TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Software & Systems Engineering
Prof. Dr. Dr. h.c. Manfred Broy

SPES 2020 Deliverable 1.3.A-1

Analysis Techniques: State of the Art in Industry and Research

Software Plattform Embedded Systems 2020

Author:	Alarico Campetelli
Version:	1.1
Date:	20 April 2010
Status:	Released

Technische Universität München - Fakultät für Informatik - Boltzmannstr. 3 - 85748 Garching

About the Document

In the development of software and hardware systems it is important to guarantee a correct behaviour. Damages due to errors may cost money and even endanger human lives. There are analysis techniques to verify, test and validate systems. In this document we explain the different approaches which—altogether—can be referred to as *analysis techniques*. The state-of-the-art of both, research and industry, will be stated for testing and verification.

The purpose of this document is give an overview and introduction to existing analysis techniques. It is one of the objective of work package ZP-AP 1.3, to integrate such techniques into the modelling theory.

Contents

1 Analysis Techniques

During the last years, the functionality of safety-critical systems grew dramatically. This includes software as well as hardware, and holds in the same time for reliability requirements. Therefore, we are now facing an enormous system complexity. Likewise to the growing complexity, the possibility of unintentional added defects grows as well. In consequence, system failures may lead to a considerable loss of money due to warranty costs or even—in the worst case—endanger human lives. Thus, the need for well-defined development theories, languages, and tools naturally follows. Their comprehensive use permits the construction of more reliable systems even though system complexity will in all likelihood continue to grow in the future.

Introduction Nowadays, computers are widespread and used to control various safety critical systems like automobiles, aircrafts, satellites, medical devices, etc. Software and hardware operating within such systems is complex, it consists of many modules or programs, each in the magnitude of thousands to millions lines of code. The existence of errors may have severe consequences. For instance, consider the famous fault occurred in 1994: the "Pentium bug", which caused Intel to recall faulty chips and take a loss of about $475 million [CMMP95]. In June 1996 another important program failure was in the Ariane 5 rocket: a computation exception occurred during the conversion of a 64-bit floating point number into a 16-bit value. The rocket exploded, less than forty seconds after its launch. The reason for this catastrophe was identified to be a software failure in a system responsible for calculating the rocket's horizontal velocity. Since these exemplary events, software and hardware analysis techniques have been more used, mostly model checkers but also theorem provers [ASM+08, BJ97].

In fact these examples (and many others) underline the need for developing reliable and sound systems. We have also to consider that through the use of analysis techniques, we have a more efficient specification and so verification of the undesired behaviours. Therefore it is possible to reduce the time of the whole development process. In the sequel, we use the term system to refer to a software or hardware system.

Definition In order to have a characterisation of the term *analysis techniques*, we refer to the definition of *Software Verification and Validation* from the IEEE [cit05]:

"Software verification and validation (V&V) is a technical discipline of systems engineering. The purpose of software V&V is to help the development organization build quality into the software during the software life cycle. V&V processes provide an objective assessment of software products and processes throughout the software life cycle. This assessment demonstrates whether the software requirements and system requirements (i.e., those allocated to software) are correct, complete, accurate, consistent, and testable. The software V&V processes determine whether the development products of a given activity conform to the requirements of that activity and whether the software satisfies its intended use and user needs. The determination includes assessment, analysis, evaluation, review, inspection, and testing of software products and processes. Software V&V is performed in parallel with software development, not at the conclusion of the development effort."

That definition is applicable to software systems only. Next we state an analogous one for hardware verification [KG99]. The desired functionality and the overall architecture of hardware design is typically defined with a high-level specification, as for instance block diagrams, tables, requirements, and informal text. In order to obtain a final design, a combination of top-down and bottom-up design techniques are applied. The specification is checked in validation and verification activities to verify that it does indeed meet physical and implemented design. This is achieved, in the normal development flow, through simulation and testing. Typically exhaustive testing for non-trivial systems is infeasible, so is model checking for large-scale systems, whereas formal verification provides a more comfortable and systematic solution. Because, formal verification, unlike testing and simulation, uses rigorous mathematical theories to show that a system meets its specification. Anyway, there are also some support tools for exhaustive testing sessions, through test case generation, which use in the verification process formal verification techniques [HSTV08, HSTV09]. Based on these definitions we consider domains and the purposes of each analysis technique.

Outline We give an overview of the major verification methods that are being used in industry and their actual state in research. In Section 2 we describe traditional testing and simulation approaches, inspections, reviews and walkthroughs in Section 3, and finally runtime verification in Section 4. Then, in Section 5 we explain the basic ideas of formal verification techniques. Section 6 sums up some successful case studies and well-known tools representing the state-of-the-art and finally in Section 7 we present the results of a survey carried out in the SPES project [HM09]. Notice, only those aspects related to analysis techniques are mentioned. In Section 8 a comparison among different types of analysis techniques is given and we conclude in Section 9.

2 Testing and Simulation

In order to introduce the typical use of testing and simulation, just consider a common development life cycle.

Development life cycle We show a typical system development life cycle in Figure 1 [Mee05]. Requirements analysis is the first step, and describes the major characteristics and system properties to be developed. The second phase is the design, which is focused on the development of a high-level architecture, which then is refined to a more detailed one describing system components. This phase is followed by coding that typically includes also testing activities of the individual modules or parts of the systems under development. Subsequent to the coding phase, system testing is applied intensively. Finally, the product will be released and maintained if all test have been passed successfully. During the development and also afterwards, in each of these phases it is possible to introduce errors or not correct behaviours that influence the correctness of the whole system. Testing and simulation are used to check that the system is correct with respect to its functionality and requirements. As exhaustive testing is in general infeasible, testing cannot show the absence of errors. Therefore, one relies on a carefully selected or generated test suite in the hope that it covers most interesting program executions.

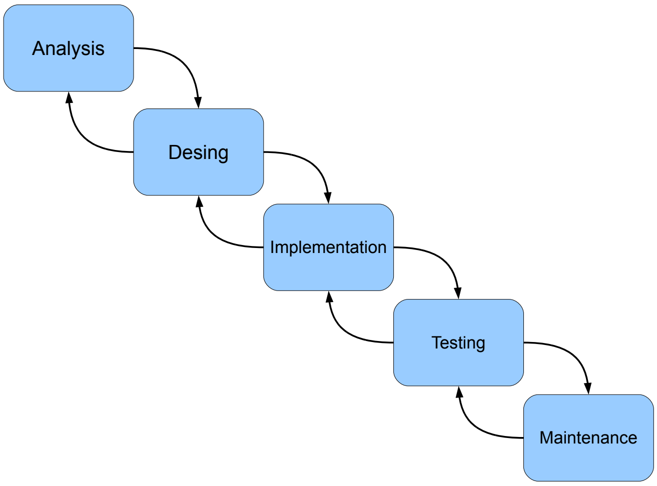


Figure 1: The development life cycle

Nowadays, testing, simulation, and code review methods are available for software written in almost all programming languages. In practice these techniques detect a high number of coding bugs and thus are considered very effective.

During the last years, Software Productivity Research[1] built up an extensive database by collecting software defects and the efficiency of their corresponding defect removal techniques [Jon97]. Considering these results, only a few defects in the final product were identified to be bugs inserted while coding, they are 35%.

For example, during testing of the Voyager and Galileo spacecraft systems, out of 197 detected critical faults, only 3 were coding errors [Lut93]. Returning to the software productivity research results, about 20% of the faults were traced back to requirements, 30% to design and the rest to other types of errors. Hence, the majority of all errors made during the overall software development process arise during the requirements and the design phase whereas less during coding.

Typically, it is not possible to cover all possible system behaviours using testing and simulation techniques. That is one of the most prominent drawbacks of these approaches. All possible system behaviours can be characterised as the evolution of internal data or variable values during the execution of a program stimulated with all possible input values. Hence, data and variable values together define a well-defined system state. All those reachable states together define the system's state space.

Considering such state definition, even small programs have a large number of states, directly dependent to the possible values that variables or data can have. For instance, a program which has a variable in the natural set $\mathbb{N}$, which is usually represented in the program with an integer variable, it can take as many values as allowed by the memory available, and for each value representing a different state.

[1] http://www.spr.com/

Without considering the obvious memory limitations, the program can be considered to have infinitely many states. Since testing and simulation are applied by explicit definition of values to variables, it is impossible or very tedious to examine all possible program behaviours. Therefore, it might be the case that critical behaviours are not considered during testing and possible defects are not detected.

3 Inspections, Reviews, and Walkthroughs

Below, we refer to the following definition: "software inspections, reviews, and walkthroughs are human based methods for analysing documents based on informal or semi-formal techniques for the purpose of early and effective defect detection during development in order to improve product quality and reduce development rework" [CLB03].

In order to have a comprehensive overview, we consider the survey from the International Software Engineering Research Network[2] and the Fraunhofer Institute for Experimental Software Engineering (IESE) initiated in 2002, and its evaluations for the state of software review practice [RCJ+08]. The results obtained evidence some important benefits from the use of software reviews: improved communication among developers, evaluation of project status, and enforcing standards. We conclude that many companies do not exploit the full potential of these methods for reducing defects and so improve the quality control process. This conclusion is due to the following considerations.

Firstly, software reviews are not systematically performed by companies during all development phases, in fact about 40% of participants accomplish reviews on requirements or design documents, and about 30% on source code.

Secondly, often the preparation or defect detection step is not systematically executed by companies. Even though empirical studies have demonstrated that a preparation phase for reviews is crucial for effective reviews [Vot93], about 60% of respondents stated that they did not regularly manage it. Considering companies conducting a preparation phase, half of them use a checklist as support, about 10% use more advanced reading techniques, and finally 40% use reading technics, which do not offer systematic support for defect detection.

Thirdly, in a systematic evaluation and improvement program, companies seldom embed reviews. In fact 40% of the companies do not collect data at all, and 18% collect data but do not analyse them. Considering the 42% who collect and analyse data, only 23% try to optimise the review process. Even though it has been proved that optimisation is crucial to accomplishing effective reviews. Today, inspections, reviews, or walkthroughs are integral parts of most industrial software development life-cycle models, procedures, and standards [RCJ+08].

