TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

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

SPES 2020 Deliverable D1.3.A

Einführung in Analysetechniken

Software Plattform Embedded Systems 2020

Author:	Judith Thyssen, TUM-SSE
Version:	1.0
Date:	31.12.2009
Status:	Released

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

Inhaltsverzeichnis

1 Einführung

Das Deliverable D1.3.A gibt eine Einführung in existierende Analysetechniken und enthält einen Vorschlag für die Auswahl von Analysetechniken, die im weiteren Projektverlauf im ZP-AP 1 bearbeitet werden sollen. Es umfasst die folgenden Teildokumente:

1. D1.3.A-1 „Analysis Techniques: State of the Art in Industry and Research“ (siehe Anhang A)
   Dieses Dokument gibt einen ausführlichen Überblick und eine Einführung in existierende Analysetechniken in der Wissenschaft und Praxis - von informellen Techniken wie Reviews, Inspections und Walkthroughs bis hin zu formalen Methoden wie Model Checking und Theorem Beweisen. Das Dokument dient als Ausgangsbasis für die Auswahl von geeigneten Analysetechniken.

2. D1.3.A-2 „Analysis Techniques suitable for SPES project“ (siehe Anhang A)
   Der Zweck dieses Dokuments ist zu skizzieren, welche Analyseverfahren im Rahmen von ZP-AP 1 in SPES mit der Modellierungstheorie FOCUS integriert werden können. Der Schwerpunkt liegt hierbei auf Methoden zur Überprüfung und Validierung von Modellen basierend auf formaler Verifikation (Model-Checking, Simulation und Theorem Beweiser).

A Deliverable D1.3.A-1

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:	1.0
Date:	October 23, 2009
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 correct behavior. The damages due to errors may cost money and even human lifes. There are methodologies known as Analysis Techniques, to verify, test and validate systems. In this document we show the different approaches used in Analysis Techniques, for testing and validating systems and then the state of the art in the research and in the industry.

The purpose of this document is to provide an introduction to existing Analysis Techniques. The integration of such techniques into the modelling theory it is one of the objective of work package ZP-AP 1.3.

Contents

1 Analysis Techniques

Hardware and software systems will inevitably grow in scale and functionality, that implies an increment of the overall complexity. Because of this increase in complexity, the likelihood of errors is much greater. Moreover, some of these errors may cause loss of money or even human lifes. A major goal of software engineering is to enable developers to construct systems that operate reliably despite this complexity.

Introduction Computers are ubiquitous and used to control various safety critical systems like aircrafts, satellites, medical instruments, etc. Software that is developed to operate these systems is complex, consisting of many modules, each with thousands of lines of code. Any errors in the functioning of these programs results in loss of life or money. An example of such a failure is that of the Ariane 5 rocket, which exploded in June 1996, less than forty seconds after it was launched. The reason for this explosion was a software error in the computer that was responsible for calculating the rocket's horizontal velocity. An exception was caused during the conversion of a 64-bit floating point number into a 16-bit signed integer. Another famous error is the "Pentium bug" in 1994, which caused Intel to recall faulty chips and take a loss of $475 million [CMMP95]. Since this event, Analisys Techniques of hardware systems have been commonplace using mostly model checkers but also theorem provers [ASM+08], [84697]. Another important aspect of these examples (and many more) indicate the need for developing reliable hardware and software systems. We have also to consider that through the use of Analysis Techniques, we have a more efficent specification and verification of the undesired behaviours. Therefore we can 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 characterization of the 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 appliable to software systems, we refer an analog for hardware verification [KG99]. Hardware design typically starts with a high-level specification, given in terms of block diagrams, tables, and informal text conveying the desired functionality. A combination of top-down and bottom-up design techniques are applied until a final design is obtained. Verification of the design involves checking that the physical design does indeed meet its specification. In a traditional design flow, this is accomplished through simulation and testing.

Because exhaustive testing for nontrivial designs is generally infeasible, testing provides at best only a probabilistic assurance. Verification, in contrast to testing, uses rigorous mathematical reasoning to show that a design meets all or parts of its specification.

Based on these definitions we consider the domains and the purposes of the Analysis Techniques.

Outline We report the major verification methods that are being used in industry and their actual state in research. In the Section 2 we describe the traditional testing and simulation, inspections, reviews and walkthroughs in Section 3 and runtime verification in Section 4. Then in the Section 5 we explain what we mean by general techniques named Formal Verification. We describe in Section 6 some successful case studies and well-known tools that represent the state of the art and then in Section 7 we report the results concerning the Analysis Techniques of the SPES survey. In Section 8 a comparison among different Analysis Techniques types and conclusions in Section 9.

2 Testing and Simulation

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

Development life cycle A typical system development life cycle as in Figure 1 [Mee05]. In this diagram the first step is the analysis of the requirements that describes properties of the system to be developed. This is followed by design phase which involves developing a high-level (followed by a detailed) layout of the system components. The next step is the coding phase which usually also includes testing of the individual modules or parts of the systems that is in development. Coding is followed by components testing and a successful results in this phase leads to the software deployment in the market for being used and so maintained. During the development and also afterwards, in each of these phases it is possible to introduct errors or not correct behaviors that influence the correctness of the whole system.

Testing and simulation are well-known techniques used to check that the system is correct respect to its functionality and requirements. Testing is any technique of checking systems including execution of test cases and program proving. Vigorous testing and code review techniques are available to test the software written in almost any programming language. These techniques are considered highly effective and few coding bugs of any significance escape detection.

Over many years, Software Productivity Research collected an extensive data base of software defects and the efficiency of various defect removal techniques [Jon97]. Considering these results few defects in the final product are due to bugs contributed during coding phase, they are 35%. For example, of 197 critical faults detected during the testing phase of the Voyager and Galileo spacecraft, just 3 were coding errors [Lut93]. About 20% of the faults were traced to requirements, 30% to design and the rest to other type of errors. The majority of faults in software development arise during requirements and design whereas less to coding.

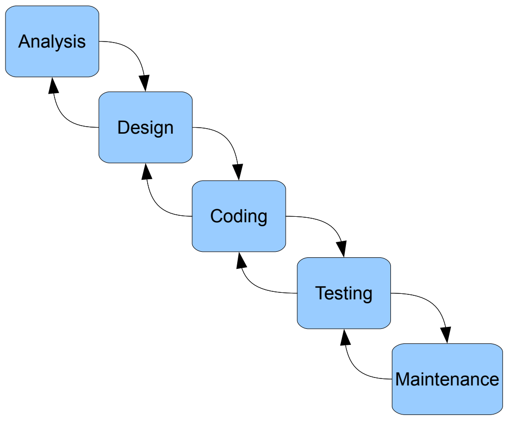


Figure 1: The development life cycle

Another problem with the techniques of testing and simulation is that they do not cover "all" possible behaviours of the system being developed. Behaviour of a system refers to the way in which the values of the variables in the program change during the execution, considering also different inputs. A state of a system typically represent the values of all the variables involved, the location where each component of the system resides in, etc. The state space of a system is the set of all possible states the system can reach.

Usually, programs have a large number of states, one for every possible value that a variable can take. Take, for example, a program which has an integer variable x, it can take as many values as allowed by the memory available, each representing a different state. If memory limitations are abstracted away, the program can be considered to have infinitely many states. Testing and simulation work by explicit substitution of values to variables and it is impossible to cover all possible behaviours for such a program. Consequently, critical behaviours might be left out while testing and these might correspond to the system failing.

3 Inspections, Reviews and Walkthroughs

We report and refer to the following definition: software inspections, reviews and walkthroughs are humanbased methods for analyzing 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 having a comprehensive overview, we consider the survey initiated in 2002, from the International Software Engineering Research Network (www.iese.fhg.de/ISERN) and the Fraunhofer Institute for Experimental Software Engineering (IESE) and its evaluations for the state of software review practice [RCJ+08]. From the results obtained the main perceived benefits of software reviews are improved communication among developers, evaluation of project status, or enforcing standards. We can conclude that many companies do not exploit the full potential of software reviews for defect reduction and quality control, due to the follow considerations. Firstly, companies don't always systematically perform reviews during all development phases. About 40 percent of respondents perform reviews on requirements or design documents, and about 30 percent on code. Second, companies often don't systematically execute the preparation, or defect detection step itself. About 60 percent of respondents stated they did not regularly conduct a preparation phase for reviews, even though empirical studies have shown this step to be crucial for effective reviews [Vot93]. Moreover, among those who conduct a preparation phase, 50 percent use a checklist to support that step, and fewer than 10 percent use a more advanced reading technique; while 40 percent use ad hoc reading; that is, they don't offer systematic support for defect detection. Finally, companies seldom embed software reviews in a systematic process evaluation and improvement program. Overall, more than half of the companies either do not collect data (40 percent) or collect data but do not analyze them (18 percent). Of the 42 percent who collect and analyze data, fewer than 23 percent try to optimize the review process, even though optimization has proved crucial to achieving highly effective reviews.

Today, inspections, reviews, or walkthroughs are integral parts of most industrial software development life-cycle models and standards [RCJ+08].

4 Runtime Verification

Runtime verification [CM04] is a verification technique that combines formal verification (Section 5) and program execution. It is the process of detecting faults in a system under scrutiny by passively observing its input/output behavior during its normal operations. The observed behavior, e.g., in terms of log traces, of the target system can be monitored and verified dynamically to satisfy given requirements. Such requirements are typically specified by a formalism which can express temporal constraints, such as LTL-formulae (Linear temporal logic) or automata and statecharts. In contrast to the classical model checking (Subsection 5.1.2) approach, where a simplified model of the target system is verified, runtime verification is performed while the real system is running. Thus, runtime verification or as it is sometimes referred to, passive testing, increases the confidence on whether the implementation conforms to its specification. Furthermore, it allows a running system to reflect on its own behavior in order to detect its own deviation from the prespecified behavior.

In contrast to other formal verification methods (Section 5) such as model checking (Subsection 5.1.2), runtime verification has to deal with finite traces only, as at an arbitrary point in time the system will be stopped and so its execution trace. The exception to this rule is a deployed runtime monitor that monitors on a potentially never ending deployed application for the purpose of detecting requirement violations while the application is running. To this end the formal specification must contain a recovery specification.

Continuous monitoring of the runtime behavior of a system can improve our confidence in the system by ensuring that the current execution is consistent with its requirements at runtime. In the literature, at least the following four reasons are mentioned in order to argue for runtime verification:

- If you check the model of your system you cannot be confident on the implementation since correctness of the model does not imply correctness of the implementation.
- Some information is available only at runtime or is convenient to be checked at runtime.

- Behavior may depend heavily on the environment of the target system; then it is not possible to obtain the information necessary to test the system.
- In the case of systems where security is important or in the case of critical systems, it is useful also to monitor behavior or properties that have been statically proved or tested.

Another good reason to use runtime verification is that it allows for formal specification (Section 5) and verification or testing (Section 2) of the properties that a system has imperatively to satisfy. Traditional testing techniques such as unit testing are ad hoc and informal. It is only a partial proof of correctness in that it does not guarantee that the system will operate as expected under untested inputs. In terms of its ability to guarantee software correctness, runtime verification is weaker than formal methods but stronger than testing. Testing can only guarantee the correctness of a limited set of inputs at implementation time. As a result, undiscovered faults may result in failures at runtime, and even allowing the system to propagate corrupted output because the failure was not detected. By always monitoring the software for correctness, such failures can be caught when they happen, for any input which causes them to occur. However, runtime verification is weaker than formal methods because such guarantees can not be made a priori.

5 Formal Verification

In Formal Verification there are advantages over testing: is exhaustive and improves the knowledge of system. However, Formal Verification is difficult and time-consuming, only as reliable as the formal models used and it´s difficult to prove that the proof is correct. Despite this complexity, during the last years, formal verification tools have been introduced, especially designed for standard development processes. The focus ranges from security related projects, over hardware circuit verification to software driver verification. In particular model checking has been very successful.

