEnvironment (env-sig s))))))
   (car (djvm::isAssignableTo
                   (obj-type (deref2 v (heap s)))
                   typ s))))
```

Figure 7.46: Relating the BCV's `IsAssignable` to DJVM's `isAssignableTo`

```
(encapsulate ()
 (local (include-book "base-bcv"))
 (local (include-book "base-bcv-djvm-getfield"))
 (defthm bcv-check-GETFIELD-ensures-djvm-check-GETFIELD
   (implies (and (bcv::check-GETFIELD
                             inst
                             (env-sig s)
                             (frame-sig
                                (current-frame s)
                                (instance-class-table s)
                                (heap s)
                                (heap-init-map (aux s))))
                 (wff-getfield inst)
                 (wff-fieldCP (arg inst))
                 (no-fatal-error? s)
                 (lookupField
                     (fieldcp-to-field-ptr (arg inst)) s)
                 ; need to assert that field is found!!
                 ; otherwise this is not true!!
                 (not (mem '*native*
                           (method-accessflags
                                 (current-method s))))
                 (consistent-state s))
            (djvm::check-GETFIELD inst s))))
```

Figure 7.47: Relating check-GETFIELD results

```
(defthm TypeListAssignable-isMatchType-prefix-class
  (implies (and ...
                (consistent-sig-stack sl icl)
                (TypeListAssignable sL gL env)
                (consistent-sig-stack gl icl)
                (good-bcv-type (prefix-class any) icl)
                (good-icl icl)
                (good-scl (classtableEnvironment env))
                (icl-scl-compatible icl
                                    (classtableEnvironment env))
                (consp gL)
                (isMatchingType (prefix-class any) gL env))
           (isMatchingType (prefix-class any) sL env)))
```

Figure 7.48: Relating `IsMatchingType` checks on two type signature states

`gl`, we can pop a value of the same type from the list `sl` — with additional side conditions that the relevant state is consistent.

Our `base-bcv-check-monotonic` book is complete. We do not expect that one needs distinctively new lemmas for proving that when bytecode verifier checks succeed on more general abstract states then the checks will succeed on more specific states.

DJVM execution remains on-track

We also need to prove leaf-level lemmas that assert that *DJVM execution remains on-track*. For each instruction, we need to prove two leaf-level lemmas: first, executing DJVM for one step and then extracting the type signature state produces a state is no more general than the type signature state obtained by first extracting the type signature state and executing the bytecode verifier for one step — *BCV next state is more general than DJVM next state.* Second, starting from a more general type signature state and executing the bytecode verifier results in a state still more general — *BCV step is monotonic*.

These two leaf-level lemmas, together with generic lemmas (3) and (4) from the previous section, are sufficient to prove that DJVM executions will remain on track.

We have created suitable ACL2 books for proving these two types of leaf-level lemmas.

The `base-bcv-step-monotonic` book helps to prove the latter — starting from a more general abstract state, executing one bytecode verifier step will reach a more general ending state. For a concrete example, see lemma `AALOAD-monotonicity` on page 281.

We created the `base-next-state-more-specific`, `base-bcv-frame--sig-expansion` and `base-frame-sig-expansion` books to prove the former: executing a DJVM step and then extracting the abstract state produces a more specific abstract state than extracting the abstract state then executing a bytecode verifier step.

We proved useful rules such as `bcv-typelistAssignable-modify--local-variable-slot` (figure 7.49) in the `base-bcv-check-monotonic` book.

In the book `base-frame-sig-expansion`, we proved rules as shown in figure 7.50 to reduce the effect of executing the DJVM operation `update-nth` on the resulting type signature state to the effect of executing a bytecode verifier operation on the extracted type signature state of the starting state.

We produced a substantial set of supporting libraries (with over 4000 supporting lemmas in over 200 files). However, we expect that new supporting lemma are still needed before one can use the library to prove all the leaf-level

```
(defthm bcv-typelistAssignable-modify-local-variable-slot
   (implies (and (typelistassignable sl gl env1)
                 (consistent-sig-locals sl icl)
                 (consistent-sig-locals gl icl)
                 (integerp i)
                 (<= 0 i)
                 (< i (len gl))
                 (isAssignable s g env1)
                 (not (equal g 'topx)))
            (typelistassignable (modifylocalvariable i s sl)
                                (modifylocalvariable i g gl)
                                env1)))
```

Figure 7.49: `ModifyLocalVariable` operation is monotonic

lemma for the remaining JVM insturctions.