4 Runtime Verification

Runtime verification [CM04] is a verification technique that combines formal verification (Section 5) and program execution. The examination process for finding the defects is made by passively observing the input/output behaviour during the normal program execution. The

[2] http://isern.iese.de

system's behaviour is monitored and so verified dynamically, by means of log traces, to satisfy the requirements. Typically, requirements are specified as temporal constraints such as LTL-formulas (Linear Temporal Logic), or automata and state charts. Different to model checking techniques (Section 5.1.2), where a model of the target system is verified, runtime verification is applied while the real system is in execution. Thus, runtime verification can determine how the implementation conforms to its specification and permits a detection of a possible deviation of the system behaviour from the pre-specified behaviour.

Formal verification methods (Section 5), as for instance model checking, also have to deal with infinite trace, whereas runtime verification only considers finite traces. A runtime monitor for detecting requirement violations that monitors on a potentially never ending system is an exception to this rule.

In order to improve our confidence in the system, continuous monitoring of the runtime behaviour checks the consistence of the current execution with the requirements. To arguing for runtime verification, the literature states at least the following four points [CM04]:

- As the correctness of the model does not imply the correctness of the implementation (e.g. errors in a code generator or compiler), one cannot be confident about the implementation.
- Some system information is convenient to be checked or is available only at runtime.
- Sometimes the behaviour depends on the environment of the target system. In fact, if an environment model is not comfortable or possible to construct it is impossible to obtain the information necessary to check the system.
- It is helpful to monitor properties or behaviour that have been already statically tested, in systems where safety and security are important or in critical systems.

Runtime verification permits formal specification (Section 5) and verification or testing (Section 2) of the properties that a system has to satisfy, whereas traditional testing techniques (such as unit testing or simulation) are mostly informal. In the latter case we have only a partial proof of correctness, which does not guarantee that the system will operate correctly with untested inputs. Runtime verification is weaker than formal methods but stronger than testing, in terms of its ability to guarantee system correctness. In fact, at implementation time, testing can guarantee the correctness only on a restricted set of inputs, as a consequence, undetected faults may result in failures at runtime. Such events even allow the propagation of corrupted output values because the failure was not detected. Contrariwise, when monitoring the system at runtime, they can be caught. Nevertheless, formal methods are stronger than runtime verification because such guarantees cannot be made a priori.

5 Formal Verification

Compared to (informal) testing, formal verification has the advantage of exhaustiveness, i.e., the whole state space is traversed. Model checking, for instance, provides a counterexample guiding the developer to the erroneous system behaviour—is there is any. This helps to gain insight into the problem and helps to fix it. Formal verification, however, has also some drawbacks: (i) It is difficult and time-consuming, (ii) it is only as reliable as the used formal models, and (iii) it's difficult to prove that a proof is correct. Despite this complexity, during the last years, formal verification tools have been introduced, in industrial development processes: from security and safety related projects, over hardware circuit verification to software driver verification. Model checking techniques has been, in some fields, very successful [CW96];

Formal verification encompasses a set of techniques available for developers to assist the development and to improve the system's quality. However, it should be used in conjunction with informal verification methods as testing, which are complementary techniques, in order to increase reliability of the system under development. Formal verification enhances reliability of the system by ensuring that it meets its functional requirements, in particular at earlier stages of system design. Nevertheless it is notable that it does not ensure that the system being developed is completely correct. Formal methods [CW96] are the theoretical basis for formal verification and are used for specifying and verifying systems through mathematically based languages, techniques, and support tools. System specification means expressing the system requirements in a mathematical or formal language. System verification is applied after the specification and formally proves that the system meets its requirements. Several tools are available for specifying and verifying system, providing the developers suitable notations, formalisms, and algorithms.

In industry formal verification of hardware and software systems has acquired popularity. In the hardware sector, formal verification techniques have been applied successfully and thus have also encouraged the software sector to consider whether similar benefits could be accomplished in program verification [OL07]. Despite proofs of correctness for computer programs are available since early times of computer science, such academic works were ignored by industry citing advances in research as "impractical" [Hoa87]. However, properties of hardware and software system are extremely different, namely the strict structure of hardware, the inherently finite state of hardware, and the limited size of hardware [SBH04]. Whether the success story of formal verification of industrial hardware can be reproduced in software is still uncertain. Some progress has been made, despite many challenges still remain [ASM+08].

Formal verification needs basically the following ingredients: first, a mathematical model of the system is needed and second, formally specified requirements have to be provided, that should be satisfied by the model. There are two fundamental verification techniques: model checking and theorem proving. In the industrial context, both techniques are applied for hardware verification to complement standard methods such as testing and simulation. Researchers and practitioners that work in the specification and verification sectors are applying progressively industrial case studies, therefore gaining the benefits of using formal methods.

Outline We subdivide formal verification techniques into (i) static analysis, techniques that elaborate information on the behaviour of a program without executing it, and (ii) theorem proving, i.e., the process of finding a proof of a property where the system itself system and its properties are expressed as formulas in some mathematical logic.

In the next subsections we describe three techniques for static analysis. We start with abstract static analysis in Section 5.1.1, these techniques are used in software development tools, e.g., for pointer analysis in modern compilers. A formal basis for such techniques is Abstract Interpretation, introduced by Cousot and Cousot [CC77b]. The second part in Section 5.1.2 is about model checking that was introduced by Clarke and Emerson [CE82], and independently by Queille and Sifakis [QS82]. The basic idea is to determine if a correctness property holds by exhaustively exploring the reachable states of a program. If the property does not hold, the model checking algorithm provides a counterexample, i.e., an execution trace leading to a state in which the property is violated. Since the state space of software programs is typically too ample to be examined in a reasonable time and memory occupation, model checking is often combined with abstraction techniques.

Finally, the third part in Section 5.1.3 is dedicated to a formal technique that performs a depth-bounded exploration of the state space. This technique is called Bounded Model Checking (BMC) since it is a special form of model checking. BMC is presented separately since the requirement to analyse the entire state space is not needed, in fact BMC explores program behaviour exhaustively, but only up to a given depth. As a consequence faulty behaviour that require longer paths are missed. Then we highlight in Section 5.2 on theorem proving [RSS95]. In theorem proving, a system and its properties are described by means of logical formulas and the system is shown by means of a logical proof to entail the desired properties.

5.1 Static Analysis

Static analysis comprises a class of techniques for automatically computing information about the behaviour of a program without executing it. Most problems and questions about the behaviour of a program are undecidable or infeasible to practically compute an answer. The basics of static analysis efficiently compute approximate but sound guarantees, guarantees that are not misleading.

5.1.1 Abstract Static Analysis

Abstract Interpretation According to Cousot, abstract interpretation “is a theory of approximation of mathematical structures, in particular those involved in the semantic models of computer systems. Abstract interpretation can be applied to the systematic construction of methods and effective algorithms to approximate undecidable or very complex problems in computer science” [Cou05]. A set of concrete values is transformed in an approximate representation named abstract domain. An abstraction function is used to map concrete values to abstract ones. In order to obtain an approximate solution, the meaning of a program is translated into an abstract domain through abstract interpretation. We can derive from a concrete interpretation an abstract interpretation by defining counterparts of concrete operations in the abstract domain. If certain mathematical constraints between the abstract and concrete domains are met, fixed points calculated in an abstract domain are guaranteed to be sound approximations of concrete fixed points [CC79]. During the last years, static analysis using abstract interpretation has been very successfully applied to verify properties of real-time and safety-critical embedded systems [CC04].

Numerical Abstract Domains According to Sliva and others, over the years, “various abstract domains have been designed, particularly for computing invariants about numeric variables” [DKW08]. The expressive power of a domain affects the class of computable invariants and thus the properties that can be proved.

It is defined a numerical abstract domain for parametrising a static analyser, that is, a set of computer-representable numerical properties with algorithms to calculate the semantics of program instructions. Examples of numerical abstract domains are: the interval domain [CC77a] that detects variable bounds, and the polyhedron domain [CH78] for affine inequalities. Each domain equilibrates some cost in respect to precision.

Shape Analysis Shape analysis is a static code analysis method, that verifies and finds in computers programs properties of data structures. Usually, it is used to prove high-level specifications and find software bugs properties at compile time. The resources required in terms of runtime limit its spread to non-academic industrial use, despite it is a powerful technique.

5.1.2 Model Checking

Model checking is a technique that builds a formal model of a system and checks whether it satisfies a desired property. Generally the check is performed as an exhaustive state space search on the model. As is is applied on finite models, it is guaranteed that the search terminates. The main challenge is to devise and improve algorithms as well as data structures, that are capable to search in large state spaces. Model checkers, i.e., tools, which perform model checking, take two inputs: a finite state model of the system and a property specified formally. A model checker proofs whether the system satisfy the property, and provides a “yes” or “no” answer. If the system does not satisfy the property, the answer is “no” and it is also provided a counterexample, i.e., a trace (system run) violating the property. Counterexample are useful in order to find bugs in the system design as they guide developers to the unintended behaviour. Figure 2 describes the model checking process. Model checking has been used mainly in hardware and protocol verification [CK96], nowadays this technique is also applied to software systems.

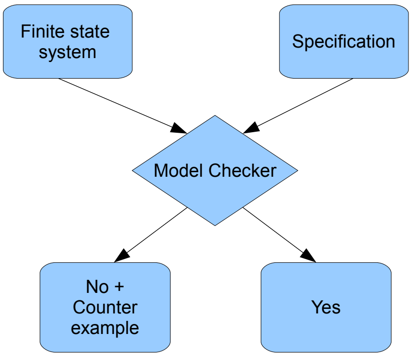


Figure 2: Overview of Model Checking

In today’s practice, two general approaches are used: the first, *temporal model checking*, is a technique developed independently in the 1980s by Clarke and Emerson [CE82] and by Queille and Sifakis [QS82]. There, the specifications φ are expressed in a temporal logic [Pnu81] and the model $\mathcal{M}$ as a finite transition system. Verification is performed through an efficient search procedure, which checks if the finite state transition system is a model for the specification, i.e. $M \overset{?}{\models} \varphi$.

In the second approach, the system and the specification are modelled as automata, then the system is compared to the specification to determine whether its behaviour conforms to the specification. Different notions of conformance have been proposed, as language inclusion [HK90, Kur94b], refinement orderings [CPS94, Ros94], and observational equivalence [CPS94, FGK+96, RS91]. The authors of [VW86] have related these two approaches, showing how the temporal logic model checking problem could be recast in terms of automata.

Model checking is completely automatic and fast, in contrast to theorem proving, sometimes producing an answer in a relative short time. We can use model checking to prove partial specifications, i.e., even if the system is not totally specified, it can provide useful information about the system correctness. Especially if the property is not satisfied, a counterexample is provided, which usually represents subtle errors in design and can be used to improve the debug phase.

