Manifesto from Dagstuhl Perspectives Workshop 10482

Modeling, Analysis, and Verification – The Formal Methods Manifesto 2010

Edited by

Jörg Kreiker[1], Andrzej Tarlecki[2], Moshe Y. Vardi[3], and Reinhard Wilhelm[4]

1 **TU München, Germany,** kreiker@in.tum.de
2 **University of Warsaw, Poland,** tarlecki@mimuw.edu.pl
3 **Rice University, U.S.,** vardi@cs.rice.edu
4 **Saarland University, Germany,** wilhelm@cs.uni-saarland.de

Abstract

This manifesto represents the results of the Dagstuhl Perspectives Workshop 10482 *"Formal Methods – Just a Euro-Science?"* held from November 30 to December 3, 2010 at Schloss Dagstuhl, Germany. We strive to clarify the terminology and categorize the abundance of concepts and methods in order to reduce misunderstandings prevalent in the research community and in communication with industry. We discuss the industrial acceptance of formal methods and how to increase it by targeted research and improved education. Finally, we state a few challenges and provide perspectives of the field.

This document is opinionated in nature and biased towards the experiences and views of the participants listed in the appendix, further distilled by the authors.

Seminar 30. November–03. December, 2010 – www.dagstuhl.de/10482

1998 ACM Subject Classification B.4.4 Performance Analysis and Design Aids, D.2.4 Software/Program Verification, F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Formal methods, Verification, Analysis, Modeling, Design for Verifiability

Digital Object Identifier 10.4230/DagMan.1.1.21

Executive Summary

Formal methods are employed during system-development process to improve the quality of the system, to increase the efficiency of the development process, or to derive guarantees about qualities of the system. The term *"formal methods"* has traditionally been used for a number of different approaches, including *modelling* and *specification languages*, as well as *methods and tools to derive properties of systems.* Because of the vagueness of the term "formal methods", it may perhaps, be desirable to replace it by *"modelling, analysis, and verification".*

A good recent overview of industrial projects concentrating on the early phases of specification and design has been given in a recent survey article: Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, John S. Fitzgerald: Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4), 2009.

The Dagstuhl Perspectives Workshop, held in December 2010, concentrated mostly on methods for system analysis and verification. These are employed in the design phase as well as in later phases of system development. Model checking, abstract interpretation, equivalence checking, and verification by deduction –all developed in academia– are the most impressive success stories.

After a very long gestation period, formal methods for the derivation of program properties have finally gained some measure of industrial acceptance. There are, however, remarkable differences in the degree of this acceptance. There is a clear correlation between the criticality of systems and the costs of failure, on one hand, and the degree to which formal methods are employed in their development, on the other hand. Hardware manufacturers and producers of safety-critical embedded systems in the transportation industry are examples of areas where applications of analysis and verification methods are perhaps most visible. A semiconductor design gone wrong is just too costly for any cost argument against the use of formal design and verification tools be acceptable. Threats of liability costs are strong arguments for the use of formal methods in the development of safety-critical embedded systems. Different application areas often entail different approaches to the use of formal methods. Safety-critical systems call for the use of sound methods to dogmatically ensure correctness. General-purpose software with strong time-to-market pressures encourage a more pragmatic attitude, with emphasis on bug-chasing methods and tools.

Industrial domains with certification requirements have introduced tools based on formal methods into their development processes. Most current certification regulations are, however, still process-based; they regulate the development process and do not state the required properties of the result. Critics describe this as "Clean pipes, dirty water." The trend to use formal methods will become stronger when certification standards move from process-based assurance to product-based assurance. These new standards will specify the guarantees to be given about system properties. Several current standards for transportation systems highly recommend abstract interpretation and model checking for systems at the highest criticality level. "Highly recommend" actually means "required". The loophole is the "state-of-practice" argument. The developer can be exempted from using a highly recommended method by arguing that it is not yet the state of practice.

Several participants of the workshop have expressed the important role of champions of a formal method. A champion, enthusiastic about the potential of and competent in the use of a verification method, is often needed to introduce the method and associated tool to the development process. Often, once the champion leaves, the degree of adoption declines dramatically.

The expectations towards analysis and verification methods have always been very high, often due to unrealistic promises. These unrealistic promises have mostly been the result of the ignorance of the differentiation of roles. Three distinct *roles* are connected to a formal method: the *researcher* develops the theoretical foundations of the method; the *tool developer* implements the method; and the *users* apply the tool in an industrial setting. The different analysis and verification methods have very different requirements imposed on their users, which has implications for their acceptance in industry. Researchers and tool developers often develop their methods and tools for their own use. Subsequently, they use these tools with a high degree of expertise. The experience of such expert users is quite different from that of industrial users, who do not have such degree of expertise. Thus, reports by expert users are often quite rosy and create unrealistic expectations. The expectations towards analysis and verification methods are astonishing in the light of the known undecidability or intractability of the problems they are expected to solve; the methods and tools are expected to be at the same time fully automatic, effective and efficient, and easy to use. Disappointment is unavoidable. Nevertheless, the border between what can currently be done and what is still out of reach is permanently moving, with significant progress accomplished over the last 30 years.

One challenge for further advances is higher *degree of automation*: the different methods require different degrees of user interaction and of user qualification. Currently, with few exceptions, such as Microsoft Research's Boogie platform, there is little integration among different tools. Nevertheless, advances can be expected in the coming years from *tool integration*, starting with information exchange between tools and common exchange formats. Specifically, there is a high potential for improvement from a synergetic integration of model-based design tools with analysis and verification tools.

