Reasoning and Verification

– State of the Art and Current Trends –

Bernhard Beckert and Reiner Hähnle

Abstract

In this article we give an overview of tool-based verification of hard- and software systems and discuss the relation between verification and logical reasoning. By verification we mean reasoning-based methods to establish dependability. This is not restricted to proofs of functional correctness but includes also other scenarios such as test generation or bug finding. We describe the main verification scenarios and methods that are in usage today and the extent to which they depend on logical reasoning. From this discussion we distill current trends and new opportunities for the interaction between verification and reasoning.

Keywords: Software/program verification, hardware verification, reasoning about programs, deduction and theorem proving.

Introduction

Over the last decades the reach and power of verification methods has increased considerably, and there has been tremendous progress in the verification of real-word systems.

Partly, this is based on methodological advances: since the beginning of this century, formalisms—including program logics for real-word programming languages—that put verification of industrial software within reach became available. At the same time, suitable theories of abstraction and composition of systems make it possible to deal with complexity. Finally, increased performance and the degree of automation of verification systems is also due to the availability of efficient SMT (satisfiability modulo theories) solvers. These provide efficient reasoning capabilities over combinations of theories—including integers, lists, arrays, bit vectors, etc.—, which is an ubiquitous sub-task of hard- and software verification.

Verification systems are now commercially used in industrial applications (see Table 1). Even highly complex system software can be formally verified when sufficient effort is spent, as is demonstrated in the L4.verified[1] and Verisoft[2] projects.

In this article we first describe the main scenarios of how verification is employed to ensure dependability of real-word systems. Then we give an overview of the various

[1] `www.ertos.nicta.com.au/research/l4.verified`

[2] `www.verisoftxt.de`

Table 1: Examples of commercially successful verification systems

SDV Microsoft's Static Driver Verifier	SDV is integrated into Visual Studio and routinely used to find bugs and ensure compliance of Windows driver software
Astrée Abstract interpretation-based static analyzer	Astrée 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 was used to formally verify correctness of commercial micro processor systems for high-assurance applications
HOL Light	Various floating-point algorithms implemented in Intel processors have been formally verified with the HOL Light system
Pex	Pex is a glassbox test generation tool for C# and part of Visual Studio Power Tools

reasoning methods that are in use today. We conclude with an outlook on current trends in the area of verification. Table 2 summarizes the main results of our analysis.

To stay with the theme of this special issue, we focus on verification scenarios requiring a non-trivial amount of logical reasoning, i.e., we do not consider static analyses based on type systems, propagation rules, dependency graphs, etc. For the same reason we do not discuss runtime assertion checking. There is also a certain emphasis on software (rather than hardware) verification, which is now growing and maturing rapidly. If it is still lagging behind applications in hardware or close to hardware, this can be partly explained by the fact that the hardware industry embraced formal methods already 20 years ago. Another reason is that less expressive and hence decidable formalisms can be usefully employed to model hardware, while more expressive formalisms are needed for software verification.

Verification Scenarios

The Overall Picture

Different verification targets and specifications Verification scenarios differ in various ways. The verification target, i.e., the formal description of the system that is actually being verified, can be an abstract system model (e.g., an automaton or a transition system), it can be program source code, byte code, or machine-level code, or it can be written in some hardware-description language. Likewise, the requirement specification, i.e., 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 may refer only to the initial and the final state of a system run, i.e., the system's input/output behavior ("if the input is x, then the output is $x+1$"), or they may 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 is the bottleneck For many years the term *formal verification* was almost synonymous with *functional verification*. In the last decade it became more and more 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 that can handle realistic systems, it finally became obvious just how difficult and time consuming the specification of functionality of real systems is. 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 usefulness of the verification result. Examples are verification methods for *finding* bugs instead of for proving their absence or methods for the combination of verification and testing. Verifying generic and uniform properties also reduces the amount of functional specifications that need to 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 is often difficult to extract the required system knowledge from legacy code and its (typically incomplete) documentation. More generally, systems that have not been designed with verification in mind may not provide an appropriate component structure. Even if they obey principles such as information hiding and encapsulation, their components may not be of the right granularity or may have too many interdependencies.

