Program Verification: Lecture 1

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

Program Verification: What and Why

What is Program Verification? Why is it needed?

- testing can show errors but not their absence
- software errors in critical systems can cause major disasters
- mathematics can be used to state program properties and to prove a program correct for all inputs
- we then say that the program has been verified correct

By Hand vs. Tool-Assisted Verification

One can reason mathematically about a program and prove it correct; this can be done even with pencil and paper.

This is the most important task; it cannot be replaced by machines for many reasons, including the fact that the choice of what properties to prove can be an ethical choice.

However, to err is human, and it is a fact of life that even very experienced professionals make programming mistakes and can similarly make mistakes in:

- specification, i.e., in modeling a system mathematically (system specification) and/or in stating the properties to be verified (property specification); and
- proof, i.e., in the actual proof of correctness.

By Hand vs. Tool-Assisted Verification (II)

Tools that mechanize the deduction process can greatly help in avoiding proof mistakes and, to some extent, specification mistakes:

- tools supporting executable specification can help in debugging specs; and
- proof assistants and other property verification tools (for example, model checkers) can ensure proof correctness.
- proof checkers and other certification methods can verify that the proofs carried out by verification tools are correct, so the tools themselves need not be trusted.

Limitations of Program Verification

Exagerated claims about what can be achieved through program verification are dangerous: they can lead to a dangerous overconfidence in a system's correctness.

We have only limited ways of convincing ourselves that we have given the right specification: there can be mistakes in the specification and in capturing the informal requirements.

Even with the right specification, all we can prove at best is the correctness of a mathematical abstraction, never of the system running in the real world.

All we can say at best is that if the compiler and the hardware design are correct, and the hardware behaves according to its specifications, then the program will execute correctly.

Limitations of Program Verification (II)

Of all the above preconditons for correct execution the first two—compiler correctness and correctness of the hardware design—deal after all with properties reducible to mathematical abstractions and can therefore be included within the program verification project.

This is because a compiler is just another program; and because a hardware design, as opposed to a hardware physical implementation, is just an abstraction that can be mathematically described just as software can, and at that level of abstraction the software/hardware distinction evaporates.

Limitations of Program Verification (III)

The biggest and most irreducible if is whether the hardware will happen to behave according to its specifications. And this for at least two reasons:

- those specifications assume normal operating conditions which can be violated by a wide range of accidental causes such as: defects in the materials or in the fabrication process, cosmic rays, changes in temperature, power outages, floods, earthquakes, etc.
- the engineering design rules are ultimately based on physico-mathematical models of the physical world which are, and always will be, both aspectual and fallible approximations of reality.

Limitations of Program Verification (IV)

In spite of the above-mentioned limitations, program verification is one of the best engineering ways that we have of gaining high confidence in the correctness of critical software systems.

And this for the exact same reason why using mathematical models is our best way to know what we are doing in science and engineering.

Due to pragmatic and economic reasons connected with the labor-intensive nature of program verification, it is often not feasible to fully verify all systems down to the hardware design, or even to fully verify the software.