The state space explosion problem is the main disadvantage of model checking. In 1987 McMillan proposed Bryant's ordered binary decision diagrams (BDDs) [Bry86] to represent the state transition systems in a more efficient way. That increased the size of these systems that could be proven. In order to ease the state space explosion problem, other approaches have been suggested, as for instance the exploitation of partial order information [Pel94], localisation reduction [Kur94b, Kur94a], and semantic minimisation [JCE97] to eliminate unnecessary information from a system model.

Today, model checkers are routinely expected to work with systems having between 100 and 200 state variables. Systems with 10^{120} reachable states have been checked by model checkers [BCL+94] and also systems with an essentially unlimited number of states using abstraction techniques [CGL94]. Therefore it is now powerful enough to be used in industry.

5.1.3 Bounded Model Checking

In the semiconductor industry bounded model checking (BMC) is one of the most commonly applied formal verification technique. The success of this technique comes from brilliant performance of propositional SAT solvers. Biere and others have introduced it in 1999, as a complementary method to BDD-based and unbounded model checkers [BCCZ99]. BMC visits the states within a bounded number of steps, say k, for this reason it is called bounded. When using BMC, the propositional formula is built by unwinding the system for k times in conjunction with a property. Next the obtained formula is passed to an efficient SAT solver. If and only if there is a trace of length k that contradicts the property the formula is satisfiable. If the propositional formula is unsatisfiable, we cannot be certain about the correctness of the verification, because there may be counterexamples longer than k steps. However, many bugs have been detected that otherwise would not have been found. Bounded model checking is applied in many areas very successfully.

Considering a propositional formula, a satisfying assignment corresponds to a path from the initial state to a state that violates the property. BMC provides a full counterexample trace in case of the property is not satisfied, and it is one of the best techniques to identify shallow bugs. The range of programs that are supported by BMC is wide, included also dynamically allocated data structures. In this case, BMC does not require built-in knowledge about the data structures that the program maintains. But then completeness of such a technique is only obtainable on very "shallow" programs, i.e., programs without deep loops.

5.2 Theorem Proving

Theorem proving uses formulas in some mathematical logic or theory, to express the system and its properties [CW96]. A *formal system* gives this logic, which defines a set of axioms and a set of inference rules. In practice, theorem proving finds from the axioms of the system a proof of a property. The proof is composed of steps which invoke the axioms and rules, and possibly derived definitions and intermediate lemmas.

Theorem proving approaches perform verification using a calculi that is based on two principal works. The first is by C.A.R. Hoare which presents a calculus to reason about program correctness in terms of pre and post conditions [Hoa69]. The second is by E.G. Dijkstra, who extended Hoare's ideas in the notion of "predicate transformers" which begins with a post condition and uses the program code to determine the pre condition, instead of starting with a pre condition. This post condition is needed to create the pre condition [Dij75]. Hoare's approach introduced the concept of a "Hoare triple", which is a formula in the following form:

$$\{PRE\}\ P\ \{POST\}$$

We can read this formula as "if property PRE holds before program P starts, $POST$ holds after the execution of P". Depending on the portion of the system to be verified, the program P can refer to an entire program or to a function call. In Hoare's calculus, the derivation of $POST$ based on PRE and P, is performed using axioms and rules of inference. In the case of software verification, the Hoare's syntax of P may correspond to an imperative language with usual constructs (assignments, conditional branches, loops, and sequential statements).

Nowadays, theorem proving is progressively used in the verification of safety-critical properties of hardware and software systems. We can coarsely categorise theorem provers into the classes (i) highly automated, (ii) general-purpose programs, and (iii) interactive systems with special-purpose capabilities. On the one hand, approaches based on automatic verification have been useful as search procedures and were successfully applied in solving combinatorial problems. On the other hand, interactive systems were more appropriate for mechanising formal methods and the formal development of mathematics [CW96].

Theorem proving can deal with systems having an infinite state spaces, whereas model checking cannot. In order to prove properties on infinite domains, it entrusts on techniques such as structural induction. The interactive theorem proving process is slow and often error-prone, because, as the name suggests, it requires interaction with a human. However, the user often gains useful insight into the system or the property being verified, during the process of finding the proof.

6 Case studies

In the previous sections, we introduced the range of verification methodologies and specification types. In order to provide a sense of the state-of-the-art in verification, we now report selected case studies, primarily taken from an industrial setting.

6.1 Inspections, Reviews, and Walkthroughs

Important and successfully operating companies such as NASA's Software Engineering Laboratory (SEL), Allianz, Motorola, or IBM, are using these techniques. They report up to 95% defect detection rates even before testing, 50% cost reduction for newly written lines of code, and even a reduction of the delivery times by up to 50% [RCJ+08].

Industrial case studies The NASA Software Engineering Laboratory (SEL) was founded in 1976, and its scope is to introduce state-of-the-art techniques and practices, and check their value for software development organisations. In the course of its 25 years of work, SEL managed to reduce defects rates by 82% during development, and decrease development cost by 74%, despite the complexity and functionality of the systems raised fivefold [RCJ+08].

The SEL chose inspections as one potential cutting-edge technology [RCJ+08]; they introduced inspections based on Fagan's paper in 1976. These code inspections used multiple reviewers and were based on ad-hoc reading. They did not necessarily follow a specific protocol or use a moderator for the inspection meetings, and they did not collect any data. These are the main differences compared to Fagan's inspections [Fag99]. Altogether, the decision makers were convinced form the available evidence that formal inspections were successful, and formal Fagan inspections were defined as a standard around 1980. Therefore, every project had to use inspections or justify why they did not use.

6.2 Static Analysis

6.2.1 Abstract Static Analysis

Lint [DKW08] is a popular static analysis tool released in 1979 by Bell Labs. It is used for detecting errors in C programs. The kind of found errors, produced warnings, and the user experience are extended and inspire many modern tools. For instance, FindBugs [DKW08] is a modern tool with analogous features for Java. These two tools are significant and have influenced other static analysis tools, nevertheless they are unsound and provide no rigorous guarantees.

Industrial case studies We refer to a market survey reported in [DKW08] and list some worthy tools from it.

- CodeSonar by Grammatech is a tool based on inter-procedural analyses to check for template errors in C++ code. The errors detected are: buffer overflows, memory leaks, redundant loops, and branch conditions. There is the K7 tool from KlocWork, with a similar characteristic and the support for Java [DKW08].
- PREVENT is a static analyser with features—similar to CodeSonar and EXTEND—a tool to implement coding standards, both are produced by Coverity. Coverity tools were utilised, in January 2006, in over 30 open source projects to check errors [DKW08]. This experiment was part of an United States Department of Homeland Security sponsored initiative.
- Abstract interpretation was used to detect the error leading to the failure of the Ariane 5 [LMR+98]. The Astrée static analyser [BCC+02] has been used to prove the flight control software of an Airbus aeroplane. It also implements abstract domains to identify buffer overflows and undefined results in numerical operations.
- NASA has developed the C Global Surveyor (CGS), a static analyser specially developed for space mission software. The software used on the Mars Path-Finder, Deep Space One, and Mars Exploration Rover, has been analysed using the CGS and C/Ada static analysers produced by Polyspace Technologies.
- PAG is a program analyser generator marketed by AbsInt GmbH. It is used to analyse architecture-dependent properties such as worst case execution time, cache performance, stack usage, and pipeline behaviour [DKW08].
- Program annotations such as types, pre- and post-conditions, and loop invariants are required for several tools. Such information may increase the burden of the model, but they may reduce the complexity and at the same time improve the verification precision. The ESC/Java tool family [FLL+02] and Microsoft's PREfix and PREfast [LBD+04] use these approaches. SPARK [Bar03] and Spec# [BLS04] have build in annotation and verification support right in the programming languages and development environments. Thus, annotations become part of the system design.

6.2.2 Model Checking

In the following we list some well-known model checkers, roughly categorised according to whether the property (to verify) is given as a logical formula or as a state machine:

Temporal logic model checkers Chronologically, the first two model checkers were EMC [CE82, CES86, BCDM86] and CÆSAR [QS82, FGK+96]. The model checkers Murϕ [DDHY92] and UV [Kal94] are based on Unity programming language [Cha88]. The Concurrency Workbench [CPS94] checks properties expressed as μ-calculus formulas in systems specified as CSS processes. SVE [FSS+94], FORMAT [DJS95, Klo01], and CV [DB95] are tools concentrated on verification of hardware systems. There are model checkers specialised on hybrid systems, as for example HyTech [AHH96], and on real-time systems, for instance Kronos [DY95, HNSY94].

SPIN (Simple Promela INterpreter) [GPV+95, Hol91] is a successful and widely used model checker, that won the ACM software system award in 2001. SPIN is open source and used to validate large scale applications, like communication protocols for distributed systems, network applications, multi-threaded code, and for academic purposes, as well. In order to reduce the state space explosion problem, SPIN implements partial order reduction [HP95, Pel94]. Systems are specified using the Promela language, as a set of automata and the properties to check are expressed in linear-time temporal logic (LTL) formulas. Some verification tools translate source code to Promela and then use SPIN to verify it, as for instance an early version of the Java Pathfinder (JPF) for Java code [VHB+03].

SMV (Symbolic Model Verifier) [McM92] is the first model checker that uses binary decision diagrams (BDD) in order to reduce the state space. Systems are modelled in an own automaton notation and the desired properties are expressed as branching-time temporal logic (CTL) formulas. The UPPAAL [BDL04] model checker is based on the theory of timed automata [AD90], systems are modelled as real-time systems and it has been used successfully in case studies ranging from communication protocols to multimedia applications.

CMC [MPC+02] and Microsoft Research's Zing [AQRX04] are two representatives of the explicit-state software model checkers. In order to avoid the state space explosion problem the VeriSoft verification tool discards the visited states [God97], therefore the visited states may be repeatedly visited and explored because they are not stored. This approach has to limit the depth of its search to eschew non-termination and is state-less. The SLAM toolkit [BCLR04] is a predicate abstraction model checker, developed by Microsoft Research. It represents the first successful implementation of predicate abstraction theory. It verifies a set of about 30 predefined, system specific properties of Windows device drivers, such as "a thread may not acquire a lock it has already acquired, or release a lock it does not hold". It incorporates the predicate abstraction tool C2BP [BPR01, BMMR01], the BDD-based model checker Bebop [BR00a] for boolean programs [BR00b], and the simulation and refinement tool Newton [BR02]. It is possible to use the BDD-based model checker Moped [ES01] instead of Bepop, to prove temporal logic specifications. Moreover, SLAM can check properties that depend on bit-vector arithmetic, if it is combined with Cogent [CKS05a], which is a decision procedure for machine arithmetic.

