ZESZYTY NAUKOWE POLITECHNIKI BIAŁOSTOCKIEJ 2009
Informatyka – Zeszyt 4

Kazimierz Trzęsicki[1]

LOGIC IN FORMAL VERIFICATION OF COMPUTER SYSTEMS. SOME REMARKS.

Abstract: Various logics are applied to specification and verification of both hardware and software systems. The problem with finding of proof is the most important disadvantage of proof-theoretical method. The proof-theoretical method presupposes the axiomatization of the logic. Proprieties of a system can also be checked using a model of the system. A model is constructed with the specification language and checked using automatic model checkers. The model checking application presupposes the decidability of the task.

Keywords: Logic, Verification, Proof-theoretical Method, Model Checking

1. Logic in Computer Science

Connections between logic and computer science (*CS*) are wide-spread and varied. Notions and methods from logic can fruitfully be applied within *CS*. Logic plays the same role in *CS* as the calculus plays in physics. Logic is „the calculus of computer science" [74,16,29].

On the one hand, logic permeates more and more its main areas. On the other hand we may notice that [59, p. 181]:

> Until the invention of the digital computer, there were few applications of formal mathematical logic outside the study of logic itself. In particular, while many logicians investigated alternative proof systems, studied the power of various logics, and formalized the foundations of mathematics, few people used formal logic and formal proofs to analyze the properties of other systems. The lack of applications can be attributed to two considerations: (i) the very formality of formal logic detracts from its clarity as a tool of communication and understanding, and (ii) the "natural" applications of mathematical logic in the pre-digital world were in pure mathematics and there was little interest in the added value of formalization. Both of these considerations

[1] University of Bialystok

> changed with the invention of the digital computer. The tedious and precise manipulation of formulas in a formal syntax can be carried out by software operating under the guidance of a user who is generally concerned more with the strategic direction of the proof.

The logical methods are applicable for the design, specification, verification[2] and optimization of programs, program systems and circuits. Logic has a significant role in computer programming. While the connections between modal logic[3] and *CS* may be viewed as nothing more than specific instances, there is something special to them.

In 1974 the British computer scientist Rod M. Burstall first remarked on the possibility of application of modal logic to solve problems of *CS*. The Dynamic Logic of Programs has been invented by Vaughan R. Pratt [81]:

> In the spring of 1974 I was teaching a class on the semantics and axiomatics of programming languages. At the suggestion of one of the students, R. Moore, I considered applying modal logic to a formal treatment of a construct due to C. A. R. Hoare, "$p\{a\}q$", which expresses the notion that if p holds before executing program a, then q holds afterwards. Although I was skeptical at first, a weekend with Hughes and Cresswell[4] convinced me that a most harmonious union between modal logic and programs was possible. The union promised

[2] Some authors distinguish between ***V**alidation* and ***V**erification* and refer to the overall checking process as V&V. Validation is answering to the question: *Are we trying to make the right thing?*. Verification answer the question: *Have we made what we were trying to make?* In general methodology of sciences the term "verification" denotes establishing correctness. The term "falsification" (or "refutation") is used in meaning: to detect an error. In *CS* "verification" covers both the meanings and refers to the two-sided process of determining whether the system is correct or erroneous.

For Dijkstra [33] the verification problem is distinct from the pleasantness problem which concerns having a specification capturing a system that is truly needed and wanted. Emerson observes that [36, p. 28]:

> The pleasantness problem is inherently pre-formal. Nonetheless, it has been found that carefully writing a formal specification (which may be the conjunction of many sub-specifications) is an excellent way to illuminate the murk associated with the pleasantness problem.

[3] The traditional modal logic deals with three 'modes' or 'moods' or 'modalities' of the copula 'to be', namely, *possibility*, *impossibility*, and *necessity*. Related terms, such as *eventually*, *formerly*, *can*, *could*, *might*, *may*, *must*, are treated in a similar way, hence by extension, logics that deals with these terms are also called modal logics.

The basic modal operator $\Box$ (necessarily) is not rigidly defined. Different logics are obtained form different definition of it. Here we are interested in temporal logic that is the modal logic of temporal modalities such as: *always, eventually*.

[4] The book Pratt is talking about is *An Introduction to Modal Logic* [55].

> to be of interest to computer scientists because of the power and mathematical elegance of the treatment. It also seemed likely to interest modal logicians because it made a well-motivated and potentially very fruitful connection between modal logic and Tarski's calculus of binary relations.

This approach was a substantial improvement over the existing approach based on the pre-condition/post-condition mechanism provided by Hoare's logic.[5] Kripke models, the standard semantic structure on which modal languages are interpreted, are nothing but graphs. Graphs are ubiquitous in *CS*.

The connection between the possible worlds of the logician and the internal states of a computer is easily described. In possible world semantics, ϕ is possible in some world w if and only if ϕ is true in some world w' accessible to w. Depending on the properties of the accessibility relation (reflexive, symmetric, and so on), there will be different theorems about possibility and necessity. The accessibility relation of modal logic semantics can thus be understood as the relation between states of a computer under the control of a program such that, beginning in one state, the machine will (in a finite time) be in one of the accessible states. In some programs, for instance, one cannot return from one state to an earlier state; hence state accessibility here is not symmetric.