Scalability of the methods and tools is still considered a problem. The *exploitation of large-scale parallelism* may increase the size of verifiable systems. A clear *identification of application areas* for the various methods rather than the search for universal methods, doomed to fail, will avoid user disappointment.

The embedded-system industry has already realised that badly structured systems written in obscure programming style cannot be effectively maintained. Similarly, it cannot be expected that verification methods would cope with such systems. Systems should be *designed for verifiability.*

While formal methods have often been dismissed by many as "Euro-Science" –a rather abstract research with little chance for industrial adoption– decades of research, both basic research and tool development have started to bear fruits, attracting an increasing level of industrial interest. This interest is often accompanied by unrealistic expectations, but, at the same time, provides an opportunity and challenge to researchers working in this area, as more basic research and good tools engineering are needed to solve the challenges outlined above.

Table of Contents

1 Introduction

Formal methods are employed during system-development process to improve the quality of the system, to increase the efficiency of the development process, or to derive guarantees about qualities of the system. After a very long gestation period, formal methods for the derivation of program properties have finally gained some measure of industrial acceptance. There are, however, remarkable differences in the degree of this acceptance. There is a clear correlation between the criticality of systems and the costs of failure, on one hand, and the degree to which formal methods are employed in their development, on the other hand. Hardware manufacturers and producers of safety-critical embedded systems in the transportation industry are examples of areas where applications of analysis and verification methods are perhaps most visible. Different application areas often entail different approaches to the use of formal methods. Safety-critical systems call for the use of sound methods to dogmatically ensure correctness. General-purpose software with strong time-to-market pressures encourage a more pragmatic attitude, with emphasis on bug-chasing methods and tools.

The expectations towards analysis and verification methods are astonishing in the light of the known undecidability or intractability of the problems they are expected to solve; the methods and tools are expected to be at the same time fully automatic, effective and efficient, and easy to use. Disappointment is unavoidable. Nevertheless, the border between what can currently be done and what is still out of reach is permanently moving, with significant progress accomplished over the last 30 years.

One challenge for further advances is higher *degree of automation*: the different methods require different degrees of user interaction and of user qualification. There is little integration among different tools. Nevertheless, advances can be expected in the coming years from *tool integration*, starting with information exchange between tools and common exchange formats. Specifically, there is a high potential for improvement from a synergetic integration of model-based design tools with analysis and verification tools.

Scalability of the methods and tools is still considered a problem. The *exploitation of large-scale parallelism* may increase the size of verifiable systems. A clear *identification of application areas* for the various methods rather than the search for universal methods, doomed to fail, will avoid user disappointment.

The embedded-system industry has already realised that badly structured systems written in obscure programming style cannot be effectively maintained. Similarly, it cannot be expected that verification methods would cope with such systems. Systems should be *designed for verifiability*.

While formal methods have often been dismissed by many as “Euro-Science” –a rather abstract research with little chance for industrial adoption– decades of research, both basic research and tool development have started to bear fruits, attracting an increasing level of industrial interest. This provides an opportunity and challenge to researchers working in this area, as more basic research and good tools engineering are needed to solve the challenges outlined above.

2 Concepts

Defining formal methods is easy. I did it 100 times.[1]

[1] Adapted from a quote by Mark Twain on quitting smoking.

We understand the area of *modelling, analysis and verification* to be mathematically founded *techniques* and *tools* that aid *humans* to *construct systems* of a higher *quality* with less *resource* usage.

Systems

Systems we consider consist of software, hardware, or a combination thereof. We use the term *data* to abstract the interaction of systems with the physical world (such as input from users or sensors). A system together with data *executes*, resulting in a (number of) mathematically defined *run(s)* of the system.

Verification

(Functional) Verification is concerned with the behavior *intended by the constructor*: 'A system should do, what I want it to do' Intended behavior is a set of acceptable runs. Acceptable runs can be defined implicitly or explicitly. Explicit definitions of acceptable runs are called *specifications.* Typically, specifications are (parts of) programs written in a formal language. The *verification* of a system formally proves its conformance with its *specification.*

Verification is the most abused term in this arena. Almost any activity to convince oneself or a client of desired system properties is called *verification* or even *formal verification.* Most notably (non-exhaustive) testing and bug finding sail under this false color. What contributes to this confusion is the polyvalent nature of several formal-method approaches. Model checking, advertised as a verification technique, is, for complexity reasons, mostly used for bug finding. We will use *verification* in the strict sense, defined in Subsection 3.2.6.

Resources

We distinguish two kinds of resource. On the one hand, there are resources consumed during the *construction* and for the *maintenance* of a system. On the other hand, there are resources consumed during the *runs* of the system. Examples of resources are time, energy, person months, and money.

Roles

Humans are involved with formal methods in several roles.

- *researchers* develop the foundations of a formal method,
- *tool developers* realize a tool based on some formal foundation,
- *users* apply tools to systems under study.

Some tools are push-button tools. However, given the undecidability of the program verification problem, more or less input from the tool user is needed as will be detailed when individual methods are described. The formal-methods area has long suffered from the fact that the researcher was also the tool developer, and worse, the user. This type of user has made unusually good experience with his own method and tool, and often raised too high expectations.