The Berkeley Lazy Abstraction Software Verification Tool (BLAST) is a software model checking tool for C programs. BLAST uses lazy abstraction, i.e., only relevant parts of the original program are re-abstracted in the refinement step [HJMS02]. Abstraction is built by the CEGAR (Counterexample guided abstraction refinement) iterations. Each iteration has a speedup due to the close integration of the verification and refinement phases. BLAST uses Craig interpolation to derive refinement predicates from counterexamples [HJMM04] and provides a language to specify reachability properties.

SATABS is a verification tool for C/C++ programs, where the abstractions and the symbolic simulation of counterexamples is performed using a SAT-solver [CKSY03]. The reachable states of the abstract program are computed with the SAT-based model checker Boppo, which depends on a QBF-solver for fixed point detection [CKS05b]. As C programs are prone to arithmetic overflow, and allow to model arrays and strings, SATABS automatically produces and verifies proof conditions for array bound violations, invalid pointer dereferencing, division by zero, and assertions provided by the user. SATABS can verify concurrent programs that communicate via shared memory. Boppo combines symbolic model checking and partial order reduction to handle concurrency [CKS05b], but this method has scalability issues [WBKW07].

YASM is a software model-checker which implements the abstraction-refinement process for program analysis and liveness properties [GC06]. The Terminator tool also checks liveness properties and is based on SLAM [CPR05]; it is not dependent on predicate abstraction and can be founded on any software model checker. Sagar Chaki's MAGIC framework is a compositional model checkers that separates the model to verify into several smaller components. On those components, verification is applied separately [CCG+03]. It does not support shared memory, but can check concurrent programs that utilise message passing. There is an experimental version of SLAM, modified for checking concurrent programs. In such version Bebop can be replaced by either the explicit state model checker Zing [AQRX04] or the SAT-based tool Boppo [CKS06].

Behaviour conformance checkers The Cospan/FormalCheck is a commercial model checker based on showing inclusion between ω-automata [DPG96, HK90]. FDR (Failures-Divergence Refinement) is a model-checking tool for CSP programs [Ros94], which has been used to verify the Needham-Schroeder authentication protocol [Low96]. The Edinburgh Concurrency Workbench (CWB) is an automated tool for the manipulation and analysis of concurrent systems [CPS94]. CWB checks a notion similar to FDR of refinement between CCS programs, and in particular it allows for various equivalence, preorder and model checking using a variety of different process semantics. CWB and the tool Auto [RS91] can determine whether two systems are observably equivalent and minimise systems with respect to observational equivalence.

Combination checkers Hojafi et al. [HBK93] combine model checking with language inclusion. Stanford's STeP is used to verify reactive, real-time, and hybrid systems based on their temporal specification [BBC+96]. STeP combines deductive methods with model checking. VIS (Verification Interacting with Synthesis) is a toolkit for formal verification, synthesis, and simulation of finite state systems [BHSV+96], which compounds logic synthesis with model checking [BHSV+96]. The PVS (Prototype Verification System) theorem prover integrates a model checker, used to verify μ-calculus properties [RSS95]. Finally, MetaFrame is an environment for formal methods-based, application-specific software design, which supports model checking in the entire software development process [SMCB96].

Industrial case studies In industrial-size settings, case studies have been accomplished using model checking. They attest that model checking is starting to become applicable in industry, using existing tools or constructing its own model checkers.

- In the early 1990s AT&T used model checking techniques in the development of the International Telecommunication Union (ITU) ISDN User part. A part of the protocol written in SDL (about 7500 lines) were checked to satisfy 145 requirements; about 55% of them were detected to be logically inconsistent. Moreover, 112 errors were detected [CW96].

- In 1992, the model checker SMV was used from a research group at CMU for the verification of the IEEE Futurebus+ standard 896.1-1991. This is also the first employment of model checking techniques to verify an IEEE standard [CW96].
- NASA's Deep Space One flight software was verified in 2001 using the SPIN model checker. The verification reveals a know error in the launch software, but more important, discovered another scenario under which a related error could happen. SPIN is also used in several other industrial case studies [GH02].
- In 1999, the verification of a protocol for controlling the switching between power on/off states in audio/video devices was successfully performed by UPPAAL [HLS99].
- Currently the Static Driver Verifier (SDV) tool, based on SLAM, is part of the beta of the Windows Driver Development Kit (DDK) [DKW08].

6.2.3 Bounded Model Checking

In the software verification community, BMC has been widely implemented, Currie et al. [CMH+00] provided the first implementation of a depth-bounded symbolic search for software. At the Carnegie Mellon University CBMC has been developed, as one of the first solutions of BMC for C programs, and the only one that also supports C++, SpecC and SystemC [CKY03, CKL04]. One of the principal application of CBMC is the verification of consistency between system-level circuit models given in SystemC or C and an implementation given in Verilog. F-SOFT is a tool for C code analysis, which implements SAT-based verification for BMC decision problems, static analyses and predicate abstraction [ISGG05].

Another variants of a BMC-based implementations, is a version of CBMC that generates a decision problem for an SMT solver for integer linear arithmetic [AMP09]. In [CR06], the authors evaluated the performance of constraint solvers on such decision problems.

Saturn is a SAT-based framework for C programs [XA05]. It has been used for to verify two selected properties of the Linux kernel code: NULL-pointer dereferences and locking API conventions. The authors showed that the tool is scalable enough to analyse the entire Linux kernel.

The EXE tool checks C programs for finding bugs in system-level software, combining explicit execution and path-wise symbolic simulation [YST+06]. It can check programs that contain arbitrary pointer type-casts, because it uses a low-level memory model.

TCBMC (Threaded-C Bounded Model Checking) is an extension of CBMC for concurrent C programs proposed by IBM [RG05]. They performed preliminary experiments: a naive concurrent implementation of bubble sort has been checked using TCMB.

6.3 Theorem Proving

Similar to model checking, the interest in theorem proving has been growing during the last years. In fact an increasing number of theorem provers and examples have been applied. In the following, we categorise some theorem provers with respect to their degree of automation:

User-guided automatic deduction tools Several tools are guided by a sequence of lemmas and definitions but each theorem is proved automatically using built-in heuristics for induction, lemma-driven rewriting, and simplification. Systems based on this approach are ACL2 [KM02], Eves [CKM+88], LP [GG88], Nqthm [BM88], Reve [Les83], and RRL [KM87]. In particular, Nqthm has been used to verify a proof of Gödel's first incompleteness theorem and in other large-scale verification problems. Jahob [KR06] is an example of a system for program correctness based on theorem proving. This system analyses annotated programs written in a subset of Java describing annotations with formulas in a subset of Isabelle [WPN08] as the specification language. VCC [DMS+09] is an automatic tool that takes as input annotated C code and verifies partial correctness. The annotation language is classical first-order predicate logic with C-like syntax, such that developers can easily understand the annotations, which can be kept inline with the code. Automatic theorem provers such as E [SLKL04], SPASS [Wei01], and Vampire [RV01] are based on the resolution principle. They have gained a high degree of sophistication, however they reason in first-order logic that may be a limitation in some area. TPS [AB96] and LEO-II [BPTF08] are higher-order logic automatic theorem provers. TPS can only cope with small problems, because it is based on an old architecture, whereas LEO-II is designed for collaboration with provers for first-order and propositional logic. An example for an first-order automatic theorem prover's verification had been the braking of security application programming interfaces in the Cambridge-MIT work [YAB+05]. Cohen's security protocol verifier [Coh00] is another example which uses SPASS.

Interactive theorem provers The Edinburgh Logical Framework (LF) [HHP87] is a widely used meta-language for representing proof systems. The proof checking is reduced to LF type checking. The Edinburgh Logical Framework is based on a general treatment of syntax, rules, and proofs by means of a typed λ-calculus with dependent types. The logical systems are constructed by a formal system that generates formal presentations. LF also includes an informal method of finding such presentations. The Twelf system relies on the LF type theory [PS99]. It is a tool for experimentation in the theory of programming languages and logics. The specification of object languages and their semantics, implementation of algorithms manipulating object-language expressions, deductions, and formal development of the meta-theory of an object language are supported by Twelf.

In the 1970s, the Stanford LCF [GMW79, CAA+86] has been developed as one of the first proof checkers. The general purpose programming language ML was introduced in LCF, to permit users to write theorem proving tactics. There are propositions of a special "theorem" abstract data type, in LCF, to specify theorems. In order to guarantee that theorems are derived using only the inference rules given by the operations of the abstract type, the system relies on the ML type system. Successors of LCF include Coq, HOL and Isabelle.

Coq [FHB+97] is a proof assistant for type theory, allowing the development of computer programs consistent with their formal specification. Coq supports to express mathematical assertions, and then mechanically proves the proofs of these assertions. It works to individuate formal proofs, and from the constructive proof of its formal specification, proves a program. Coq comprises automatic theorem proving tactics and decision procedures, despite it is not an automated theorem prover. LEGO [LP92] is a tool for interactive proof development in the natural deduction style and it supports refinement proof as a basic operation. It has been employed in a proof of the strong normalisation theorem of the second-order λ-calculus, the proof of Tarski's fixed point theorem, program specifications, program correctness proofs, and other scientific problems.

HOL (Higher Order Logic) [Gor87] denotes a set of interactive theorem proving systems sharing similar logics and implementation strategies. Initially, HOL was destined to be used for the specification and verification of hardware systems, nevertheless it has been applied also to other sectors. At Cambridge University, Mike Gordon has modified the Edinburgh LCF system creating the original HOL system, by adding a version of Church's higher order logic as the object language of the system. Edinburgh ML (implemented on top of Lisp) was the meta-language used for encoding the logic. After some years of further development, in 1988 an improved version of HOL named HOL88 was released. Slind at the University of Calgary elaborated a port of HOL88 to SML: HOL90 (released in 1990) that ran on Poly/ML and SML/NJ. In this version the Lips substrate was removed and also some LCF technologies was reimplemented or trimmed off. In 1998, HOL98 was released. It was based on a new design and introduced a port to MoscowML [NS]. The last version of HOL is HOL4 [SN08], which is currently running on top of Poly/ML and MoscowML and which is also the supported version of the system for the international HOL community. Compared to the precedent HOL releases, HOL4 brought in more novelties.