The question of using of temporal logic (TL) to software engineering was undertaken by Kröger [61,62,63,64]. The development of TL as applied to *CS* is due to Amir Pnueli. He was inspired by „Temporal Logic", a book written by Rescher and Urquhart [84].[6] „The Temporal Logic of Programs" [79], a paper by Pnueli,[7] is the classical source of TL for specification and verification of programs. This work is commonly seen as a crucial turning point in the progress of formal methods for the verification of concurrent and reactive systems. Amir Pnueli argues that temporal logic can be used as a formalism to reason about the behavior of computer programs and, in particular, of non-terminating concurrent systems.[8] In general, properties are

[5] Hoare's logic views a program as a transformation from an initial state to a final state. Thus it is not eligible to tackle problems of reactive or non-terminating systems, such as operating systems, where the computation does not bring to a final state.

[6] See [42, p. 222].

[7] Pnueli received the Turing Award in 1996:

> for seminal work introducing temporal logic into computing science and for outstanding contributions to program and system verification.

[8] A system is said to be concurrent when its behavior is the result of the interaction and evolution of multiple computing agents. The initial interest in concurrent systems was motivated by the speed improvements brought forth by multi-processor computers.

mostly describing correctness or safety of the system's operation. For Clarke [20, p. 1] works of Pnueli [79], Owicki and Lamport [78]:

> demonstrated convincingly that Temporal Logic was ideal for expressing concepts like mutual exclusion, absence of deadlock, and absence of starvation.

There is a difference between logician and computer scientists approach to systems of logics [14, p. 315]:

> Decidability and axiomatization are standard questions for logicians; but for practitioner, the important question is model-checking.

In opinion of Dijkstra:[9]

> The situation of programmer is similar to the situation of mathematician, who develops a theory and proves results. [...] One can never guarantee that a proof is correct, the best one can say, is: "I have not discovered any mistakes". [...] So extremely plausible, that the analogy may serve as a great source of inspiration. [...]
> Even under the assumption of flawlessly working machines we should ask ourselves the questions: "When an automatic computer produces results, why do we trust them, if we do so?" and after that; "What measures can we take to increase our confidence that the results produced are indeed the results intended?"

In another work [32, p. 6] Dijkstra says:

> Program testing can be used to show the presence of bugs, but never to show their absence.

Formulated in terms of Turing Machines, the verification problem was already considered by Turing [88]. He demonstrated that there is no general method of proving of correctness of any program.

Application of a computer system may cause not only material losses, e.g., in e-banking, but also may be dangerous for life, e.g., in health care, transportation, especially air and space flights.[10] Correctness of design is a very important factor of

[9] See Dijkstra E. W., *Programming Considered as a Human Activity*, `http://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD117.html`.

[10] A famous example: The Ariane-5 launch on June 4, 1996; it crashed 36 seconds after the launch due to a conversion of a 64-bit floating point into a 16-bit integer value. In 2008 it was announced that the Royal Navy was ahead of schedule for switching their nuclear submarines to a customized Microsoft Windows solution dubbed *Submarine Command System Next Generation*. In this case any error may have an unimaginable aftermath.

systems for preventing economical and human losses caused by minor errors. The reduction of errors in computer systems is one of the most important challenges of *CS* [64, p. V]. It has long been known that [36, p. 27]:

> computer software programs, computer hardware designs, and computer systems in general exhibit errors. Working programmers may devote more than half of their time on testing and debugging in order to increase reliability. A great deal of research effort has been and is devoted to developing improved testing methods. Testing successfully identifies many significant errors. Yet, serious errors still afflict many computer systems including systems that are safety critical, mission critical, or economically vital. The US National Institute of Standards and Technology has estimated that programming errors cost the US economy $60B annually[11].

Computer systems are more and more complicated. Verification of digital hardware designs has become one of the most expensive and time-consuming components of the current product development cycle. Empirical testing and simulation is expensive, not ultimately decisive and sometimes excluded for economical or ethical reasons. Formal methods are the most notable efforts to guarantee a correctness of system design and behaviors. Thus the formal specification and computer aided validation and verification are more and more indispensable. Formal methods have gained popularity in industry since the advent of the famous Intel Pentium *FDIV* bug in 1994, which caused *Intel* to recall faulty chips and take a loss of $475 million [28]. Digital computers are intended to be abstract discrete state machines and such machines and their software are naturally formalized in mathematical logic.