Formal verification is a set of techniques that developers can use to construct systems that operate reliably. It is complementary to testing and should be used in conjunction with testing to increase reliability of the system being developed. It is worth noting that formal verification is not a way to ensure that the system being developed is 100% correct; it enhances reliability of the system by ensuring that it meets defined functional requirements, particularly at earlier stages of design. Formal verification is based on formal methods [CW96] which are mathematically based languages, techniques and tools for specifying and verifying systems. Specifying a system means writing down the requirements about the system in a mathematical language. Verification is the a step in the aftermath of specification and involves formally proving that the system meets its requirements. There are various tools that aid in specification and verification. These tools provide notations and algorithms for a system developer to formally specify and/or verify a system.

Formal Verification of hardware and software systems has gained popularity in industry. The benefits reaped in the hardware sector have led the software sector to consider whether similar benefits could be achieved in the context of program correctness [Oui08]. Proofs of correctness about computer programs exist since the early days of computer science, but academic developments were routinely ignored by industry citing advances in research as “impractical”

[Hoa87]. There are drastic differences between the properties of software and the properties of hardware, namely the strict structure of hardware, the inherently finite state of hardware, and the restricted size of hardware [SBH04]. While there are doubts about whether the success of formal verification of industrial hardware can be replicated in the software sector, some progress has been made though numerous challenges still remain [ASM+08].

Formal verification relies on building a mathematical model of the system and on formally specifying the requirements to be checked against the system. Verification tools, i.e. tools performing formal verification, take two inputs: model of a system and its specification and check if the system satisfies the specification.

Depending on how the modelling and the checking are done, there are two fundamental techniques in formal verification: model checking and theorem proving. For hardware verification, industry is adopting both these techniques to complement the more traditional one of simulation. In specification and verification areas, researchers and practitioners are performing more and more industrial-sized case studies, and thereby gaining the benefits of using formal methods.

Subsections We subdivide the Formal Verification techniques in Static Analysis, techniques that elaborate information on the behavior of a program without executing it, and Theorem Proving, the process of finding a proof of a property where system and its desired 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 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 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 generates a counterexample, an execution trace leading to a state in which the property is violated. As the state space of software programs is typically too large to be analyzed directly, model checking is often combined with abstraction techniques. The third part 5.1.3 is dedicated to a formal technique that performs a depth-bounded exploration of the state space. As this technique is a form of Model Checking, it is called Bounded Model Checking (BMC). We present BMC separately, as the requirement to analyze the entire state space is relaxed: BMC explores program behavior exhaustively, but only up to a given depth. Bugs that require longer paths are missed. Then we highlight in Section 5.2 on Theorem Proving [RSS95]. In the theorem proving a system and its properties are described by means of logical formulae and the system is shown by means of a logical proof to entail the desired properties.

5.1 Static Analysis

Static analysis encompasses a family of techniques for automatically computing information about the behavior of a program without executing it. Most questions about the behavior of a program are either undecidable or it is infeasible to compute an answer. Thus, the essence of static analysis is to efficiently compute approximate but sound guarantees: guarantees that are not misleading.

5.1.1 Abstract Static Analysis

Abstract Interpretation Abstract Interpretation [CC77b] is a theory of approximation of mathematical structures, in particular those involved in the semantic models of computer systems. An abstract domain is an approximate representation of sets of concrete values. An abstraction function is used to map concrete values to abstract ones. An abstract interpretation translates the meaning of a program on an abstract domain to obtain an approximate solution. An abstract interpretation can be derived from a concrete interpretation by defining counterparts of concrete operations, such as addition or union, in the abstract domain. If certain mathematical constraints between the abstract and concrete domains are met, fixed points computed in an abstract domain are guaranteed to be sound approximations of concrete fixed points [CC79]. Abstract interpretation can be applied to the systematic construction of methods and effective algorithms to approximate undecidable or very complex problems in computer science as for instance the semantics, the proof, the static analysis, the verification, the safety and the security of software or hardware computer systems. In particular, static analysis by abstract interpretation, which automatically infers dynamic properties of computer systems, has been very successful these last years to automatically verify complex properties of real-time, safety-critical embedded systems [CC04].

Numerical Abstract Domains Over the years, various abstract domains have been designed, particularly for computing invariants about numeric variables. The class of invariants that can be computed, and hence the properties that can be proved, varies with the expressive power of a domain. A static analyzer is thus parameterized by a numerical abstract domain, that is, a set of computer-representable numerical properties together with algorithms to compute the semantics of program instructions. There already exist quit a few numerical abstract domains. Well-known examples include the interval domain [CC77a] that discovers variable bounds, and the polyhedron domain [CH78] for affine inequalities. Each domain achieves some cost versus precision balance. In particular, non-relational domains—e.g., the interval domain—are much faster but also much less precise than relational domains—able to discover variable relationships.

Shape Analysis Shape analysis is a static code analysis technique that discovers and verifies properties of linked, dynamically allocated data structures in (usually imperative) computer programs. It is typically used at compile time to find software bugs or to verify high-level correctness properties of programs. In Java programs, it can be used to ensure that a sort method correctly sorts a list. For C programs, it might look for places where a block of memory is not properly freed. Although shape analyses are very powerful, they usually take a long time to run. For this reason, they have not seen widespread acceptance outside of universities and research labs (where they are only used experimentally).

5.1.2 Model Checking

Model checking is a technique that relies on building a finite model of a system and checking that a desired property holds in that model. Roughly speaking, the check is performed as an exhaustive state space search that is guaranteed to terminate since the model is finite. The technical challenge in model checking is in devising algorithms and data structures that allow to handle large search 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 checks if the system satisfies the property and gives a "yes" or "no" answer. If the answer is "no", i.e., if the system does not satisfy the property, model checkers also output a counterexample, i.e., a run of the system which violates the property. The counterexample can be analyzed to discover bugs in the system design. Figure 2 describes the process of model checking. Model checking has been used primarily in hardware and protocol verification [CK96]; the current trend is to apply this technique to analyzing specifications of software systems.

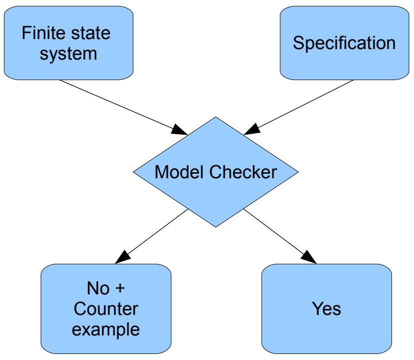


Figure 2: Overview of Model Checking

Two general approaches to model checking are used in practice today. The first, *temporal model checking*, is a technique developed independently in the 1980s by Clarke and Emerson [CE82] and by Queille and Sifakis [QS82]. In this approach specifications are expressed in a temporal logic [Pnu81] and systems are modeled as finite state transition systems. An efficient search procedure is used to check if a given finite state transition system is a model for the specification.

In the second approach, the specification is given as an automaton; then the system, also modeled as an automaton, is compared to the specification to determine whether its behavior conforms to that of the specification. Different notions of conformance have been explored, including language inclusion [HK90], [Kur94b], refinement orderings [CPS94] and [Ros94], and observational equivalence [CPS94], [FGK^{+}96] and [RS91]. [VW86] showed how the temporal-logic model-checking problem could be recast in terms of automata, thus relating these two approaches.

In contrast to theorem proving, model checking is completely automatic and fast, sometimes producing an answer in a matter of minutes. Model checking can be used to check partial specifications, and so can provide useful information about a system's correctness even if the system has not been completely specified. Above all, model checking's tour de force is that it produces counterexamples, which usually represent subtle errors in design, and thus can be used to aid in debugging.

The main disadvantage of model checking is the state explosion problem. In 1987 McMillan used Bryant's ordered binary decision diagrams (BDDs) [Bry86] to represent state transition systems efficiently, thereby increasing the size of the systems that could be verified. Other promising approaches to alleviating state explosion include the exploitation of partial order information [Pel94], localization reduction [Kur94b] and [Kur94a], and semantic minimization [JCE97] to eliminate unnecessary states from a system model.

Model checkers today are routinely expected to handle systems with between 100 and 200 state variables. They have checked interesting systems with 10^{120} reachable states [BCL+94] and, by using appropriate abstraction techniques, they can check systems with an essentially unlimited number of states [CGL94]. As a result, model checking is now powerful enough that it is becoming widely used in industry to aid in the verification of newly developed designs.

5.1.3 Bounded Model Checking

Bounded model checking (BMC) is one of the most commonly applied formal verification techniques in the semiconductor industry. The technique owes this success to the impressive capacity of propositional SAT solvers. It was introduced in 1999 by Biere et al. as a complementary technique to BDD-based unbounded model checking [BCCZ99]. It is called bounded because only states reachable within a bounded number of steps, say k, are explored. In BMC, the design under verification is unwound k times and conjoined with a property to form a propositional formula, which is passed to a SAT solver. The formula is satisfiable if and only if there is a trace of length k that refutes the property. The technique is inconclusive if the formula is unsatisfiable, as there may be counterexamples longer than k steps. Nevertheless, the technique is successful, as many bugs have been identified that would otherwise have gone unnoticed. A satisfying assignment to the propositional formula corresponds to a path from the initial state to a state violating the property.

BMC is the best technique to find shallow bugs, and it provides a full counterexample trace in case a bug is found. It supports the widest range of program constructions. This includes dynamically allocated data structures; for this, BMC does not require built-in knowledge about the data structures the program maintains. On the other hand, completeness is only obtainable on very 'shallow' programs, i.e., programs without deep loops.

5.2 Theorem Proving

Theorem proving [CW96] is a technique by which both the system and its desired properties are expressed as formulas in some mathematical logic. This logic is given by a *formal system*, which defines a set of axioms and a set of inference rules. Theorem proving is the process of finding a proof of a property from the axioms of the system. Steps in the proof appeal to the axioms and rules, and possibly derived definitions and intermediate lemmas. Although proofs can be constructed by hand, here we focus only on machine-assisted theorem proving.

Theorem provers are increasingly being used today in the mechanical verification of safety-critical properties of hardware and software designs. Theorem provers can be roughly classified in a spectrum from highly automated, general-purpose programs to interactive systems with special-purpose capabilities. The automated systems have been useful as general search procedures and have had noteworthy success in solving various combinatorial problems. The interactive systems have been more suitable for the systematic formal development of mathematics and in mechanizing formal methods.

In contrast to model checking, theorem proving can deal directly with infinite state spaces. It relies on techniques such as structural induction to prove over infinite domains. Interactive theorem provers, by definition, require interaction with a human, so the theorem proving process is slow and often error-prone. In the process of finding the proof, however, the human user often gains invaluable insight into the system or the property being proved.

6 Case studies

In the previous sections, we introduced a variety of specification styles and verification methodologies. We now present selected case studies, mostly industrial designs in order to provide a sense of the state-of-the-art in verification.

6.1 Inspections, Reviews and Walkthroughs

Successful companies such as Allianz, NASA's Software Engineering Laboratory (SEL), Motorola or IBM report up to 95% defect detection rates before testing, overall cost reduction for newly developed lines of code by 50%, and even shortening of delivery times by up to 50% [RCJ+08].

Industrial case studies The NASA Software Engineering Laboratory (SEL) was founded in 1976, with the goal to define effective practices for softwaredeveloping organizations, and to introduce state-of-the-art techniques as well as determine their value. Over its 25 years of existence, the SEL managed to reduce defect rates during development by 82%, and reduce development cost at the same time by 74%, although the functionality of the systems increased fivefold [6].