Isabelle [WPN08] is a generic proof assistant and a successor of HOL. It comprises tools for proving mathematical formulas expressed in a formal language, through a logical calculus. The formalisation of mathematical proofs and, in particular, formal verification is the main application of Isabelle. In particular, this includes proving the correctness and properties of systems, computer languages and protocols. Isabelle/HOL is nowadays the most widespread instance of Isabelle, which supports a higher-order logic theorem proving environment and comprises a powerful specification tool for data types, inductive definitions, and functions with complex pattern matching [Kra10]. The language Isar is a structured proof language used in Isabelle for conducting proofs, understandable for both humans and computers. Following the general idea of natural deduction, the Isabelle/Pure meta-logic permits the formalisation of the syntax and inference rules of a wide range of object logics [Pau86, Pau90]. Isabelle logical core is implemented concording to the "LCF approach" of secure inferences as abstract datatype constructors in ML [GMW79].

Sophisticated extra-logical infrastructure supporting structured proofs and specifications are provided in Isabelle/Isar, including ideas for modular theory development. Isabelle/ZF (Zermelo-Fraenkel set-theory, see [Pau98, Pau00]) and Isabelle/HOLCF (Scott's domain theory within HOL) [MNOS99] are other worthy object logics, that have been applied to formalise and verify hard problems in mathematics and in program verification.

Some hard number-theoretic problems due to Ramanujam have been successfully checked by Analytica, which combines theorem proving with the symbolic algebra system Mathematica [CZ92]. Powerful decision procedures and model checking are compounded with interactive proof in the tools PVS [ORS92] and STeP [BBC+96]. Notable examples of the PVS verification using a high-order logic, are: hardware designs, reactive real-time, and fault-tolerant algorithms.

Industrial case studies

- The Pentium FDIV bug that caused Intel to take a $475 million charge against revenues, was a problem in the lookup table of an SRT divider. The SRT is a popular method for division in many microprocessor implementations. After that bug, various communities have successfully verified the system [Pra95]. A group from Stanford Research Institute provided a verified treatment of the general theory of SRT division, using the PVS theorem prover.
- In the mid nineties, Motorola has formally specified its Complex Arithmetic Processor used for Digital Signal Processing (DSP), working with the ACL2 theorem prover. During the design process of various DSP algorithms, their binary micro code has been continuously verified.
- In the Verisoft project a computer system from the hardware, up to an operating system kernel [ASS08] and a compiler for a C-dialect [LP08] have been formalised. The L4 operating system microkernel was proved in the L4.verified project [HEK$^+$07, TKN07], relating an abstract specification, a Haskell model, and the C code.

7 SPES Survey Results

On April 23rd 2009 the Technische Universität München distributed a survey to all partners of the SPES project. The scope of the survey was to obtain an idea on which tools and methods are currently used in industry, for the development of embedded systems, and so derive requirements for a next generation of industrial tools. The 25 questions cover programming languages, tools, operating systems, and technologies to develop embedded systems. The project partners are industries, universities, and research centres in the domain of avionics, medical, automotive, energy, and automation engineering. We contacted all the SPES partners and until July 31th, 2009, we altogether received 24 completely filled out questionnaires.

There are two questions concerning analysis techniques, for collecting the methods and tools utilised by the partners. The first (Figure 3) collects the methods and the second one (Figure 4) the tools, used for quality assurance. For both the questions the companies had the possibility of response with multiple answers. Almost every project performs manual tests and code reviews for quality assurance (96% and 83%, respectively). Other formal methods like test case generation and runtime verification are adopted in 63% of the projects. The respondents also named other methods like airplane testing, automated testing, design review, model review, requirements review, based on the guidelines for development of aviation software RTCA/DO-178B, system simulation, and system testing. With a share of 38%, Polyspace (cf. Section 6) is the most popular quality assurance tool. Although MATLAB/Simulink itself is widespread, the Simulink Design Verifier is not used in many cases. 38% of the respondents have not specified a tool, and thus do not seem to use a tool for quality assurance at all. 21% of the respondents named other tools like Automatic Test Sequenzer, FTI test environment, PC-lint, TAU, TechSat ADS-2 Test-Benches, and XLINT. The high number of tools used for quality assurance provides evidence for the heterogeneity of the tool landscape in industry.

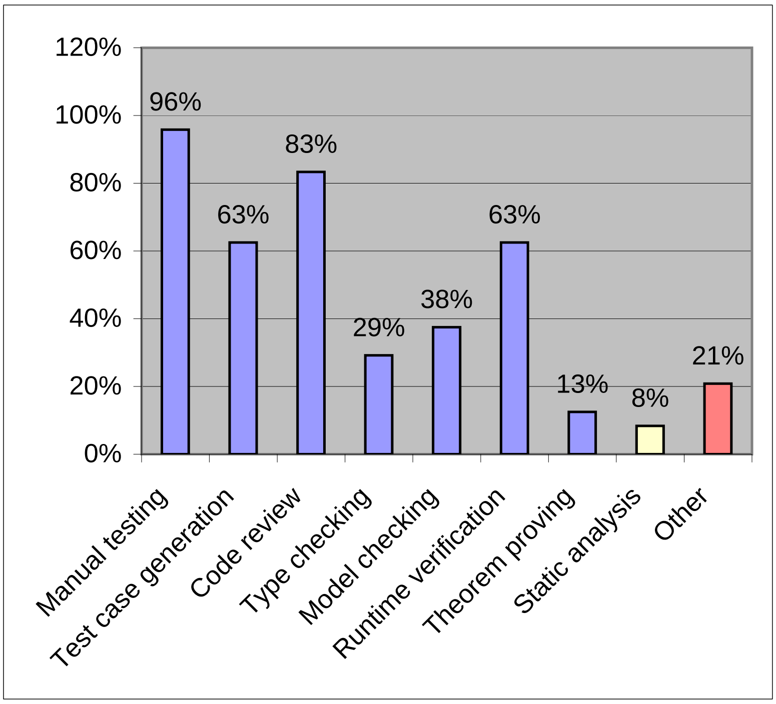


Figure 3: Methods used for quality assurance

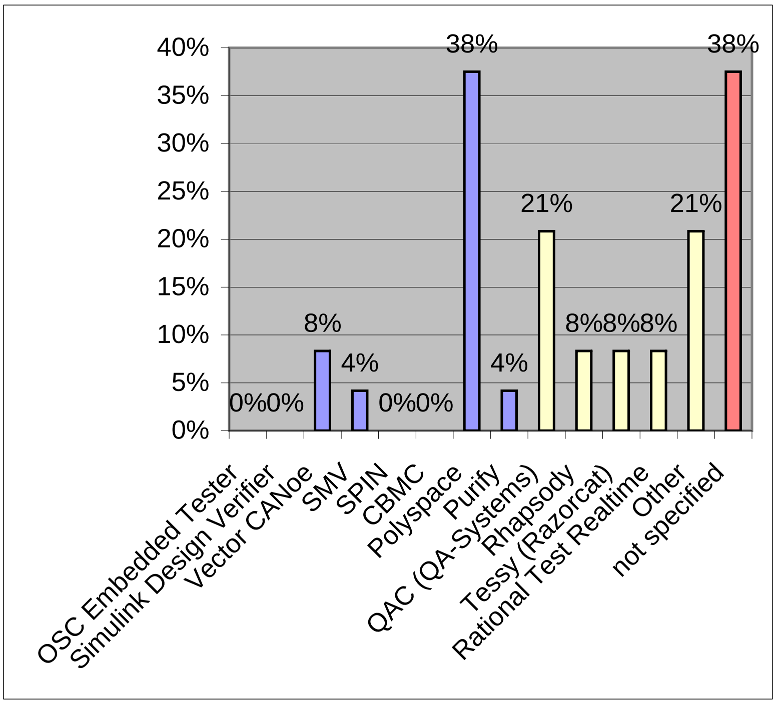


Figure 4: Tools used for quality assurance

8 Techniques Comparison

The presented analysis techniques have different characteristics and purposes, we summarise these differences making some significant direct comparisons between a couple of theories.

Runtime Verification versus Model Checking There are many similarities between runtime verification and model checking, nevertheless there are also important differences. In model checking, to answer whether the system satisfies a given property all execution paths have to be analysed. In terms of language theory, model checking corresponds to the language inclusion problem. Instead, runtime verification deals with the word problem, that has in most of the logical systems a lower complexity than the inclusion problem [BLS09].

Runtime verification performs the verification on finite traces, whereas model checking, specially in case of LTL, deals with infinite traces. Moreover, the finite executions in runtime verification (especially with online monitoring), have an increasing size, instead in model checking the model is complete for considering arbitrary positions of the trace.

Runtime Verification versus Testing Runtime verification is not an exhaustive technique, because it does not evaluate each execution of a system, but only a finite subset. This is an important characteristic that is in common with testing, so both the methods are usually incomplete. In testing, usually a test-case considering a finite number of input-output sequences of the system is defined. Next the test is executed using the input values and it is checked whether the resulting output values correspond to the expected ones.

However, there is a type of testing that is similar to runtime verification: oracle-based testing. There, the produced output values of the system under test, for a given test case input, are compared to those provided by an oracle. In this sense, the system is observed by the oracle and hence in terms of runtime verification the oracle behaves as a monitor. Runtime verification can be interpreted as a form of testing, nevertheless there are differences in the emphases. Typically, an oracle is defined directly, rather than generated from some high-level specification. Contrariwise in runtime verification, we do not consider the preparation of a set of input sequences to "exhaustively" test a system [BLS09].

Model Checking versus Theorem Proving There is a fundamental difference between theorem proving and model checking. We have to point out that theorem provers do not necessitate to exhaustively visit the system's state space to verify a property.

Since theorem provers reason about constraints on states, and not instances of states; they can deal with infinite state spaces and state spaces regarding complex data types and recursion. Furthermore, the proofs in theorem proving are explored in the syntactic domain, which is typically smaller than the semantic domain explored by model checkers. Therefore, systems with complex data structures, but simple information flow are adequate for theorem provers. Even though theorem proving is limited in the support for fully automatic verification in some cases [Duf91], a satisfactory level of automation is reached with matured theorem provers [DDN$^+$03]. The verification of inductive structures, for instance trees, stacks, or lists, cannot be automated, however can be computed through mathematical induction. Anyway, the lack of automation can be acceptable in some cases for increased capabilities, also because this type of analysis cannot be performed by model checkers.