> Given the formal descriptions of such systems, it is then natural to reason about the systems by formal means. And with the aid of software to take care of the myriad details, the approach can be made practical. Indeed, given the cost of bugs and the complexity of modern hardware and software, these applications cry out for mechanical analysis by formal mathematical means. [59, p. 181–182]

Errors should already be detected at design stage. It is very important to specify the correctness property of system design and behavior, and an appropriate property must be specified to represent a correct requirement. It is estimated that 70% of design-time is spent to minimize the risk of errors [86], see [76]. Formal methods, model checkers as well theorem provers, are proposed as efficient, safe and less expensive tools [59,15]. According to Emerson [36, pp. 27–28]:

[11] See: National Institute of Standards and Technology, US Department of Commerce, "Software Errors Cost U.S. Economy $59.5 Billion Annually", NIST News Release, June 28, 2002.

> Given the incomplete coverage of testing, alternative approaches have been sought. The most promising approach depends on the fact that programs and more generally computer systems may be viewed as mathematical objects with behavior that is in principle well-determined. This makes it possible to specify using mathematical logic what constitutes the intended (correct) behavior. Then one can try to give a formal proof or otherwise establish that the program meets its specification. This line of study has been active for about four decades now. It is often referred to as *formal methods*.

2. Formal methods of verification

2.1 Formal methods

Formal methods include: formal specification, specification analysis and proof, transformational development, program verification. The principal benefits of formal methods are in reducing the number of faults in systems. Consequently, their main area of applicability is in critical systems engineering. There have been several successful projects where formal methods have been used in this area. The use of formal methods is most likely to be cost-effective because high system failure costs must be avoided. Nevertheless formal methods have not become mainstream software development techniques as was once predicted. Other software engineering techniques have been successful at increasing system quality. Hence the need for formal methods has been reduced. Market changes have made time-to-market rather than software with a low error count the key factor. Formal methods do not reduce time to market. Moreover, the scope of formal methods is limited. They are not well-suited to specifying and analyzing user interfaces and user interaction. Formal methods are still hard to scale up to large systems. Nevertheless as it is stressed by Edmund M. Clarke [56, p. ix], one of the prominent researcher in the field of formal methods in *CS*:

> Formal methods have finally come of age! Specification languages, theorem provers, and models checkers are beginning to be used routinely in industry.

The formal methods to be appropriate need to be properly adapted. Temporal logic and its language are of particular interest in the case of reactive[12], in particular concurrent systems. The language of TL is one that fulfills three important criteria. It:

[12] Systems can be divided into two categories: transformational programs (data intensive) and reactive systems (control intensive). The systems of the second type maintain an ongoing interaction with their environment (external and/or internal stimuli) and which ideally never terminate. Their specifications are typically expressed as constraints on their behavior over time.

– has the ability to express all sorts of specification (expressiveness);
– has reasonable complexity to evaluate the specified rules (complexity);
– due to its resemblance to natural language is easy to learn (pragmatics).

The knowledge of TL is indispensable in practice, tough, as it is remarked by Schnoebelen [87]:

> In today's curricula, thousands of programmers first learn about temporal logic in a course on model checking!

TL languages can be used to specification of widely spectrum of systems. Methods of TL can be applied to verification [72]. In the case of reactive systems TL is more useful than Floyd-Hoare logic that is better in the case of "input-output" programs. TL languages [64, p. 181]:

> provide general linguistic and deductive frameworks for *state systems* in the same manner as classical logics do for mathematical systems.

There are two main methods: proof-theoretical and model-theoretical [26].

2.2 Proof-theoretical approach

Already in the works of Turing the mathematical methods were applied to check correctness of programs [83]. By the end of sixties of last century Floyd [37], Hoare [48] and Naur [77] proposed axiomatic proving sequential programs with respect to their specification. Proof-theoretical method based on TL was proposed by Pnueli and Manna [72].

This method is used to prove a correctness of system through logical proving about system constraints or requirement for safe system behavior. Propositions specifying the system are joined as premisses to the thesis of deduction system of logic. Proofs can be "described" a variety of ways, e.g., by giving the inference steps, by specifying tactics or strategies to try, by stating the "landmark" subgoals or lemmas to establish, etc. Often, combinations of these styles are used within a single large proof project. Verification is positive if the proposition expressing the desired property is proved by using formal axioms and inference rules oriented towards sequential programs. Correctness of formal derivations could be "mechanically" checked, but finding a proof needs some experience and insight.

> But all proofs of commercially interesting theorems completed with mechanical theorem proving systems have one thing in common: they require a great deal of user expertise and effort. [59, pp. 182–183]

For example [59, p. 182]:

> The proof, constructed under the direction of this paper's authors and Tom Lynch, a member of the design team for the floating point unit, was completed 9 weeks after the effort commenced. About 1200 definitions and theorems were written by the authors and accepted, after appropriate proofs were completed by the *ACL2*[13] theorem prover.

At the time of its introduction in the early 1980's, a "manual" proof-theoretic approach was a prevailing paradigm for verification. Nowadays proofs are supported by semi-automatic means[14], *provers* and *proof checkers*. Interactive provers are used to partially automate the process of proving. Among the mechanical theorem proving systems used to prove commercially interesting theorems about hardware designs are *ACL2*[15], *Coq*[16], *HOL*[17], *HOL* Light[18], Isabelle[19], and *PVS*[20]. The proof assistant approach is a subject of research projects, e.g. *BRICKS* `http://www.bsik-bricks.nl/research_projects_afm4.shtml`.