Different ways of handling complexity Of course, there is a limit to simplification of verification scenarios lest they become useless. At some point, one has to face the complexities of real-world systems. There are two fundamental approaches to deal with complex verification targets (typically used in combination): abstraction and (de-)composition. Abstraction means to consider an abstract model of the verification target that is less complex than the target system itself. Decomposition means that the verification target is subdivided into components that are small enough, such that their properties can be verified separately.

Neither abstraction nor composition come for free: a suitable abstract model, respectively, suitable components must be identified and their properties specified. Both, ab-

straction and composition, lead to additional sources of errors or to additional effort to show that an abstract model indeed is a valid abstraction, i.e., that all properties of the abstract model hold for the actual target system. For composition, one has to show that the verified properties of the components imply the desired property for the composed system.

Functional Correctness

To verify functional correctness of a system requires formally proving that all possible runs of the system satisfy a declarative specification of what the system is supposed to do, i.e., of its externally observable behaviour. 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 pre-condition, then the system's final state must satisfy a given post-condition ("if the input is non-negative, then the output is the square root of the input"). To handle the frame problem, pre-/post-condition pairs are often accompanied by a description of which variables (or heap locations) a system is allowed to change (otherwise one would have to specify explicitly that all untouched variables remain unchanged).

Pre-/post-condition pairs describe the input/output behaviour of programs. They cannot specify the behavior in intermediate states. This is problematic if the functionality of concurrent or reactive systems is to be verified, as it is observable what such systems do in intermediate states. In addition, such systems are not necessarily intended to terminate (servers, for example). For that reason, extensions of the pre-/post-condition approach allow to 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, or KIV[3], can prove functional correctness at the source-code level for programs written in industrial languages such as Java and C. 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 syntax and the semantics of the verification target 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 type safety of the target language. Formalisations exist, for example, for Java and C in Isabelle/HOL.

As explained above, full functional verification is limited by the specification bottleneck. Non-trivial systems need to be decomposed to handle their complexity. The components are

[3]Further information on all the verification systems mentioned here is summarized in Table 3. For space reasons we cannot list all extant systems, but give a representative selection of systems that were historically influential and/or that represent the state of the art.

then verified separately (typical components are individual functions, methods, or classes). This is possible using the tools and methods available today, but auxiliary specifications need to be created that describe the functional behaviour of the components. Typically, the amount of auxiliary annotations needed is a multiple (up to about 5 times) of the target code to be verified (measured in lines of code) [1].

Safety and Liveness Properties

This verification scenario 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 the system 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 (i.e., propositional temporal logics, possibly with timing expressions).

Though both the system model and the specification use languages of limited expressivity, the specification bottleneck persists. It can be alleviated by using pattern languages and specification idioms for frequently used properties (see, e.g., [3]). But even then, model checking safety and liveness properties is far from being an "automatic" or a "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 have been developed for that purpose (e.g., HyTech, KeYmaera). Hybrid systems have discrete as well as continuous state transitions, as is typical for cyber-physical systems, automotive and avionics applications, robotics, etc. An important instance of hybrid automata are timed automata, where the continuous variables are clocks representing the passing of time [5].

Refinement

Refinement-driven verification starts out with a declarative specification of the functionality of the target system. This is, for example, expressed in typed first-/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 main difference to functional verification is that the refinement spans more levels and starts at the most abstract level. For non-trivial systems, dozens of refinement steps might be necessary. The advantage of the higher number of levels is that the "distance" between adjacent levels is smaller than that between 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 (the required property is often called the *coupling invariant*).

The use of 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 has been used to develop transport schedulers.

