Looking Ahead

Amir Pnueli

New York University and Weizmann Institute of Sciences

Palo Alto, February 2005

A Disclaimer

A Talmudic quotation:

> Since the destruction of the temple,
> Prophecy has been taken from the prophets and given to the fools

A Plea for Reinstituting the Name: A Verifying Compiler

Arguments in favor:

- The term verification, whether formal or informal, is unambivalently interpreted as being applied post-facto.
- A verifying compiler can also be used to check that a program has been developed according to a preferred development methodology. Thus it can also cover a multitude of correct by construction approaches.
- Discussions both before and during the workshop, seem to suggest that development approaches should not be ruled out.
- The view that a program should be allowed to run only if it is syntactically correct, type safe, and semantically correct, is a very appealing concept.

Impressive Progress in Formal Verification

Is due, as we heard, to

- Faster machines.
- Brilliant new ideas and algorithms.
- Lowering the expected level of verified properties.

Impressive Progress in Formal Verification

Is due, as we heard, to

- Faster machines.
- Brilliant new ideas and algorithms.
- Lowering the expected level of verified properties.

to which I wish to add:

- Lowering the expected degree of automation.

In fact, it is enough to reduce the measure $|properties| \times |automation|$.

The Serious Goal of Formal Verification

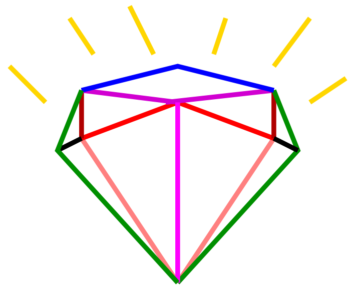
Has always been software verification.

The unanticipated success of hardware verification has been a 15 year exciting diversion (and distraction).

- It gave us most valuable techniques such as BDD's, SAT, and model checking.
- It also cultivated the false illusion that most verification tasks can be solved by a press-button methods

Now that we have honed many interesting techniques on the toy problems of hardware verification, it is time to go back to the main goal of software verification.

In FM'99 I Presented A Talk



Deduction is Forever

I am now ready to tone it down to

User Guidance is Unavoidable

The Law of Conservation of User Guidance

- One may replace deduction by abstraction, but then the user has to provide the abstraction mapping.
- You may try to base the abstraction mapping on predicate abstraction, but the the user has to provide the predicate base.
- There are automatic way to construct an initial base an performs abstraction refinement, but we just heard that in the interesting cases, there is no replacement for manual refinement

The Above Picture is Exaggerated in Order to Make A Point

In fact, there is continued improvement in he degree of involvement of the user in the verification process.

The improvement is not in the reduction of responsibility, but it is in the direction of:

> Let the user do what humans do best, and leave the computer to handle the combinatorics and tedious searches.

A case best illustrating such a successful synergy is predicate abstraction, where the user provides the predicate base and the model checker finds the best boolean combination of these predicates which forms an inductive assertion.

Other cases: polynomial invariants, abstract interpretation, etc.

My View of the Future Verifying Compiler

is that it will be a proof checker in which the user will guide the proof steps. Each step can invoke different tools and methods such as a model checker, various program analyses, and deductive proof steps.

Some of the user guidance can be prepared a priori (off-line) in the form of assertions, annotations, or proof strategies.

Progress in our verification ability will be expressed in being able to take bigger proof steps that will be automatically performed.

So What Should We Do?

Obviously, any possible collaboration must be loose, where smaller groups which share common approaches could hold tighter cooperation.

Tying the effort together, we must work on means for integration, including:

- A common set of benchmark programs to be verified.
- Agreements on formats for interchange of specifications, models, and partial results. For example, for temporal verification of reactive systems, we recommended the use of fair transition systems. If you want to deal with procedures, this model has to be extended.
- Wherever possible, automatic translation between models, specifications, and logics.
- A better interface for inter-operability of tools. Examples:
  - Deep embedding of LTL and CTL* within PVS.
  - In the context of computing existential abstraction, shipping a query to PVSand receiving back a complete partially resolved goal tree.

Why Don’t We Try it on Linux?

- Like the human genome, a focusing single project.
- As an open source code, it is publicly available, and morally deserves our support.
- As divorced from commercial interests as possible. Nobody will give us 2M to work on it (perhaps we can be given money to desist).
- This is an opportunity to test Tony’s hypothesis that, given a verified and unverified versions of a program, people will always prefer the verified one.
- A major effort that will take significant time and have multiple benefits is the development of a specification.