The proof-theoretic framework is one-sided. It is possible only to prove that a proposition is a thesis. If we do not have a proof, we are entitled only to say that we could not find a proof, and nothing more. However, theorem proving can deal with an infinite state space, i.e., system with infinitely many configurations. Nevertheless this method is also indispensable in some intractable cases of finite state systems. Though today's model checkers are able to handle very large state spaces, eg. 10^{120} [[59, p. 183], [25]] but it does not mean that these states are explored explicitly. The above discussed theorem about *FDIV* (see p. 157) could be checked by running the microcode on about 10^{30} examples. Since in this case there are no reduction techniques, if it is assumed that one example could be checked in one femtosecond (10^{-15} seconds — the cycle time of a petahertz processor), the checking of the theorem will take more than 10^7 years [59, p. 183].

For Emerson [36, p. 28]:

> The need to encompass concurrent programs, and the desire to avoid the difficulties with manual deductive proofs, motivated the development of model

[13] See [71,12].

[14] Until the artificial intelligence problem is solved, human interaction will be important in theorem proving.

[15] See `http://www.cs.utexas.edu/~moore/acl2/`, [59].

[16] See `http://coq.inria.fr/`.

[17] See `http://www.cl.cam.ac.uk/research/hvg/HOL/`, [39].

[18] See `http://www.cl.cam.ac.uk/~jrh13/hol-light/`.

[19] See `http://www.cl.cam.ac.uk/research/hvg/Isabelle/`.

[20] See `http://pvs.csl.sri.com/`.

checking. In my experience, constructing proofs was sufficiently difficult that it did seem there ought to be an easier alternative.

2.3 Model-theoretical approach

Both the idea of automatic verification of concurrent programs based on model-theoretic approach and the term "model checking" were introduced by Clarke and Emerson in [21],[21] and independently the idea of model checking was conceived by Quille and Sifakis [82].[22] The idea was developed in works by Clarke, Emerson, Sistla and other [22,23,24,11,27].

Model checking is a verification technique that is preferred to theorem proving technique. This method, similarly as it is in the case of logical calculi, is more effective comparatively to proof-theoretic method. It is one of the most active research areas because its procedures are automatic and easy to understand.

According to Edmund M. Clarke [20, p. 1]:

> Model Checking did not arise in a historical vacuum. There was an important problem that needed to be solved, namely concurrent program verification.

In another place he continues:[23]

> Existing techniques for solving the problem were based on manual proof construction from program axioms. They did not scale to examples longer than a page and were extremely tedious to use. By 1981 the time was ripe for a new approach to the problem, and most of necessary ideas were already in place.

Model checking bridges the gap between theoretical computer science and hardware and software engineering. Model checking does not exclude the use of proof-theoretical methods, and conversely, the proof-theoretical methods do not exclude using of model checking. In practice one of theses methods is complementary to the other at least at the heuristic level. On the one hand, failed proofs can guide to the discovery of counterexamples. Any attempt of proving may be forego by looking for counterexamples. Counterexamples of consequences of a theorem can help to reformulate it. Examples may aid comprehension and invention of ideas and can be used as a basis for generalization being expressed by a theorem. The role of decision procedures is often essential in theorem proving. There has been considerable interest in

[21] See [36, p. 9].

[22] E. M. Clarke and E. A. Emerson interpreted concurrent system as finite Kripke structure/transition system and properties were expressed in *CTL* language. J.-P. Queille and J. Sifakis based on Petri nets and properties were expressed in language of branching time logic.

[23] See `http://events.berkeley.edu/index.php/calendar/sn/coe.html?event`.

developing theorem provers that integrate *SAT* solving algorithms. The efficient and flexible incorporating of decision procedures into theorem provers is very important for their successful use. There are several approaches for combining and augmenting of decision procedures. On the other hand, the combination of model checking with deductive methods allows the verification of a broad class of systems and, as it is in the case of eg. *STeP* [73], not restricted to finite-state systems. The question of combining proof-theoretical and model checking methods and the general problem of how to flexibly integrate decision procedures into heuristic theorem provers are subjects of many works [13].

In model checking the first task is to convert a system to a formal model accepted by a model checker. We model a system as a finite state machine. It is a model in the form of a Kripke structure[24] or labeled graph of state transitions — that has to accurately describe the behavior of the checked system. To do this formal languages defined by formal semantics must be used. To draw an abstract model many techniques are applied. Many methods are used to reduce states of a system. In practice, this process is not automated.

The second task is to specify properties that that must be satisfied by the real system. Mechanically assisted verification of properties of a complex system requires an accurate formal model of the system. The specification usually is given in some logical formalism. Generally, temporal logics are used to represent a temporal characteristic of systems.