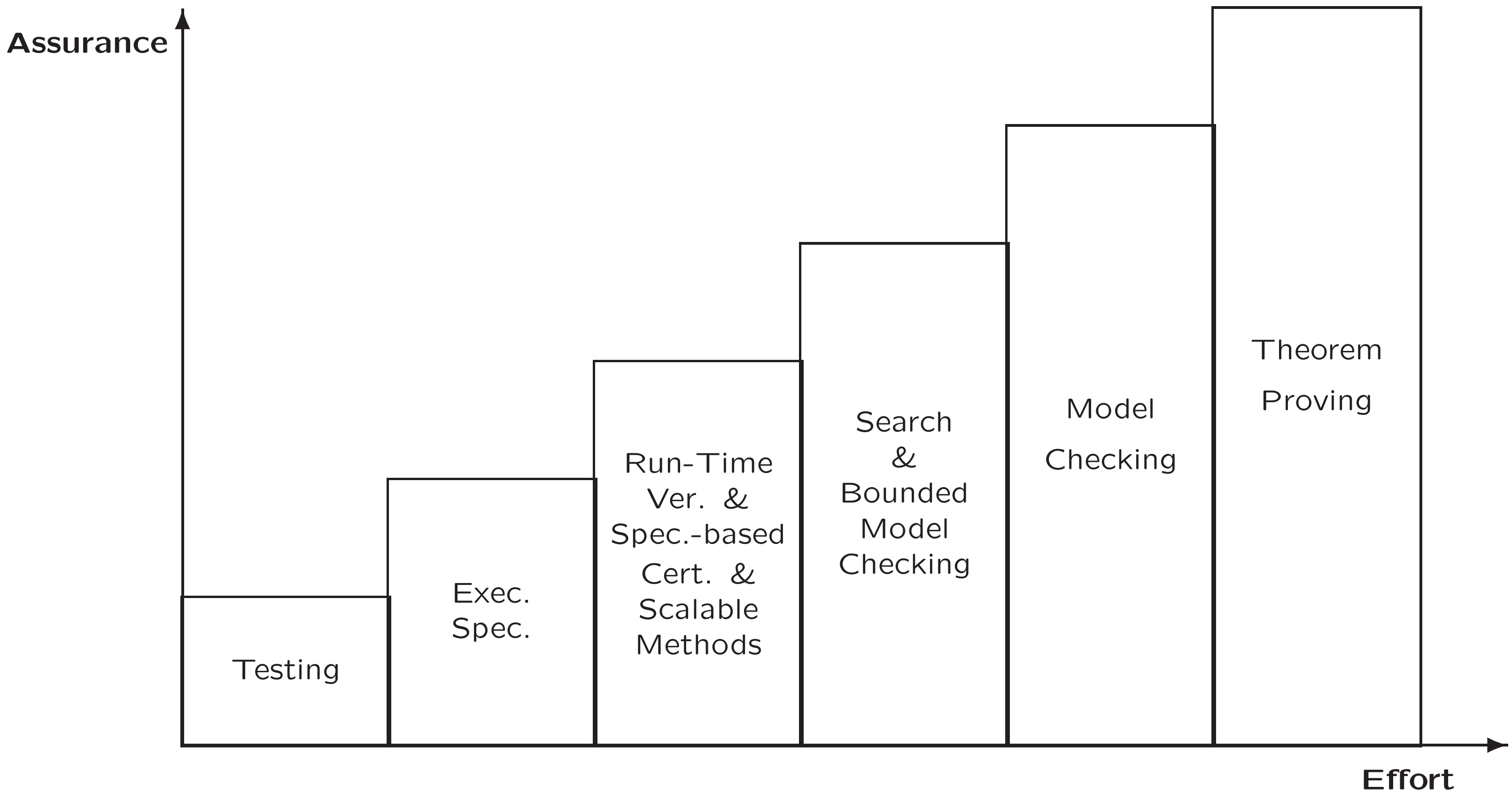
Limitations of Program Verification (V)

The fact of life is that not all systems are equally critical. Therefore it is a question of good judgement, and at times also an ethical question, to decide how much effort should be spent in program verification.

At one end of the spectrum we have testing, as a weak form of program validation. At the other end of the spectrum we have testing plus full program verification, say down to the hardware design.

In the middle we have a wide range of methods of partial program verification such as, for example, symbolic simulation, model checking, and runtime verification. The key point is that even a modest amount of program verification can go a long way in increasing software quality.

Formal Methods: their Cost and Assurance



Deterministic vs. Concurrent

Programs come in many different languages and styles. This in fact impacts both the level of difficulty and the verification techniques suitable in each case.

A first useful distinctions is deterministic vs. concurrent:

- deterministic programs, for each input either yield an answer or loop; they are usually written in sequential programming languages and run on sequential computers, but sometimes they can be parallelized;

- concurrent programs may yield many different answers, or no answer at all, in the sense of being reactive systems constantly interacting with their environment; they usually run simultaneously on different processors.

Imperative vs. Declarative

A second useful distinction is imperative vs. declarative:

- imperative programs are those of most conventional languages; they involve commands changing the state of the machine to perform a task;

- declarative programs give a mathematical axiomatization of a problem, as opposed to low-level instructions on how to solve it; they can be based on different logical systems.

Of course, the deterministic vs. concurrent and the imperative vs. declarative are orthogonal distinctions: all four combinations are possible.

The Declarative Advantage

For program reasoning and verification purposes, declarative programs have the important advantage of being already a piece of mathematics. Specifically:

- a declarative program P in a language based on a given logic is typically a logical theory in that logic.
- the properties that we want to verify are satisfied by P can be stated in another theory Q; and
- the satisfaction relation that needs to be verified is a semantic implication relation $P \models Q$ stating that any model of P is also a model of Q.

The Imperative Program Verification Game

By contrast, imperative programs are not expressed in the language of mathematics, but in a conventional programming language like C, C^{++}, Java, or whatever, with all kinds of idiosyncrasies.

Thefore, the first thing that we crucially need to do in order to reason about programs in an imperative programming language $\mathcal{L}$ is to define the mathematical semantics of $\mathcal{L}$.

This we can always do in informal mathematics, but for tool assistance purposes it is advantageous to axiomatize the semantics of $\mathcal{L}$ as a logical theory $T_{\mathcal{L}}$ in a logic.

The Imperative Program Verification Game (II)

Then, given a program P in $\mathcal{L}$, the properties we wish to verify about P can typically be expressed as a logical theory $Q(P)$, involving somehow the text of P.

In the imperative case the satisfaction relation can again be understood as a semantic implication between two theories, namely, the axiomatization of the language and the desired properties: $T_{\mathcal{L}} \models Q(P)$.

