Reasoning and Verification: State of the Art and Current Trends

Bernhard Beckert, *Karlsruhe Institute of Technology*

Reiner Hähnle, *Technische Universität Darmstadt*

In this article, the authors give an overview of tool-based verification of hardware and software systems and discuss the relation between verification and logical reasoning.

Over the past few decades, the reach and power of verification methods have increased considerably, and we've seen tremendous progress in the verification of real-word systems. Partly, this is due to methodological advances: since the beginning of this century, the availability of formalisms—including program logics for real-word programming languages—put verification of industrial software within reach. At the same time, suitable theories of abstraction and composition of systems make it possible to deal with complexity. Finally, the availability of efficient satisfiability modulo theories (SMT) solvers has increased verification system performance and automation. SMT solvers provide efficient reasoning capabilities over combinations of theories—including integers, lists, arrays, and bit vectors—which is a ubiquitous subtask of hard- and software verification.

Verification systems are now commercially used to formally verify many industrial applications (see Table 1). Even highly complex system software can be formally verified when sufficient effort is spent, as the L4.verified (www.ertos.nicta.com.au/research/l4.verified) and Verisoft (www.verisoftxt.de) projects demonstrate.

Here, we describe how verification is employed to ensure dependability of real-word systems, and then offer an overview of the various reasoning methods in use today. In keeping with this special issue's theme, we focus on verification scenarios requiring a nontrivial amount of logical reasoning—that is, we don't consider static analyses based on type systems, propagation rules, dependency graphs, and so on. For the same reason, we don't discuss runtime assertion checking. We do place more emphasis on software (rather than hardware) verification, which is now growing and maturing rapidly. If it's still lagging behind hardware applications or hardware-related applications, this is partly because the hardware industry embraced formal methods 20 years ago. Another reason is that less expressive—and hence, decidable—formalisms can be usefully employed to model hardware, while software verification requires more expressive formalisms.

Verification Scenarios

Verification scenarios differ in various ways. The verification target—that is, the formal

1541-1672/14/$31.00 © 2014 IEEE
Published by the IEEE Computer Society

description of the system that's actually being verified—can be an abstract system model (such as an automaton or a transition system); program source code, byte code, or machine-level code; or written in some hardware-description language.

Likewise, the requirement specification—the formal description of the properties to be verified—can take various forms. Specifications can be algorithmic (executable), describing *how* something is to be done, or they can be declarative, describing *what* the (observable) output should look like. They might refer only to the initial and the final state of a system run—that is, to the system's I/O behavior ("if the input is x, then the output is $x + 1$")—or they might refer to the system's intermediate states and outputs ("if in some state the output is x, then in all later states the output must be some y with $y > x$").

Specification Bottleneck

For many years, the term *formal verification* was almost synonymous with *functional verification*. In the past decade, it has become increasingly clear that full functional verification is an elusive goal for almost all application scenarios. Ironically, this became clear through the *advances* of verification technology: with the advent of verifiers that fully cover and precisely model industrial languages and can handle realistic systems, it's finally become obvious just how difficult and time consuming it is to specify real systems' functionality. Not verification but specification is the real bottleneck in functional verification.[1]

Because of this, "simpler" verification scenarios are often used in practice. These relax the claim to universality of the verified properties, thus reducing the complexity of the required specifications, while preserving the verification result's usefulness; examples include verification methods for *finding* bugs instead of proving their absence, and methods that combine verification and testing. Verifying generic and uniform properties also reduces the amount of functional specifications that must be written.

Finally, the problem of writing specifications is greatly alleviated if the specification and the verification target are developed (or generated) in tandem. In contrast, writing specifications for legacy systems is much harder. It's often difficult to extract the required system knowledge from legacy code and its (typically incomplete) documentation. More generally, systems that haven't been designed with verification in mind might not provide an appropriate component structure. Even if they obey principles such as information hiding and encapsulation, their components might not be of the right granularity or might have too many interdependencies.

Table 1. Examples of commercially successful verification systems.

Static Driver Verifier (SDV)	Microsoft's SDV is integrated into Visual Studio and routinely finds bugs and ensures compliance of Windows driver software.
Astrée	This abstract-interpretation-based static analyzer has been used to prove the absence of run-time errors in the primary flight-control software of Airbus planes.
ACL2	The ACL2 theorem prover has been used to verify the correctness of commercial microprocessor systems for high-assurance applications.
HOL Light	The HOL Light system has been used to formally verify various floating-point algorithms implemented in Intel processors.
Pex	This glassbox test generation tool for C# is part of Visual Studio Power Tools.