We perform a model checker whether the system satisfies its properties as expressed by temporal logic formulas. The answer is positive only if all runs are models of the given temporal logic formula. The technique is based on the idea of exhaustive exploration of the reachable state space of a system. For this reason it can only be applied to systems with a finite state space, i.e., systems with finitely many configurations, and — for practical limitations (tractability) — with not too many states. The verification is completely automatic with the abstract model and properties. Thus it is possible to verify the correctness of very complicated and very large systems manual checking of which is almost not possible. We can verify a complex system as a hardware circuit or communication protocol automatically. The verification re-

[24] Kripke or relational semantics of modal logics has been conceived in fifties of the last century. This semantics was philosophically inspired nevertheless it has found application in *CS*. In *CS* Kripke structure is associated with a transition system. Because of the graphical nature of the state-space, it is sometimes referred to as the state graph associated with the system. Similarly as in modal logics this role may be played by Hintikka frames [8]. A Kripke frame consists of non-empty set and a binary relation defined on this set. In modal logics elements of the set are called possible worlds and the relation is understood as accessibility of one world from another. In the case of TL as applied in *CS* the Kripke semantics is based on computational time.

sults are correct and easy to analysis. However, it does need human assistance to analyze the result of model checking. If logic is complete with respect to the model and is decidable, then in the case of any proposition that specifies the behavior of the system the procedure of checking is finite. But if the model is too detailed the verification becomes intractable. A model checker verifies the model and generates verification results, "True" or counterexample if the result is "False". If the proposition is satisfied the system is verified. If the proposition is not valid the construction results in a counterexample — this is one of important advantages of model checking. The counterexample provides an information about an error (bug) in the system. The model checker can produce a counterexample for the checked property, and it can help the designer in tracking down where the error occurred.

The counterexample gives us a new precondition or a negative result in the following way: When we obtain a counterexample, we analyze it and as far as this trace could not occur in real system we add new preconditions to the formula. We may obtain a counterexample again which often results to many preconditions. In this case, analyzing the error trace may require a modification to the system and reapplication of the model checking process. The error can also result from incorrect modeling of the system or from an incorrect specification. The error trace can also be useful in identifying and fixing these two problems.

Model checking comes in two varieties depending on the way the proprieties are expressed. If theory of automata is employed the system as well as its specification are described by automaton. Questions concerning system and its specification are reduced to the question about the behavior of automaton. In other words, when we say "automata theoretic approach" we mean:

- specifying systems using automata
- reducing model checking to automata theory.

In the case of TL model checking the system is modeled as a finite-state automaton, while the specification is described in temporal language. A model checking algorithm is used to verify whether the automaton has the proper temporal-logical proprieties. In other words, if TL is applied [76, p. 2–3]:

> Model checking involves checking the truth of a set of specifications defined using a temporal logic. Generally, the temporal logic that is used is either CTL^* or one of its sublogics, CTL [...] [23] or LTL [...] [80].

Various model checkers are developed. They are applied to verification of large models, to real-time systems, probabilistic systems, etc. [50,66,24,10] — see [87].

Software is usually less structured than hardware and, especially in the case of concurrency, asynchronous. Thus the state space is bigger in the case of software than in hardware. Consequently, Model Checking has been used less frequently for software verification than for hardware verification [20, p. 18]. The limits of models checking are pushed by employing work-station clusters and *GRID*s, e.g. the *VeriGEM* project aims at using the storage and processing capacity of clusters of workstations on a nation-wide scale `www.bsik-bricks.nl/research_projects_afm6.shtml`. Despite being hampered by state explosion, since its beginning model checking has had a substantive impact on program verification efforts.

It is worth mentioning some of the applications of model checking elsewhere. These include understanding and analyzing legal contracts, which are after all prescriptions for behavior [31]; analyzing processes in living organisms for systems biology [43]; e-business processes such as accounting and workflow systems [91]. Model checking has also been employed for tasks in artificial intelligence such as planning [38]. Conversely, techniques from artificial intelligence related to SAT-based planning [60] are relevant to (bounded) model checking.

Let us repeat after Emerson some interesting remarks concerning model checking [36, p. 42]:

> Edsger W. Dijkstra commented to me that it was an "acceptable crutch" if one was going to do after-the-fact verification. When I had the pleasure of meeting Saul Kripke and explaining model checking over Kripke structures to him, he commented that he never thought of that. Daniel Jackson has remarked that model checking has "saved the reputation" of formal methods.

3. Model checkers

By a model checker we mean a procedure which checks if a transition system system is a model for a formula expressing a certain property of this system [23].

There is a wide variety of model checkers available, with a number of different capabilities suited to different kinds of problems. Some of these are academic tools, others are industrial internal tools, and some are for sale by *CAD* vendors. The variety is of great benefit to practitioners. They have to know which tools are available and which tools to chose for a particular problem. Today, software, hardware and *CAD* companies employ several kinds of model checkers. In software, *Bell Labs*, *JPL*, and *Microsoft*, government agencies such as *NASA* in USA, in hardware and *CAD*, *IBM*, *Intel* (to name a few) have had tremendous success using model checking for verifying switch software, flight control software, and device drivers.