- The NASA Software Engineering Laboratory (SEL) picked inspections as one potential cutting-edge technology [RCJ+08]; they introduced inspections Based on Fagan´s paper in 1976. These were "informal" or basic code inspections based on ad-hoc reading, using multiple reviewers. In contrast to Fagan´s inspections [Fag02], they did not necessarily follow a specific protocol or use a moderator for the inspection meeting, and they did not record any data. All in all, the available evidence convinced the decision makers that formal inspections were successful, and around 1980, formal Fagan inspections were defined as a standard. This meant that there was a process description, and every project had to do inspections or justify why they did not.

6.2 Static Analysis

6.2.1 Abstract Static Analysis

An early, popular static analysis tool for finding simple errors in C programs is Lint [DKW08], released in 1979 by Bell Labs. Several modern tools emulate and extend Lint in terms of the kind of errors detected, warnings provided and user experience. FindBugs [DKW08] for Java is a notable modern tool with similar features. We mention Lint and FindBugs because of their significance and influence on static analysis tools. However, these tools are unsound and provide no rigorous guarantees so we do not discuss them further.

Industrial case studies Though several commercial static analyzers are available, the details of the techniques implemented and their soundness is unclear, so we refrain from a full market survey, and only list a few exemplary tools.

- Grammatech Inc. produces CodeSonar, a tool using inter-procedural analyses to check for template errors in C/C++ code. These include buffer overflows, memory leaks, redundant loop and branch conditions. The K7 tool from KlocWork has similar features and supports Java [DKW08].
- The company Coverity produces Prevent, a static analyzer, and Extend, a tool for enforcing coding standards. Prevent has capabilities similar to those of CodeSonar, but also supports Microsoft COM and Win32 APIs and concurrency implemented using PThreads, Win32 or WindRiver VxWorks. In January 2006, as part of a United States Department of Homeland Security sponsored initiative, Coverity tools were used to find defects in over 30 open source projects [DKW08].
- Abstract interpretation tools were used to identify the error leading to the failure of the Ariane 5 rocket [LMR+98]. Abstract domains for finding buffer overflows and undefined results in numerical operations are implemented in the Astrée static analyzer [BCC+02], used to verify Airbus flight control software.
- Polyspace Technologies markets C and Ada static analyzers. The C Global Surveyor (CGS), developed at NASA, is a static analyzer specially developed for space mission software. The Polyspace tool and CGS have been used to analyze software used on the Mars Path-Finder, Deep Space One and Mars Exploration Rover.
- AbsInt GmbH is a company selling PAG, a program analyzer generator used to analyze architecture-dependent properties such as worst case execution time, cache performance, stack usage and pipeline behavior [DKW08].
- Several tools require annotations such as types, pre- and post-conditions, and loop invariants to be provided. Annotations may reduce the information the static analyzer has to compute and improve its precision, but increase the burden on the programmer. This approach is used to varying degrees in the (ESC/Java) tool family [FLL+02] and Microsoft's PREfix and PREfast [LBD+04]. The annotation and verification mechanism has been integrated in programming languages and development environments such as SPARK [Bar03] and Spec# [BLR+04], to make writing annotations a natural part of programming.

6.2.2 Model Checking

Listed in the following are some well-known model checkers, roughly categorized according to whether the specification they check is given as a logical formula or as a machine:

Temporal logic model checkers The very first two model checkers were EMC [CE82], [CES86] and [BCDM86] and CÆSAR [QS82] and [FGK+96]. Murphi [ADHY92] and UV [Kal94] are based on the Unity programming language [Cha88]. The Concurrency Workbench [CPS94] verifies CCS processes for properties expressed as mu-calculus formulas. SVE [FW94], FORMAT [DJS95] and [Klo01], and CV [DB95] all focus on hardware verification. HyTech [AHH96] is a model checker for hybrid systems; Kronos [DY95] and [HNSY94], for real-time systems. SPIN (Simple Promela INterpreter) [GPV+95] and [Hol91] is an open source model checking tool used for verifying distributed systems like communication protocols, network applications, multi-threaded code etc. The SPIN system uses partial order reduction to reduce the state explosion problem [HP95] and [Pel94]. The systems are specified in the language Promela as a network of communicating automata and properties given as a linear-time temporal logic formulas. SPIN is one of the most widely used model checkers and won the ACM software system award in 2001. Some software model checkers, such as an early version of the Java Pathfinder (JPF), translate Java code to PROMELA and use SPIN for model checking [VHB+03].

SMV (Symbolic Model Verifier) [McM92] is a model checker used to verify hardware designs and is freely available for academic use. The system is modelled using binary decision diagrams and the property is given as branching-time temporal logic formula. It is the first model checker which uses BDDs. UPPAAL [BDL04] is a model checker used for the verification of real-time systems and it has been applied successfully in case studies ranging from communication protocols to multimedia applications. The tool is based on the theory of timed automata.

Two prominent representatives of the class of explicit-state software model checkers are CMC [MPC+02] and Microsoft Research's Zing [AQRX04]. The VeriSoft software verification tool attempts to eschew state explosion by discarding the states it visits [Bel97]. Since visited states are not stored, they may be repeatedly visited and explored. This method is state-less and has to limit the depth of its search to avoid non-termination. The success of predicate abstraction for software model checking was initiated by the Microsoft Research's SLAM toolkit [BCLR04]. SLAM checks 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". SLAM comprises 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]. The BDD-based model checker Moped [ES01] can be used in place of Bebop to check temporal logic specifications. When combined with Cogent [CKS05a], a decision procedure for machine arithmetic, SLAM can verify properties that depend on bit-vector arithmetic.

The tool BLAST uses lazy abstraction: The refinement step triggers the re-abstraction of only relevant parts of the original program [RHJM02]. The tighter integration of the verification and refinement phases enables a speedup of the CEGAR (Counterexample guided abstraction refinement) iterations. Unlike SLAM, BLAST uses Craig interpolation to derive refinement predicates from counterexamples [HJMM04]. Like SLAM, BLAST provides a language to specify reachability properties.

The verification tools mentioned above use general purpose theorem provers to compute abstractions and BDDs for model checking. SATABS uses a SAT-solver to construct abstractions and for symbolic simulation of counterexamples [CKSY03]. The bitlevel accurate representation of C programs makes it possible to model arithmetic overflow, arrays and strings. SATABS automatically generates and checks proof conditions for array bound violations, invalid pointer dereferencing, division by zero, and assertions provided by the user. SATABS uses a SAT-based model checker Boppo to compute the reachable states of the abstract program. Boppo relies on a QBF-solver for fixed point detection [CKS05b]. SATABS can verify concurrent programs that communicate via shared memory. To deal with concurrency, Boppo combines symbolic model checking with partial order reduction [CKS05b]. Unfortunately, this approach still has severe scalability issues [WBKW07].

Gurfinkel's model checker YASM can construct sound abstractions for liveness properties [GC06]. The SLAM-based Terminator tool also checks liveness properties [ACPR05]. However, the latter is not dependent on predicate abstraction and can be based upon any software model checker. Sagar Chaki's MAGIC framework [CCG+03] uses a compositional approach and decomposes the program into several smaller components which are verified separately. Furthermore, MAGIC is able to check concurrent programs that use message passing, but does not support shared memory. In an experimental version of SLAM for concurrent programs, Bebop can be replaced by either the explicit state model checker Zing [AQRX04] or the SAT-based tool Boppo [CK06].

Behavior conformance checkers The Cospan/FormalCheck system [DG96], [HK90] is based on showing inclusion between omega automata. FDR [Ros94] checks refinement between CSP programs; most recently, it has been used to verify and debug the Needham-Schroeder authentication protocol [Low96]. The Concurrency Workbench [CPS94] checks a similar notion of refinement between CCS programs; it and the tool Auto [RS91] may also be used to minimize systems with respect to observational equivalence and to determine if two systems are observably equivalent.

Combination checkers Berkeley's [HBK93] combines model checking with language inclusion; Stanford's STeP [BBC+96] system, with deductive methods; and VIS [BHSV+96], with logic synthesis. The PVS theorem prover [ORS92] has a model checker for the modal mu-calculus [RSS95]. METAFrame [SMCB96] is an environment that supports model checking in the entire software development process.

Industrial case studies Some successful industrial-size case studies exist in model checking. They evidence that model checking is starting to become applicable in industry, by building its own model checkers or simply using existing ones.

- AT & T applied model checking to the development of the International Telecommunication Union (ITU) ISDN User part protocol in the early 1990s. About 7500 lines of source code of the protocol written in SDL were verified to satisfy 145 requirements. About 55%

of the original design requirements were found to be logically inconsistent and a total of 112 errors were discovered [CW96].

- In 1992, a research group at CMU found errors in the IEEE Futurebus+ standard 896.1-1991 using the model checker SMV. That is the first time model checking was used to find errors in an IEEE standard [CW96].
- In 2001, the model checker SPIN was used to verify legacy flight software from NASA's Deep Space One mission. The process not only unearthed a known error in the launch software but also discovered a second scenario under which a similar error could occur. There are various other successful case studies from the industry [GH02].
- In 1999 the model checker UPPAAL has been successfully applied to the verification of a protocol for controlling the switching between power on/off states in audio/video components [HLGS99].
- An incarnation of SLAM, the Static Driver Verifier (SDV) tool, is currently part of a beta of the Windows Driver Development Kit (DDK).

6.2.3 Bounded Model Checkeing

There are a number of BMC implementations for software verification. The first implementation of a depth-bounded symbolic search in software is due to Currie et al. [CMH+00]. One of the first implementations of BMC for C programs is CBMC [CKY03], [CKL04], developed at CMU; it emulates a wide range of architectures as environment for the design under test. It includes options to export the formula to various word-level formats. It is the only tool that also supports C++, SpecC and SystemC. The main application of CBMC is checking consistency of system-level circuit models given in C or SystemC with an implementation given in Verilog.

The only tool that implements an unwinding of the entire transition system is F-SOFT [ISGG05], developed at NEC Research. It features a SAT solver customized for BMC decision problems. The benchmarks that have been reported are system-level UNIX applications such as pppd (Point-to-Point Protocol daemon).

A number of variants of these implementations have been reported. Armando et al. implement a version of CBMC that generates a decision problem for an SMT solver for integer linear arithmetic [AMP09]. The performance of off-the-shelf constraint solvers on such decision problems is evaluated in [CR06].

SATURN is a specialized implementation of BMC, tailored to the properties it checks [XA05]. The authors have applied it to check two different properties of Linux kernel code: NULL-pointer dereferences and locking API conventions. They demonstrate that the technique is scalable enough to analyze the entire Linux kernel. Soundness is relinquished for performance; SATURN performs at most two unwindings of each loop. Bugs that require more than two unwindings are missed.

The EXE tool is also specialized to bug-hunting [YST+06]. It combines explicit execution and path-wise symbolic simulation to detect bugs in system-level software such as file system code. It uses a very low-level memory model, which permits checking programs that contain arbitrary pointer type-casts.

Industrial case studies

- IBM has developed a version of CBMC for concurrent programs [RG05].

6.3 Theorem Proving

As with model checking, an increase in the number and kinds of theorem provers provides evidence for a growing interest in theorem proving. There has been a corresponding increase in the number and kinds of examples to which theorem provers have been applied. Following are some well-known theorem provers, categorized roughly by their degree of automation:

User-guided automatic deduction tools Systems like ACL2 [KM02], Eves [CKM+88], LP [GG88], Nqthm [BM88], Reve [Les83], and RRL [KM87] 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. Nqthm, the Boyer-Moore theorem prover, has been used to check a proof of Gödel's first incompleteness theorem, and in a variety of large-scale verification efforts.