Models

Systems are the end-product of a construction process. A *model* is then viewed as an abstraction of a system. It should be formally defined (possibly in the same language as the

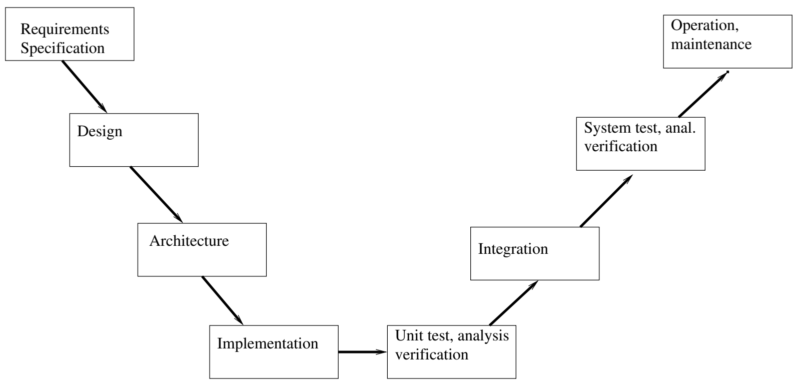
Requirements Specification
Design
Architecture
Implementation
Unit test, analysis verification
Integration
System test, anal. verification
Operation, maintenance

Figure 1 The V cycle for software development

system). Formal methods study the relation between a model and the system *implementing* the model.

Having introduced general definitions, we instantiate them with existing systems, specifications and methods in the next section.

2.1 Development Process

The construction of systems is organized according to some development process. One such process is depicted in Figure 1. Typically, different formal methods contribute to that process and, typically, each step during the development is associated with preferred methods.

Modeling and specification languages are associated with the first three phases, Requirements Specification, Design, and Architecture. Code synthesis in model-based approaches is associated with Implementation. Model checking, abstract interpretation, and program verification may be appplied at the model level during the Design phase and at the unit and the system level.

3 A Survey of Formal Methods

In this section, we list typical formal methods in no particular order. It must be noted that different application domains require different methods and that there is not one ideal method. Therefore, we delineate typical application domains with each method.

Method Signatures

Following Dines Bjørners proposal at the workshop, we ask developers and users of formal methods to provide a *signature* of their method. A signature is a type expression over the type constructors: Sys, Model, Data, and Spec, $\rightarrow$, and $\times$. As an example consider

$$\mathit{compiler} :: \mathsf{Model} \rightarrow \mathsf{Sys}.$$

That is, a compiler takes a model (here: a program) and produces a system (here: binary code). Note that binary code could serve as a model (for instance in worst-case execution time analysis) and that a program could serve as a system (if one is interested in the development resulting in the program, say, from a higher specification by model-based development).

3.1 Specification Languages

Specification languages, as the term indicates, are formalisms to write *specifications* that capture properties required of the system (or better, of its behaviour). Therefore, the basic functionality they offer is to built specifications that capture designer's initial, informal ideas about desired system behaviour:

$$\textit{specify} :: \mathsf{Informal\_Ideas} \to \mathsf{Spec}.$$

Once specifications are given, to make them of any use, it must be possible to determine whether or not a given system *satisfies* a specification, or in other words, *realizes* it correctly:

$$\textit{correct} :: \mathsf{Sys} \times \mathsf{Spec} \to \mathbb{B}.$$

Checking this relationship is the matter of *verification* of a system w.r.t. its specification.

Refinement is a relation between specifications, whereby a specification gets refined by incorporating some further design and programming decisions, with the basic requirement that any system that correctly realizes the refined specification satisfies the original specification as well:

$$\textit{refines} :: \mathsf{Spec} \times \mathsf{Spec} \to \mathbb{B}.$$

The approaches to capturing the required system behaviour (or some of its aspects) vary vastly.

At one extreme, we have the use of standard (high-level) programming languages to define a specific behaviour, which is then viewed as a presentation of the desired behaviour of the system. This idea led to the development of *model-oriented* specification languages, where a particular model of a system is built (whether using some programming formalisms, or some more abstract mathematical notation) and then viewed as a specification: any system that displays a behaviour conformant with that of the given model is its correct realization. Development then takes a form of *model refinement*, or reduces to *synthesizing* a program from a model. An archetypical example of a model-oriented specification formalism, which heavily influenced further developments is VDM [20].

Model-based design is extensively used in the embedded-systems industry. It offers a unique chance, namely that code generators are tailored towards analyzability and verifyability. It is not the programmer who has to be forced to produce disciplined code. Code synthesis, as experienced with several code generators for Scade, can be designed to guarantee analyzability and verifiability.

Another corner is occupied by logical formalisms, where a specification is given by simply listing the required properties as formulae of some logic. Specifications for programs-in-the-small are often given in this form, based on a programming logic (with Hoare's logic [19] as a classical example). Not much more than pure formulae of some temporal logics have traditionally been used in model checking. Finally, various algebraic languages were put forward. The purpose was to specify abstract data types by listing axioms that link the operations (and constants) of a data type with each other. They thus specify the behaviour of functions that are to implement the operations [16]. These ideas spurred development of *property-oriented* specification languages, like Z [33], which apart from such basic specifications, built by listing the axioms, offer various mechanisms to combine specification systematically, thus building them in a structured manner. This was perhaps first stressed in the work on CLEAR [10] and developed in full in a whole line of algebraic specification languages, with CASL [6, 26] as a relatively recent fully-fledged example of such a formalism, with complete formal semantics, development and verification methodologies, and support tools under intensive development.