Handling Complexity

Of course, we must limit the simplification of verification scenarios, lest they become useless. At some point, we must face the complexities of real-world systems. There are two fundamental approaches, which are typically combined, to deal with complex verification targets: abstraction and (de)composition. *Abstraction* considers an abstract model of the verification target that's less complex than the target system itself. *Decomposition* subdivides the verification target into components so that their properties are small enough to be verified separately.

Neither abstraction nor decomposition come for free: a suitable abstract model and suitable components, in turn, must be identified and their properties specified. Both abstraction and decomposition lead to additional sources of errors or additional effort to show that the abstract model is indeed a valid abstraction—that is, that all properties of the abstract model hold for the actual target system. For decomposition, we must show that the components' verified properties imply the desired property for the composed system.

Functional Correctness

To verify a system's functional correctness requires formally proving that all possible system runs satisfy a declarative specification of the system's externally observable behavior. The system must satisfy the specification for all possible inputs and initial system states.

The standard approach is to use contract-based specifications. If the input and the initial state, in which the system is started, satisfy a given precondition, then the system's final state must satisfy a given postcondition, such as, "If the input is nonnegative, then the output is the square root of the input." To handle the frame problem, pre-/postcondition

pairs are often accompanied by a description of which variables (or heap locations) a system is allowed to change (otherwise, you'd have to specify explicitly that all untouched variables remain unchanged).

Pre-/postcondition pairs describe programs' I/O behavior, but they can't specify the behavior in intermediate states. This is problematic if you need to verify the functionality of concurrent or reactive systems, as what such systems do in intermediate states is observable. In addition, such systems aren't necessarily intended to terminate (as with servers, for example). For that reason, extensions of the pre-/postcondition approach let you specify properties of whole traces or histories (all states in a system run) or properties of all the state transitions (two-state invariants).

State-of-the-art verification systems, such as KeY, Why, and KIV, can prove functional correctness at the source-code level for programs written in industrial languages such as Java and C (Table 2 shows further information on many of the verification systems mentioned here). Programs are specified using formalisms that are specific to the target language, such as the Java Modeling Language for Java or the ANSI/ISO C Specification Language (ACSL) and the VCC language for C.

A different approach to functional verification is to formalize both the verification target's syntax and semantics in an expressive logic and formulate correctness as a mathematical theorem. Besides functional verification of specific programs, this permits expressing and proving meta properties such as the target language's type safety. Formalizations exist, for example, for Java and C in Isabelle/HOL.

Although verifying non-trivial systems is possible using today's tools and methods, they need to be decomposed and auxiliary specifications must be created to describe the components' functional behavior. Typically, the amount of auxiliary annotations required is a multiple (up to five times) of the target code to be verified (measured in lines of code).[1]

Safety and Liveness Properties

The verification of safety and liveness properties is closely related to model checking techniques.[2] Typically, the verification target is an abstract system model with a finite state space. The goal is to show that the system never reaches a critical state (safety), and that it will finally reach a desired state (liveness). Specifications are written in variants of temporal logics that are interpreted over state traces or histories. Mostly, the specifications are written in decidable logics (that is, propositional temporal logics, possibly with timing expressions).

Although both the system model and specification use languages of limited expressiveness, the specification bottleneck persists. It can be alleviated using pattern languages and specification idioms for frequently used properties.[3] Even then, however, model checking for safety and liveness properties is far from an automatic or push-button verification scenario. Often, problems need careful reformulation before model checkers can cope with them.

Lately, there has been growing interest in the verification of safety and liveness properties for hybrid systems,[4] and various methods and tools—such as HyTech and KeYmaera—have been developed for that purpose. Hybrid systems have discrete as well as continuous state transitions, as is typical for cyber-physical systems, automotive and avionics applications, robotics, and so on. An important instance of hybrid automata are *timed automata*, in which the continuous variables are clocks representing the passing of time.[5]

Refinement

Refinement-driven verification begins with a declarative specification of the target system's functionality. This is, for example, expressed in typed first- or higher-order logic plus set theory. In a series of refinement steps, the specification is gradually turned into an executable system model. Provided that each refinement step preserves all possible behaviors, the final result is guaranteed to satisfy the original specification.

The approach's main difference from functional verification is that the refinement spans more levels and starts at the most abstract level. For nontrivial systems, dozens of refinement steps might be necessary. The advantage of more levels is that the "distance" between adjacent levels is smaller than that between the specification and target system in functional verification. Hence, the individual steps in refinement-driven verification tend to be easier to prove. To ensure correctness, only certain kinds of refinement are permitted and each refinement step must be accompanied by a proof that behavior is preserved.