Proof checkers One of the first Proof checker is LCF ([GMW79], [CAA+86]) started in the 1970s with Stanford LCF. It introduced the general purpose programming language ML to allow users to write theorem proving tactics. Theorems in the system are propositions of a special "theorem" abstract datatype. The ML type system ensures that theorems are derived using only the inference rules given by the operations of the abstract type. Successors include Coq, HOL and Isabelle. Coq [FHB+97] is a proof assistant for higher-order logic, allowing the development of computer programs consistent with their formal specification. It allows the expression of mathematical assertions, mechanically checks proofs of these assertions, helps to find formal proofs, and extracts a certified program from the constructive proof of its formal specification. Coq works within the theory of the calculus of inductive constructions, a derivative of the calculus of constructions. Coq is not an automated theorem prover but includes automatic theorem proving tactics and various decision procedures. LEGO [LP92] is a tool for interactive proof development in the natural deduction style and it supports refinement proof as a basic operation. LEGO has been used in many scientific problem, for example a proof of the strong normalization theorem of the second-order lambda-calculus, the proof of Tarski's fixpoint theorem, program specifications and program correctness proofs.

HOL (Higher Order Logic) [Gor87] denotes a family of interactive theorem proving systems sharing similar logics and implementation strategies. Systems in this family follow the LCF approach as they are implemented as a library in some programming language. This library implements an abstract data type of proven theorems so that new objects of this type can only be created using the functions in the library which correspond to inference rules in higher-order logic. As long as these functions are correctly implemented, all theorems proven in the system must be valid. In this way, a large system can be built on top of a small trusted kernel. The primary application area of HOL was initially intended to be the specification and verification of hardware designs. However, the logic does not restrict applications to hardware; HOL has been applied to many other areas. HOL is a predecessor of Isabelle. Isabelle [WPN08] is a generic proof assistant. It allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus. The main application is the formalization of mathematical proofs and in particular formal verification, which includes proving the correctness of systems and proving properties of computer languages and protocols. The most widespread instance of Isabelle nowadays is Isabelle/HOL, providing a higher-order logic theorem proving environment ready to use for sizable applications. Isabelle/HOL includes powerful specification tools, e.g. for datatypes, inductive definitions and functions with complex pattern matching. Proofs are conducted in the structured proof language Isar, allowing for proof text naturally understandable for both humans and computers. The Isabelle/Pure meta-logic allows the formalization of the syntax and inference rules of a broad range of object logics following the general idea of natural deduction [Pau86], [Pau90]. The logical core is implemented according to the well-known "LCF approach" of secure inferences as abstract datatype constructors in ML [GMW79]; explicit proof terms are also available [BN00]. Isabelle/Isar provides sophisticated extra-logical infrastructure supporting structured proofs and specifications, including concepts for modular theory development. Other notable object logics are Isabelle/ZF (Zermelo-Fraenkel set-theory, see [Pau98], [Pau00]) and Isabelle/HOLCF [MNOS99](Scott's domain theory within HOL). These tools have been used to formalize and verify hard problems in mathematics and in program verification.

Combination provers Analytica [CZ92], which combines theorem proving with the symbolic algebra system Mathematica, has successfully proved some hard number-theoretic problems due to Ramanujam. Both PVS [ORS92] and STeP [BBC+96] combine powerful decision procedures and model checking with interactive proof. PVS has been used to verify a number of hardware designs and 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 with a problem in the lookup table of an SRT divider, was successfully identified, verificating the system (after the occurrence of the bug!) by various communities. A formally verified treatment of the general theory of SRT division was given using the PVS theorem prover by a group from Stanford Research Institute.
- In the mid nineties, Motorola's Complex Arithmetic Processor used for Digital Signal Processing (DSP) was formally specified using the ACL2 theorem prover. The design was tracked as it was evolving and the binary micro code of various DSP algorithms were verified in the process.
- The Verisoft formalized a whole computer system from the hardware up to an operating system kernel [ASS08] and a compiler for C-dialect [LP08]. The *L4.verified* project [HEK+07], [TKN07] verifies the L4 operating system microkernel, relating an abstract specification, a Haskell model, and the C code.

7 SPES Survey Results

On April 23rd 2009 Technische Universität München distributed a survey to all partners of the SPES project. The scope of the survey was to having 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 questions regarding programming languages, tools, operating systems and technologies to develop embedded systems.

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.

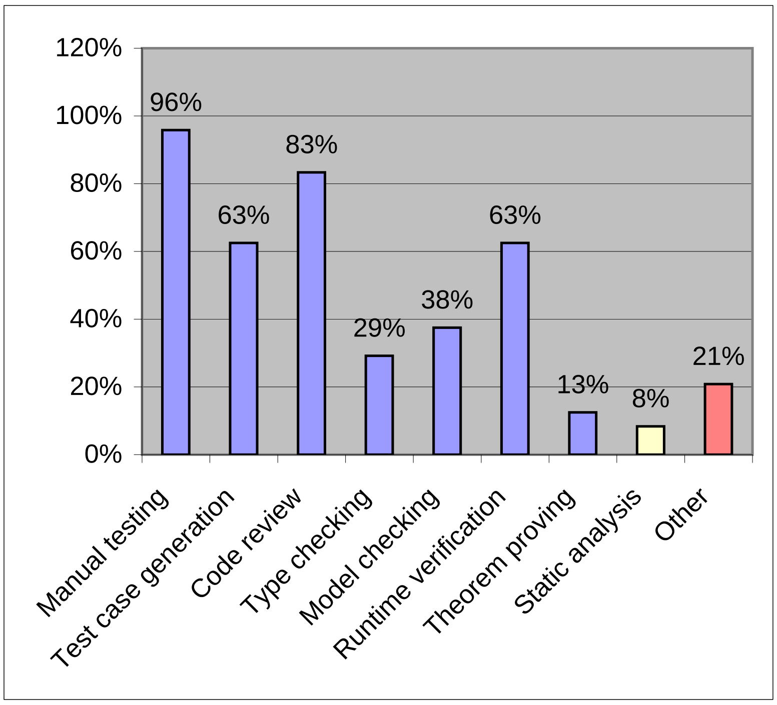


Figure 3: Methods used for quality assurance

Almost every project performs manual tests and code reviews for quality assurance (96% and 83%, respectively). But also more 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, RTCA/DO-178B, System simulation, and System testing.

With a share of 38%, Polyspace (we discussed about it on Section 6) is the most popular quality assurance tool. Although Simulink itself is widespread, the Simulink Design Verifier is not used in a lot of 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

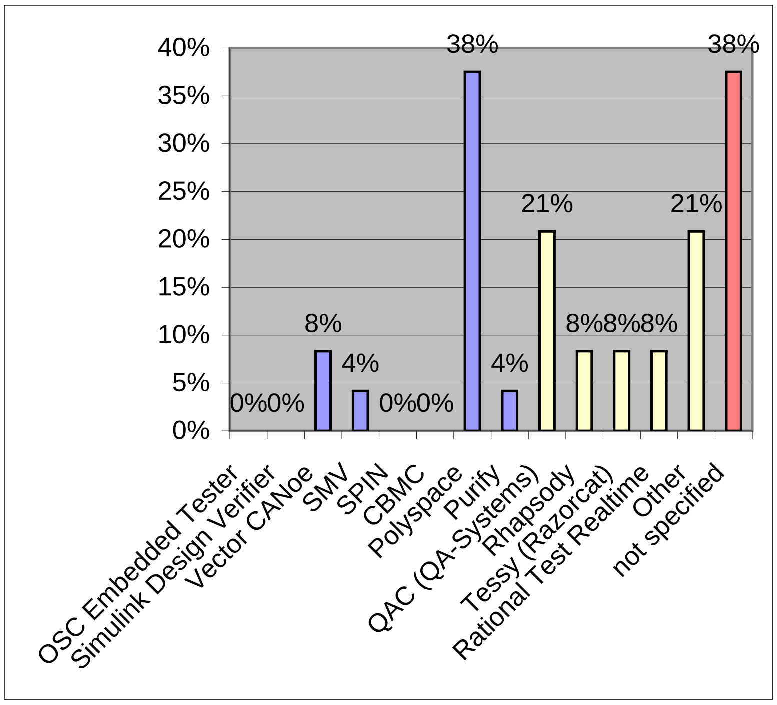


Figure 4: Tools used for quality assurance

for the heterogeneity of the tool landscape in industry.

8 Techniques Comparison

The presented Analysis Techniques have different characteristics and purposes, we summarize these differences making some significant direct comparisons between couple of theories.

Runtime Verification versus Model Checking While runtime verification shares also many similarities with model checking, there are important differences. In model checking, all executions of a given system are examined to answer whether these satisfy a given property. This corresponds to the language inclusion problem. In contrast, runtime verification deals with the word problem. For most logical frameworks, the word problem is of far lower complexity than the inclusion problem [BLS09].

Model checking, especially in case of LTL, considers infinite traces, runtime verification deals with finite traces. While in model checking a complete model is given allowing to consider arbitrary positions of a trace, runtime verification, especially when dealing with online monitoring, considers finite executions of increasing size. For this, a monitor should be designed to consider executions in an incremental fashion.

Runtime Verification versus Testing As runtime verification does not consider each possible execution of a system, but just a single or a finite subset, it shares similarities with testing: both are usually incomplete. Typically, in testing one considers a finite set of finite input-output sequences forming a test suite. Test-case execution is then checking whether the output of a system agrees with the predicted one, when giving the input sequence to the system under test.

A different form of testing, however, is closer to runtime verification, namely oracle-based testing. Here, a test-suite is only formed by input-sequences. To make sure that the output of the system is as anticipated, a so-called test oracle has to be designed and "attached" to the system under test. This oracle then observes the system under test and checks a number of properties, i. e. in terms of runtime verification the oracle acts as a monitor. Thus, in essence, runtime verification can be understood as this form of testing. There are, however, differences in the foci of runtime verification and oracle-based testing: In testing, an oracle is typically defined directly, rather than generated from some high-level specification. On the other hand, in the domain of runtime verification, we do not consider the provision of a suitable set of input sequences to "exhaustively" test a system [BLS09].

Model Checking versus Theorem Proving A key difference between the theorem approach to software verification and the model checking approach to software verification is that theorem provers do not need to exhaustively visit the program's state space to verify properties [CGP99].

Consequently, a theorem prover approach can reason about infinite state spaces and state spaces involving complex datatypes and recursion. This can be achieved because a theorem prover reasons about constraints on states, not instances of states. Theorem provers search for proofs in the syntactic domain, which is typically much smaller than the semantic domain searched by model checkers. Consequently, theorem provers are well-suited for reasoning about "data-intensive" systems with complex data structures but simple information flow. Although theorem provers support fully automated analysis in restricted cases only [Duf91], a mature theorem proving system can provide an acceptable level of automation [DDN+03]. Reasoning about inductive structures of arbitrary size (e.g., trees, lists, or stacks) can be achieved through mathematical induction but cannot be automated [KS96]. Nevertheless, this tradeoff is acceptable in certain instances since this type of analysis cannot be performed by model checkers, but is still vital to the verification effort. In certain instances, lack of automation can be tolerated for increased capabilities.

While theorem provers have distinct advantages over model checkers, namely in the superior size of the systems they can be applied to and their ability to reason inductively, deductive systems also have their drawbacks. The proof system of a theorem prover for a system of practical size can be extremely large [ASM+08]. Furthermore, the generated proofs can be large and difficult to understand. An often cited drawback of theorem provers is that they require a great deal of user expertise and effort [AVM03]. This requirement presents perhaps the greatest barrier to widespread adoption and usage of theorem provers.

Although theorem proving and model checking appear to be contradictory approaches to software verification, there has been considerable effort in the past 15 years to incorporate model checking and theorem proving ([Sha00] and [AKMR03]). Because theorem provers and model checkers each provide complementary benefits in terms of automation and scalability, it is likely that this trend will follow and that model checkers will continue to be useful on systems or manageable size while theorem provers will be used on large systems [Hol03].