In a refinement-based scenario, one always co-constructs the multi-level specification and the target system. This avoids the problems related to verifying legacy systems and is an important reason for the viability of refinement-based methods.

Besides systems that refine from an abstract specification down all the way to executable code, there are methods and systems for relating different model levels to each other that are all abstract (e.g., the Alloy Analyzer). This leads to less complex models and proofs as platform- and implementation language-specific details are not considered. On the other hand, errors that involve such details cannot be uncovered.

Uniform, Generic and Light-weight Properties

The need to write requirement specifications can be reduced by using generic or uniform specifications. These specifications do not describe the specific functionality of the target system but only express properties that are desirable for a general class of system. Besides reducing the amount of 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, division by zero etc. In the case, of SDV (see Table 1), a set of general properties was devised such that a device driver satisfying these properties, cannot cause the operating system to crash. This is possible, because the ways in which a driver may crash the operating system are known in general and do not depend on a particular driver's functionality.

Simple, "light-weight" properties can be formalised 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 light-weight properties that have been added as assertions to the target program. Verification of light-weight properties succeeds in many cases without auxiliary specifications.

Non-functional properties can often be specified in a uniform way even if they are not completely generic. This includes limits on resource consumption such as time, space, and energy. A further example concerns security properties. A verification target may be forbidden to call certain methods, or information-flow properties may 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), where code that is downloaded from an untrusted source (e.g., an applet downloaded from

an untrusted web site) is accompanied by a verification proof. That proof can be checked on the host system before running the code to ensure that the code satisfies the host's security policies and has other desirable properties. The PCC scenario requires that a predefined set of properties exists that is shared by the host and the untrusted source.

Relational Properties

Relational properties do not use declarative specifications but 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 such as a simulation relation (one system is a refinement of the other) or bisimulation (both systems exhibit the same behaviour), which corresponds to compiler correctness. Another example of a relational property is non-interference: If it is provable that any two runs of a system that differ in the initial value of some variable x result in the same output, then consequently the variable x does not interfere with the output (the system does not reveal information about the initial value of x).

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

Bug Finding

The idea of the bug-finding scenario is to give up on the claim to the universality of verification.

One variation on this theme is to use failed proof attempts to generate bug warnings. If a verification attempt fails because some sub-goals cannot be proved, then instead of declaring failure, one gives warnings to the user that are extracted from the open sub-goals. These warnings indicate that there may be a problem at the points in the verification target related to the open sub-goals. In case the sub-goals could not be closed due to missing auxiliary specifications or a time-out, even though in fact a proof exists, *false positives* are produced. It is important for the usefulness of this scenario that not too many spurious warnings are created. To do so, some systems (e.g., ESC/Java) give also up on soundness, i.e., they do not show all possible warnings.

A second variation on bug finding is to not prove correctness for all runs of the program and all inputs. Then, if the verification succeeds, this only indicates the absence of errors in many but not in 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 instance of the latter approach is *bounded* verification, i.e., imposing a finite bound

on the domains of system variables or on the number of execution steps of 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 sub-components and loop invariants. In particular, loop invariants are not needed as they can be considered to be induction hypotheses for proving (by induction) that the loop works for all numbers of required loop iterations. Since 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 by using verification technology that is based on symbolic execution to implement symbolic debuggers. Such symbolic debuggers cover all possible execution paths, and there is no need to initialise input values.

Test Generation

Verification and testing are different approaches to improve the dependability of software that can both complement and support each other. There are several scenarios where verification methods can be used to help with testing.

For example, verification methods such as symbolic execution can be used to 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, one can generate tests that exercise particular program paths, satisfy various code coverage criteria, or cover all disjunctive case distinctions in the specification.

There is an important dimension where testing goes *beyond* verification: the latter ensures correctness of the target system, but not of the runtime environment or the compiler backends, however, testing can also exhibit bugs that are *not* located in the target system itself. For this reason, testing cannot be replaced by verification in all cases.

