Java Program Verification via a JVM Deep Embedding in ACL2

Hanbing Liu and J Strother Moore

University of Texas at Austin

{hbl,moore}@cs.utexas.edu

Overview of the Work

- We modeled a realistic JVM to formalize the semantics of Java bytecode programs

Overview of the Work

- We modeled a realistic JVM to formalize the semantics of Java bytecode programs
- We proved simple properties of single threaded Java programs

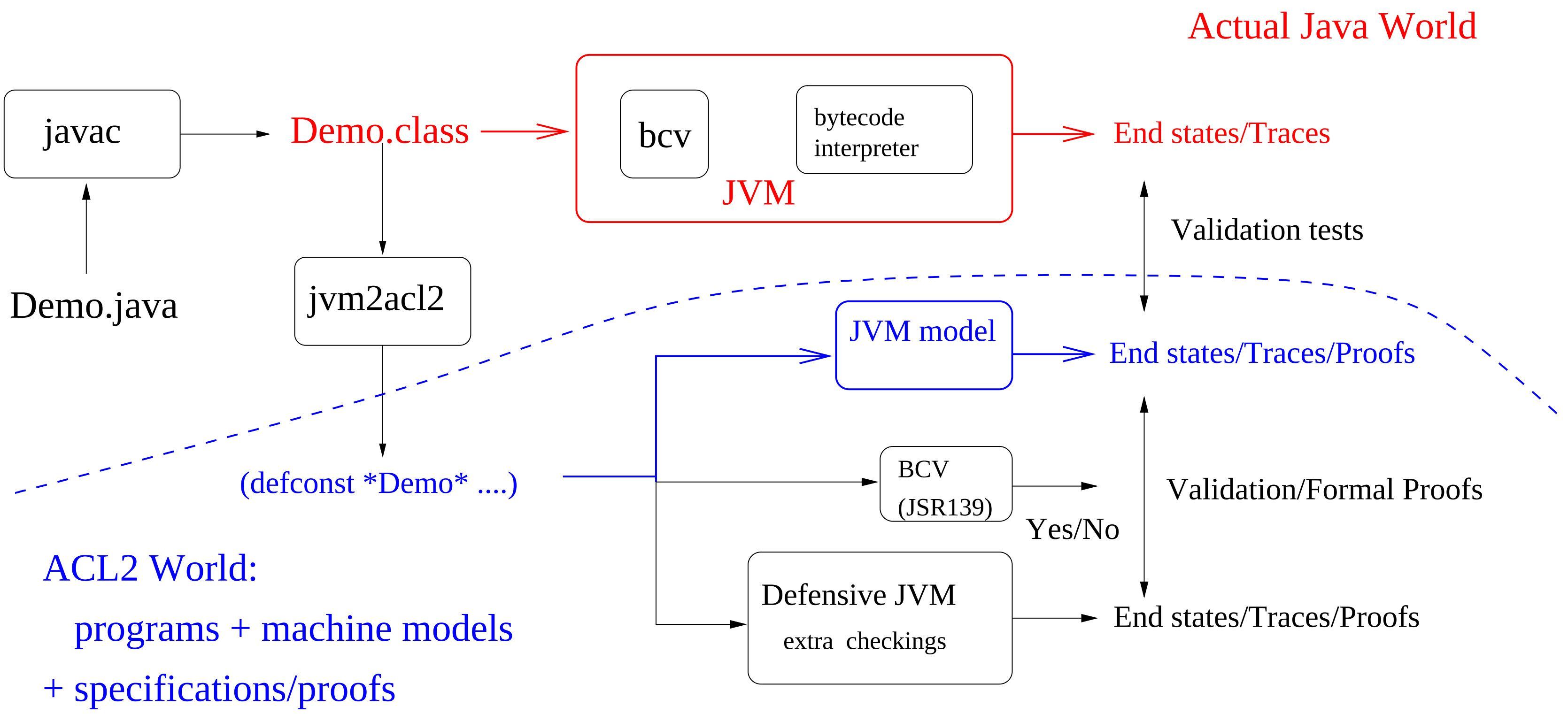
Overview of the Work

- We modeled a realistic JVM to formalize the semantics of Java bytecode programs
- We proved simple properties of single threaded Java programs
- We hold that for Java verification our approach provides better assurance than the “shallow” embedding approach