9 Conclusions

We presented the major Analisys Techniques and their actual state in research and industry. We conclude that the use of Analisys Techniques has proved successful in verifying safety-critical software, protocol standards and hardware designs. Due to the commercial pressure to produce high-quality software, the role of Analisys Techniques is increasing in the system development process. In future with simpler tool more and more companies will use those techniques in the development process. Today they are not so much widespread used, due to the difficult, to understand the tools and the specification of the property. In fact lot of research is concentrated on making the tools and notations accessible to system developers who are non-experts in this area.

A Formal Verification Taxonomy

We summarize in Figure 5 the major tools used in Formal Verification.

References

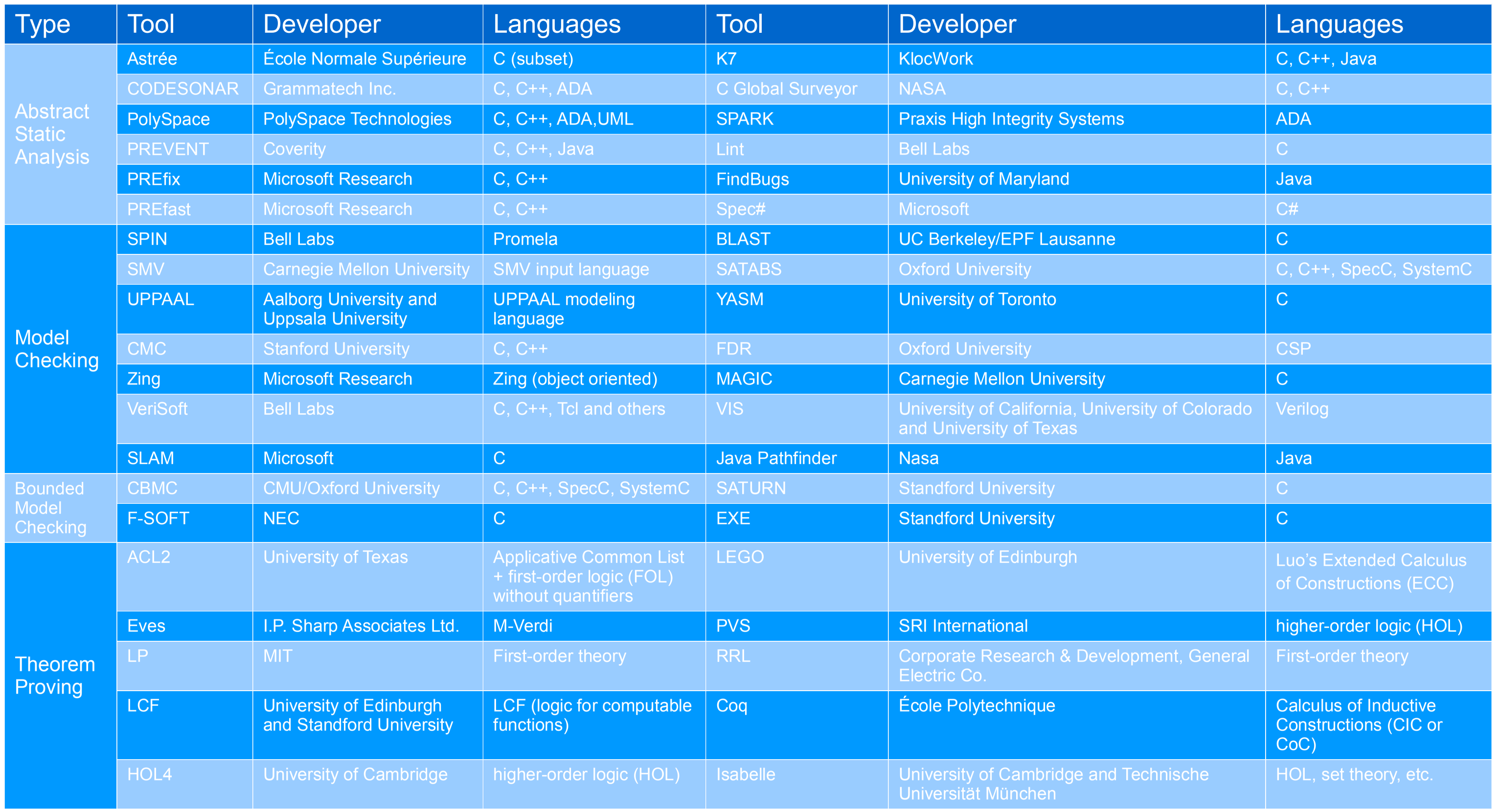
[84697] Formally specifying and mechanically verifying programs for the motorola complex arithmetic processor dsp. In *ICCD '97: Proceedings of the 1997 International Conference on Computer Design (ICCD '97)*, page 31, Washington, DC, USA, 1997. IEEE Computer Society.

[ACPR05] Ron Cook Andreas, Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction Refinement for Termination. In *In SAS'2005: Static Analysis Symposium, volume 3672 of LNCS*, pages 87–101. Springer, 2005.

[ADHY92] David Dill Andreas, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol Verification as a Hardware Design Aid. In *In IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, 1992.

[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 (LNCS)*, pages 21–33, Malente, Germany, May 2003 2003.



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

[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 *CONCUR*, pages 1–15, 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 *TACAS*, pages 109–123, 2008.

[AVM03] M. Archer, B. Di Vito, and C. 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 Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs, 1999.

[BCDM86] M C Browne, E. M. Clarke, D L Dill, and B 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 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[Bel97] Patrice Godefroid Bell. Model Checking for Programming Languages using VeriSoft. In *In Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186. ACM Press, 1997.

[BHSV^{+}96] Robert K. Brayton, Gary D. Hachtel, Alberto Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu-Tsung Cheng, Stephen Edwards, Sunil 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. pages 428–432. Springer-Verlag, 1996.

[BLR^{+}04] Mike Barnett, K. Rustan M. Leino, 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 Millstein, and Sriram K. Rajamani. Automatic Predicate Abstraction of C Programs. In *In Proc. ACM PLDI*, pages 203–213. ACM Press, 2001.

[BN00] Stefan Berghofer and Tobias Nipkow. Proof terms for simply typed higher-order logic. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics*, volume 1869, pages 38–52, 2000.

[BPR01] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and Cartesian Abstraction for Model Checking C Programs. pages 268–283. Springer, 2001.

[BR00a] Thomas Ball and Sriram K. Rajamani. Bebop: A Symbolic Model Checker for Boolean Programs. 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] T. Ball and S. 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] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. 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-Verlag.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. 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.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* The MIT Press, January 1999.

[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 verificiation and validation. Technical report, 2005.

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

[CK06] Byron Cook and Daniel Kroening. Over-approximating boolean programs with unbounded thread creation. In *In: Formal Methods in Computer-Aided Design (FMCAD), IEEE*, pages 53–59, 2006.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *In Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[CKM$^+$88] D. Craigen, S. Kromodimoeljo, I. Meisels, A. Neilson, B. Pase, and M. 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 *in SPIN*, pages 75–90. Springer, 2005.

[CKSY03] Edmund 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 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.

[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 *TACAS*, pages 182–196, 2006.

[CW96] Edmund 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 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 *In CHARME'95, volume 987 of Lecture Notes in Computer Science*, pages 293–310. Springer Verlag, 1995.

[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.

[DG96] G. DEPALMA and A. GLASER. Formal verification augments simulation. In *Electrical Engineering Times 56*, 1996.

[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.

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

[DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with kronos. In *In Proc. 1995 IEEE Real-Time Systems Symposium, RTSS'95*, 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-Verlag, 2001.

[Fag02] Michael Fagan. Design and code inspections to reduce errors in program development. pages 575–607, 2002.

[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-Verlag.

[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.

[FW94] SCHNEIDER H. SCHOLZ A. STRASSER A. FILKORN, T. 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 Holger Hermanns and Jens Palsberg, editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 212–226. Springer, 2006.

[GG88] S.J. Garland and J. 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] P.R. Gluck and G.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.

[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] R. Hojati, R. K. Brayton, and R. 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.

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

[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.

[HLGS99] Klaus Havelund, Kim Guldstrand Larsen, Kim Guldstr, and Arne Skou. Formal verification of a power controller using the real-time model checker uppaal. In *In the Proceedings of the 5th International AMAST Workshop on Real-Time and Probabilistic Systems*, pages 277–298. Springer-Verlag, 1999.

[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.

[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.

[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] C. 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-Verlag New York, Inc., 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.

[KS96] Deepak Kapur and Mahadevan Subramaniam. Lemma discovery in automated induction. In *CADE-13: Proceedings of the 13th International Conference on Automated Deduction*, pages 538–552, London, UK, 1996. Springer-Verlag.

[Kur94a] R. 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 F?hndrich, 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] P. Lacan, J. N. Monfort, L. V. Q. Ribal, A. Deutsch, and G. 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-Verlag, 1996.

[LP92] Z. Luo and R. 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.

[ORS92] S. Owre, J. M. Rushby, , and N. 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-Verlag.

[Oui08] Martin Ouimet. Formal software verification: Model checking and theorem proving, 2008.

[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-Verlag.

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

[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-Verlag.

[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.

[RHJM02] Thomas Henzinger Ranjit, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Lazy abstraction. In *In POPL*, pages 58–70. ACM Press, 2002.

[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-Verlag.

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

[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-Verlag.

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

[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.

[WPN08] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The isabelle framework. In *TPHOLs '08: Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, pages 33–38, Berlin, Heidelberg, 2008. Springer-Verlag.

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

B Deliverable D1.3.A-2

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

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

SPES 2020 Deliverable 1.3.A-2

Analysis Techniques suitable for SPES project

SPES

Software Plattform Embedded Systems 2020

Author:	Alarico Campetelli
Version:	1.0
Date:	January 11, 2010
Status:	Released

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

About the Document

In SPES project we have introduced Focus modelling approach for the development of reactive systems. The purpose of this document is to provide the actual research status of analysis techniques that can be integrated and used in Focus. We present methods for verifying and validating model definitions and requirements by formal verification.

We list formal techniques based on model checking, simulation and theorem proving, which have been studied in the Verisoft XT project. In the framework of SPES project we propose mainly model checking techniques, considering the research work already done. The model checker proposed is an our own solution based on the notion of abstraction and refinement of the model to verify. The integration of analysis techniques into the modelling theory is one of the objectives of work package ZP-AP 1.3.

Contents

1 Introduction

In the last years, the wide spread deployment of embedded software systems within human activities depend, it has raised many interests about safety issues. A combination of fault prevention, fault tolerance, fault removal and fault forecasting techniques are commonly used in order to achieve a high degree of reliability. Anyway there are no standard methods for performing analysis techniques of such systems. In fact, every industry, considering different application fields, tends to use their own particular development methodologies and so techniques for analysing and enhancing reliability.

We have reported in the SPES Deliverable 1.3.A-1 [Cam09], the major analysis techniques and their state in research and industry. The techniques that we have analized are variegated, and with different characteristics and purposes. Therefore the choice of the solution is made offsetting these characteristics and the needs of the project. We want to evaluate in this document which methods might be used in a modelling theory for SPES project. In our opinion the use of a Formal Verification [CW96] technique can improve the overall reliability of the systems. With such techniques the whole development process can be made in an efficient way, using less time and so with economic saves. Besides, formal verification is exhaustive, provides a formal proof of the system´s correctness and improves the knowledge of system. However, it is difficult and time-consuming, only as reliable as the formal models used and it´s difficult to prove the proof correctness.

In order to analyse the behaviour and the functionality of a system by formal methods [CW96], also the specification should have a rigorous definition, so it is not enough to use only formal verification. Actually in the industry formal methods are used seldom, despite the increasing of the recognition in international guidelines and standard for the development of safety-critical system. For the industries to implement formal methods in the whole life cycle development actually needs to much investments. So industries prefer to use formal verification within the existing development process. In the SPES project we want to provide a complete formal modelling theory, inclusive formal verification techniques.