Using many refinement levels can easily lead to an excessive effort for specification and proving. To alleviate this, refinement-based methods often work with patterns and libraries and, for this reason, work best in specific application domains. For example, Event-B is optimized for reactive systems while Specware is used to develop transport schedulers.

In a refinement-based scenario, it's important to always co-construct the multilevel specification and the target system. This avoids problems related to verifying legacy systems and is an important reason for the viability of refinement-based methods.

In addition to systems that refine from an abstract specification down all the way to executable code, there are methods and systems—such as the Alloy Analyzer—that relate different abstract model levels to each other. This creates less complex models and proofs, as it doesn't consider platform- and implementation language-specific details. On the other hand, it can't uncover errors that involve those details.

Uniform, Generic, and Lightweight Properties

Using generic or uniform specifications can reduce the need to write requirement specifications. Rather than describing the specific functionality of the target system, these specifications express only properties that are desirable for a general class of system. In addition to reducing the specification overhead for individual systems, this allows the use of simpler and less-expressive specification languages. An important class of generic properties is the absence of typical errors—such as buffer overflows, null-pointer exceptions, and division by zero. In the case of SDV, a set of general properties was devised such that a device driver satisfying these properties can't cause the operating system to crash. This is possible, because the ways in which a driver might crash the OS are generally known and don't depend on a particular driver's functionality.

Simple, lightweight properties can be formalized using (Boolean) expressions of the target programming language without the need for quantifiers or higher-order logic features. Systems such as Spec# and CBMC allow the verification of lightweight properties that have been added as assertions to the target program. Verification of lightweight properties succeeds in many cases without auxiliary specifications.

Further, *non-functional* properties can often be specified in a uniform way even if they aren't completely generic. This includes limits on resource consumption such as time, space, and energy. A further example concerns security properties. A verification target might be forbidden to call certain methods, or information-flow properties might be specified to ensure that no information flows from secret values to public output.

An important variation of the generic-property scenario is *proof-carrying code* (PCC), in which code that's downloaded from an untrusted source—such as an applet downloaded from an untrusted website—is accompanied by a verification proof. The host system can check that proof before running the code to ensure that the code satisfies the host's security policies and has other desirable properties. The PCC scenario requires a predefined set of properties be shared by the host and the untrusted source.

Relational Properties

Relational properties don't use declarative specifications, but rather relate different systems, different versions of the same system, or different runs of the same system to each other.

Typically, the verified relation between systems is functional—examples include a simulation relation (one system is a refinement of the other) or bisimulation (both systems exhibit the same behavior)—which corresponds to compiler correctness. Another example of a relational property is noninterference: If it's provable that any two system runs that differ in the initial value of some variable x result in the same output, then the variable x doesn't interfere with the output (the system doesn't reveal information about the initial value of x).

Verifying relational properties avoids the bottleneck of having to write complex requirement specifications. However, verification might still require complex auxiliary specifications that describe the functionality of subcomponents or detail the relation between the two systems (coupling invariants).

Bug Finding

The bug-finding scenario's concept is to give up on formal verification's claim to universality. One variation on this theme is to use failed proof attempts to generate bug warnings. If a verification attempt fails because some subgoals can't be proven, then instead of declaring failure, the verification system gives warnings to the user that are extracted from the open subgoals. These warnings indicate that a problem might exist at the points in the verification target related to the open subgoals. If the subgoals could not be closed due to missing auxiliary specifications or a time-out, even though in fact a proof exists, *false positives* are produced. If this scenario is to be useful, there can't be too many spurious warnings. To ensure this, some systems (such as the Extended Static Checker for Java) also give up on soundness—that is, they don't show all possible warnings.

A second variation on bug finding is to prove correctness for only part of the program runs and inputs. So, if the verification succeeds, it indicates the absence of errors in many—but not all—cases. On the other hand, if a verification attempt fails with a counter example (and not just a time out), then the counter example indicates a bug in the verification target (or the specification) and, moreover, describes when and how the bug makes the system fail.

One example of the latter approach is *bounded* verification: imposing a finite bound on the domains of system variables or on the number of execution steps in the target system, which yields relative verification results that

hold only up to the chosen bound. Bounded verification reduces the need for decomposition—and thus the need to write auxiliary specifications, such as contracts for subcomponents and loop invariants. In particular, loop invariants aren't needed, as they can be considered as induction hypotheses for proving (by induction) that the loop works for all numbers of required loop iterations. Because the number of loop iterations is bounded, no induction is needed.