Verification Methods

The Overall Picture

Most verification approaches fall into one of four methodologies (deductive verification, model checking, refinement and code generation, abstract interpretation) that we now discuss in turn. Let us introduce some dimensions along which they can be classified and that influence the nature of the reasoning that happens during verification.

Arguably, the main tradeoff that influences the design of a verification method is automation of proofs search versus expressivity 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 expressivity, 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 ex-

pressivity of a logic and the computational complexity of its decision problems are directly related to each other. For undecidable languages, such as first-order logic, full automation cannot be expected, but even for decidable languages, such as temporal logic, problems quickly become infeasible as the size of the target system grows.

There is, however, 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 there are instances for which 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 employ 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 one typically works with an abstract system model from the start, which may have to be refined and adapted many times during the verification process. As we will see below, it is fruitful to combine both approaches.

Another dimension in the design of verification methods, heavily influenced by expressivity, is the verification workflow: assuming a decidable modeling language and a feasible target system size, it is 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 mislabelled as "push button" verification) based on cycles of failed verification attempt, failure analysis, followed by modifications to the target system, specification, or instrumentation, until a verification attempt turns out to be successful.

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.

A further distinction is the precision of the verification method, i.e., 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 that 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 abstract syntax and the semantics of the target language. In systems (sometimes called *proof assistants)*, such as HOL

or Isabelle, real-life languages of considerable scope have been modelled in this manner, including, e.g., the floating point logic of x86 processors, a non-trivial fragment of the Java language, the C language, and an operating system 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 normally based on first-order logic and target language objects occur directly as part of logical expressions without encoding. The semantics of the target language 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-/post-condition 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 does not hold, are added to the pre-condition (we assume that the execution of B has no side effects). There is typically at least one such proof rule for each syntactic element of the target language. Such calculi have been implemented for functional (ACL2, VeriFun), as well as for imperative programming languages (KeY, 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 approach is called VCG (for Verification Condition Generator) architecture and realized, e.g., in Dafny and Why.

Common to all three architectures is that they need detailed and many auxiliary specifications, including loop invariants and/or induction hypotheses and that they can be used for proving functional correctness of systems. Due to their general nature and their expressivity, proof assistants for higher-order logic tend to require more user interaction than the other two. However, in the last years external automated theorem provers are increasingly employed to decrease the necessary amount of interaction. To make this work, one must 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 Boyer & Moore in the predecessor of the ACL2 system [8]. This 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 often their internal working is not very 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, 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 is also hard to implement aggressive simplification of intermediate results.

Model Checking

Model checking [2] is based on the idea to view the execution model of a soft- or hardware system as a finite transition system, i.e., as a state automaton whose states are propositional variable assignments. Since 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 execution of the system 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 progresses in state representation were made that in many cases are 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 model checkers SPIN and NuSMV. 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 fundamental limitation to finite state systems of the standard approaches. To go beyond the finite state barrier (or simply deal with finite but large systems), several techniques have been suggested: sound abstraction (see also Abstract Interpretation below), abstraction with additional checks, and incomplete approaches such as bounded model checking [10]. Yet another possibility is offered by symbolic execution engines that enumerate reachable states without loss of precision, such as KeY, VeriFast, Java PathFinder, or Bogor. The logic-based techniques for infinite state representation realized in the latter kind of systems employ 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 under "model checking" that use reasoning technology such as SMT and propositional satisfiability (SAT) solving instead of model checking as defined above (an example is the CBMC system, see

Table 3). In this article, we use the notion of "model checking" in a more narrow sense and consider the latter kind of systems under the heading of "deductive verification."

Refinement and Code Generation

Verification can also be achieved by gradual refinement of 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 non-deterministic concepts, conveniently expressed in set theory, must be refined into operational ones. For example, there may be a proof obligation relating a set comprehension to an iterator. Hence, refinement over multiple levels for non-trivial systems creates a large number of proof obligations over set theory.