Overview of the Work

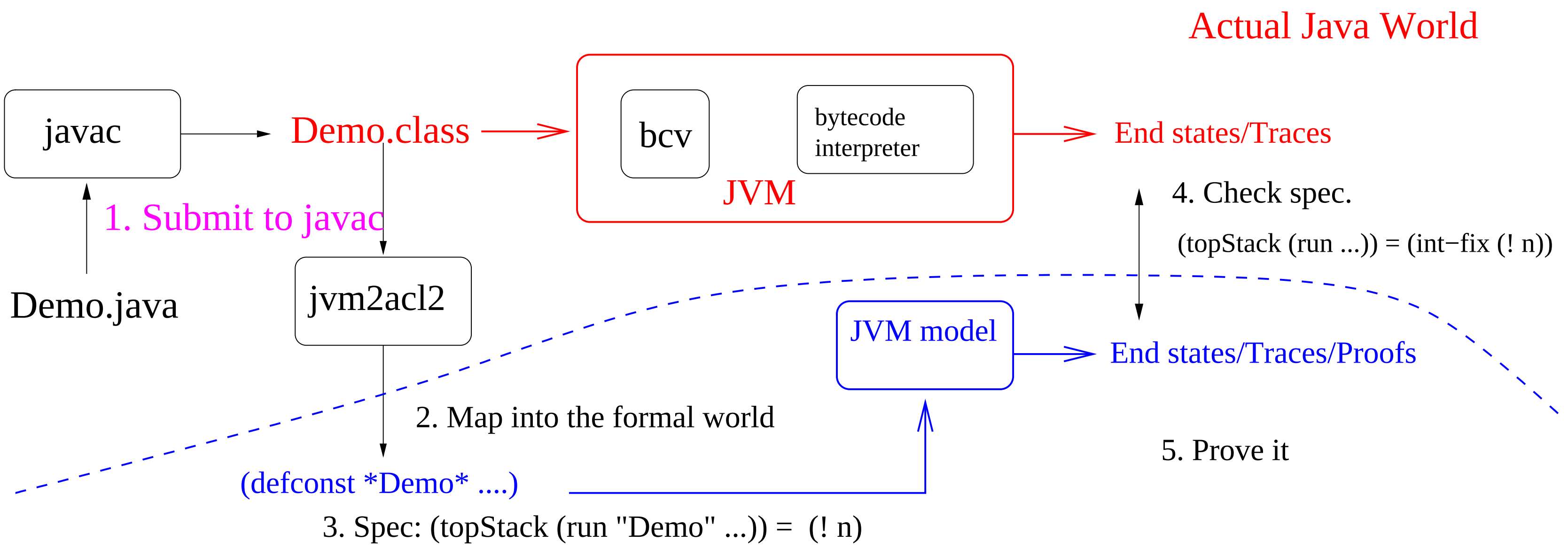
- We modeled a realistic JVM to formalize the semantics of Java bytecode programs
- We proved simple properties of single threaded Java programs
- We hold that for Java verification our approach provides better assurance than the “shallow” embedding approach
- We show that, for single threaded program at least, the complexity of deep embedding approach can be effectively managed with automatic theorem proving support