The advantages of theorem provers over model checkers are, viz the superior dimension of the system to verify and the possibility to reason inductively. But there are also disadvantages, as for instance their proof can be highly large [ASM+08] and difficult to understand, furthermore they require a great portion of user expertise and effort to be utilised [AVM03]. In particular this last point may be the greatest obstacle to the diffusion of theorem proving.

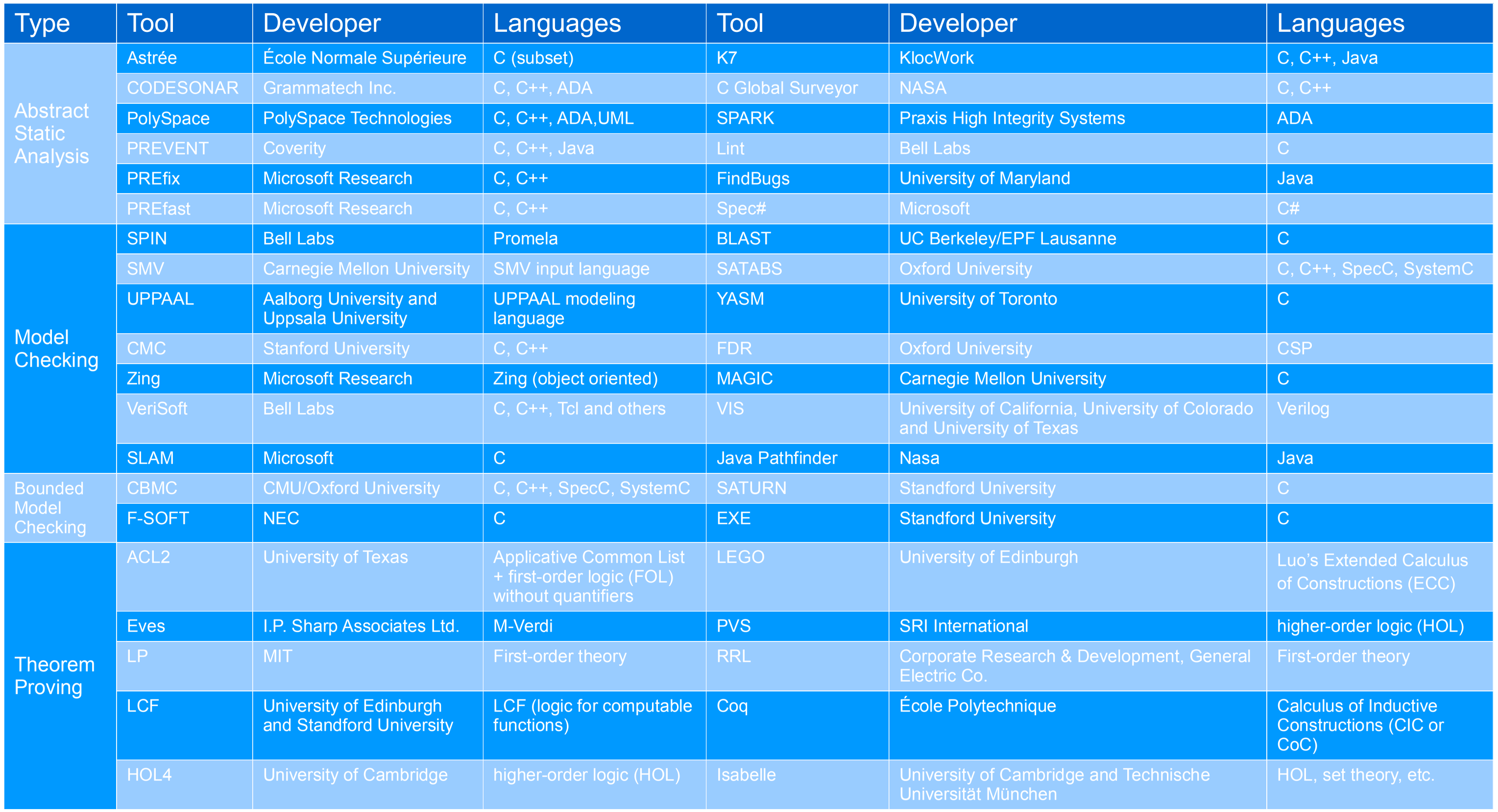
There have been appreciable attempts in the past, to integrate model checking and theorem proving [Sha00, AKMR03], despite they seam to be contradictory approaches. Their characteristics in terms of automation and scalability give complementary benefits, therefore it is possible that this trend will continue and model checking will be used on systems with a reasonable size, whereas theorem proving will be used on sizeable systems [Hol03].

9 Conclusion

We presented major analysis techniques and their actual state in research and industry. We conclude that the use of analysis techniques has proven to be successful in verifying safety-critical software, protocol standards, embedded systems, and industrial hardware designs. The role of analysis techniques is increasing in the system development process, due to the benefits of their use and to the commercial interest to produce high-quality systems. In the future, when specification and analysis tools will be more effectively integrated into the development tools, companies will utilise those methods in the development process more and more. Today they are not widespread employed, also due to the difficulties to understand the tools and the specification of the properties to be analysed. For formal verification, in particular, research is directed in developing tools and notations that are easy to use and comprehensible for developers, that are no domain experts in the field of formal verification.

A Formal Verification Taxonomy

We summarise in Figure 5 the major tools used in formal verification.



Type	Tool	Developer	Languages	Tool	Developer	Languages
Abstract Static Analysis	Astrée	École Normale Supérieure	C (subset)	K7	KlocWork	C, C++, Java
	CODESONAR	Grammatech Inc.	C, C++, ADA	C Global Surveyor	NASA	C, C++
	PolySpace	PolySpace Technologies	C, C++, ADA,UML	SPARK	Praxis High Integrity Systems	ADA
	PREVENT	Coverity	C, C++, Java	Lint	Bell Labs	C
	PREfix	Microsoft Research	C, C++	FindBugs	University of Maryland	Java
	PREfast	Microsoft Research	C, C++	Spec#	Microsoft	C#
Model Checking	SPIN	Bell Labs	Promela	BLAST	UC Berkeley/EPF Lausanne	C
	SMV	Carnegie Mellon University	SMV input language	SATABS	Oxford University	C, C++, SpecC, SystemC
	UPPAAL	Aalborg University and Uppsala University	UPPAAL modeling language	YASM	University of Toronto	C
	CMC	Stanford University	C, C++	FDR	Oxford University	CSP
	Zing	Microsoft Research	Zing (object oriented)	MAGIC	Carnegie Mellon University	C
	VeriSoft	Bell Labs	C, C++, Tcl and others	VIS	University of California, University of Colorado and University of Texas	Verilog
	SLAM	Microsoft	C	Java Pathfinder	Nasa	Java
Bounded Model Checking	CBMC	CMU/Oxford University	C, C++, SpecC, SystemC	SATURN	Standford University	C
	F-SOFT	NEC	C	EXE	Standford University	C
Theorem Proving	ACL2	University of Texas	Applicative Common List + first-order logic (FOL) without quantifiers	LEGO	University of Edinburgh	Luo's Extended Calculus of Constructions (ECC)
	Eves	I.P. Sharp Associates Ltd.	M-Verdi	PVS	SRI International	higher-order logic (HOL)
	LP	MIT	First-order theory	RRL	Corporate Research & Development, General Electric Co.	First-order theory
	LCF	University of Edinburgh and Standford University	LCF (logic for computable functions)	Coq	École Polytechnique	Calculus of Inductive Constructions (CIC or CoC)
	HOL4	University of Cambridge	higher-order logic (HOL)	Isabelle	University of Cambridge and Technische Universität München	HOL, set theory, etc.

Figure 5: Formal Verification Taxonomy

References

[AB96] Peter B. Andrews and Matthew Bishop. On sets, types, fixed points, and checkerboards. In *TABLEAUX '96: Proceedings of the 5th International Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, pages 1–15, London, UK, 1996. Springer.

[AD90] Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In Mike Paterson, editor, *ICALP*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer, 1990.

[AHH96] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho. Automatic Symbolic Verification of Embedded Systems. *IEEE Transactions on Software Engineering*, 22:181–201, 1996.

[AKMR03] Konstantine Arkoudas, Sarfraz Khurshid, Darko Marinov, and Martin Rinard. Integrating Model Checking and Theorem Proving for Relational Reasoning. In *Seventh International Seminar on Relational Methods in Computer Science (RelMiCS 2003)*, volume 3015 of *Lecture Notes in Computer Science*, pages 21–33, Malente, Germany, May 2003 2003.

[AMP09] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *Int. J. Softw. Tools Technol. Transf.*, 11(1):69–83, 2009.

[AQRX04] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, and Yichen Xie. Zing: Exploiting Program Structure for Model Checking Concurrent software. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2004.

[ASM$^+$08] Rev R. Acad, Cien Serie, A. Mat, Matt Kaufmann, and J Strother Moore. Ciencias de la Computación / Computational Sciences Some Key Research Problems in Automated Theorem Proving for Hardware and Software Verification. 2008.

[ASS08] Eyad Alkassar, Norbert Schirmer, and Artem Starostin. Formal Pervasive Verification of a Paging Mechanism. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 109–123. Springer, 2008.

[AVM03] Myla Archer, Ben Di Vito, and César Muñoz. Developing User Strategies in PVS: A Tutorial. In *Proceedings of Design and Application of Strategies/Tactics in Higher Order Logics STRATA'03*, NASA/CP-2003-212448, NASA LaRC,Hampton VA 23681-2199, USA, September 2003.

[Bar03] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[BBC$^+$96] Nikolaj Bjørner, Anca Browne, Eddie Chang, Michael Colón, Arjun Kapur, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe. STeP: Deductive-Algorithmic Verification of Reactive and Real-time Systems, 1996.

[BCC+02] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. pages 85–108, 2002.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, London, UK, 1999. Springer.

[BCDM86] Micheal C. Browne, Edmund M. Clarke, David L: Dill, and Bud Mishra. Automatic Verification of Sequential Circuits Using Temporal Logic. *IEEE Trans. Comput.*, 35(12):1035–1044, 1986.

[BCL+94] Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. Mcmillan, and David L. Dill. Symbolic Model Checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13:401–424, 1994.

[BCLR04] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In *IFM*, pages 1–20. Springer, 2004.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on Uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04, Proceedings)*, number 3185 in Lecture Notes in Computer Science, pages 200–236. Springer, September 2004.

