Formal Verification: From Challenges to Solutions

Marius Minea

Department of Computing, "Politehnica" University of Timişoara
Institute e-Austria Timişoara
marius@cs.utt.ro

Abstract

Formal verification has evolved from theoretical research topic to widespread practical acceptance and success. We present basic aspects and discuss starting points for industrial application.

Why formal methods? As humans construct increasingly complex systems, ranging from safety-critical to infrastructure to everyday consumer electronics or software, the goal of ensuring their correct functionality becomes more difficult to fulfill. Formal methods are "mathematically-based languages, techniques and tools for specifying and verifying such systems" [CW96]. They improve upon more traditional techniques such as simulation and testing in several ways.

A first advantage is *completeness*. Whether based on *theorem proving* or algorithmic state-space exploration called *model checking*, formal verification techniques consider all possible execution scenarios of the system. Both simulation and testing cannot generally be exhaustive for systems of realistic size. They are ill-suited for handling concurrent, distributed or timed systems, with hard reproducible behaviors depending on the relative timing of events. It is here that formal methods can find elusive errors which may occur after a particular sequence of tens or hundreds of execution steps.

Another advantage is *rigor*. Once the model of the system has been chosen and validated, any formal verification results obtained on this model are mathematically guaranteed to be precise and sound, while simulation typically gives only an indication of real system behavior. To solve the issue of having an accurate system model, it is possible to maintain a correspondence (namely refinement) between the model for verification and the design or implementation, and in some cases even to synthesize implementations which are "correct by construction".

Last but not least, formal methods can lead to significant gains in efficiency. A verification run can be significantly faster than a suite of simulations or tests. But more importantly, applying formal methods early on can uncover specification inconsistencies and design errors which could otherwise propagate to the implementation, requiring costly tests to detect and remove.

Application domains and success stories

Hardware design is the classical example of an area where formal verification has impacted an entire industry [CK96]. Combinational equivalence checking is nowadays a standard part of every tool suite, and sequential verification is pursued in every major design company, with research groups using in-house or university-developed tools.

Formal verification is paramount in the area of *safety-critical systems*, being in some cases required by standards or governments. This is also a prime area where formal specification has successfully replaced natural language documents, with measurable benefits [CW96]. For *security protocols*, formal methods have discovered attacks in protocols that had been in wide use for years.

Formal methods have been used: in *telecommunications*, to specify features and interaction, and to verify protocols, e.g., written in SDL; in *embedded and real-time systems*, to verify a large part of the time-triggered architecture for automotive systems using the theorem prover PVS; in *consumer electronics*, where within less than two years, the state of the art advanced from tenuous manual proof for a part of an audio control protocol to automatic verification in its entirety, in a few seconds.

More recently, focus has been on software, with hundreds of errors found in operating system code by METAL, a defect detection tool based on user-specified patterns. Verification tools exist for C (SLAM, BLAST) and Java (ESC/Java, Bandera, Java Pathfinder), integrating techniques from model checking, static analysis and theorem proving. Verification support for design languages and models, as well as precise analysis methods scalable to large code are some of the emerging trends [JR00].

Complexity challenges and solutions

Complexity remains perhaps the major challenge for verification, much as it is for design. However, several key techniques and developments, coupled with the increase in computing power have helped formal methods keep up with design complexity.

The major hurdle is the so-called *state space explosion* problem: the number of states in a system increases exponentially in the number of components. *Symbolic model checking*, which represents state sets according to their relevant properties rather than individually has pushed the size of verifiable systems to a few hundreds of state bits.

For asynchronous concurrent systems, *partial order reduction* avoids the exponential increase in the number of possible executions by considering only relevant representatives, often yielding orders of magnitude in improvements.

The tools that introduced these techniques, SMV and Spin respectively, have been refined by a decade of evolution and are among the most powerful verification systems.

Theorem proving offers an approach that does not directly depend on the size of the state space, and can be used to verify also parameterized or infinite-state systems.

Abstraction is the key for scaling verification to large systems: rather than analyzing the original complex system, a simplified *abstract* system is built by techniques guaranteeing that correctess of the abstract system implies correctness of the initial one. *Compositional reasoning* allows to deduce properties of a system from properties of its components, avoiding the complexity of building the full system model. With these techniques, otherwise hopelessly complex systems can be decomposed and simplified yielding verification tasks of manageable size.

Are formal methods for you?

With a wide array of techniques, tools and existing applications, the potential for using formal methods exists virtually regardless of company size and profile. An application which is first in the design flow concerns specification and modeling. Even just the benefits of having a formal and thus well-defined specification can be significant. Modeling languages such as UML see wide usage, but often as little more than an intuitive graphical notation. With the definition of a formal semantics, verification support for UML is emerging and could be used to check initial properties of a design.

Another easy step is the use of so-called "lightweight formal methods", e.g., error detection tools which have seen applicability for system code, detecting violation of locking or system call policies. Static analysis tools can uncover problems starting with null pointer assignments, array bound violations, or unhandled exceptions, but in the presence of programmer annotations they can verify much more complex properties, including functional specifications such as invariants.

Perhaps the easiest candidates for full-fledged verification are systems written in a language for which there is direct tool support, such as C or Java. Some existing verifiers are quite mature, have good user interface and documentation, and an easy learning curve including template support for writing specifications and performing abstractions.

Even for design languages with no existing tools, the road to a verification solution need not be long. A number of powerful verification toolkits are built modularly around intermediate-level description languages. Thus, the only missing link to construct is a compiler front-end targeting the intermediate language closest in flavor, and thus giving access to a full set of verification algorithms. Finally, in some cases, manual model construction may be necessary. This can still be a cost-efficient approach for verifying a key part of a design, most frequently in the case of a concurrent, distributed or real-time system, for which simulation or testing does not give the needed confidence.

Formal verification can both efficiently detect errors and offer the guarantee of correctness. For systems of increasing complexity, where the starting assumption must be of an error hidden somewhere, it has gradually evolved from curiosity to acceptance to necessity.

References

[CK96] E. M. Clarke and R. P. Kurshan. Computer Aided Verification. *IEEE Spectrum*, 33(6):61–67, 1996.

[CW96] E. M. Clarke and J. M. Wing. Formal Methods. State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[FM03] J. R. Bowen (editor). The WWW Virtual Library: Formal Methods. `http://www.afm.sbu.ac.uk`

[JR00] D. Jackson and M. Rinard. Software Analysis: A Roadmap. *Proc. International Conference on Software Engineering*, pp. 133–145, June 2000.