The distinction between model- and property-oriented approaches to specifications was blurred from the very beginning. Logic programming languages, as well as other declarative languages viewed as specification tools found their early place between these two worlds. So did many mature formalisms, like RAISE [17], that incorporate both approaches.

One problem any specification method has to face is how exactly the general specification mechanisms used relate to the programs in a specific programming language. One solution to alleviate this, perhaps first put forward in Larch [18], is to extend explicitly the generic specification language by "interface languages" that link it to various programming languages. Another idea is to develop formalisms that interleave specification with programs, with examples like a number of languages devoted to *design by contract*, like Eiffel [24] and Extended ML [31, 21], or various extensions to programming languages to allow assertions to be inserted, which underlie a number of program verification systems and tools, like JML [9].

As far as applications are concerned, a recent survey on the use of formal methods [37] indicates that the specification/modeling activities were a crucial part of practically all reviewed projects that used any formal methods at all. No surprises here: even if the use of formal methods stops at providing a precise specification (necessarily involving some specification formalisms), the benefits of clear statement of the requirements on and properties of the system or component cannot be overestimated. In the early Transputer project, just the specification of the floating-point arithmetic [3] helped to discover problems with the standard to be implemented, and the full strength of the formal methods in use was highlighted even more when it came to formal verification of its Occam implementation and development of the chip microcode from it. Quite similar experience was brought for instance by a project to specify and verify the AAMP5 microprocessor in PVS (a specification language integrated with support tools and a theorem prover [28]). Two errors in the microcode were identified when the specification was given, and then a verification of the microcode was carried out [34].

These examples, as well as other examples (coming from different application areas, as for instance listed in [37]), show that this indeed is a typical pattern: a precise formal specification, often used to identify and clarify problems with informal design, offers a precise complete description of the systems (or their components) to be developed, a valuable asset on its own. Then formal methods and tools, often very specifically linked with the specification formalisms in use (with dependencies going in either direction) can be employed to support the development and verification of the software; see subsequent sections on model checking, verification, abstract interpretation, and the like. It is interesting that reports in [37] do not give a uniformly positive view of the benefits of the use of specification formalisms and related formal methods: while everybody agrees that their use results in an increased confidence and a higher quality of the product, their cost effectiveness is less clear. The major cost factor is, however, the high learning curve necessarily involved at the early stages of the project, when a (new) specification formalism is brought in and has to be learned to build the specifications; clearly, this overhead should decrease considerably with time when the use of a particular specification formalism is repeated and the use formal methods in general becomes an expected and required standard.

3.2 Verification Methods

We now describe briefly several major verification methods, abstract interpretation, model checking, equivalence checking, and verification by deduction. The properties to be proven are undecidable most of the time. In these cases, soundness—no false claim produced—and completeness—every property that holds is actually derived—can not be reached at the same time. The different methods deal with this in different ways. All are sound, will not derive correctness claims if, indeed, the system is faulty (*false positives*). They may be incomplete, producing warnings (*false negatives*) while, indeed, the system is correct, and/or they may require input from the user.

3.2.1 Abstract Interpretation

Abstract interpretation [12] computes an approximation of the program semantics. Alternatively, one might say that it computes *invariants*, properties that hold of every run of the system regardless of the environment. For example: At program line 17, variable `x` always has value 5, or the memory load at program point p is always a cache hit. As a signature we suggest

$$abstr\_int :: \mathsf{Sys} \rightarrow \mathsf{Spec}.$$

Abstract interpretation *per se* typically works on implementations without knowledge of input data and without specification. It infers properties that hold *for all possible executions*. In particular, let $abstr\_int(s) = \varphi$, and let $R = [\![\varphi]\!]$ be the set of runs denoted by φ. An abstract interpretation is called an *under-approximation*, if $R \subseteq [\![s]\!]$ and an *over-approximation* if $R \supseteq [\![s]\!]$. An abstract interpretation is always an over- or an under-approximation of the program semantics, $[\![s]\!]$. As an example, consider the safe classification of memory accesses as cache hits and cache misses as needed for a timing analysis of hard real-time programs described below. To classify memory accesses as cache hits one needs an under-approximation of the cache contents, while for cache misses one needs an over-approximation of the cache contents.

3.2.1.1 Derived Properties

Often, the specification (invariant) inferred by abstract interpretation is not the one one is actually interested in. Let φ_{spec} be the desired specification and let φ_{inv} be the inferred one. An abstract interpretation is *sound*, if $\varphi_{spec} \Rightarrow \varphi_{inv}$. In this case, one may obtain *false positives*, that is, indications that a specification is violated even though it is not. On the other hand, soundness allows the proof of absence of defects. If $\varphi_{inv} \Rightarrow \varphi_{spec}$ then the abstract interpretation is *complete*. If an error is found, it is definitely an error. Such a method is ideal for bug-hunting. On the other hand, a complete method might suffer from *false negatives*, that is, it might fail to uncover an error. While possible sound and complete abstract interpretations are possible, they are rare.

Often abstract interpretation is used with *implicit* specifications such as absence of bugs like division-by-zero, array-out-of-bounds-accesses, stack-overflow, null-pointer-dereferences. Typically, people deal with sound methods producing false positives rather than false negatives. On the other hand, abstract interpretation allows to *prove* the absence of certain defects making it attractive for certification of systems with respect to authority standards (e.g. in avionics).