[BHSV+96] Robert K. Brayton, Gary D. Hachtel, Alberto L. Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu-Tsung Cheng, Stephen A. Edwards, Sunil P. Khatri, Yuji Kukimoto, Abelardo Pardo, Shaz Qadeer, Rajeev K. Ranjan, Shaker Sarwary, Thomas R. Shiple, Gitanjali Swamy, and Tiziano Villa. Vis: A system for verification and synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *8th International Conference on Computer Aided Verification (CAV'96, Proceedings)*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer, 1996.

[BJ97] Bishop Brock and Warren A. Hunt Jr. Formally Specifying and Mechanically Verifying Programs for the Motorola Complex Arithmetic Processor DSP. In *ICCD*, pages 31–36, 1997.

[BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. pages 49–69. Springer, 2004.

[BLS09] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime Verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2009. accepted for publication.

[BM88] Robert S. Boyer and J. Strother Moore. *A computational logic handbook.* Academic Press Professional, Inc., San Diego, CA, USA, 1988.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *PLDI*, pages 203–213, 2001.

[BPR01] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking c programs. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 268–283, London, UK, 2001. Springer.

[BPTF08] Christoph Benzmüller, Lawrence C. Paulson, Frank Theiss, and Arnaud Fietzke. LEO-II - A Cooperative Automatic Theorem Prover for Classical Higher-Order Logic (System Description). In *IJCAR '08: Proceedings of the 4th international joint conference on Automated Reasoning*, pages 162–170, Berlin, Heidelberg, 2008. Springer.

[BR00a] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130. Springer, 2000.

[BR00b] Thomas Ball and Sriram K. Rajamani. Boolean Programs: A Model and Process for Software Analysis. Technical report, February 2000.

[BR02] Thomas Ball and Sriram K. Rajamani. Generating Abstract Explanations of Spurious Counterexamples in C Programs. Technical Report MSR-TR-2002-09, Microsoft Research, 2002.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[CAA$^+$86] Robert L. Constable, Stuart F. Allen, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, Scott F. Smith, James T. Sasaki, and S. F. Smith. Implementing Mathematics with The Nuprl Proof Development System, 1986.

[CC77a] Patrick Causot and Radhia Cousot. Static determination of dynamic properties of generalized type unions. In *Proceedings of an ACM conference on Language design for reliable software*, pages 77–94, 1977.

[CC77b] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM Press, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282, New York, NY, USA, 1979. ACM.

[CC04] Patrick Cousot and Radhia Cousot. Basic concepts of abstract interpretation. pages 359–366. 2004.

[CCG$^+$03] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in c, 2003.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer.

[CES86] Edmund M. Clarke, E. Allen Emerson, and Aravinda Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96, New York, NY, USA, 1978. ACM.

[Cha88] K. Mani Chandy. *Parallel program design: a foundation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[cit05] IEEE Std 1012 - 2004 IEEE Standard for Software Verification and Validation. Technical report, 2005.

[CK96] Edmund M. Clarke and Robert P. Kurshan. Computer-aided verification. *IEEE Spectr.*, 33(6):61–67, 1996.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[CKM$^+$88] Dan Craigen, Sentot Kromodimoeljo, Irein Meisels, Andy Neilson, Bill Pase, and Mark Saaltink. m-EVES: A tool for verifying software. In *ICSE '88: Proceedings of the 10th international conference on Software engineering*, pages 324–333, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[CKS05a] Byron Cook, Daniel Kroening, and Natasha Sharygina. Cogent: Accurate theorem proving for program verification. In *Proceedings of CAV 2005, volume 3576 of Lecture Notes in Computer Science*, pages 296–300. Springer, 2005.

[CKS05b] Byron Cook, Daniel Kroening, and Natasha Sharygina. Symbolic model checking for asynchronous boolean programs. In *SPIN*, pages 75–90. Springer, 2005.

[CKS06] Byron Cook, Daniel Kroening, and Natasha Sharygina. Over-Approximating Boolean Programs with Unbounded Thread Creation. In *FMCAD '06: Proceedings of the Formal Methods in Computer Aided Design*, pages 53–59, Washington, DC, USA, 2006. IEEE Computer Society.

[CKSY03] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Predicate Abstraction of ANSI-C Programs using SAT. In *Formal Methods in System Design (FMSD), 25:105– 127, September–November*, page 2004, 2003.

[CKY03] Edmund M. Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and verilog programs using Bounded Model Checking. In *DAC '03: Proceedings of the 40th annual Design Automation Conference*, pages 368–371, New York, NY, USA, 2003. ACM.

[CLB03] Marcus Ciolkowski, Oliver Laitenberger, and Stefan Biffl. Software reviews: The state of the practice. *IEEE Software*, 20(6):46–51, 2003.

[CM04] Séverine Colin and Leonardo Mariani. Run-Time Verification. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 525–555. Springer, 2004.

[CMH$^+$00] David W. Currie, Billerica Ma, Alan J. Hu, Sreeranga Rajan, Masahiro Fujita, and Sunnyvale Ca. Automatic Formal Verification of DSP Software, 2000.

[CMMP95] Tim Coe, Terje Mathisen, Cleve Moler, and Vaughan Pratt. Computational aspects of the pentium affair. *Computing in Science and Engineering*, 2(1):18–31, 1995.

[Coh00] Ernie Cohen. Taps: A first-order verifier for cryptographic protocols. *Computer Security Foundations Workshop, IEEE*, 0:144, 2000.

[Cou05] Patrick Cousot. The verification grand challenge and abstract interpretation. In Bertrand Meyer and Jim Woodcock, editors, *VSTTE*, volume 4171 of *Lecture Notes in Computer Science*, pages 189–201. Springer, 2005.

[CPR05] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction refinement for termination. In Chris Hankin and Igor Siveroni, editors, *SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 2005.

[CPS94] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15, 1994.

[CR06] Hélène Collavizza and Michel Rueher. Exploration of the Capabilities of Constraint Programming for Software Verification. In Hermanns and Palsberg [HP06], pages 182–196.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, December 1996. Report by the Working Group on Formal Methods for the ACM Workshop on Strategic Directions in Computing Research.

[CZ92] Edmund M. Clarke and Xudong Zhao. Analytica - A Theorem Prover for Mathematica. Technical report, Pittsburgh, PA, USA, 1992.

[DB95] David Déharbe and Dominique Borrione. Semantics of a Verification-Oriented Subset of VHDL. In *CHARME'95, volume 987 of Lecture Notes in Computer Science*, pages 293–310. Springer, 1995.

[DDHY92] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol Verification as a Hardware Design Aid. In *ICCD '92: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, pages 522–525, Washington, DC, USA, 1992. IEEE Computer Society.

[DDN$^+$03] David Detlefs, David Detlefs, Greg Nelson, Greg Nelson, James B. Saxe, and James B. Saxe. Simplify: A theorem prover for program checking. Technical report, J. ACM, 2003.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[DJS95] Werner Damm, Bernhard Josko, and Rainer Schlör. Specification and verification of VHDL-based system-level hardware designs. pages 331–409, 1995.

[DKW08] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1165–1178, July 2008.

[DMS$^+$09] Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. VCC: Contract-based Modular Verification of Concurrent C, 2009.

[DPG96] G. De Palma and A. Glaser. Formal verification augments simulation. In *Electrical Engineering Times 56*, 1996.

[Duf91] David A. Duffy. *Principles of automated theorem proving.* John Wiley & Sons, Inc., New York, NY, USA, 1991.

[DY95] Conrado Daws and Sergio Yovine. Two examples of verification of multirate timed automata with Kronos. In *IEEE Real-Time Systems Symposium (RTSS'95, Proceedings)*, pages 66–75. IEEE Computer Society Press, 1995.

[ES01] Javier Esparza and Stefan Schwoon. A BDD-based Model Checker for Recursive Programs. In *In Proc. CAV'01, LNCS 2102*, pages 324–336. Springer, 2001.

[Fag99] Michael E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 38(2/3):258–287, 1999.

[FGK$^+$96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, and Mihaela Sighireanu. CADP - A Protocol Validation and Verification Toolbox. In *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*, pages 437–440, London, UK, 1996. Springer.

[FHB$^+$97] Jean-Christophe Filliâtre, Hugo Herbelin, Bruno Barras, Bruno Barras, Samuel Boutin, Eduardo Giménez, Samuel Boutin, Gérard Huet, César Muñoz, Cristina Cornes, Cristina Cornes, Judicaël Courant, Judicael Courant, Chetan Murthy, Chetan Murthy, Catherine Parent, Catherine Parent, Christine Paulin-mohring, Christine Paulin-mohring, Amokrane Saibi, Amokrane Saibi, Benjamin Werner, and Benjamin Werner. The Coq Proof Assistant - Reference Manual Version 6.1. Technical report, 1997.

[FLL$^+$02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM.

[FSS$^+$94] T. Filkorn, H. Schneider, A. Scholz, A. Strasser, and P. Warkentin. SVE User's Guide. In *Tech. Rep. ZFE BT SE 1-SVE-1*, Munich, Germany, 1994. Siemens AG, Corporate Research and Development.

[GC06] Arie Gurfinkel and Marsha Chechik. Why Waste a Perfectly Good Abstraction? In Hermanns and Palsberg [HP06], pages 212–226.

[GG88] Stephen J. Garland and John V. Guttag. Inductive methods for reasoning about abstract data types. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 219–228, New York, NY, USA, 1988. ACM.

[GH02] Peter R. Gluck and Gerard J. Holzmann. Using SPIN Model Checking for Flight Software Verification. In *In the Proceedings of 2002 IEEE Aerospace Conference*, pages 1–105–1–113 vol.1, 2002.

[GMW79] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[God97] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186, New York, NY, USA, 1997. ACM.

[Gor87] Mike Gordon. HOL : A proof generating system for higher-order logic. Technical Report UCAM-CL-TR-103, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500, January 1987.

[GPV$^+$95] Rob Gerth, Doron Peled, Moshe Y. Vardi, R. Gerth, Den Dolech Eindhoven, D. Peled, M. Y. Vardi, and Pierre Wolper. Simple On-the-fly Automatic Verification of Linear Temporal Logic. In *In Protocol Specification Testing and Verification*, pages 3–18. Chapman & Hall, 1995.

[HBK93] Ramin Hojati, Robert K. Brayton, and Robert P. Kurshan. BDD-Based Debugging of Designs Using Language Containment and Fair CTL. In C. Courcoubetis, editor, *Computer Aided Verification: Proc. of the 5th International Conference CAV'93*, pages 41–58. Springer, Berlin, Heidelberg, 1993.