Overview Analysis Techniques in Focus In [HHR09] we introduced the Focus [Bro07] approach for the development of reactive systems. One of the goals of SPES project is to provide a complete formal modelling theory and therefore we aim to integrate formal verification. The aim of the current paper is to provide analysis techniques suitable for Focus specifications. We present the methodologies considering the results reached within Verisoft XT project[1]. In this project are described verification techniques for embedded systems, with technical issues as well as case studies on embedded control systems. They have considered the verification of hardware, operating systems and application software. The modelling theory used is in the framework of Focus meanwhile AutoFocus 3 [SPHP02] is used as support tool. The verification is made from an informal specification through multiple transformation steps to obtain a verified formal specification. This research work is focused on application of simulation, model checking and theorem proving methods [CW96].

In this work we define the ideas for implementing analysis techniques in Focus, in two different phases. First, validation and verification activities are provided translating the formal model

[1]http://www.verisoftxt.de

for the Isabelle/HOL theorem prover [WPN08]; Second, verification of the translation of the model and its corrisponding C code generation with a model checker against a specification obtained from the formal requirements. With regard to the model checker we propose an our own solution, based on actual research work.

Model Checking In few words model checking is an automated technique which, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) the model. One important characteristic is that the verification is exhaustive, i.e. all possible input combinations and states are taken into account, and a counterexample is usually generated in case a certain property does not hold. Respect other verification techniques Model Checking is fully automatic. Verification is usually carried out by using algorithms to demonstrate the satisfiability of properties formalized as logical formulae over the model of the system.

Model checking approach suffers from "state space explosion" problem, since are considered all the possible input and states in the verification. Recent advances in techniques, however, have managed to deal with very large state spaces by using a symbolic representation of the state space, therefore there is no explicit enumeration of the states; and by abstraction techniques, pratically removing information from the concrete system. Such tools have been successfully applied to very large state spaces in hardware verification. We have considered model checking techniques for embedded systems also because such systems use fault tolerance, i.e. the property of a system to provide, through redundancy, a service that complies with the specification despite the occurrence of faults. Therefore its characteristics, such as the use of redundancy, might help to reduce the state space explosion problem applying abstraction techniques.

An important technique to reduce the state explosion problem is in fact abstraction, wherein a smaller abstract model is verified that keeps an approximation of the behaviors of the original system. The major problem in applying this technique is in constructing such an abstract model. A successfull technique is counterexample guided abstraction-refinement (CEGAR) [CGJ+00] that addresses this problem by constructing abstractions automatically by starting with a coarse abstraction of the system, and progressively refining it, based on invalid counterexamples seen in prior model checking runs, until either an abstraction proves the correctness of the system or a valid counterexample is generated.

Our Model Checker In the same framework of abstraction and refinement for model checking we propose for AutoFocus specifications an our own technique, which is based on three-valued logic [CGLT09]. Our Three-valued Abstraction Refinement (TVAR) approach extends the existing work on the three-valued model checking [BG99].

Outline We present the analysis techniques applied to Focus specifications and its layers in Section 2. In Section 3 we describe model checking and in Section 4 is presented our model checker solution. Future work are discussed in Section 5 and finally conclusions in Section 6.

2 Overview Analysis Techniques in Focus

We know that many embedded systems are used in safety-critical environments and so it is important to apply them an exhaustive verification and validation. Our proposal is based on formal verification techniques. A full testing of such applications is often not possible because it is too time consuming or too expensive. However, errors or fault state in these systems may lead to severe events. Besides a replacement of the software while in operation is often too difficult or not possible for the nature of such systems and anyway it is not simple as in common desktop software. So the interest from the industries for formal verification is grown and several research projects have produced tools, but they are still not widely used and are not a standard in the development of embedded systems, due also to the complexity of formal verification. Another factor that has limited the diffusion of such technique is also the difficulty to provide a standard tool considering that most of available tools use a proprietary input model. There are for instance few model checking tools that provide interfaces to standard development tools, as Statemate, MATLAB/Simulink etc or that are integrated into such tools for instance SCADE Suite.

Our idea is to provide a complete modelling theory with verification techniques integrated. We have introduced for SPES the FOCUS modelling theory, we describe now the different layers that compose a model specification, and which analysis techniques to apply them. The layers analysed are: FOCUS, AutoFOCUS and C code. Figure 1 contains a draft of the methodologies described in the next sections.

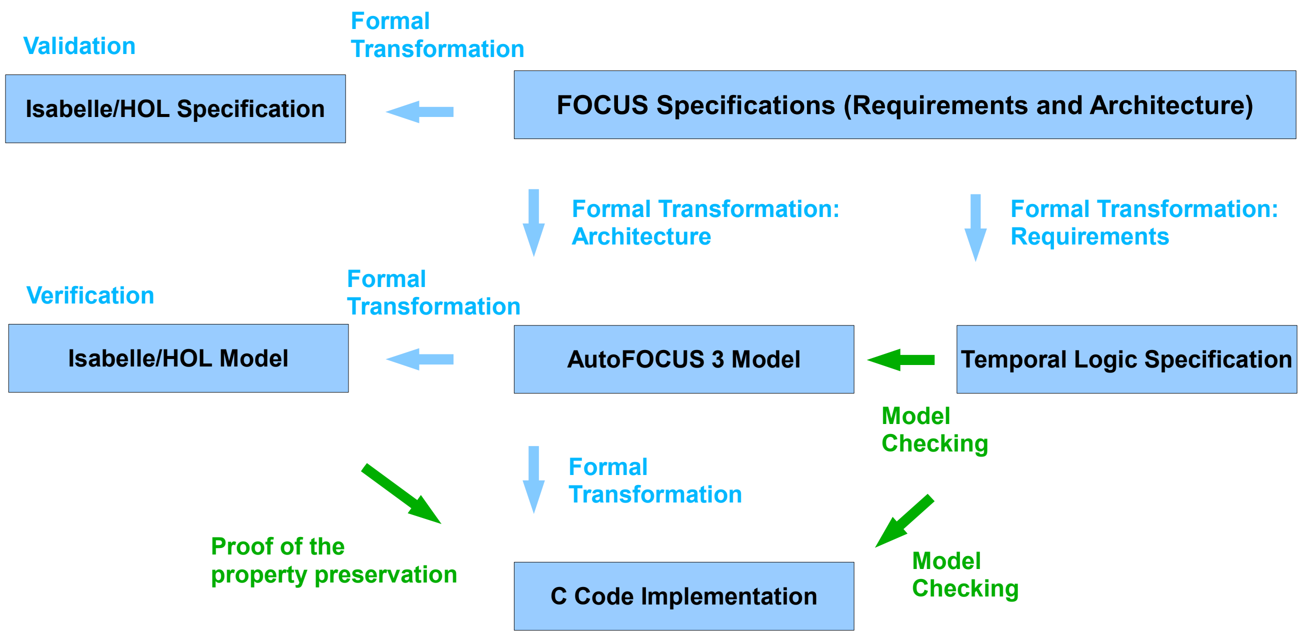

Validation
Formal Transformation
Isabelle/HOL Specification
FOCUS Specifications (Requirements and Architecture)
Formal Transformation: Architecture
Formal Transformation: Requirements
Verification
Formal Transformation
Isabelle/HOL Model
AutoFOCUS 3 Model
Temporal Logic Specification
Model Checking
Formal Transformation
Proof of the property preservation
C Code Implementation
Model Checking

Figure 1: Overview of Analisys Techniques based on the Verisoft XT approach.

2.1 Focus layer

Focus is a framework for formal specifications and development of distributed interactive systems. This framework has an integrated notion of time and modelling techniques for unbounded networks, provides a number of specification techniques for distributed systems and concepts of refinement. Moreover, Focus specifications are much more readable and manageable, in fact the advantage of graphical notation is extremely important when we are dealing with systems of industrial size. Focus supports a variety of specification styles, which describe system components by logical formulas or by diagrams and tables representing logical formulas.

Theorem Proving In general we represent in Focus two kinds of specifications: a requirements specification of the system and its architecture specification (corresponding to the black and the glass box view on the system, respectively). In Verisoft XT approach these representations prepare the ground to verify the system architecture specifications against the system requirements by translating both to the theorem prover Isabelle/HOL via the framework "Focus on Isabelle" [Spi07], and the refinement relation between them can be validated. The case when one needs to prove a single property of a system specification can also be seen as a refinement relation: this property can be defined as a Focus specification itself and then one just needs to show that the system specification is its refinement. Using this approach one can perform automatic correctness proofs of syntactic interfaces for specified system components. Verifying Focus specifications using Isabelle/HOL means proving that the semantics of the system architecture specification (white-box behavior) implies the semantics of the system requirements specification (black-box behavior).

2.2 AutoFocus 3 layer

We have described therem proving for verifying and validating the Focus layer, but in some cases inconsistencies can still remain in the specification, model or code even after verifying certain properties. Thus, not only theorem proving techniques, but also testing, model checking and simulation must belong to the development process. In order to apply such techniques in Verisoft XT approach the Focus architecture specification is translated to a representation in the related CASE tool AutoFocus 3[2]. AutoFocus 3 is a scientific research prototype tool, implementing a modelling language based on a graphical notation and a restricted version of the formal Focus semantics. This tool provides simulation and model-checking facilities.

Model checking In AutoFocus 3 functional properties can be specified using temporal logic notations, especially LTL (Linear Temporal Logic). Temporal properties can be checked using model checking tools, e.g., SMV (Symbolic Model Verifier) [McM92]. That way, system properties expressible in LTL can be verified by exporting AutoFocus 3 models to SMV and model checking the corresponding temporal formulae, as has been performed, for instance, for selected safety-critical properties in [FFH+09].

[2]http://af3.in.tum.de/

Besides the functional properties we want also to verify the Focus requirements specification. We can translate it or another kind of specification to temporal logic and again using the AutoFocus 3 model exported for SMV, we can check these temporal formulae. The transformations from Focus to temporal logic and to the AutoFocus 3 representation are dimostrated in a formal and schematic way in Verisoft XT approach. In this work we present our own TVAR model checking solution (described in Section 4). We propose as future work (Section 5) to use TVAR for verifying the AutoFocus 3 model exported in a representation for TVAR and the properties expressed by temporal logic, as already made for SMV.

Theorem Proving Within Verisoft XT project a code generator has been developed for creating Isabelle/HOL representations to prove the AutoFocus 3 models, as described in [Tra09]. We can formulate properties using more powerful notations expressible in HOL. Therefore a property is formulated as an Isabelle/HOL theorem and proved for the Isabelle/HOL representation of the system.

Simulation We can validate the model using the AutoFocus 3 simulator to get a first impression of the system under development and possibly find implementation errors that we introduced during the manual transformation of the Focus specification into a AutoFocus 3 model.

2.3 C code layer

As illustrated in [Höl09] the final product of the development process is C code generated from the AutoFocus 3 models. The AutoFocus 3 model is transformed to a corresponding C code by a code generator. More precisely a generator for C0 code, a C language subset constructed for usage with the Isabelle/HOL verification environment as discussed in [Sch06]. It is shown in [Höl09] that this trasformation step preserves the properties of the model. C0 differs from C by restricting of the language. Well-known, hazardous features, like pointer arithmetic, are forbidden in C0, while other restrictions, like the non-nested use of function calls, ease the reasoning and verification with Isabelle/HOL. The C0 code generated from AutoFocus 3 models does not need language features still available in C0 like dynamic memory allocation, pointers or arrays.

Model Checking We want to verifiy the C0 code representation with model checking techiques for a further verification of the implemented model. This approach is already applied in other model checkers based on C source code with applicability to embedded software. We propose our own TVAR model checker (Section 4) to make this verification step. We are working for providing the support to C0 programs, which will be completed in future work (Section 5).