Some programs are grouped as it is in the case of MODEL-CHECKING KIT `http://www.fmi.uni-stuttgart.de/szs/tools/mckit/overview.shtml`. This is a collection of programs which allow to model a finite-state system using a variety of modeling languages, and verify it using a variety of checkers, including deadlock-checkers, reachability-checkers, and model-checkers for the temporal logics *CTL* and *LTL*. The most interesting feature of the Kit is that:

> Independently of the description language chosen by the user, (almost) all checkers can be applied to the same model.

The counterexamples produced by the checker are presented to the user in terms of the description language used to model the system.

The Kit is an open system: new description languages and checkers can be added to it.

The description languages and the checkers have been provided by research groups at the *Carnegie-Mellon University*, the *University of Newcastle upon Tyne*, *Helsinki University of Technology*, *Bell Labs*, the *Brandenburg Technical University at Cottbus*, the *Technical University of Munich*, the *University of Stuttgart*, and the *Humboldt-Universität zu Berlin*.

Problems of techniques and tools of verification of *ICT* systems are subjects of research projects. E.g., in the scheme of *BRICKS* `http://www.bsik-bricks.nl/index.shtml` under theme *Algorithms and Formal Methods* there are developed

- Advancing the Real Use of Proof Assistants
- Infinite Objects: Computation, Modeling and Reasoning
- A Verification Grid for Enhanced Model Checking
- Modeling and Analysis of QoS for Component-Based Designs
- A Common Framework for the Analysis of Reactive and Timed Systems

Many of the research problems originating from industrial parties.

Below we give a few examples of model checkers. Usually they description will be taken from they website home pages.

Two of the most popular on-the-fly, explicit-state-based model checkers are SPIN (**S**imple **P**romela **IN**terpreter) and MURϕ or MURPHI [35,34].

SPIN is:

> a popular open-source software tool, used by thousands of people worldwide, that can be used for the formal verification of distributed software systems. The tool was developed at *Bell Labs* in the original UNIX group of the Computing Sciences Research Center, starting in 1980. The software has been

available freely since 1991, and continues to evolve to keep pace with new developments in the field. In April 2002 the tool was awarded the prestigious *System Software Award* for 2001 by the *ACM*. http://spinroot.com/spin/whatispin.html

SPIN continues to evolve to keep pace with new developments in the field. The DSPIN tool [57] is an extension of SPIN, which has been designed for modeling and verifying object-oriented software (JAVA programs, in particular).

Murϕ is a system description high-level language and model checker developed to formally evaluate behavioral requirements for finite-state asynchronous concurrent systems [35,34], http://sprout.stanford.edu/dill/murphi.html. *Mur*ϕ is developed by a research group at the University of Utah http://www.cs.utah.edu/formal_verification/.

SMV http://www.cs.cmu.edu/~modelcheck/smv.html (**S**ymbolic **m**odel **v**erifier) is a model checker that accepts both the temporal logics *LTL* and *CTL*. It is the first and the most successful *OBDD*-based symbolic model checker [75]. SMV has been developed by The Model Checking Group that is a part of Specification and Verification Center, Carnegie Mellon University http://www-2.cs.cmu.edu/~modelcheck/index.html.

CADENCE SMV http://www.kenmcmil.com/smv.html is a symbolic model checking tool released by Cadence Berkeley Labs. CADENCE SMV is provided for formal verification of temporal logic properties of finite state systems, such as computer hardware designs. It is an extension of SMV. It has a more expressive mode description language, and also supports synthesizable VERILOG as a modeling language.

NUSMV http://nusmv.irst.itc.it, http://nusmv.fbk.eu is an updated version of SMV [18,17]. The additional features contained in NUSMV include a textual interaction shell and graphical interface, extended model partitioning techniques, and facilities for *LTL* model checking. NUSMV [19] has been developed as a joint project between Formal Methods group in the Automated Reasoning System division at Istituto Trentino di Cultura, Istituto per la Ricerca Scientifica e Tecnologica in Trento, Italy), the Model Checking group at Carnegie Mellon University, the Mechanized Reasoning Group at the University of Genoa and the Mechanized Reasoning Group at the University of Trento.

NUSMV 2 is open source software. It combines BDD-based model checking with SAT-based model checking. It has been designed as an open architecture for model checking. NUSMV 2 exploits the CUDD library developed by Fabio Somenzi at Colorado University and SAT-based model checking component that includes an

RBC-based Bounded Model Checker, connected to the SIM SAT library developed by the University of Genova. It is aimed at reliable verification of industrially sized designs, for use as a back-end for other verification tools and as a research tool for formal verification techniques.

An enhanced version of SMV, RULEBASE `www.haifa.ibm.com/projects/verification/RB_Homepage/` [7] is an industry-oriented tool for the verification of hardware designs, developed by the IBM Haifa Research Laboratory. In an effort to make the specification of *CTL* properties easier for the non-expert, RULEBASE supports its own language, Sugar. In addition, RULEBASE supports standard hardware description languages such as VHDL and VERILOG. RULEBASE is especially applicable for verifying the control logic of large hardware designs.