```
(defthm |local-sig-frame-set-locals-update-nth-plus-
                                      invalidate-1-strong|
  (implies (and (< 0 i)
                (< i (len (locals (current-frame s))))
                (integerp i)
                (consistent-state s)
                (consistent-value-x v (instance-class-table s)
                                      (heap s))
                (equal (type-size (tag-of v)) 1)
                (equal (type-size
                        (tag-of
                          (nth (- i 1)
                               (locals (current-frame s)))))
                       1))
    (equal (locals-sig (UPDATE-NTH i v
                                   (locals (current-frame s)))
                       (instance-class-table s)
                       (heap s)
                       (heap-init-map
                                (aux s))
                                (method-ptr (current-frame s)))
           (bcv::modifylocalvariable
            i
            (value-sig v
                       (instance-class-table s)
                       (heap s)
                       (heap-init-map (aux s))
                       (method-ptr    (current-frame s)))
            (locals-sig (locals (current-frame s))
                        (instance-class-table s)
                        (heap s)
                        (heap-init-map (aux s))
                        (method-ptr (current-frame s)))))))
```

Figure 7.50: Effect of `update-nth` on type signature state

Chapter 8

Conclusion

We studied the Java Virtual Machine (JVM) [45] as an abstraction layer. We formalized the safety guarantee provided by this layer. We studied how the safety guarantee may be correctly and efficiently provided by a specification compliant JVM implementation.

The official JVM specification (JVMSpec) describes the safety guarantee of the layer as (1) that all reachable states meet a set of constraints (see section 4.8.2,*Structural Constraints*, of the JVMSpec [45]) and (2) that JVM instructions are always executed with their preconditions met.

We identified the implicit guarantee of the JVM specification that, as long as JVM implementors correctly implement the operationally specified JVM operations (including the bytecode verifier and class loader), their JVM implementation will provide the declaratively specified safety guarantees.

We moved on to prove that this implicit guarantee is in fact true. In such efforts, we realized that in order to prove that the JVM provides the original form of the safety guarantee, we first need to prove by mathematical induction that the JVM satisfies a stronger version of the safety guarantee.

As a necessary intermediate step, we identified a *consistent state* that needs to be maintained by the JVM. We observed that a consistent state not only needs to represent a sensible JVM (by satisfying a set of consistency constraints among its own components), it also needs to satisfy an *on-track* requirement. The on-track requirement demands that every call frame of the JVM is approximated by some corresponding abstract state observed by the bytecode verifier.

8.1 Contributions

This dissertation presents our work towards formalizing these aforementioned concepts and observations, and proving that our model of the JVM and its bytecode verifier have desired properties. We made the following contributions:

Detailed JVM model

We built a detailed executable formal model of the JVM.

It serves as a platform for reasoning about properties of the JVM and properties of Java programs [14]. Simpler JVM models have been used for reasoning about Java programs that do not rely on features available in this JVM model [32].

JBook [35] describes a detailed JVM model of comparable complexity. Our JVM model, M6, is written in ACL2. We benefit from the fact that proofs about M6 can be mechanically checked by the ACL2 theorem prover. The ACL2 theorem prover acts both as a meticulous critic and a personal assistant in finding a machined checked proof.

Our JVM model is also built with an emphasis to capture what is specified in the official JVM specification. We specified "guards" for JVM operations, which capture all the side conditions for the JVM to execute that operation safely.

Detailed bytecode verifier model

We created an executable CLDC bytecode verifier by translating the Prolog-style rules used in the official bytecode verifier specification [11].

It is the only formal bytecode verifier model built directly from the 2003 bytecode verifier specification. Differing from related work [12, 21], we aimed at proving properties of the official bytecode verifier. Type hierarchy information is explicitly encoded in a data structure, the `class-table`.

Useful JVM safety specification

We introduced an alternative formulation of the JVM safety guarantees. We specified the JVM safety guarantees in terms of (1) that an inductive invariant on the runtime state that will be preserved and (2) that all guards that we have defined for the JVM operations will be met.