A further use of verification for bug finding is to enhance the debugging process using verification technology based on symbolic execution to implement symbolic debuggers. Such symbolic debuggers cover all possible execution paths, and there's no need to initialize input values.

Test Generation

Verification and testing are different approaches to improving software dependability that can both complement and support each other. Verification methods can be used to help with testing in several scenarios. For example, verification methods such as symbolic execution can generate tests from the specification and the source code (glass-box testing) or from the specification of the verification target alone (black-box testing). Using reasoning techniques, it's possible to generate tests that exercise particular program paths, satisfy various code coverage criteria, or cover all disjunctive case distinctions in the specification.

Testing goes *beyond* verification in an important dimension: verification ensures correctness of the target system, but not of the runtime environment or the compiler backends; testing, however, can also find bugs that are located outside the target system itself. For this reason, testing can't be replaced by verification in all cases.

Verification Methods

Most verification approaches fall into one of four methodologies: deductive verification, model checking, refinement and code generation, and abstract interpretation. We'll now introduce some dimensions that are useful for classifying these approaches and we discuss how they influence the type of reasoning that occurs during verification.

Arguably, the main tradeoff that influences a verification method's design is automation of proof search versus expressiveness of the logic formalism used for specification and reasoning. Most verification systems use a logic-based language to express properties. Common logics, ordered according to their expressiveness, include propositional temporal logic, finite-domain first-order logic (FOL), quantifier-free FOL, full FOL, FOL plus reachability or induction schemata, dynamic logic, higher-order logic, or set theory. The expressiveness of a logic and the computational complexity of its decision problems are directly related. For undecidable languages, such as first-order logic, full automation can't be expected; yet even for decidable languages, such as temporal logic, problems quickly become infeasible as the target system's size grows.

However, there's a difference between the theoretical complexity of the decision problem of a logic and the efficiency/effectiveness of provers in practice. In reality, typical instances of undecidable problems are hard—but not impossible—to solve. Theory tells us that, in some instances, either no (finite) proof or no counter example exists (otherwise the problem would be decidable). But in practice, such problem instances are few and far between. Even for undecidable problems, the real difficulty is to find—existing—proofs.

Verification methods that use abstraction can take different forms: while abstract interpretation attempts to find a sound abstraction of the target system, for which the desired properties are still provable, in model checking, the users typically work with an abstract system model from the start, and might have to refine and adapt it many times during the verification process. As we show later, it's fruitful to combine both approaches.

Another dimension of verification method design is verification workflow, which is heavily influenced by expressiveness: assuming a decidable modeling language and a feasible target system size, it's possible to automatically verify a system provided that the specified property actually holds, that the verifier is suitably instrumented, and that the system is suitably modeled. This approach, typically realized in model checking,[2] enables a *batch mode* workflow (often mislabeled as "push button" verification) based on cycles of failed verification attempt and failure analysis, followed by modifications to the target system, specification, or instrumentation, until a verification attempt succeeds.

Verification systems for expressive formalisms (first-order logic and beyond) require often more fine-grained *human interaction*, where a user gives hints to the verifier at certain points during an attempted proof. Such hints could be quantifier instantiations or auxiliary specifications, such as loop invariants or induction hypotheses.

Finally, a further distinction is the verification method's precision—that is, whether it might yield false positives[6] (and if so, to what extent).

Deductive Verification

Under deductive verification, we subsume all verification methods that use an expressive (at least first-order) logic to state that a given target system is correct with respect to some property. Logical reasoning (deduction) is

then used to prove validity of such a statement. Perhaps the best-known approach along these lines is Hoare logic,[7] but it represents only one of three possible architectures.

The most general deductive verification approach is to use a highly expressive logical framework, typically based on higher-order logic with inductive definitions. Such logics permit the definition of not only properties, but also the target language's abstract syntax and semantics. In so-called proof-assistants, such as HOL and Isabelle, real-life languages of considerable scope have been modeled in this manner, including, for example, the floating point logic of x86 processors, a nontrivial fragment of the Java language, the C language, and an OS kernel.

A second deductive verification approach is provided by program logics, where a fixed target language is embedded into a specification language. The latter is usually based on first-order logic, and target language objects occur directly as part of logical expressions without encoding. The target language's semantics is reflected in the calculus rules for the program logic. For example, the task to prove that a program "if (B) Q else R; S" is correct relative to a pre-/postcondition pair is reduced to prove correctness of the two programs "Q;S" and "R;S," respectively, where *additional* assumptions that the path condition B respectively holds and doesn't hold, are added to the precondition (we assume that B's execution has no side effects). Typically, at least one such proof rule exists for each syntactic element of the target language. Such calculi have been implemented for functional languages (in ACL2 and VeriFun), as well as for imperative programming languages (in KeY and KIV).