Evidently, most proof obligations generated during refinement-based verification can be discharged with automated theorem provers. Yet the interaction with the automated-reasoning community has been surprisingly little. 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 for that reason discharging of proof obligations is outsourced to Isabelle, but not to first-order provers.

There is almost no work done by the verification community regarding code generation by compilation and optimization of executable, yet abstract system models. Of course, there is an abundance of model-driven software development approaches. However, most of the involved notations (like UML) are not rigorous enough to permit formal verification. The same is true for code generation from languages like MathWorks, SystemC, VHDL, or Simulink, although the SCOOT system[4] is able to extract abstract models from SystemC. Recently, it has been 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 constitutes a vast potential for the reasoning and formal verification communities to employ their techniques.

Abstract Interpretation

Abstract interpretation [13] is a method to reason soundly and 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 (i.e., type), but over a more abstract, smaller one. For example, an integer variable might only have 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", etc. The abstract domains and operations

[4] `www.cprover.org/scoot`

Table 2: Executive summary of the analysis provided in this article

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 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 is a convergence of finite-state/abstract (model checking, abstract interpretation) and infinite state/precise (deductive verification, refinement) methods
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 is no single best verification tool

are chosen in such a way that the semantics is preserved: 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) one can 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 is, of course, 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, since abstract interpretation can be seen as a very general method to render infinite computations finite in a sound manner, it is 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 is 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. That has been realized in the KeY system and allows the exploitation of synergies between abstract interpretation-style constraint solving and deductive verification-style logical reasoning[15].

Trends and Opportunities

We close this overview paper by a brief discussion of the main trends and opportunities for reasoning in the context of verification. The main points of our analysis are also given in the form of ten concise statements in Table 2.

Non-Functional Properties

From the somewhat sobering insight that full functional verification is too expensive for most application scenarios due to 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. The required specifications (including invariants) can be in many cases 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 every-day scenarios and can easily be mapped to business cases: e.g., quality-of-service parameters such as response time or resource consumption of cloud applications [17].

Convergence of Methods

From our discussion of verification methods above one can see that there is much to be gained from a closer collaboration of the various subcommunities. We give two examples: first, to verify large industrial systems it is necessary to use both, methods optimized for finite state systems (such as model checking) and 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 is a vast opportunity for verification in correct code generation from modeling languages such as Simulink or SystemC. Even though first steps have been made [18], this is a (so far) 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 the performance of verification systems. SMT solvers combine efficient theory reasoning over variable-free expressions with heuristically driven quantifier instantiation. Importantly, they are also able to detect counter examples for invalid problems. Similar techniques had been implemented as part of monolithic verifiers such as ACL2 or KIV for decades, but stand-alone SMT solvers are much easier to maintain and they also benefit from progress in SAT solving. As a consequence, there is currently the happy situation that the verification and SMT solving

communities drive each others 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 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 changes of the verification target. During system development and maintenance such changes are normal and occur frequently. They are triggered by feature requests, environment changes, refactoring, bug fixes, etc. Any change in the target system has the potential to invalidate the complete verification effort that has been spent already. If re-verification is expensive, this is a major threat against the practical usefulness of any but fully automatic and lightweight verification methods. One solution can be verification methods that are aware of changes [19]. In this case, re-verification—in particular of those parts that remain unchanged—can be replaced with automated reasoning.

Table 3: An overview of reasoning and verification systems, which may serve as a starting point for further exploration.

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, light-weight 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

System/URL	Method	Verification Scenario
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/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, light-weight 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

System/URL	Method	Verification Scenario
VeriFun `www.verifun.org`	deductive verification (batch), induction proofs	functional correctness
VCC `research.microsoft.com/projects/vcc`	deductive verification (batch)	functional correctness, bug finding

Conclusion

The future looks bright for the collaboration of verification and reasoning. Recent advances in both fields and increasingly tight interaction already gave 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 already in the hardware domain.