Theorem Proving The C0 code generation is performed by the AutoFocus 3 tool, more precisely by a Java program that has not been formally verified. Therefore we cannot formally guarantee the correctness of the C0 code, even after the correctness has been shown for the AutoFocus 3 model and/or its Isabelle/HOL representation. In order to obtain this formal guarantee, the behavioural equivalence of the Isabelle/HOL representation of the AutoFocus 3 model and generated C0 code can be proved in Isabelle/HOL. The construction of a verification environment for C0 code in Isabelle/HOL was presented in [Sch06]. Such a verification environment can also be used for directly proving system properties for the generated C0 code.

2.4 SPES Domain

In the framework of SPES project we consider systems which are constructed from specifications to AutoFocus 3 models. Our proposal is therefore based on verification techniques in the layers AutoFocus 3 and C0 code as shown in Figure 1. We aim to implement model checking and simulation methods.

3 Model Checking

Model checking is a technique that relies on building a finite model of a system and checking that a desired property holds in that model. Roughly speaking, the check is performed as an exhaustive state space search that is guaranteed to terminate since the model is finite. The technical challenge in model checking is in devising algorithms and data structures that allow handling large search 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 checks if the system satisfies the property and gives a "yes" or "no" answer. If the answer is no, i.e., the system does not satisfy the property, model checkers also output a counterexample, i.e., a run of the system which violates the property. The counterexample can be analysed to discover bugs in the system design. Figure 2 describes the process of model checking. Model checking has been used primarily in hardware and protocol verification [CK96], the current trend is to apply this technique to analysing specifications of software systems.

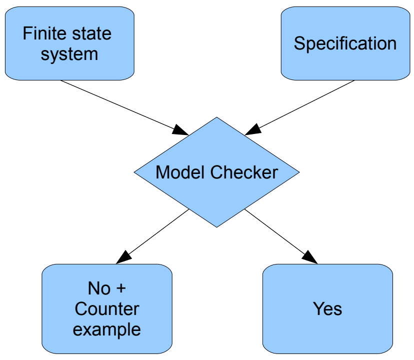


Figure 2: Overview of Model Checking

The specification to verify is usually expressed by temporal logic. Temporal logic is a class of modal logic, which extends propositional logic to incorporate time operators, in the sense that formulas can evaluate to different truth values over time. There are different types of temporal logic notations that correspond to different views of time (branching vs. linear, discrete vs. continuous, past vs. future, real time). Examples of temporal logic formal languages are μ-calculus, Linear Temporal Logic (LTL), Computational Tree Logic (CTL) and Timed CTL. Typically, the system is abstractly represented by a state machine, where the nodes represent the system's states and the arcs represent possible transitions between the states. Because state machine alone might be too weak to provide a complete and interesting description, the states and the transitions are annotated with more specific information. A common approach is to use Kripke structures [CGP99].

3.1 Kripke Structure

The Kripke structures are widely used in model checking, the semantics of temporal theories are traditionally defined in terms of Kripke structures. A Kripke structure is basically a graph having the reachable states of the system as nodes and state transitions of the system as edges. A labeling function maps each node to a set of properties that hold in the corresponding state. Kripke structures can be seen as describing the behavior of the modeled system in a modelling language independent manner. Therefore, temporal logics are really modelling formalism independent; the definition of state´s properties is the only thing that could need to be adjusted for any formalism.

3.2 Properties verified

Temporal logic is a powerful tool to express several properties about systems. Examples of properties that can be expressed and so verified by model checkers are:

- Reachability: some particular situation can be reached
- Safety: something wrong never occurs
- Liveness: something desirable will ultimately occur
- Fairness: something shall (or shall not) occur infinitely often
- Deadlock-freeness: the system can always evolve to a successor state

While a violation of a safety property can be detected by a finite sequence of executions steps in the system, a violation of a liveness property may be detected only by an infinite execution of the system, pratically finding a loop where the property is never reached. Consequently, liveness properties are harder to be verified by model checkers. The temporal logics listed have different ways to express these properties, and some property cannot be expressed from some logics.

3.3 State space explosion

The size of the systems to verify grows exponentially with the number of parallel system components, which limits the applicability of model checking techniques. In the last years one of the main goals of the research has been to extend the size of systems that can be effectively model checked. This problem is the major obstacle in applying model checking to industry. We list two approches used to limit such problem.

3.3.1 Symbolic Model Checking

Symbolic Model Checking [McM92] was introduced around 1987 and is considered an important breakthrough in model checking techniques. Using this technique the representation of the system to verify are reduced, in order to handle the state space explosion problem. This technique utilizes a symbolic representation for sets of states and state transitions, which are represented by a Boolean encoding. The size of this encoding is greatly reduced by representing such structures by Binary Decision Diagrams (BDDs) [Büc62], which are traditionally used to represent Boolean functions. The verification for symbolic model checking is made using fixpoint operators, which operate on the BDDs. The introduction of symbolic model checking was very successful in the last decade in the research community as well as in industry application especially in the verification of hardware designs.

3.3.2 Abstraction

Another improvement with respect to classical model checking algorithms is when only a restricted part of the whole state space is explored. An important approach consists of reducing the size of the model to be verified by abstraction, therefore replacing the concrete model of the system with a smaller abstract model. Abstract interpretation [CC77] is a theory of approximation of mathematical structures, in particular those involved in the semantic models of computer systems. Traditionally, abstraction techniques are designed to be conservative for property that hold. Thus, if a property is verified in the abstract system, it also holds for the concrete system. If, however, a formula does not hold for the abstract system, this may be because too much information has been hidden; say the abstraction is too coarse. In this case the abstraction needs to be refined, adding information which avoids having again the same spurious counterexample.

4 TVAR Model Checker

We first describe the major successful abstraction technique for model checking, i.e. the counterexample guided abstraction refinement (4.1) and then our solution based on three valued logic (4.2).

4.1 Counterexample guided abstraction refinement

One of the successfull abstraction technique is Counterexample guided abstraction-refinement (CEGAR) [CGJ+00]. The paradigm of iterative abstraction and refinement has gained momentum in recent years as a particularly effective approach for the scalable verification of complex hardware and software systems. CEGAR is an efficient automatic refinement technique which uses information obtained from erroneous counterexamples. The initial abstraction and the refinement steps are efficient and entirely automatic. The refinement procedure is guaranteed to eliminate spurious counterexamples while keeping the state space of the abstract model small. The advantages of this methodology have been demonstrated by experimental results and the interest received in research and application.

4.1.1 The CEGAR loop

This technique constructs an abstract model of the system, and then a traditional model checker is used to determine whether properties hold in the abstraction. If the answer is yes, then the concrete model also satisfies the property. If the answer is no, then the model checker generates a counterexample. Since the abstract model has more behaviors than the concrete one, the abstract counterexample might not be valid in the concrete system, viz a spurious counterexample.

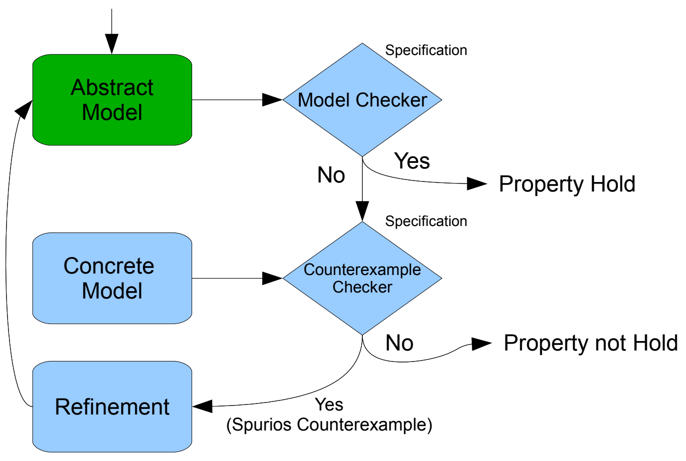


Figure 3: CEGAR Loop

Therefore they provide a symbolic algorithm to determine whether an abstract counterexample is spurious. If the counterexample is not spurious, we report it to the user and loop is stopped. If the counterexample is spurious, the abstraction must be refined to eliminate it. Instead when the property is true in the abstract model is it true also in the concrete model, so the property holds. In Figure 3 is represented the CEGAR verification loop.

4.2 Three-valued logic abstraction refinement

We propose for SPES the integration of a model checking techniques, with abstraction and refinement of the system. Our work may be considered as a generalization of the CEGAR (Section 4.1) in the setting of abstractions: Three-valued Abstraction Refinement (TVAR) [CGLT09] is based on three valued logic and we are presently implementing it. The specifications to verify are expressed by μ-calculus, which is a temporal logic, and the algorithm that checks these specifications is used for symbolic (BDD-based) model-checking (Subsection 3.3.1). Therefore we combine the major techniques used for reduce the state space explosion problem.

4.2.1 Model Abstraction

Our model checker is based on notion of model abstraction (Section 3.3.2). An abstraction is conservative with respect to the property evaluation if the results in the abstraction verification are correct in the corrispondent concrete system. Traditionally, abstraction techniques are designed to be conservative when the result is *true*, that is the property is verified by the abstraction. Thus, if a property is verified in the abstract system, it holds for the concrete system. If, homever, a formula does not hold for the abstract system this may be because too much information has been hidden, say the abstraction is too coarse. In fact in CEGAR, Figure 3, a counterexample obtained in the abstract system is concretized and stepwise replayed, on the concrete system. This either shows that the property is indeed violeted in the concrete system or, it identifies a transition present in the abstract state machine system but not in the concrete. In this latter case, the abstraction may be refined by eliminating the corresponding transition in the abstract state machine system.

In multi-valued logic a formula evaluates no longer to just true or false but to one of many truth values. In our work we use three-valued logic, so the value are $true$, $false$ and $don't\ know$. In three-valued model checking, abstractions are considered to be conservative for $true$ and $false$ evaluations. It means that only when we have a $don't\ know$ result we have to refine our abstraction, in the other cases the problem is solved also for the concrete system.

In order to have such framework Kripke structures are extended to the three-valued logic by assigning to each proposition in each state the values $true$, $false$ and $don't\ know$ and the same to each transition which means there is a transition, there is not, or may be. Abstract systems are so no longer normal Kripke structure but Larsen´s and Thomsen´s Kripke modal transition systems [LT88].

Kripke modal transition systems A Modal Transition System (MTS) [LT88] is a specification in state-transition form, where *loose* transitions (that is, transitions that may or may not be present in the final implementation) are labeled as may-transitions, and *tight* transitions, which must be preserved in the final implementation, are labeled as $must$-transitions. Since a transition that must be present in the final implementation may be present as well, every $must$-transition is by definition a may-transition. This is a foundation for three-valued program analysis.

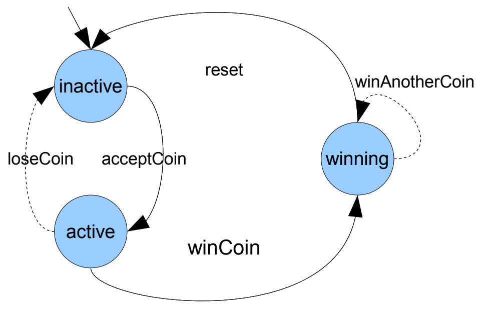


Figure 4: Kripke modal transition system example.

We show an example of MTS in Figure 4, a specification of a slot machine, where some behaviors of the final implementation are fixed (the *must*-transitions) and some are uncertain (the *may*-transitions). Kripke modal transition systems, are a generalization of MTS, and are used for represent the abstract model meanwhile for the concrete one are used common Kripke structures. Transitions in the Kripke modal transition systems which are labeled as *may*-transitions may or may not be present in the concrete Kripke structure; transitions which are labeled as *must*-transitions must be present in the concrete structure. This leads to three possible values for a transition: It is there for sure, it is not there for sure, or it may be there. The notions of *must* and *may* transitions determine an over and under-approximation of the system.

State abstraction We consider abstractions of Kripke structures induced by joining states to form abstract states, as is made in traditional logic systems. In this manner we can reduce the dimension of the Kripke structure joining concrete state.