Hoare logic[7] is a representative of a third architecture: here, a set of rewrite rules specifies how first-order correctness assertions about a given target system are reduced to purely first-order verification conditions using techniques such as weakest precondition reasoning. For example, if an assertion P holds immediately after an assignment "$x = e$;", then this is propagated to the assertion $P(x/e)$ (denoting P where all occurrences of x are replaced with e) that must hold just before the assignment. This approached, called verification condition generator (VCG) architecture, is realized, for example, in Dafny and Why.

All three architectures need numerous and detailed auxiliary specifications, including loop invariants and/or induction hypotheses; all three can also be used for proving functional correctness of systems. Due to their general nature and their expressiveness, proof assistants for higher-order logic tend to require more user interaction than the other two. However, in the past years, external automated theorem provers are increasingly employed to decrease the amount of required interactions. To make this work, it's necessary to translate between first- and higher-order logic—hence, a loosely coupled system architecture is used and the granularity (complexity) of problems handed over to external reasoners tends to be large.

An interesting fact is that the designers of all verifiers that use a dedicated program logic felt the need to add sophisticated first-order reasoning capabilities to their systems, starting with the seminal work by Robert S. Boyer and J. Strother Moore in the predecessor of the ACL2 system.[8] Adding these capabilities was necessary because mainstream automated reasoning systems for first-order logic lacked central features required for verification, such as types, heuristic control, and induction. The coupling of these "internal reasoners" is tight, so that intermediate results can be constantly simplified without translation overhead (fine problem granularity). The downside is that internal reasoners are difficult to use independently of their host systems and their internal workings are not typically well documented.

In contrast to logical frameworks and program logics, VCG systems admit workflow in batch mode: in the first phase, a verification problem is reduced to a (typically very large) number of first-order queries. These are then solved by external reasoners, which are often run competitively in parallel. The advantage is a modular architecture that can exploit the latest progress in automated reasoning technology. The disadvantage is that it can be difficult to relate back the failure of proving a verification condition to its root cause. It's also hard to implement aggressive simplification of intermediate results.

Model Checking

In model checking,[2] the execution model of a soft- or hardware system is viewed as a finite transition system—that is, as a state automaton whose states are propositional variable assignments. Because finite transition systems are standard models of propositional temporal logic, to check that a finite transition system T is a model of a temporal logic formula P means to ensure that every possible system execution represented by T meets the property expressed with P. Hence, model checking can be used for system verification.

The bottleneck is the explosion of the number of possible states that occurs even for small systems when an explicit representation of states is chosen. Since the mid-1980s, enormous progress has been made in state representation that, in many cases, is able to avoid state explosion. First, encodings based on binary decision diagrams (BDDs)[9] made vast

improvements possible, later Buechi automata, symmetry reduction, abstraction refinement, modularization, and many other techniques pushed the boundaries.[2] Many of these are implemented in the widely used SPIN and NuSMV model checkers. Systems such as UPPAAL extended temporal logic with timing conditions and can be used to model real-time systems.

Traditionally, automata-based techniques and efficient data structures to represent states played a much more prominent role in model checking than logical reasoning. This is about to change, as the model checking community strives to overcome the standard approaches' fundamental limitation to finite state systems. To go beyond the finite state barrier (or simply deal with finite but large systems), several techniques have been suggested, including sound abstraction (related to abstract interpretation, which we discuss later) and abstraction with additional checks, as well as incomplete approaches, such as bounded model checking.[10] Yet another possibility is offered by symbolic execution engines, which enumerate reachable states without loss of precision; examples include KeY, VeriFast, Java PathFinder, and Bogor. The logic-based techniques for infinite state representation realized in the latter systems use automated reasoning to bound state exploration.[11] We expect the combination of ideas from deductive verification and model checking to enable further advances in the coming years.

Lately, there has been a trend to subsume verification tools and methods that use reasoning technology, such as SMT and propositional satisfiability (SAT) solving under the term "model checking"; an example of such a system is CBMC (see Table 2). Here, we use the term "model checking" in a narrower sense and consider this other type of system under the "deductive verification" heading.

Refinement and Code Generation

Verification can also be achieved by gradually refining an initial system model (that directly reflects the requirements) into an executable model, provided that each refinement step preserves the properties of the preceding one. Declarative and highly nondeterministic concepts, conveniently expressed in set theory, must be refined into operational ones. For example, there might be a proof obligation relating a set comprehension to an iterator. Hence, refinement over multiple levels for nontrivial systems creates a large number of proof obligations about set theory.