This is not the conventional way to think of it. Conventionally, a "logic of programs" such as Hoare's logic is used, with triples of the form $\{A\}P\{B\}$ with P the program and A, B formulas. But we shall see that the conventional approach can be subsumed in the above one.

The Equational/Rewriting Logic Framework

A very good and nontrivial question is what logic to use as the framework logic for program verification. There are many choices with different tradeoffs.

In this course we will use equational logic to axiomatize the semantics of (declarative or imperative) deterministic programs, and rewriting logic to axiomatize the semantics of (declarative or imperative) concurrent programs.

To axiomatize the properties satisfied by such programs we will allow more expressive logics, such as full first-order logic plus inductive principles, or even temporal logic (for concurrent programs).

The Equational/Rewriting Logic Framework (II)

The above choice has the following advantages:

1. suitable subsets of equational and rewriting logic are efficiently executable, giving rise, respectively, to a declarative deterministic functional language, and a declarative concurrent language;

2. equational logic is very well suited to give executable axiomatizations of deterministic languages, including imperative sequential languages;

3. rewriting logic is likewise very well suited to give executable axiomatization of (declarative or imperative) concurrent languages;

4. therefore, we can specify all the four kinds of programs in an executable way within the combined framework.

Initiality and Induction

Yet another key advantage is that equational and rewriting logic theories have initial models. That is, theories in these logics have an intended or standard model, (also called initial) which is the one corresponding to our computational intuitions.

Inductive reasoning principles, such as the different induction schemes, are then sound principles to infer other properties satisfied by the standard model of a theory.

The two crucial satisfaction relations for declarative, resp. imperative, program verification, namely, $P \models Q$, resp. $T_{\mathcal{L}} \models Q(P)$, should be understood as inductive satisfaction relations, corresponding to the initial model of P, resp. $T_{\mathcal{L}}$.

Maude

Maude is a declarative language and high-performance interpreter based on rewriting logic that is very well suited for concurrent specification and programming.

Since equational logic is a sublogic of rewriting logic, Maude has a functional programming sublanguage.

We will use Maude and its tools in the course to experiment with and verify both determinisitc (functional) and concurrent declarative programs.

We will also use Maude and its tools to give executable axiomatizations of imperative sequential and concurrent programming languages and to verify imperative programs.

Course Outline

1. Equational logic and functional programming in Maude.

2. Initiality, induction, and verification of Maude functional programs.

3. Algebraic semantics of a simple sequential imperative language and verification of its programs.

4. Rewriting logic and concurrent programming in Maude.

5. Verification of Maude concurrent programs and of imperative concurrent programs.

What You Can Get out of this Course

1. basics of equational logic, rewriting logic, inductive theorem proving, Hoare logic, temporal logic, and model checking;

2. basics of functional and concurrent declarative programming in Maude;

3. equational/rewriting methods for giving executable semantics to imperative programming languages;

4. basic program verification principles and experience for deterministic declarative and imperative programs, and for concurrent declarative and imperative programs.

Set Theory Prerequisites

Set theory is the language of modern mathematics. In some countries, students are introduced to set-theoretic notation in high school or even earlier. In some others, even some graduate students in engineering have somehow been cheated out of this very basic training and are quite unfamiliar with even the most elementary set-theoretic notions.

As for any part of mathematics, also for logic we will need to use elementary set theory notions (and corresponding notations) including:

- set, subset, union, intersection, complement, etc.
- functions, injective, surjective, bijective, etc.

Set Theory Prerequisites (II)

- ordered pairs and cartesian products
- sets of functions from one set to another
- binary operations on a set
- relations, including reflexive, symmetric, and transitive relations
- equivalence relations, quotient sets, and partitions

Set Theory Prerequisites (III)

This is not a remedial course on elementary set theory; it is a course on program verification. Therefore, all the above set theory notions and notations will be assumed known by all the students.

Elementary set theory is not rocket science. It is indeed quite elementary, so if you were cheated out of this most basic mathematical training up to now, you can pick it up in a short time.

In fact, you must pick it up very soon in order for you to be able to follow the course.

Set Theory Prerequisites (IV)

To help you in this task, in case you need it, the following things may be useful and may help you focus your efforts:

1. Begin by studying Chapters 1–5 of J. Meseguer, *Set Theory and Algebra in Computer Science A Gentle Introduction to Mathematical Modeling*. Then do the exercises in homework 1.

Other Suggested Readings

Besides the suggested catching up reading on set theory, to help you with the course itself:

1. Browse through Chapters 1, 2, and 3 of "All About Maude," and get Maude itself up and running on your machine by downloading it from the Maude web page (`http://maude.cs.uiuc.edu`).

2. Part II of *Set Theory and Algebra in Computer Science A Gentle Introduction to Mathematical Modeling* can also be quite helpful to study the algebra and term rewriting ideas covered in the course.

3. As side reading, you can also benefit from Peter Ölveczky's lecture notes at the Univ. of Oslo, available in the course web page.