VEREOFY `http://www.vereofy.de/` was written at Technische Universität Dresden. It is developed in the context of the EU project CREDO. VEREOFY is a formal verification tool of checking of component-based systems for operational correctness.

Model checking tools were initially developed to reason about the logical correctness of discrete state systems, but have since been extended to deal with real-time and limited forms of hybrid systems. Real-time systems are systems that must perform a task within strict time deadlines. Embedded controllers, circuits and communication protocols are examples of such time-dependent systems. The hybrid model checker HYTECH [44] is used to analyze dynamical systems whose behavior exhibits both discrete and continuous change. HYTECH automatically computes the conditions on the parameters under which the system satisfies its safety and timing requirements.

The most widely used dense real-time model checker (in which time is viewed as increasing continuously) is UPPAAL `www.uppaal.com/` [70]. Models are expressed as timed automata [2] and properties defined in UPPAAL logic, a subset of Timed Computational Tree Logic (*TCTL*) [1]. UPPAAL is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays, etc.). The tool is developed in collaboration between the Department of Information Technology at Uppsala University, Sweden and the Department of Computer Science at Aalborg University in Denmark.

Another real-time model checker is KRONOS `http://www-verimag.imag.fr/TEMPORISE/kronos/` [92]. KRONOS is developed at VERIMAG, a leading research center in embedded systems in France. KRONOS checks whether a real-time system modeled by a timed automaton satisfies a timing property specified by a formula of the **T**imed **C**omputational **T**ree **L**ogic *TCTL*, a timed extension of *CTL*.

Model checking requires the manual construction of a model, via a modeling language, which is then converted to a Kripke structure or an automaton for model checking. Model checking starts with translation to model checker language. In model checking considerable gains can be made by finding ways to extract models directly from program source code. There have been several promising attempts to do so.

VERISOFT `http://cm.bell-labs.com/who/god/verisoft/` is the first model checker that could handle programs directly.

The first version of BLAST (**B**erkeley **L**azy **A**bstraction **S**oftware verification **T**ool) `http://mtc.epfl.ch/software-tools/blast/`, `http://www.sosy-lab.org/~dbeyer/blast_doc/blast001.html`, `www.sosy-lab.org/~dbeyer/blast_doc/blast.pdf` [45] was developed for checking safety properties in C programs at University of California, Berkeley. The BLAST project is supported by the National Science Foundation. BLAST is a popular software model checker for revealing errors in `Linux` kernel code. BLAST is relatively independent of the underlying machine and operating system. It is free software, released under the Modified *BSD* license `http://www.oss-watch.ac.uk/resources/modbsd.xml`. BLAST is based on similar concepts as SLAM `http://research.microsoft.com/en-us/projects/slam/`. BLAST and SLAM are relatively new. SLAM was developed by Microsoft Research around 2000, i.e., earlier than BLAST, which was developed around 2002. Both the checkers have many characteristics in common. One key difference between SLAM and BLAST is the use of lazy abstraction in BLAST.

SLAM has been customized for the `Windows` product `StaticDriverVerifier`, SDV, a tool in the `WindowsDriverDevelopmentKit`.

SLAM and BLAST differ from other model checking tools in many ways. First of all, the traditional approach to model-checking (followed by SPIN and KRONOS) has been to first create a model of a system, and once the model has been verified, move on to the actual implementation. SLAM and BLAST fall in the category of the "modern" approach in model checking. The user has already completed the implementation and wishes to verify the software. The objective then is to create a model from the existing program and apply model checking principles, such that the original program is verified.

The FEAVER (**Fea**ture **Ver**ification system) `http://cm.bell-labs.com/cm/cs/what/feaver/` tool grew out of an attempt to come up with a thorough method to check the call processing software for a commercial switching product, called the PATHSTAR® access server [54,51]. It allows models to be extracted mechanically

from the source of software applications, and checked using SPIN. SPIN allows C code to be embedded directly within a PROMELA specification [53,52].

The Time Rover `http://www.time-rover.com/` is a specification based verification tool for applications written in C, C++, JAVA, VERILOG and VHDL. The tool combines formal specification, using *LTL* and *MTL*, with conventional simulation/execution based testing. The Temporal Rover is tailored for the verification of complex protocols and reactive systems where behavior is time dependent. The methodology and technology are based on the Unified Modeling Language (UML) and are currently in active use by NASA and the national Missile Defense development team.

Since Pnueli introduced temporal logic to computer science, the logic has been extended in various ways to include probability. Probabilistic techniques have proved successful in the specification and verification of systems that exhibit uncertainty. Early works in this field were focusing on the verification of qualitative properties. These included work of [30] which considered models of two types, **D**iscrete-**T**ime **M**arkov **C**hains (DTMCs) and **M**arkov **D**ecision **P**rocesses (MDPs).