3.2.1.2 Four Tools

Probably the largest industrial systems to which abstract interpretation was applied were safety-critical systems of the Airbus A380 plane. These systems consist of several hundred thousand lines of code. The four abstract interptretation based static analyzers described below, Polyspace Verifier, Astrée, Stackanalyzer, and aiT were able to analyze tasks of this size in times acceptable for the developers in the aeronautics and automotive industries.

Astrée is a sound static analyser for the programming language C designed to prove the absence of run-time errors such as division-by-zero, index-out-of-bounds, overflow and underflow, null, mis-aligned or dangling pointers [8]. Astrée was also designed to be complete, i.e., produce no false alarms, but only for the type of software found in critical real-time synchronous embedded control systems (e.g. synthesized from SCADE). Astrée[2] has been used with success in verifying aeronautic, aerospace, and automotive applications, such as electric flight control or space-vessels maneuvers, on programs up to 10^6 lines of code, without false alarms.

The sound static analysis tool with the largest user base is Polyspace Verifier[3]. It is in routine use in a decent set of development laboratories of the safety-critical embedded systems domain. Polyspace Verifier is based on abstract interpretation and analyzes programs written in C/C++ and Ada. Compared to Astrée it checks for absence of fewer errors and is less configurable. Its policy, “Green follows Orange” means that the analysis continues after a warning as if nothing happened. This means that several iterations are necessary to discover all problems.

Stackanalyzer[4] determines safe upper bounds on the size of system and user stacks. It determines the worst-case stack usage of the tasks in in the code under verification and displays the results as annotations of the call graph and control-flow graph.

Finally, aiT[5] determines safe upper bounds on the execution time of real-time programs [14]. Several different abstract interpretations are used, the most complex being the derivation of invariants about the set of all execution states of the execution platform. These invariants are used to bound the execution times of instructions. Depending on the complexity of the execution platform, aiT has shown an over-estimation of the execution times of between 8% for simple microcontrollers and 25% for complex high-performance mono-processors [35], while tasks of several million instructions can be analyzed within one day.

3.2.1.3 Roles in Abstract Interpretation

An abstract interpreter, as realized by a *tool developer* (based on foundations laid by the *researcher*) is able to analyze systems for a specific set of properties and nothing else. Any given tool is not universal, in contrast to underlying theory. Hopefully, the tool developer aims at the right set of properties for a relevant set of systems. The *user*, in principle, gets a push-button system. However, the analysis results may be much better if he/she gives a little help to the analyzer. This help may consist in configuring the system for the particular characteristics of the system, e.g. describing the ranges for environment variables and combining the right set of abstract domains for an embedded-control system in Astrée.

[2] http://www.absint.de/astree/
[3] http://www.mathworks.de/products/polyspace/
[4] http://www.absint.de/stackanalyzer/
[5] http://www.absint.de/ait/

It may also consist in supplying necessary properties about the execution platform, e.g. the type of used memory, or properties of the system under analysis, e.g. loop bounds, to aiT.

3.2.2 Model Checking

Model checking [11, 30] is understood by the following signature:

$$mc :: \mathsf{Model} \times \mathsf{Spec} \times \mathsf{Data} \rightarrow \mathbb{B} \times [\mathsf{Run}].$$

This means that given a model (or a system really) and a specification and input data, a model checker either provides a run to witness a possible error or indicates a successful check. Hence the signature has the "optional" result type [Run] with an extra bit representing satisfiable/unsatisfiable.

3.2.2.1 Model Checking in Industry

The most widespread use of model checking is in the semiconductor industry. Typical use cases were described by Cindy Eisner from IBM. A chip consists of several units. A *unit* is the smallest component of a processor architecture that has a functionality. A specification describing the functionality could be given for a unit. However, the state space to be exhaustively elaborated by a model checker currently is too large. Instead, *blocks*, parts of units, are checked. They may not have a specifiable functionality. So, only *local properties* are checked; for instance 14,000 local properties for the Pentium 4. These local properties express local correctness conditions.

Blocks have many different *environments* or *contexts*, in which they can be activated, in fact, too many to do this exhaustively. So, currently *blocks are checked for local properties in restricted sets of environments.*

Despite the dominance of model-checking use cases in hardware industry, there are examples from software and other industries as well. The *Static Driver Verifier Research Platform* [2] is a tool suite provided by Microsoft to verify Windows drivers. It is based on the software model checker SLAM.

3.2.2.2 Roles in Model Checking

Model checking has a different distribution of obligations from that of abstract interpretation. It places a higher burden on the user, who has to write a specification in the form of a finite-state machine or a temporal-logic formula, both not the native languages of most developers. In some cases, the user also has to supply an abstract model of the system, an often non-trivial task. This task may actually be alleviated by the recently begun cooperation of static analysts and model checkers. Abstraction of systems may be done based on the theory of abstract interpretation. The resulting abstract systems can then be model-checked.

3.2.3 Equivalence Checking

While model checking compares a model with its specification, equivalence checking compares two models. An adequate signature for equivalence checking also takes into account the provision of a counterexample in case that a difference between two systems is detected:

$$ec :: \mathsf{Spec}_1 \times \mathsf{Spec}_2 \rightarrow \mathbb{B} \times [\mathsf{Run}].$$

Spec_1 is typically referred to as the *Golden Implementation* and plays the role of a (formal) specification. Spec_2 is referred to as *Implementation*. Equivalence checking is mainly applied in the design of hardware systems, combinational or sequential circuits. The *Implementation* may be the result of adapting an existing design to a new semiconductor technology, performance optimization, compilation from register-transfer-level to gate-level, and the like. Very often, Spec_1 and Spec_2 share a lot of structural similarities, smoothing the way for the efficient application of graph-based data structures and algorithms (e.g., And-Inverter-Graphs or Binary Decision Diagrams) [22], as well as the application of SAT-solvers (for solving a satisfiability problem) [32] and ATPG-tools (for solving Automatic Test Pattern Generation problems), or combinations thereof [29].