We argued that this formulation of the JVM safety guarantees is useful for JVM implementers (1) to pick a low level representation for JVM states, (2) to implement the JVM operations with low level primitives, and (3) to check that their implementation can provide the safety guarantee (see Chapter 6).

We also observe that defining this alternative safety specification is a necessary intermediate step for proving that the operationally specified JVM can provide the declaratively specificied safety guarantees in JVMSpec.

We note that our safety specification is not yet complete (see Section 1.3.

A framework for proving a JVM is safe

We designed an overall approach and a supporting lemma library for proving that the JVM is safe and the bytecode verifier is effective.

We demonstrated the approach by proving that a simple machine is safe and that its CLDC-style static checker is effective (see Section 7.1). The proof is mechanically verified by ACL2.

The key observations obtained are (1) we need to identify the *on-track* property as a part of the global invariant that a safe JVM execution will maintain and (2) we need to introduce an alternative bytecode verifier to separate the procedural aspects of the CLDC bytecode verifier specification from the essential reasons for the static checking to be effective.

We have identified leaf-level lemmas that we need to prove about every instruction. We have created an (incomplete) library for proving such leaf-level lemmas.

We have defined the alternative bytecode verifier. We proved a *reduction theorem* which asserts that all programs verified by the original CLDC bytecode verifier will also be verified by our non-procedural alternative bytecode verifier. This theorem is one important step in our overall approach to show that the JVM is safe.

8.2 Challenges and Lessons Learnt

We identify the following major challenges and remark on how one can improve on the approach that we have taken.

- M6 is too complex.

M6 contains many details that are necessary for using it as a realistic JVM simulator. However these details are not directly relevant for showing that the bytecode verifier is effective.

For example, M6's `INVOKE`-family instruction may call a *synchronized* method. To implement this correctly, M6 defines a set of operations for acquiring a lock, setting up the wait queue associated with the lock, and updating thread execution status.

We consider that a better way to proceed is to define a cut-down version of M6. The cut-down version of M6 will have only one thread. It will have neither native methods nor "magic" operations for setting up an initial state for executing a program.

- The type hierarchy information used by M6 and the bytecode verifier is not exactly the same.

  Our CLDC bytecode verifier checks a method against the external class table. However, DJVM conducts its runtime checks against its internal class table.

  This adds the difficulty of relating the static checks done by the bytecode verifier to the corresponding defensive checks done by DJVM.

  Furthermore, we have decided to be accurate to the facts (1) that classes are dynamically loaded and (2) we only know that the internal class table encodes a *consistent type hierarchy*. We have decided not to assume any properties about the type hierarchy information encoded in the external class table (which the bytecode verifier uses).

  As a result, in order to derive consistency properties of the type hierarchy information encoded in the external class table, we have to resort to the facts (1) that internal class table encodes consistent type hierarchy information and (2) the internal class table is loaded from the static class table by an operationally specified class loader.

This choice adds much difficulty to our proof. In retrospect, considering the fact that the real bytecode verification of a method uses the internal class table, we would recommend that if one were to continue our work, one should consider either defining the bytecode verifier to use the internal class table or asserting a similar consistency requirement on the type hierarchy information encoded in the external class table.

- ACL2 has the cumbersome requirement that the guard of a definition must itself have a verifiable guard of `t`.

  Instead of just proving meaningful JVM operations can be guard verified, we need to first guard verify the guard of the operation. When the guard for an operation becomes complicated, to verify the guard of the guard also becomes more complicated. We need to guard verify every operation used in defining the guard.

  We hope that the ACL2 theorem prover can be updated to remove this "unnecessary" requirement . [1]

  Alternatively, we may extend the ACL2 programming language to allow attaching pre-conditions and post-conditions to every function def-

[1] We note that although this requirement is unnecessary — when one is only using the guard to write specifications for ACL2 programs, however, ACL2 users does often write and verify guards of their program for a different purpose — to allow efficient and safe execution of the ACL2 programs that they wrote. In this latter senarios, such a requirement is necessary.

inition. We may extend the ACL2 theorem prover to systematically generate proof obligations to show that the function does meet its specification: (1) when input meets the precondition, the output meets the post-condition and (2) all sub-operations are invoked with their precondition met.

- The full scale of this project is big. Manging the proofs itself becomes a challenge.