[HEK$^+$07] Gernot Heiser, Kevin Elphinstone, Ihor Kuz, Gerwin Klein, and Stefan M. Petters. Towards trustworthy computing systems: taking microkernels to the next level. *SIGOPS Oper. Syst. Rev.*, 41(4):3–11, 2007.

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40:194–204, 1987.

[HJMM04] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. *SIGPLAN Not.*, 39(1):232–244, 2004.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 58–70, New York, NY, USA, 2002. ACM.

[HK90] Zvi Har'El and Robert P. Kurshan. Software for analytical development of communications protocols. *AT&T Bell Laboratories Technical Journal*, 69(1):45–59, 1990.

[HLS99] Klaus Havelund, Kim Guldstrand Larsen, and Arne Skou. Formal Verification of a Power Controller Using the Real-Time Model Checker UPPAAL. In Joost-Pieter Katoen, editor, *ARTS*, volume 1601 of *Lecture Notes in Computer Science*, pages 277–298. Springer, 1999.

[HM09] Markus Herrmannsdoerfer and Stefano Merenda. Result of the tool questionnaire. Technical Report TUM-I0929, Technische Universität München, 2009.

[HNSY94] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:394–406, 1994.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, October 1969.

[Hoa87] C. A. R. Hoare. An overview of some formal methods for program design. *Computer*, 20(9):85–91, 1987.

[Hol91] Gerard J. Holzmann. *Design and validation of computer protocols.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[Hol03] Gerard J. Holzmann. Trends in software verification. In *In: Proceedings of the Formal Methods Europe Conference*, 2003.

[HP95] Gerard J. Holzmann and Doron Peled. An improvement in formal verification. In *Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques VII*, pages 197–211, London, UK, UK, 1995. Chapman & Hall, Ltd.

[HP06] Holger Hermanns and Jens Palsberg, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*, volume 3920 of *Lecture Notes in Computer Science*. Springer, 2006.

[HSTV08] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. FShell: systematic test case generation for dynamic analysis and measurement. In *CAV*, pages 209–213, 2008.

[HSTV09] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. Query-driven program testing. In *VMCAI*, pages 151–166, 2009.

[ISGG05] Franco Ivanicic, Ilya Shlyakhter, Aarti Gupta, and Malay K. Ganai. Model Checking C Programs Using F-SOFT. In *ICCD '05: Proceedings of the 2005 International Conference on Computer Design*, pages 297–308, Washington, DC, USA, 2005. IEEE Computer Society.

[JCE97] J. Baugh Jr., R. Cleaveland, and W. Elseaidy. Modeling and verifying active structural control systems. *Sci. Comput. Program.*, 29(1-2):99–122, 1997.

[Jon97] Capers Jones. Software quality in 1997: What works and what doesn´t. 1997.

[Kal94] Markus Kaltenbach. Model Checking for UNITY - The UV System Revision 1.10, 1994.

[KG99] Christoph Kern and Mark R. Greenstreet. Formal verification in hardware design: a survey. *ACM Trans. Des. Autom. Electron. Syst.*, 4(2):123–193, 1999.

[Klo01] Carlos Delgado Kloos. *Practical Formal Methods for Hardware Design.* Springer, Secaucus, NJ, USA, 2001.

[KM87] Deepak Kapur and David R. Musser. Proof by consistency. *Artif. Intell.*, 31(2):125–157, 1987.

[KM02] Matt Kaufmann and J Moore. A Computational Logic for Applicative Common Lisp. In *A Companion to Philosophical Logic*, pages 724–741, 2002.

[KR06] Viktor Kuncak and Martin Rinard. An Overview of the Jahob Analysis System - Project Goals and Current Status. In *In NSF Next Generation Software Workshop*, 2006.

[Kra10] Alexander Krauss. Partial and nested recursive function definitions in higher-order logic. *J. Autom. Reason.*, 44(4):303–336, 2010.

[Kur94a] Robert P. Kurshan. The complexity of verification. In *STOC '94: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 365–371, New York, NY, USA, 1994. ACM.

[Kur94b] Robert P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach.* Princeton University Press, Princeton, NJ, USA, 1994.

[LBD$^+$04] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fähndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, 2004.

[Les83] Pierre Lescanne. Computer experiments with the reve term rewriting system generator. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 99–108, New York, NY, USA, 1983. ACM.

[LMR$^+$98] Philippe Lacan, Jean N. Monfort, Le Vinh Quy Ribal, Alain Deutsch, and Georges Gonthier. ARIANE 5 – The Software Reliability Verification Process. pages 201—-205. Paris: European Space Agency, 1998.

[Low96] Gavin Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR. pages 147–166. Springer, 1996.

[LP92] Zhaohui Luo and Robert Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, Computer Science Dept., Univ. of Edinburgh, 1992.

[LP08] Dirk Leinenbach and Elena Petrova. Pervasive compiler verification – from verified programs to verified systems. *Electron. Notes Theor. Comput. Sci.*, 217:23–40, 2008.

[Lut93] Robyn R. Lutz. Analyzing software requirements errors in safety-critical, embedded systems. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 126–133, 1993.

[McM92] Kenneth Lauchlin McMillan. *Symbolic model checking: an approach to the state explosion problem.* PhD thesis, Pittsburgh, PA, USA, 1992.

[Mee05] B. Meenakshi. Formal verification. *Resonance*, 10(5):26–38, 2005.

[MNOS99] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oskar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.

[MPC$^+$02] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: a pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, 2002.

[NS] Michael Norrish and Konrad Slind. A thread of hol development. *Computer Journal*, 45:37–45.

[OL07] Martin Ouimet and Kristina Lundqvist. Formal software verification: Model checking and theorem proving. Technical Report, Mälardalen University, March 2007.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer.

[Pau86] Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[Pau90] Lawrence C. Paulson. Isabelle: The next 700 theorem provers, 1990.

[Pau98] Lawrence C. Paulson. Set Theory for Verification: I. From Foundations to Functions. *Journal of Automated Reasoning*, 11:353–389, 1998.

[Pau00] Lawrence C. Paulson. Set Theory for Verification: II. Induction and Recursion. *Journal of Automated Reasoning*, 15:167–215, 2000.

[Pel94] Doron Peled. Combining Partial Order Reductions with On-the-fly Model-Checking. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 377–390, London, UK, 1994. Springer.

[Pnu81] Amir Pnueli. A temporal logic of concurrent programs. In *Theoretical Computer Science 13*, pages 45–60, 1981.

[Pra95] Vaughan R. Pratt. Anatomy of the pentium bug. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT*, volume 915 of *Lecture Notes in Computer Science*, pages 97–107. Springer, 1995.

[PS99] Frank Pfenning and Carsten Schürmann. System Description: Twelf - A Meta-Logical Framework for Deductive Systems. In *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, Lecture Notes in Artificial Intelligence, pages 202–206. Springer, 1999.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer.

[RCJ$^+$08] Dieter Rombach, Marcus Ciolkowski, Ross Jeffery, Oliver Laitenberger, Frank McGarry, and Forrest Shull. Impact of research on practice in the field of inspections, reviews and walkthroughs: learning from successful industrial uses. *SIGSOFT Softw. Eng. Notes*, 33(6):26–35, 2008.

[RG05] Ishai Rabinovitz and Orna Grumberg. Bounded model checking of concurrent programs. In *In Computer-Aided Verification (CAV), LNCS 3576*, pages 82–97. Springer, 2005.

[Ros94] A. W. Roscoe. Model-checking CSP. pages 353–378, 1994.

[RS91] Valérie Roy and Robert de Simone. Auto/Autograph. In *CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification*, pages 65–75, London, UK, 1991. Springer.

[RSS95] S. Rajan, Natarajan Shankar, and Mandayam K. Srivas. An integration of model checking with automated proof checking. pages 84–97. Springer, 1995.

[RV01] Alexandre Riazanov and Andrei Voronkov. Vampire 1.1 (system description). In *IJCAR '01: Proceedings of the First International Joint Conference on Automated Reasoning*, pages 376–380, London, UK, 2001. Springer.

[SBH04] Sandeep K. Shukla, Tevfik Bultan, and Constance L. Heitmeyer. Panel: given that hardware verification has been an uphill battle, what is the future of software verification? In *MEMOCODE*, pages 157–158. IEEE, 2004.

[Sha00] Natarajan Shankar. Combining theorem proving and model checking through symbolic analysis. In *CONCUR '00: Proceedings of the 11th International Conference on Concurrency Theory*, pages 1–16, London, UK, 2000. Springer.

[SLKL04] Stephan Schulz, Risc Linz, Johannes Kepler, and Universität Linz. System description: E 0.81. In *Basin and Rusinowitch*, pages 223–228. Springer, 2004.

[SMCB96] Bernhard Steffen, Tiziana Margaria, Andreas Claßen, and Volker Braun. The METAFrame'95 Environment. In *Proc. CAV'96, LNCS*, pages 450–453. Springer, 1996.

[SN08] Konrad Slind and Michael Norrish. A brief overview of hol4. In *TPHOLs '08: Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, pages 28–32, Berlin, Heidelberg, 2008. Springer.

[TKN07] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 97–108, New York, NY, USA, 2007. ACM.

[VHB$^+$03] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[Vot93] Lawrence G. Votta, Jr. Does every inspection need a meeting? In *SIGSOFT '93: Proceedings of the 1st ACM SIGSOFT symposium on Foundations of software engineering*, pages 107–114, New York, NY, USA, 1993. ACM.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge, June 1986.

[WBKW07] Thomas Witkowski, Nicolas Blanc, Daniel Kroening, and Georg Weissenbacher. Model checking concurrent linux device drivers. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 501–504, New York, NY, USA, 2007. ACM.

[Wei01] Christoph Weidenbach. Combining superposition, sorts and splitting. pages 1965–2013, 2001.

[WPN08] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle Framework. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *21st International Conference on Theorem Proving in Higher-Order Logics (TPHOLs 2008, Proceedings)*, volume 5170 of *Lecture Notes in Computer Science*, pages 33–38. Springer, 2008.

[XA05] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. pages 351–363, 2005.

[YAB$^+$05] Paul Youn, Ben Adida, Mike Bond, Jolyon Clulow, Jonathan Herzog, Amerson Lin, Ronald L. Rivest, and Ross Anderson. Robbing the bank with a theorem prover. Technical report, 2005.

[YST$^+$06] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. Automatically Generating Malicious Disks using Symbolic Execution. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 243–257, Washington, DC, USA, 2006. IEEE Computer Society.