3.2.4 Equivalence Checking in Industry

Industrial development processes of hardware designs routinely apply equivalence checking during the design process. Incremental and fine-grained design steps (as coarsely sketched in Fig. 1) ensure that the problem instances are manageable. Industrial applications of equivalence checking still require high level of user expertise for setting up the equivalence-checking framework, especially when implementations are delivered from outside customers or subcontractors.

3.2.5 Roles in Equivalence Checking

The roles in equivalence checking seem to be separated more clearly than for the other methods described in this section. The development team producing Spec_1 is typically different from the designers providing the implementation Spec_2. In the are of hardware development, the profession of a *Verification Engineer* was created. A verification engineer integrates Spec_1 and Spec_2 into the equivalence-checking framework while taking care of technological features, e.g., when some design features are deemed redundant with regard to the equivalence-checking task. Nevertheless, the verification engineer must be in close cooperation with the developers of the implementation, e.g., to get rid off *false-negative* counterexamples.

3.2.6 Verification by Deduction

Most abstractly, we describe verification by the signature

$$vbd :: \mathsf{Sys} \times \mathsf{Spec} \rightarrow \mathbb{B} \times [\mathsf{Proof}].$$

More precisely, the boolean result is either a proof that a system satisfies a given property; or a proof cannot be established. Logic is the lingua franca of *verification by deduction*. While logic is used in other approaches too (say model checking) it is really universal in verification. In interactive program verification, the user of a tool has to supply invariants at cutpoints of the program.

3.2.6.1 Academic and Industrial Practice

Verification of functional correctness by interactive theorem proving is standard practice for arithmetic units at processor manufacturers. This probably is the consequence of the Pentium bug [27, 13].

Proving the correctness of a compiler is, one could say, the "mother" of all software-verification attempts. The verification of the compiler guarantees that the safety properties proved on the source code hold for the executable compiled code as well. Xavier Leroy developed and formally verified, i.e., gave a proof of semantic preservation, of a compiler from Clight (a large subset of the C programming language) to PowerPC assembly code. He used the Coq proof assistant both for programming the compiler and for proving its correctness [23].

C.A.R. Hoare's vision of the *Verifying Compiler* led to the Verified Software Repository (VSR). This is an evolving collection of tools and challenges related to software verification. It supports a community effort to develop technology to enable the mechanical certification of computer programs [5].

The Verisoft project [1] was a research project funded by the German Federal Ministry of Education and Research (BMBF). The main goal of the project was the pervasive formal verification of computer systems. The correct functionality of systems, as they are used, for example, in the automotive domain, in security technology and in medical technology, was to be mathematically proved. The proofs are computer aided in order to prevent human error by the scientists involved.

Finally, the verification effort developed within Microsoft Research cannot go unmentioned. Microsoft makes use of its verification platform Boogie[6]. Specifically, Boogie is an intermediate language generated by a number of front-end tools for specific purposes and languages like Havoc (pointer verification in C) or Chalice (concurrent). Boogie is probably the first example of information-exchange between verification engines. It has a wide selection of provers at its disposal to verify that programs adhere to their specs. Examples include Z3, Simplify, or Isabelle/HOL.

4 Acceptance

This section addresses the acceptance of formal methods in industry. It is based on the experience and observations of the participants, some industrial, some academic.

Compelling Needs

Different application domains have different requirements for verification and, therefore, call for the application of formal methods at different degrees.

- In general-purpose computing, time-to-market may be decisive, such that the additional cost of applying formal methods may be considered inappropriate. Users may be willing to tolerate system failures once in a while.
- For safety-critical embedded systems, failure is not acceptable. However, it is unrealistic to assume that complex systems consisting of millions of lines of code could be produced free of bugs. High-integrity subsystems still may be required to be free of bugs. The application of formal methods is mandatory to achieve the highest possible quality.
- For high-security systems or system components with high-security requirements, the existence of security loopholes is not tolerated. This area has the interesting feature that bug chasing is done by an external community, whether the designer want this or not, namely the hackers.

This classification is reflected in a recent survey of applications of formal methods in industry, see [7]. It lists a large number of applications in the transportation sector where

[6] http://research.microsoft.com/en-us/projects/boogie/

safety-critical subsystems have been developed using formal methods, followed by a good number of applications in hardware design and several in the financial sector.

Motivating the Introduction of Formal Methods

Often, a formal method is introduced into industrial practice after a major desaster that it could have prevented from occurring. A premier example is the Pentium bug. Formal verification of the Pentium's floating-point arithmetic unit could have saved Intel half a billion Dollars. This experience boosted the application of formal verification in the hardware industry. Similarly, several failures in medical instruments have cost the producers high liability costs and led to the introduction of formal methods.

The Role of Champions

Participants from industry emphasized the importance of champions in industry being enthusiastic about and competent in a formal method. Without these, a formal method is seldom introduced into industrial practice. On the other hand, an introduced method and tool may fall again into oblivion once a champion leaves his or her position.