Evidently, most proof obligations generated during refinement-based verification can be discharged with automated theorem provers, yet there's been surprisingly little interaction with the automated-reasoning community. This can be partly explained by a mismatch of requirements: the support for set-theoretic reasoning in mainstream automated reasoning tools is limited. One industrially successful system, Specware, uses higher-order logic, and thus outsources the discharging of proof obligations to Isabelle, but not to first-order provers.

Almost no work exists in the verification community regarding code generation by compilation and optimization of executable, yet abstract system models. Of course, there's an abundance of model-driven software development approaches. However, most of the involved notations (such as UML) are not rigorous enough to permit formal verification. The same is true for code generation from languages such as MathWorks, SystemC, VHDL, and Simulink, although the Scoot system (www.cprover.org/scoot) can extract abstract models from SystemC. Recent research has shown that deductive verification of relational properties is a promising approach to ensure correct compilation and optimization.[12] We believe that provably correct (behavior-preserving) code generation offers vast opportunities for the reasoning and formal verification communities to employ their techniques.

Abstract Interpretation

Abstract interpretation[13] is a method to reason soundly, in finite domains, about potentially infinite state systems. The idea can be simply stated: in the target system, all variables are interpreted not over their original domain (that is, their type), but over a more abstract, smaller one. For example, an integer variable might have only the values "positive," "0," "negative," "non-positive," "non-negative," and "anything." Of course, all operations also must be replaced by operations over the abstract domain, for example, "positive" + "non-negative" yields "positive" and so on. The abstract domains and operations are chosen so as to preserve the semantics: if a property holds in the abstract system, then it must also hold in the original system.

If the abstract domain is finite (or at least has no infinite ascending chains), it's possible to show that any computation in the abstract system must finitely terminate, because loops and recursive calls reach a fixpoint after finitely many steps. The price to pay, of course, is a loss of precision and completeness: not all properties of interest might be expressible in the abstract domain and, even if they are, a property that holds for the actual system might cease to hold in its abstraction.

Reasoning in connection with abstract interpretation means constraint solving in specific abstract domains. However, because abstract interpretation can be seen as a very general method to render infinite computations finite in a sound manner, it's

natural to combine it with precise verification methods. This has been done since the late 1990s with model checking, notably in counter-example guided abstraction and refinement (CEGAR), where a suitable system abstraction is computed incrementally.[14] It's less known that symbolic program execution can be seen as abstract interpretation, which makes it possible to put sound abstraction on top of verification systems based on symbolic execution. The KeY system has realized this, and allows the exploitation of synergies between abstract interpretation-style constraint solving and deductive verification-style logical reasoning.[15]

Trends and Opportunities

We now offer a brief discussion of the main trends and opportunities for reasoning in the verification context. Our 10 key conclusions are also shown in the related sidebar.

Non-functional Properties

From the somewhat sobering insight that full functional verification is too expensive for most application scenarios due to both general difficulties and the effort required in achieving functional specification, new opportunities have arisen: *non-functional* properties of systems—such as resource (including energy) consumption or security properties—can often be schematically specified. Often, the required specifications (including invariants) can be automatically generated.[16]

This is a great opportunity for the verification community: whereas functional verification is rarely requested by industry and likely to remain a niche for high-assurance applications, non-functional properties are extremely relevant in everyday scenarios and can easily be mapped to business cases, including quality-of-service parameters such as response time or resource consumption in cloud applications.[17]

10 Key Conclusions

Based on our analysis, we have 10 key conclusions.

1. Given enough time and effort, current technology permits the formal verification of even highly complex systems.
2. The main bottleneck of functional verification is the need for extensive specifications.
3. Verification of complex systems is never automatic or "push-button."
4. Verification of non-functional properties alleviates the specification problem and is of great practical relevance.
5. Verification, bug finding, and test generation are not alternatives, but rather complement each other: all are essential.
6. Abstraction and compositional verification are key to handling complexity in verification.
7. Model-centric software development and code generation account for huge opportunities in verification and are under-researched.
8. There's a convergence of finite-state/abstract methods (model checking, abstract interpretation) and infinite state/precise methods (deductive verification, refinement).
9. Verification, SMT solving, and first-order automated reasoning form a virtuous cycle in extending the reach of verification technology.
10. There are many scenarios and variations of verification, which makes different systems hard to compare; and there's no single best verification tool

Method Convergence

From our discussion of verification methods, it's clear that there's much to be gained from a closer collaboration of the various subcommunities. Here, we offer two examples. First, to verify large industrial systems, it's necessary to use both methods optimized for finite state systems (such as model checking) and methods for infinite state systems (such as deductive verification). Abstract interpretation and symbolic execution seem to be natural bridges. Second, compilation, code generation, and code simplification are neglected areas in verification. There's a vast opportunity for verification in correct code generation from modeling languages such as Simulink and SystemC. Although first steps have been made,[18] this is (so far) a missed opportunity because existing methods and tools in deductive verification can well be applied here.