Java Program Verification Process



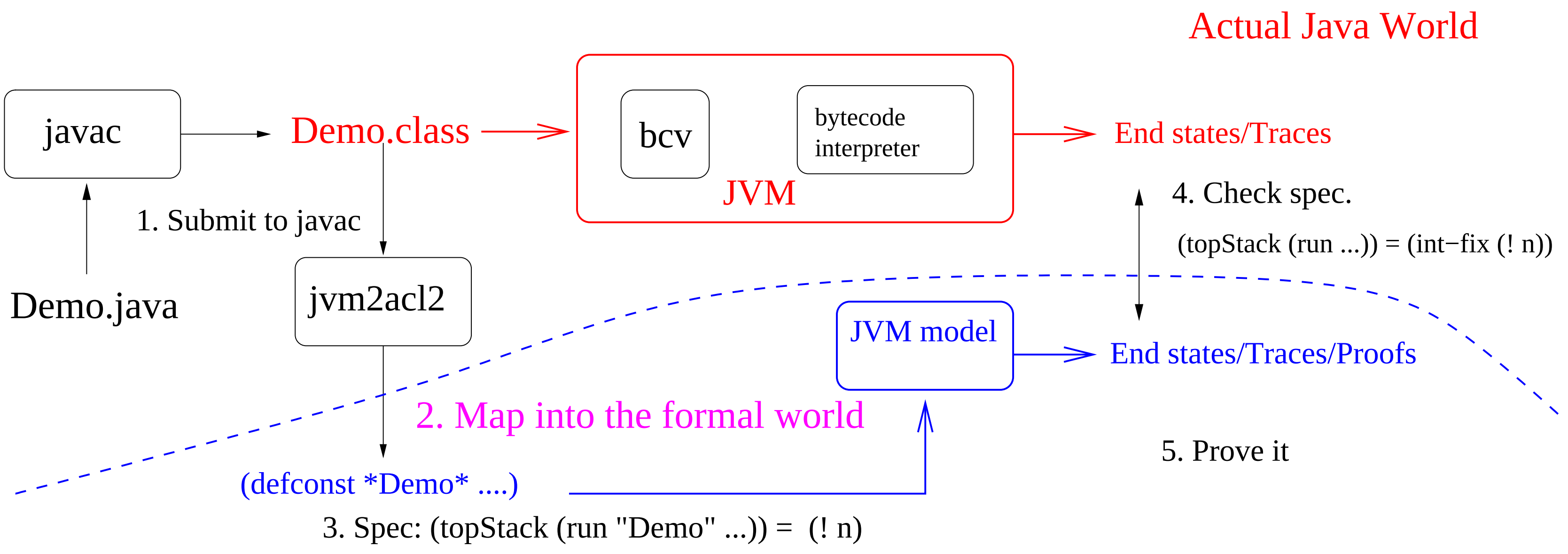
The Framework

Java Program Verification Process



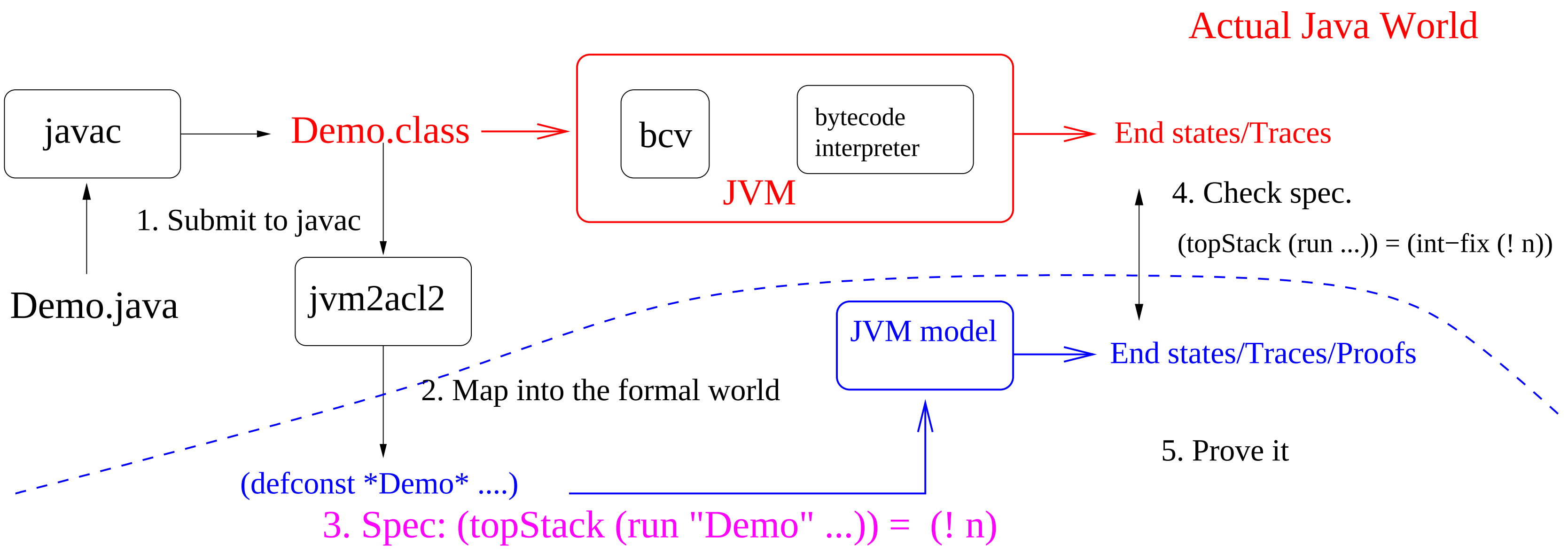
1. Submit Java programs to `javac`

Java Program Verification Process



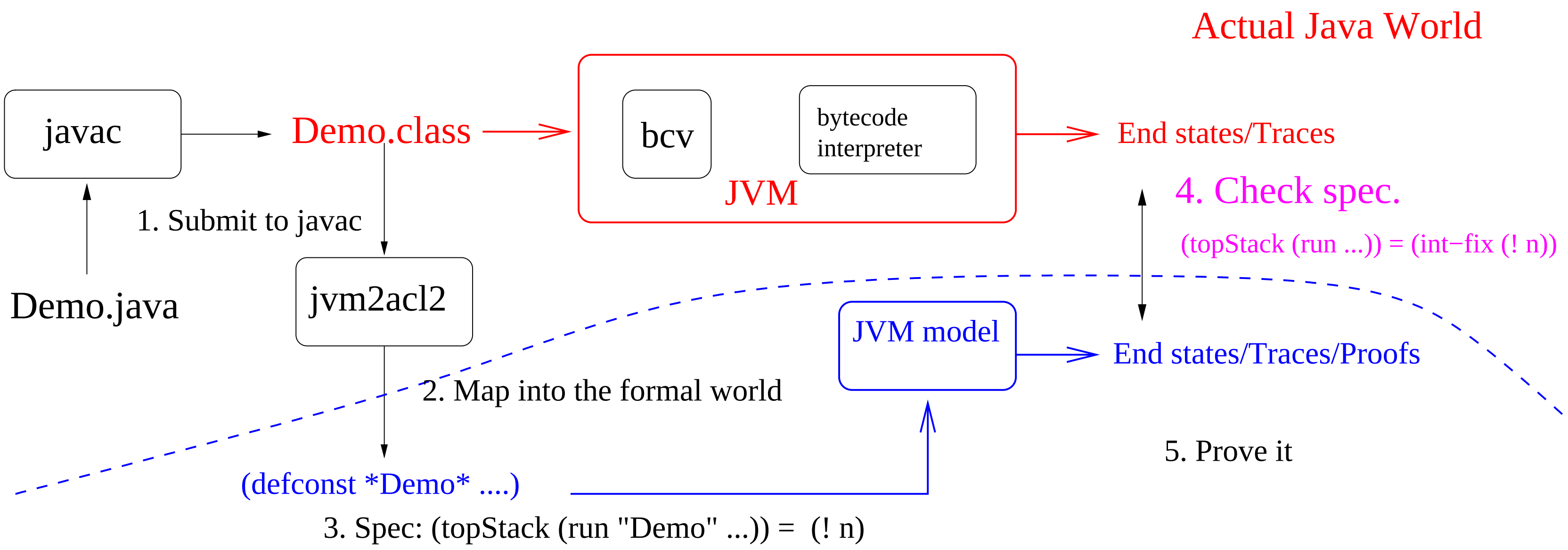
2. Map into the formal world

Java Program Verification Process



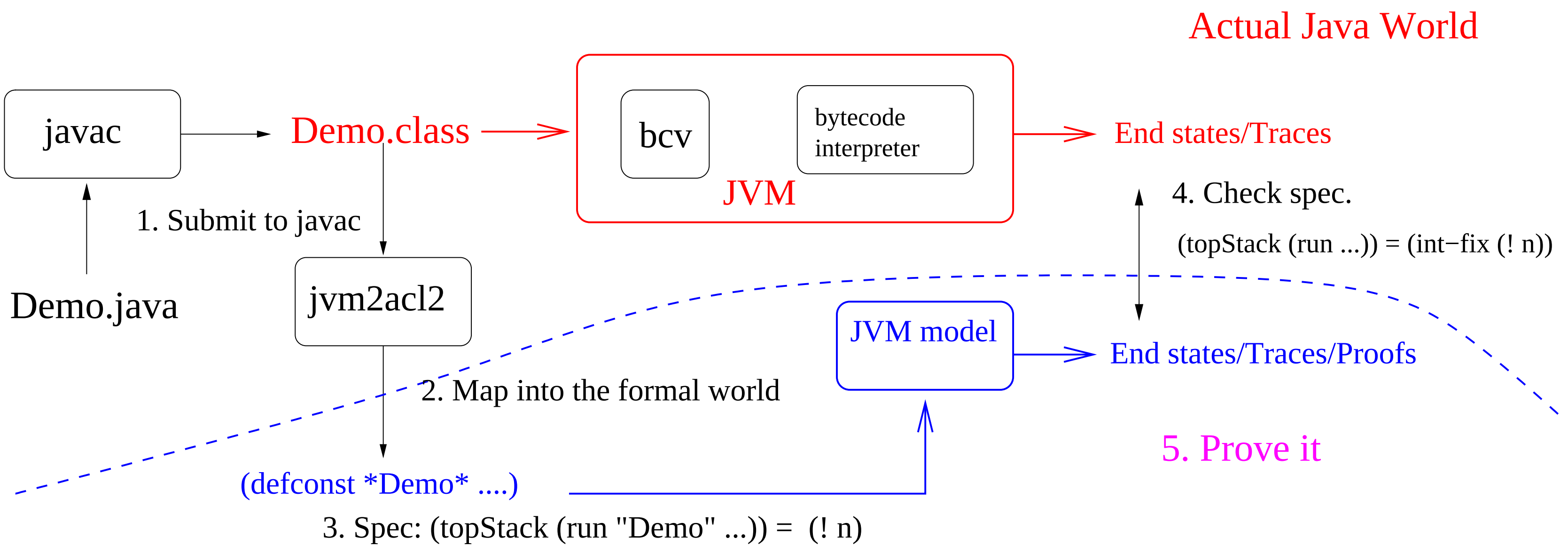
3. Write down the specification

Java Program Verification Process



4. Check the specification

Java Program Verification Process



Actual Java World

javac

Demo.class

bcv

bytecode interpreter

JVM

End states/Traces

1. Submit to javac

Demo.java

jvm2acl2

4. Check spec.

(topStack (run ...)) = (int−fix (! n))

JVM model

End states/Traces/Proofs

2. Map into the formal world

5. Prove it

(defconst *Demo* ....)

3. Spec: (topStack (run "Demo" ...)) = (! n)

5. Just prove it

Executable JVM Model in ACL2

Demo: JVM Model and Programs in Execution

Formalization: “Deep” or “Shallow”

- We used the “deep embedding” approach
  - Keep the form of the original bytecode program
  - Formalize the semantics of the *language* as opposed to assign meanings to specific programs
  - Formalize the properties of programs as assertions on execution traces and states