The Role of Education

It is decisive for the acceptance of a method and tool in industry that the competences required from an industrial user are available or can be taught without too much effort. Industrial participants emphasized that teaching student how to develop high-quality code is more important than teaching them formal methods. This goes in the same direction as our emphasis of the importance of the design for verifyability. Disciplined, well-structured code will increase the applicability of formal methods and will allow for a higher degree of automation. At the same time, it is important that students obtain adequate mathematical background, enbaling them later to master the usage of formal-methods tools.

4.1 A Spectrum of Formality

The term *verification* is heavily abused, as was said above. Every activity expected to lead to a better system quality is subsumed under it. To account for this, we describe a spectrum of methods, not all considered "formal" that are used in practice.

Testing: Testing is still very popular despite C.A.R. Hoare's statement that it can only prove the existence and not the absence of bugs. In the terminology of Section 3.2, it is unsound and imcomplete.

Unsound static analysis: This method may be very helpful in chasing bugs [4]. However, it is neither sound nor complete and therefore not suitable for verification.

Model-based design: The most heavily used modeling languages have a brittle semantics, in fact, different semantics defined by different code generators. Only the ones having a formally defined semantics, e.g. Scade, would be subsumed under formal methods.

From lite to rigorous: Dines Bjørner in [7] describes a spectrum from lite, to rigorous, to formal application. "Lite" means means specifying the problem and maybe the solution in a formal specification language. "Rigorous" means to specify additional properties and possibly the relation between different specifications. "Formal" requires proofs of specified propositions.

5 Challenges and Perspectives

Industry representatives applying formal methods issued the following wish list, mainly concerning static-analysis tools.

- Improvements in tool functionality: higher precision, i.e., less false alarms, support for functionality analysis, better configurability, a way to trade precision for performance, stronger automation, diagnosis of the error–not of the error symptom– and possibly examples exhibiting the symptom.
- Improvements in scalability: Coping with very large systems: full verification for smaller programs, defect localization for large programs.
- Process support: compliance with chracteristics of the development and the certification processes, iterative and incremental verification, exploitation of model information avaiable in model-based design of safety-critical software, support for code quality assessments.
- Tool cooperation: support for information exchange between tools exploiting synergy between tools.

Keep it simple, predictable, actionable

Quoting from Tom Ball's presentation at the workshop, we understand that tools implementing formal methods are still too hard to use. Simple counterexamples and simple proofs are needed, as are predictable behavior to avoid wild swings in performance for small changes. Tools should explain their failures so that users know what to do next. Type systems are a particularly successful example satisfying these requirements. Finally, we should learn to build on each other's work and stop reinventing the wheel!

Design for Verifiability

A very recent area of research emerging from formal methods is design for verifiability. A number of design decisions influence the possibility and if this is given, the ease of verifying properties of systems. Traditionally, this is applied in programming-language design. The programming language may have a strong influence on the possibility and the needed effort of system analysis and verification. It may enforce restrictions whose validation would otherwise require an enormous effort. For example, a major problem in the analysis of imperative programs is the determination of dependences between the statements in programs. The unrestricted use of pointers, as in the C programming language, makes this analysis of dependences very difficult due to the severe alias-problem created through pointers.

Several coding guidelines have been proposed to lead to more disciplined code. They typically restrict the use of the *dark corners* of the programming language, e.g. pointer arithmetic and function pointers. One prominent example is MISRA C [25], the C coding standard proposed by the Motor Industry Software Reliability Association. It bans the worst features of C with respect to the software verification task. However, this does not necessarily lead to programs whose timing behavior is precisely predictable [15].

The execution platform determines the analyzability of the timing behavior [36]. Most emerging multi-core platforms will make timing analysis infeasible due to the interference of threads on shared resources.

The transition from federated architectures—one computer per function—to integrated architectures—several functions integrated on one platform—as currently under way in the

Integrated Modular Avionics (IMA) and the AUTomotive Open System Architecture (AUTOSAR) standardization efforts offers a great chance to improve the verifiability of systems. Temporal and spatial partitioning is used in IMA to avoid the logical interference of functions. However, the existing implementations give away the chance to cleanly and efficiently deal with the interaction on shared resources and the resulting non-composability of the resource behavior. In the ideal case, *design meets verification*, that is, design only admits systems that can be easily verified.

Limitations

However close we get to modelling real systems, we will always talk about models abstracting from some details. The same goes for specifications. Details left unmodeled and/or unspecified cannot be verified, obviously, and remain a fundamental limitation.

Acknowledgements

We thank Marc Herbstritt for supporting us during the meeting by taking minutes which built a basis for this manifesto and for providing a draft on equivalence checking.

6 Participants

- Krzysztof Apt
CWI – Amsterdam, NL
- Thomas Ball
Microsoft Res. – Redmond, US
- Dines Bjørner
Holte, DK
- Patrick Cousot
ENS – Paris, FR
- Cindy Eisner
IBM – Haifa, IL
- Javier Esparza
TU München, DE
- Steffen Görzig
Daimler AG – Böblingen, DE
- Yuri Gurevich
Microsoft Res. – Redmond, US
- Marc Herbstritt
Schloss Dagstuhl, DE
- Manuel Hermenegildo
IMDEA Software – Madrid, ES
- Bengt Jonsson
University of Uppsala, SE
- Joseph Roland Kiniry
IT Univ. of Copenhagen, DK
- Jörg Kreiker
TU München, DE
- Wei Li
Beihang University, CN
- Wolfgang J. Paul
Universität des Saarlandes, DE
- Erik Poll
Radboud Univ. Nijmegen, NL
- Sriram K. Rajamani
Microsoft Research India – Bangalore, IN
- Jean-Francois Raskin
Univ. Libre de Bruxelles, BE
- John Rushby
SRI – Menlo Park, US
- Donald Sannella
University of Edinburgh, GB
- Wei Sun
Beihang University, CN
- Andrzej Tarlecki
University of Warsaw, PL
- Wolfgang Thomas
RWTH Aachen, DE
- Moshe Y. Vardi
Rice University, US
- Reinhard Wilhelm
Universität des Saarlandes, DE
- Jim C.P. Woodcock
University of York, GB
- Lenore Zuck
NSF – Arlington, US