Acknowledgments

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

References

[1] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Lessons learned from microkernel verification: Specification is the new bottleneck. In Franck Cassez, Ralf Huuck, Gerwin Klein, and Bastian Schlich, editors, *Proceedings, Seventh Conference on Systems Software Verification. SSV 2012, Sydney, Australia)*, number 102 in Electronic Proceedings in Theoretical Computer Science, 2012.

[2] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[3] James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In Klaus Havelund, John Penix, and Willem Visser, editors, *7th International SPIN Workshop Stanford*, volume 1885 of *LNCS*. Springer, 2000.

[4] André Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, 2010.

[5] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[6] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings, ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin*, pages 234–245. ACM Press, 2002.

[7] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.

[8] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, New York, 1988.

[9] Randal E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[10] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W. R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems. Part of European Conferences on Theory and Practice of Software, ETAPS'99, Amsterdam*, volume 1579 of *LNCS*, pages 193–207. Springer-Verlag, 1999.

[11] Bernhard Beckert and Daniel Bruns. Dynamic logic with trace semantics. In Maria Paola Bonacina, editor, *Automated Deduction, 24th International Conference on Automated Deduction, Lake Placid, USA*, volume 7898 of *LNCS*, pages 315–329. Springer, 2013.

[12] Ran Ji, Reiner Hähnle, and Richard Bubel. Program transformation based on symbolic execution and deduction. In Robert M. Hierons, Mercedes G. Merayo, and Mario Bravetti, editors, *Software Engineering and Formal Methods: 11th International Conference, SEFM 2013, Madrid, Spain*, volume 8137 of *LNCS*, pages 289–304. Springer, 2013.

[13] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles of Programming Language, Los Angeles*, pages 238–252. ACM Press, New York, January 1977.

[14] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, Chicago/IL, USA*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

[15] Richard Bubel, Reiner Hähnle, and Benjamin Weiss. Abstract interpretation of symbolic execution with explicit state updates. In Frank de Boer, Marcello M. Bonsangue, and Eric Madelaine, editors, *Post Conf. Proc. 6th International Symposium on Formal Methods for Components and Objects (FMCO)*, volume 5751 of *LNCS*, pages 247–277. Springer-Verlag, 2009.

[16] Elvira Albert, Richard Bubel, Samir Genaim, Reiner Hähnle, Germán Puebla, and Guillermo Román-Díez. Verified resource guarantees using COSTA and KeY. In *Proceedings, ACM SIGPLAN 2011 Workshop on Partial Evaluation and Program Manipulation (PEPM'11), Austin, Texas, USA*. ACM Press, 2011.

[17] Elvira Albert, Frank de Boer, Reiner Hähnle, Einar Broch Johnsen, and Cosimo Laneve. Engineering virtualized services. In Arnor Solberg, Muhammad Ali Babar, Marlon Dumas, and Carlos E. Cuesta, editors, *2nd Nordic Symposium on Cloud Computing and Internet Technologies (NordiCloud)*, pages 59–63. ACM Press, 2013.

[18] Nesrine Harrath, Bruno Monsuez, and Kamel Barkaoui. Verifying SystemC with predicate abstraction: A component based approach. In *IEEE 14th International Conference on Information Reuse & Integration, IRI, San Francisco, USA*, pages 536–545. IEEE Press, 2013.

[19] Reiner Hähnle, Ina Schaefer, and Richard Bubel. Reuse in software verification by abstract method calls. In Maria Paola Bonacina, editor, *Proceedings, 24th Conference on Automated Deduction (CADE), Lake Placid, USA*, volume 7898 of *LNCS*, pages 300–314. Springer-Verlag, 2013.

Bios

Bernhard Beckert is a professor of computer science at the Karlsruhe Institute of Technology (KIT), Germany. Contact him at `beckert@kit.edu`.

Reiner Hähnle is a professor of computer science at Technische Universität Darmstadt, Germany. Contact him at `haehnle@cs.tu-darmstadt.de`.