- We did not use the “shallow embedding” approach
  - Rephrase each bytecode program case by case
  - Nor write a “compiler” that translates programs

Why Model a JVM?

Main criticism: complexity

Considerations:

- We are more confident in correctly modeling a JVM than rephrasing Java programs case by case. *A shallow embedding maybe more suitable for descriptions in a simpler language, such as netlists descriptions for digital circuits but not for Java programs*

Why Model a JVM?

Main criticism: complexity

Considerations:

- We are more confident in correctly modeling a JVM than rephrasing Java programs case by case. *A shallow embedding maybe more suitable for descriptions in a simpler language, such as netlists descriptions for digital circuits but not for Java programs*
- We are more confident in properties directly formulated as assertions on execution traces and states.

Why Model a JVM?

Main criticism: complexity

Considerations:

- We are more confident in correctly modeling a JVM than rephrasing Java programs case by case. *A shallow embedding maybe more suitable for descriptions in a simpler language, such as netlists descriptions for digital circuits but not for Java programs*
- We are more confident in properties directly formulated as assertions on execution traces and states.
- We are interested in studying properties of the JVM and the bytecode language.

Why Model a JVM?

Main criticism: complexity

Considerations:

- We are more confident in correctly modeling a JVM than rephrasing Java programs case by case. *A shallow embedding maybe more suitable for descriptions in a simpler language, such as netlists descriptions for digital circuits but not for Java programs*
- We are more confident in properties directly formulated as assertions on execution traces and states.
- We are interested in studying properties of the JVM and the bytecode language.
- We have “automatic” mechanical theorem proving support.

Theorem Proving Support

- Executable: intuitive for defining operational semantics
- ACL2 “learns” from established theorems
  - Non-trivial efforts in solving a brand-new problem. We need about 300 lemmas for proving that a 7-instruction ADD1 program adds one to its local variable.
  - Greatly reduced efforts in solving similar problems. We need 20 extra lemmas for proving a 15-instruction straight line program that manipulates locals and operand stack.
  - Orthogonality of different operations can be captured.