The Importance of Reasoning

The advent of efficient SMT solvers has given a boost to verification system performance. SMT solvers combine efficient theory reasoning over variable-free expressions with heuristically driven quantifier instantiation. Importantly, they can also detect counter examples for invalid problems. Similar techniques had been implemented as part of monolithic verifiers such as ACL2 or KIV for decades, but standalone SMT solvers are much easier to maintain, and they also benefit from progress in SAT solving. As a consequence, there's currently the happy situation that the verification and SMT solving communities drive each other's research. With some delay, this opportunity has also been grasped by the first-order theorem proving community, as is witnessed by recent events such as the Dagstuhl Seminar 13411 on deduction and arithmetic, as well as the rise of theorem-proving methods that can create counter examples, such as instantiation-based proving.

One challenge that current verification approaches barely address is how to deal with verification target changes. During system development and maintenance, such changes

Table 2. An overview of reasoning and verification systems. (This table may serve as a starting point for further exploration. It contains a representative selection of systems that were historically influential or represent the state of the art.)

System/URL	Method	Verification scenario
Alloy Analyzer alloy.mit.edu/alloy	Refinement, deductive verification	Functional correctness, safety properties
ACL2 www.cs.utexas.edu/users/moore/acl2	Deductive verification (interactive)	Functional correctness, bug finding
Astrée www.astree.ens.fr	Static analysis	Safety properties, generic properties
Bogor bogor.projects.cis.ksu.edu	Model checking	Safety properties
CBMC www.cprover.org/cbmc	Deductive verification	Bug finding, lightweight properties
Coq www.lix.polytechnique.fr/coq	Proof assistant (interactive)	Functional correctness, safety, security properties, refinement relations
Dafny research.microsoft.com/projects/dafny	Deductive verification (batch)	Functional correctness, bug finding
ESC/Java www.kindsoftware.com/products/opensource/ESCJava2	Deductive verification	Bug finding
Event-B www.event-b.org	Deductive verification	Refinement
Frama C/Why frama-c.com	Deductive verification (batch)	Functional correctness, bug finding
HyTech embedded.eecs.berkeley.edu/research/hytech	Model checking	Safety properties of hybrid automata
Isabelle isabelle.in.tum.de	Proof assistant (interactive)	Functional correctness, safety, security properties, refinement relations
Java Pathfinder babelfish.arc.nasa.gov/trac/jpf	Model checking	Safety properties
KeY System www.key-project.org	Deductive verification (interactive)	Functional correctness, bug finding, security properties
KeYmaera symbolaris.com/info/KeYmaera.html	Deductive verification (interactive)	Safety and liveness properties of hybrid automata
KIV www.informatik.uni-augsburg.de/lehrstuehle/swt/se/kiv	Deductive verification (interactive)	Functional correctness, bug finding, security properties
NuSMV nusmv.fbk.eu	Model checking	Safety properties
Pex research.microsoft.com/projects/pex	Deductive verification	Test-case generation
PVS pvs.csl.sri.com	Proof assistant (interactive)	Functional correctness, safety, security properties, refinement relations
Spec# research.microsoft.com/projects/specsharp	Deductive verification	Bug finding, lightweight properties
Specware www.specware.org	Deductive verification	Refinement
SPIN spinroot.com	Model checking	Safety properties
TVLA www.cs.tau.ac.il/~tvla	Abstract interpretation	Safety properties, functional verification
UPPAAL www.uppaal.org	Model checking	Safety/liveness properties of temporal automata
VeriFast people.cs.kuleuven.be/~bart.jacobs/verifast	Deductive verification (batch)	Functional correctness
VeriFun www.verifun.org	Deductive verification (batch), induction proofs	Functional correctness
VCC research.microsoft.com/projects/vcc	Deductive verification (batch)	Functional correctness, bug finding

are normal and occur frequently, triggered by feature requests, environment changes, refactoring, bug fixes, and so on. Any change in the target system has the potential to completely invalidate the expended verification effort. If re-verification is expensive, this constitutes a major threat against the practical usefulness of all but fully automatic and lightweight verification methods. One solution might be verification methods that are aware of changes,[19] which lets automated reasoning replace re-verification, especially in those parts of a system that remain unchanged.