  We have defined 2104 ACL2 functions. We proved over 4764 theorems. 196 of them are "skip-proofed". To prove the rest of them, 971 inductions are necessary. The total line count of the ACL2 input is 136000. They are organized in 282 files. The dependency graph between the 282 ACL2 books has 2012 edges.

  One needs to be very conscious about how he organizes his proofs — how to group lemmas into books, how to only export effective rules outside a book, and how to use ACL2's `in-theory` hints.

- M6 has been built with "ad-hoc" data structures.

  When we first built M6, we chose the obvious way for representing different components of the M6. Many data structures have been represented as simply lists.

  Although we have defined primitive operations to access and update these data structures, we have not strictly followed the abstract data type (ADT) discipline to use only these operations to access and update them.

  Occasionally, we exploited the internal representation of the data structure to define other operations. This problem of ADT discipline violation occurs regularly when we are defining the guards for accessor operations. Ideally the guards themselves should be expressed as a set of requirements on components of the ADT and these components should be retrieved with the accessor functions. However, before we could define the accessors, we first have to define the guards for the accessors.

We feel that we have not invested enough efforts to identify an *effective strategy* to reason about these imperfect ADTs. We think that an effective strategy would tell how to define a new data structure, what rules we need to prove about the new data structure, and how to set up ACL2's current theory to prove properties about operations that use these ADTs.

Should a researcher decide to continue this effort, we think she should first decide on such an effective strategy. She then needs to follow such a strategy and define a set of books to reason about the common ADTs, such as stack, list, dictionary. She may consider rebuilding M6 using these as building blocks.

Bibliography

[1] B. Brock and W. Hunt, Jr. Formal analysis of the motorola CAP DSP. In *Industrial-Strength Formal Methods*. Springer-Verlag, 1999.

[2] P. Bertelsen. Semantics of Java Byte Code. Technical report, Technical University of Denmark, 1997.

[3] G. R. Blakley. The emperor's old armor. In *NSPW '96: Proceedings of the 1996 workshop on new security paradigms*, pages 2–16, New York, NY, USA, 1996. ACM Press.

[4] B. Brock and J S. Moore. A mechanically checked proof of a comparator sort algorithm. In *Engineering Theories of Software Intensive Systems*. IOS Press, Amsterdam, 2005.

[5] R. Cohen. Defensive Java Virtual Machine Version 0.5 alpha Release. Available from `http://www.cli.com/software/djvm/index.html`, 1997.

[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM Press, 1977.

[7] E. Börger and R. F. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer, 2003.

[8] E. G. Sirer and B. Bershad. Kimera bytecode verification. http://www.cs.cornell.edu/People/egs/kimera/verifier.html, 1997.

[9] E. Rose and K. Rose. Lightweight bytecode verification. In *OOPSLA'98 Workshop Formal Underpinnings of Java*, 1998.

[10] G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, and S. Melo de Sousa. A formal executable semantics of the javacard platform. In D. Sands, editor, *Proceedings of ESOP'01*, 2001.

[11] G. Bracha, T. Lindholm, W. Tao, and F. Yellin. CLDC byte code type-checker specification. As part of CLDC1.1 specification `http://jcp.org/en/jsr/detail?id=139`, January 2003.

[12] G. Klein and T.Nipkow. Verified lightweight bytecode verification. *Concurrency and Computation: Practice and Experience*, 13(13):1133–1151, 2001.

[13] L. Gong. *Inside Java2 Platform Security: Architecture, API Design, and Implementation.* Addison-Wesley, 1999.

[14] H. Liu and J S. Moore. Java Program Verification via a JVM Deep Embedding in ACL2. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *Proceedings of 17th International TPHOLs 2004*, volume 3223 of *LNCS*, pages 184–200, 2004.

[15] H. Liu and J S. Moore. Executable JVM model for analytical reasoning: A study. *Science of Computer Programming*, 57(3), 2005.

[16] J. Gosling, B. Joy, G. L. Steele Jr., and G. Bracha. *The Java Language Specification.* Addison-Wesley Publisher, second edition, 2000.

[17] J. Gosling and H. McGilton. The Java Language Environment. Available from `http://java.sun.com/docs/white/langenv/Intro.doc.html`, May 1996.

[18] J. Matthew, J S. Moore, S. Ray, and D. Vroon. A symbolic simulation approach to assertional program verification. Available from `http://www.cs.utexas.edu/users/sandip/publications/symbolic/main.html`.