References

1 Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert W. Schirmer, and Artem Starostin. The Verisoft approach to systems verification. In Natarajan Shankar and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments Second International Conference, VSTTE 2008, Toronto, Canada, October 6–9, 2008. Proceedings*, volume 5295 of *Lecture Notes in Computer Science*, pages 209–224, Toronto, Canada, October 2008. Springer.

2 Thomas Ball, Ella Bounimova, Vladimir Levin, Rahul Kumar, and Jakob Lichtenberg. The static driver verifier research platform. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 119–122. Springer, 2010.

3 Geoff Barrett. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering*, 15(5):611–621, 1989.

4 Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, 2010.

5 Juan Bicarregui, C. A. R. Hoare, and J. C. P. Woodcock. The verified software repository: a step towards the verifying compiler. *Formal Asp. Comput.*, 18(2):143–151, 2006.

6 Michel Bidoit and Peter D. Mosses, editors. *CASL User Manual.* Number 2900 in Lecture Notes in Computer Science. Springer, 2004.

7 Dines Bjøner. A spring 2011 survey of applications of formal methods in industry, a draft version. http://www2.imm.dtu.dk/ db/fm-industry-project-survey/fm-applics-survey.pdf, 2011.

8 Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207. ACM, 2003.

9 L. Burdy, Y. Cheon, D.C. Cok, M.R. Ernst, J.R. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, 2005.

10 R.M. Burstall and J.A. Goguen. An informal introduction to specifications using Clear. In R.S. Boyer and J.S. Moore, editors, *The Correctness Problem in Computer Science*, pages 185–213. Academic Press, 1981. Also in: *Software Specification Techniques* (eds. N. Gehani and A.D. McGettrick), Addison-Wesley, 1986.

11 Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

12 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

13 Alan Edelman. The mathematics of the Pentium division bug. *SIAM Rev.*, 39:54–67, 1997.

14 Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise WCET determination for a real-life processor. In *EMSOFT*, volume 2211 of *LNCS*, pages 469–485. Springer, 2001.

15 Gernot Gebhard, Christoph Cullmann, and Reinhold Heckmann. Software structure and WCET predictability. In *Proc. of the workshop Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, Grenoble, 2011.

16 Joseph Goguen, James Thatcher, and Eric Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. Technical Report RC 6487,

IBM Watson Research Center, Yorktown Heights NY, 1976. Also in: *Current Trends in Programming Methodology. Volume IV (Data Structuring)* (ed. R.T. Yeh), Prentice-Hall, 80–149, 1978.

17 The RAISE Language Group. *The RAISE Specification Language.* The BCS Practitioners Series. Prentice-Hall, 1992.

18 John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer, 1993.

19 C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the Association for Computing Machinery*, 12:576–580,583, 1969.

20 Cliff B. Jones. *Software Development: A Rigorous Approach.* Prentice-Hall, 1980.

21 Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The definition of Extended ML: A gentle introduction. *Theoretical Computer Science*, 173:445–484, 1997.

22 Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay K. Ganai. Robust boolean reasoning for equivalence checking and functional property verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 21(12):1377–1394, 2002.

23 Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

24 Bertrand Meyer. *Eiffel : The Language.* Prentice-Hall, 1991.

25 MISRA. *Guidelines for the Use of the C Language in Critical Systems*, October 2004. ISBN 0 9524156 2 3.

26 Peter D. Mosses, editor. *CASL Reference Manual.* Number 2960 in Lecture Notes in Computer Science. Springer, 2004.

27 Thomas R. Nicely. The Pentium division flaw. *Virginia Scientists Newsletter*, 1:3, 1995.

28 Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Automated Deduction — CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings*, volume 607 of *LNCS*, pages 748–752. Springer, 1992.

29 Viresh Paruthi and Andreas Kuehlmann. Equivalence checking combining a structural sat-solver, bdds, and simulation. In *Proc. IEEE Int'l Conference On Computer Design (ICCD), Austin, USA*, pages 459–464, 2000.

30 Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in Cesar. In *Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.

31 Donald Sannella and Andrzej Tarlecki. Program specification and development in Standard ML. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pages 67–77, 1985.

32 João P. Marques Silva and Karem A. Sakallah. Boolean satisfiability in electronic design automation. In *DAC*, pages 675–680, 2000.

33 J. Michael Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, second edition, 1992.

34 Mandayam K. Srivas and Steven P. Miller. Applying formal verification to the AAMP5 microprocessor: A case study in the industrial use of formal methods. *Formal Methods in System Design*, 8(2):153–188, 1996.

35 Lili Tan. The worst-case execution time tool challenge 2006. *STTT*, 11(2):133–152, 2009.

36 Lothar Thiele and Reinhard Wilhelm. Design for timing predictability. *Real-Time Systems*, 28(2-3):157–177, 2004.

37 Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John S. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4), 2009.