The future looks bright for the collaboration of verification and reasoning. Recent advances in both fields and increasingly tight interaction have already given rise to industrially relevant verification tools. We predict that this is only the beginning, and that within a decade tools based on verification technology will be as useful and widespread for software development as they are today in the hardware domain. ▬

Acknowledgment

We thank the anonymous reviewers for their careful reading of this article and numerous valuable suggestions for improvement.

References

1. C. Baumann et al., "Lessons Learned from Microkernel Verification: Specification Is the New Bottleneck," *Proc. 7th Conf. Systems Software Verification,* 2012, pp. 18–32.
2. E.M. Clarke, O. Grumberg, and D.A. Peled, *Model Checking*, MIT Press, 1999.
3. J.C. Corbett et al., "A Language Framework for Expressing Checkable Properties of Dynamic Software," *Proc. 7th Int'l SPIN Workshop Stanford*, vol. 1885, 2000, pp. 205–223.
4. A. Platzer, *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*, Springer, 2010.
5. R. Alur and D.L. Dill, "A Theory of Timed Automata," *Theoretical Computer Science*, vol. 126, no. 2, 1994, pp. 183–235.
6. C. Flanagan et al., "Extended Static Checking for Java," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 2002, pp. 234–245.
7. C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," *Comm. ACM*, vol. 12, no. 10, 1969, pp. 576–580.
8. R.S. Boyer and J.S. Moore, *A Computational Logic Handbook*, Academic Press, 1988.
9. R.E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams," *ACM Computing Surveys*, vol. 24, no. 3, 1992, pp. 293–318.
10. A. Biere et al., "Symbolic Model Checking without BDDS," *Tools and Algorithms for the Construction and Analysis of Systems,* W.R. Cleaveland, ed., LNCS 1579, Springer, 1999, pp. 193–207.
11. B. Beckert and Daniel Bruns, "Dynamic Logic with Trace Semantics," *Proc. Int'l Conf. Automated Deduction,* LNCS 7898, Springer, 2013, pp. 315–329.
12. R. Ji, R. Hähnle, and R. Bubel, "Program Transformation Based on Symbolic Execution and Deduction," *Proc. 11th Int'l Conf. Software Eng. and Formal Methods*, LNCS 8137, Springer, 2013, pp. 289–304.
13. P. Cousot and Radhia Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," *Proc. 4th ACM Symp. Principles of Programming Language,* 1997, pp. 238–252.
14. E.M. Clarke et al., "Counterexample-Guided Abstraction Refinement," *Proc. 12th Int'l Conf. Computer Aided Verification,* LNCS 1855, Springer, 2000, pp. 154–169.
15. R. Bubel, R. Hähnle, and B. Weiss, "Abstract Interpretation of Symbolic Execution with Explicit State Updates," *Proc. 6th Int'l Symp. Formal Methods for Components and Objects*, LNCS 5751, Springer, 2009, pp. 247–277.
16. E. Albert et al., "Verified Resource Guarantees Using COSTA and Key," *Proc. ACM SIGPLAN 2011 Workshop Partial Evaluation and Program Manipulation,* 2011, pp. 73–76.
17. E. Albert et al., "Engineering Virtualized Services," *Proc. 2nd Nordic Symp. Cloud Computing and Internet Technologies (Nordicloud)*, 2013, pp. 59–63.
18. N. Harrath, B. Monsuez, and K. Barkaoui, "Verifying SystemC with Predicate Abstraction: A Component Based Approach," *Proc. 14th Int'l Conf. Information Reuse & Integration,* 2013, pp. 536–545.
19. R. Hähnle, I. Schaefer, and R. Bubel, "Reuse in Software Verification by Abstract Method Calls," *Proc. 24th Conf. Automated Deduction,* LNCS 7898, Springer, 2013, pp. 300–314.

THE AUTHORS

Bernhard Beckert is a professor of computer science at the Karlsruhe Institute of Technology (KIT), Germany. His research interests include formal specification and verification, security, and automated deduction. Beckert has a PhD in computer science from the University of Karlsruhe (now KIT). Contact him at beckert@kit.edu.

Reiner Hähnle is a professor of computer science at Technische Universität Darmstadt, Germany. His research interests include formal verification, formal modeling, automated debugging, and cloud computing. Hähnle has a PhD in computer science from University of Karlsruhe (now KIT). Contact him at haehnle@cs.tu-darmstadt.de.