[19] J S. Moore and G. Porter. An executable formal java virtual machine thread model. In *Proceedings of 2001 JVM Usenix Symposium*, Monterey, California, April 2001. USENIX.

[20] M. Kaufmann and J S. Moore. ACL2: An industrial strength version of nqthm. In *Proceedings of 11th Annual Conference on Computer Assurance (COMPASS'96)*, pages 23–34, Gaithersburg, MD, 1996. National Institute of Standards and Technology.

[21] G. Klein. *Verified Java Bytecode Verification.* PhD thesis, Institut für Informatik, Technische Universität München, 2003.

[22] H. Liu. Formal specification and verification of a jvm and its bytecode verifier (supporting material). Available from `http://www.cs.utexas.edu/~hbl/dissertation/`.

[23] H. Liu and J S. Moore. Executable JVM model for analytical reasoning: a study. In *Proceedings of the 2003 workshop on Interpreters, Virtual Machines and Emulators*, pages 15–23. ACM Press, 2003.

[24] M. Kaufmann. *Computer-aided Reasoning: ACL2 Case Studies*, chapter Modular Proof: The Fundamental Theorem of Calculus. Kluwer Academic Publishers, 2000.

[25] M. Kaufmann and J S. Moore. A precise description of the acl2 logic. `http://www.cs.utexas.edu/users/moore/publications/km97a.ps.gz`.

[26] M. Kaufmann and J S. Moore. Structured theory development for a mechanized logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001.

[27] M. Kaufmann and J S. Moore. A proof of the correctness of a towers of hanoi program. With the ACL2 system in *books/misc/hanoi.lisp*, April 2003.

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

[29] G. McGraw and E. Felten. *Securing Java*. John Wiley & Sons, Inc., 1999. `http://www.securingjava.com/toc.html`.

[30] J S. Moore. Inductive assertions and operational semantics. In D. Geist, editor, *Proceedings of CHARME 2003*, Lecture Notes in Computer Science, pages 289–303. Springer Verlag, 2003.

[31] J S. Moore. Proving theorems about Java and the JVM with ACL2. In M. Broy, editor, *Lecture Notes of the Marktoberdorf 2002 Summer School*, LNCS. Springer, 2003. `http://www.cs.utexas.edu/users/moore/publications/marktoberdorf-03`.

[32] J S. Moore and G. Porter. The apprentice challenge. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):193–216, 2002.

[33] G. Porter. A commuting diagram relating threaded and non-threaded JVM models. Technical report, Honors Thesis, Department of Computer Sciences, University of Texas at Austin, 2001.

[34] Z. Qian. A formal formal specification of Java Virtual Machine instructions. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, pages 271–312. Springer-Verlag, 1999.

[35] R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer, 2001.

[36] S. Freund and J. Mitchell. A type system for the java bytecode language and verifier. *Journal of Automated Reasoning*, 2003.

[37] S. Ray and J S. Moore. Proof styles in operational semantics. In *Proceedings of FMCAD'04*, Austin, TX, 2004. Springer LNCS.

[38] V. Saraswat. Java is not type-safe. Available from `http://www.cis.upenn.edu/~bcpierce/courses/629/papers/Saraswat-javabug.%html`, 1997.

[39] J. Schmid. Executing ASM specifications with ASMGofer. `http://www.tydo.de/AsmGofer/`, 1999.

[40] D. A. Schmidt. Abstract interpretation in the operational semantics hierarchy. BRICS RS-97-2, Department of Computer Science, University of Aarhus, Denmark, 1997.

[41] R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proceedings of POPL'98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 149–160, San Diego, CA, 1998.

[42] Connected Limited Device Configuration (CLDC) Specification 1.1. `http://jcp.org/en/jsr/detail?id=139`.

[43] The K Virtual Machine. `http://java.sun.com/products/cldc/`.

[44] Announcing java (first press release). Available from `http://www.sun.com/smi/Press/sunflash/1995-05/sunflash.950523.3421.html%`, May 1995.

[45] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley Publisher, second edition, 1999.

[46] Frank Yellin. Low level security in Java. Available from `http://java.sun.com/sfaq/verifier.html`, Decemeber 1995.