“Factorial computes factorial”

```
(defthm factorial-computes-factorial
    (implies (and (poised-to-invoke-fact s)
                  (wff-state-regular s)
                  (wff-thread-table-regular (thread-table s))
                  (no-fatal-error? 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 (! n))
                                           (popStack s))))))
```

Difficulties in Proofs

- The “frame” problem — what does not change
  - Identify equivalence relations
  - Prove that operations preserve equivalence relations
  - Express desired properties as properties on equivalence classes
  - Prove congruence rules about operations
- Identify the implicit assumptions about the domain. Example: `next-inst`

Related Work

Combine rewrite engine and operational semantics for:

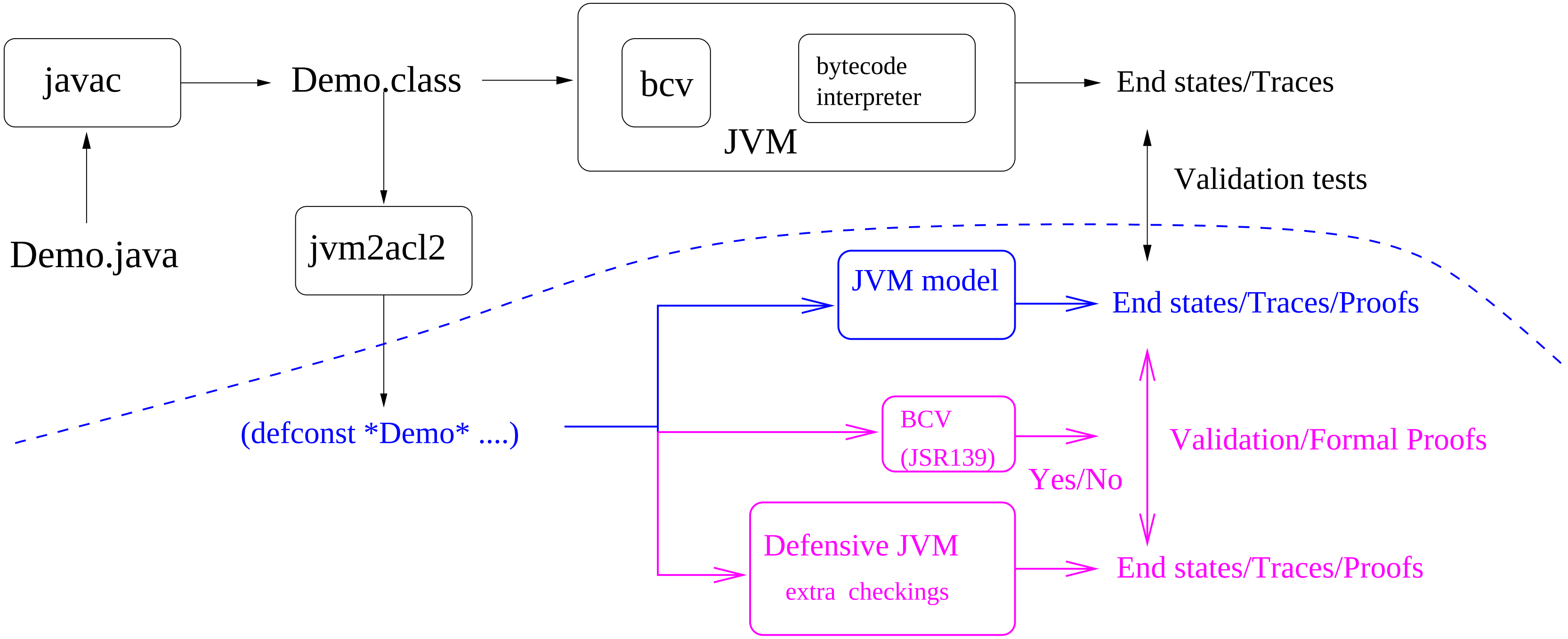
- Verification condition generation
  — without a verification condition generator

  See *Inductive Assertions and Operational Semantics*, CHARME'03.

- "Shallow embedding"
  — without a compiler

  The idea is to use the symbolic execution to reduce bytecode programs to their "effect" functions that directly modify the states.

Other Work: Verify a JVM



Verify the JVM and its bytecode verifier

Conclusion

- Formalized the Java bytecode language by modeling a realistic JVM
- Proved properties of simple Java programs
- Gained a better assurance via:
  - Direct formulation of simple facts
  - Machine checked derivation steps
  - Validation tests that one can execute
- Managed the complexity with support from ACL2
- We are working on the verification of a JVM with its bytecode verifier
- Question?

Other Demos

- “Shadow embedding” without a compiler
- Example: reasoning about `next-inst`
- Example: Orthogonality captured by ACL2 theorems