In Figure 5 there is an example of Kripke structure which shows state abstraction. The dotted transitions between abstract states represent *may*-transitions and the others are *must*-transitions. They induce an over- and under- approximation of the system. The model checking procedure is made on this structure.

4.2.2 TVAR Model Checking

Model checking of the abstract system either yields *true* or *false*, which is called a definite result, or, the answer is *don't know* for an indefinite result. In the latter case, a *cause* for the indefinite result may be identified, which is a transition or state of the abstract system, for which it is only known that it *may* be available in the concrete system or, respectively, that it may satisfy some properties. Moreover, intuitively, this lack of knowledge is a reason for the indefinite result. The idea of identifying causes useful for refinement was first made precise in the context of three-valued CTL model checking by Grumberg and Shoham [SG07]. In case that model checking yields a definite result, i.e. *true* or *false*, this result also holds for the concrete system. If the model checking result is *don't know*, we moreover obtain a set of *causes* following the cause annotated semantics.

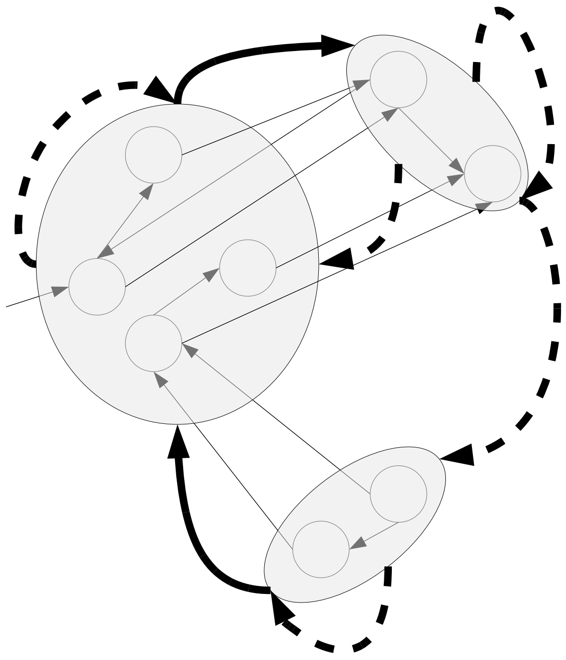


Figure 5: Over- and under- system approximation example.

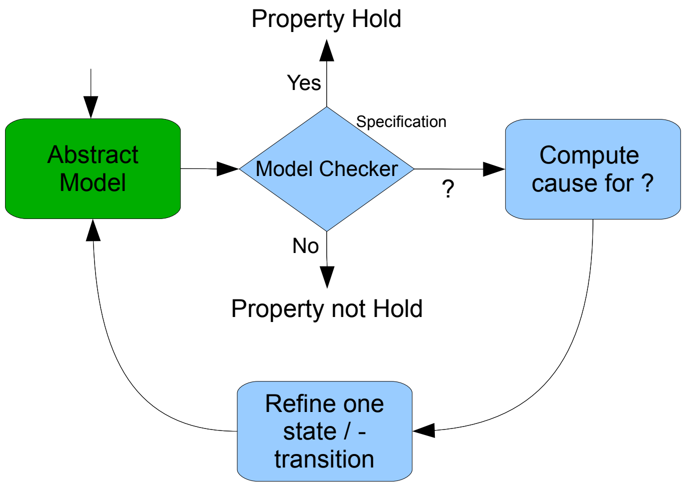


Figure 6: TVAR Loop

The resulting abstraction/refinement loop is shown in Figure 6 and is called Three-valued abstraction refinement loop. In comparison to CEGAR, TVAR has some advantages, thanks to the simultaneous over- and under approximation of the concrete system:

- TVAR allows verifying also existential properties that cannot be verified using CEGAR.

- While within CEGAR, a counterexample has to be replayed on the concrete system to obtain information for refinement, the cause for indefinite results can directly be obtained in the abstract system in TVAR and requires no further processing on the concrete system.

The price to pay for TVAR is that an underlying theorem prover/SMT solver, which is typically both employed in CEGAR and TVAR to reason on the existence of transitions, has to be more powerful in TVAR than in CEGAR.

4.2.3 Implementation

We have implemented the TVAR approach within the TVARC tool. It presently reads first-order system descriptions and our goal is to extend the input traslating AutoFocus 3 model and adding support to C program (initially to C0). The properties are specified by μ-calculus and is implemented the TVAR loop, shown in the previous section. Many temporal logics, e.g., CTL, LTL, and CTL* can be easily translated into μ-calculus [Eme96], therefore we have a strong expressivity for the properties definition.

We have implemented the TVAR loop based on the SMT solver Z3 [dMB08], whose rich API allow a convient integration into our tool TVARC. The model checker is implemented on top of the JavaBDD-library. Instead of the usual state sets our model checkerd has to operate on elements of the three-valued logic introduced above. One component of our implementation maps three-valued structures to standard BDDs.

Results We have checked several existential properties of protocols, both with NuSMV [CCG+02] and TVARC, where we used integers of finite range for NuSMV and no restriction for TVARC. The general message of the experiments is that as long as NuSMV has no memory problems, it checks much faster than TVARC. While TVARC is tipically slower than the highly-efficent model checker NuSMV, it uses far less memory. Thus, TVARC is able to verify systems in a reasonable amount of time systems that are far beyond the capabilities of NuSMV. Hence, this is a prove that TVAR will considerably extend the range of today´s pratical verification problems.

5 Future Work

Formal Verification Isabelle/HOL can be used to formulate and prove refinement respect equivalence relations between different system specification resulting from different development phases and techniques. Interesting topics for future work are formal refinement/equivalence relations between Focus and AutoFocus 3 systems specifications as well as behavioural equivalence of AutoFocus 3 models and generated C0 code.

Model Checking In Verisoft XT approach, as we explained in Section 2, the AutoFocus 3 models are verified implementing model checking techniques. We propose such approach also for SPES, by an integration with the model checkers SMV and TVAR. Besides in such integration a future improvement would be the implementation of counterexample analysis, i.e. when the verification returns a negative result, the informations provided by the counterexample might be related with the model and presented in a graphical way in the tool, in order to be useful for the debug phase. The TVAR tool is actually working in progress. We are implementing the support for C language programs as input, providing before the support to C0 programs which are generated from AutoFocus 3 models. The following step is to support a direct conversion from an AutoFocus 3 model to a representation for TVAR, in this way is it possible to make the model verification. It will be interesting to work with a concrete tool in order to consider the performances in time and memory occupation, compared with the existing model checking tools (as for instance SMV).

Domain Analysis In order to provide a better understanding of our methodologies, we propose as future work a complete case study with properties examples whereby applying the proposed analysis techniques. We are interested also to study the definition of classes of properties, which may be verified, referred to an application domain and/or pattern specifications suitable for SPES project. It will be also interesting to study the meaning (in terms of system reliability) for the failure or success results of the analysis techniques in selected classes of properties.

6 Conclusions

We have presented a methodology for developing safety-critical embedded systems extended with pervasive verification techniques. In this document we have shown an overview of analisys techniques studied for Focus theory and its layers, and based on it a proposal for SPES project. The artifacts of the different stages can be created through schematic transformations, which are, depending on the development state of the tools, automatic or at least tool supported. These artifacts are verified during each development phase: we apply both automatic and semi-automatic interactive verification techniques. Our proposal is mainly based on formal verification, applied to the AutoFocus and C0 layers of a Focus specification. We opt for methods in the framework of model checking and simulation. The advantages of such techniques fit with the criticality of embedded systems development.

We propose an our own model checker based on Three-valued Abstraction Refinement (TVAR). Model checking is automatic, logics used can easily express many concurrency properties and in case of a violation provide counterexamples that can help in the debugging phase. The main disadvantage compared to other verification techniques is the state space dimension. In fact it may grow exponentially to the number of used variables or concurrent components, the known state explosion problem, due to the exhaustive analysis performed. The common technique for reduce such problem are based on model abstraction. Our solution is in the same branch of CEGAR, but our work is based on three-valued logic system instead of common two valued logic, with notions of abstraction and refinement.

The proposed development methodologies make possible to perform formal verification on different levels of system abstraction and development, ranging from formalisation of system requirements to program code. The major verification techniques reported here are model checking, simulation and interactive theorem proving. We suggest to use model checking and simulation because they provide different features needed for different verification tasks and complementing each other. While model checking provides automatic verification for property notations of limited expressiveness like LTL, simulation allows to validate the model showing its execution. Besides the implementation of such techiques as future work we want to provide an easy specification way for the properties to verify and a conterexample analysis for improve the debug phase.

References

[BG99] Glenn Bruns and Patrice Godefroid. Model checking partial state spaces with 3-valued temporal logics. In *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*, pages 274–287, London, UK, 1999. Springer-Verlag.

[Bro07] Manfred Broy. Two Sides of Structuring Multi-Functional Software Systems: Function Hierarchy and Component Architecture. pages 3–12, 2007.

[Büc62] J. Richard Büchi. On a decision method in restricted second order arithmetic. In *Logic, Methodology and Philosophy of Science (Proc. 1960 Internat. Congr .)*, pages 1–11. Stanford Univ. Press, Stanford, Calif., 1962.

[Cam09] Alarico Campetelli. SPES 2020: Analysis Techniques: State of the Art in Industry and Research. Technical report, Technische Universität München, 2009.

[CC77] 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.

[CCG$^+$02] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[CGJ$^+$00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement, 2000.

[CGLT09] Alarico Campetelli, Alexander Gruler, Martin Leucker, and Daniel Thoma. *Don't know* for multi-valued systems. In Zhiming Liu and Anders P. Ravn, editors, *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis (ATVA'09)*, volume 5799, pages 289–305. Springer, 2009. to appear.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* The MIT Press, January 1999.

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

[CW96] Edmund 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.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[Eme96] E. Allen Emerson. Model checking and the mu-calculus. In Neil Immerman and Phokion G. Kolaitis, editors, *Descriptive Complexity and Finite Models*, volume 31 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 185–214. American Mathematical Society, 1996.

[FFH$^+$09] M. Feilkas, A. Fleischmann, F. Hölzl, C. Pfaller, K. Scheidemann, M. Spichkova, and D. Trachtenherz. A Top-Down Methodology for the Development of Automotive Software. Technical report, Technische Universität München, 2009.

[HHR09] Alexander Harhurin, Judith Hartmann, and Daniel Ratiu. Motivation and Formal Foundations of a Comprehensive Modeling Theory for Embedded Systems. Technical Report TUM-I0924, Technische Universität München, 2009.

[Höl09] F. Hölzl. The AutoFocus 3 C0 Code Generator. Technical Report TUM-I0918, Technische Universität München, 2009.

[LT88] Kim Guldstrand Larsen and Bent Thomsen. A modal process logic. In *LICS*, pages 203–210, 1988.

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

[Sch06] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL.* PhD thesis, Technical University of Munich, April 2006.

[SG07] Sharon Shoham and Orna Grumberg. A game-based framework for ctl counterexamples and 3-valued abstraction-refinement. *ACM Trans. Comput. Log.*, 9(1), 2007.

[SPHP02] Bernhard Schätz, Alexander Pretschner, Franz Huber, and Jan Philipps. Model-based development of embedded systems. In *OOIS Workshops*, pages 298–312, 2002.

[Spi07] M. Spichkova. *Specification and Seamless Verification of Embedded Real-Time Systems: FOCUS on Isabelle.* PhD thesis, 2007.

[Tra09] David Trachtenherz. Ausführungssemantik von AutoFocus-Modellen: Isabelle/HOL-Formalisierung und Äquivalenzbeweis. Technical report, Institut für Informatik, Technische Universität München, January 2009.

[WPN08] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle Framework. In *TPHOLs '08: Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, pages 33–38, Berlin, Heidelberg, 2008. Springer-Verlag.