Tools concerning model checking probabilistic systems such as PRISM (**PR**obabilistic **S**ymbolic **M**odel Checker) `http://www.cs.bham.ac.uk/~dxp/prism/`, [67,69,68] have been developed and applied to several real-world case studies. Other tools include ETMCC [46], CASPA [65] and MRMC (**M**arkov **R**eward **M**odel **C**hecker) [58].

ETMCC [90] requirements against action-labeled continuous time Markov chains. Probabilistic Model Checker ETMCC (**E**rlangen-**T**wente **M**arkov **C**hain **C**hecker) [46] is developed jointly by the Stochastic Modeling and Verification group at the University of Erlangen-Nürnberg, Germany, and the Formal Methods group at the University of Twente, the Netherlands. ETMCC is the first implementation of a model checker for **D**iscrete-**T**ime **M**arkov **C**hain**s** (DTMs) and **C**ontinuous-**T**ime **M**arkov **C**hain**s** (CTMCs). It uses numerical methods to model check *PCTL* [41] and **C**ontinuous **S**tochastic **L**ogic (*CSL*)[25] formulas respectively for DTMCs and CTMCs.

Markov Reward Model Checker **M**arkov **R**eward **M**odel **C**hecker (MRMC) `http://www.mrmc-tool.org/trac/` has been developed by the Formal Methods & Tools group at the University of Twente, The Netherlands and the Software Modeling and Verification group at RWTH Aachen University, Germany under the guidance of Joost-Pieter Katoen [5, Ch. 10 Probabilistic systems]. MRMC is a successor of ETMCC, which is a prototype implementation of a model checker for continuous-time Markov chains.

[25] A branching-time temporal logic a´ la *CTL* with state and path formulas [4,6,3].

PRISM stands for Probabilistic Symbolic Model Checker http://www.prismmodelchecker.org/. It is the internationally leading probabilistic model checker being implemented at the University of Birmingham [67,69,68], http://www.cs.bham.ac.uk/~dxp/prism/. First public release: September 2001.

There are three types of probabilistic models that PRISM can support directly: **D**iscrete-**T**ime **M**arkov **C**hain**s**, Markov decision processes and **C**ontinuous-**T**ime **M**arkov **C**hain**s**.

PRISM [69,85] allows time to be considered as increasing either in discrete steps or continuously. Models are expressed in PRISM own modeling language and converted to a variant of a Markov chain (either discrete- or continuous-time). Properties are written in terms of *PCTL* or *CSL*, respectively. Models can also be expressed using PEPA (**P**erformance **E**valuation **P**rocess **A**lgebra) [47] and converted to PRISM. PRISM is free and open source, released under the *GNU* General Public License (*GPL*), available freely for research an teaching.

References

[1] Alur R., Courcoubetis C., Dill D. L. (1990): Model-checking for real-time systems, *in* 'Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science', IEEE Computer Society Press, Philadelphia, PA, pp. 414–425.

[2] Alur R., Dill D. (1993): A theory of timed automata', *Inf. Comput.* **194**, 2–34.

[3] Aziz A., Sanwal K., Singhal V., Brayton R. (2000): Model checking continuous time Markov chains, *ACM Trans. Computational Logic* **1**(1), 162–170.

[4] Aziz A., Sanwal K., Singhal V., Brayton R. K. (1996): Verifying continuous time Markov chains, *in* R. Alur, T. A. Henzinger, eds, 'Eighth International Conference on Computer Aided Verification CAV 1996', Vol. 1102 of *Lecture Notes in Computer Science*, Springer Verlag, New Brunswick, NJ, USA, pp. 269–276.

[5] Baier C., Katoen J. P. (2008): *Principles of Model Checking*, The MIT Press. Foreword by Kim Guldstrand Larsen.

[6] Baier C., Katoen J.-P., Hermanns H. (1999): Approximate symbolic model checking of continuous-time Markov chains, *in* 'International Conference on Concurrency Theory', pp. 146—-161.

[7] Beer, I., Ben-David, S., Eisner, C., Landver, A. (1996): Rulebase: An industry-oriented formal verification tool, *in* 'Proceedings of the 33rd Conference on Design Automation (DAC'96)', ACM Press, Las Vegas, NV, pp. 655—660.

[8] Ben-Ari, M., Manna, Z., Pnueli, A. (1981): The temporal logic of branching time, *in* 'Proc. 8th ACM Symposium on Principles of Programming Languages', ACM Press, New York, pp. 164–176. Por. [9].

[9] Ben-Ari, M., Manna, Z., Pnueli, A. (1983): 'The temporal logic of branching time', *Acta Informatica* **20**, 207–226. Por. [8].

[10] Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P. (2001): *Systems and Software Verification. Model-Checking Techniques and Tools*, Springer.

[11] Bidoit, B. M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L. , Schnoebelen, P. (2001): *Systems and Software Verification: Model-checking Techniques and Tools*, Springer.

[12] Boyer, R. S., Moore, J. S. (1979): *A Computational Logic*, Academic Press, New York.

[13] Boyer, R. S., Moore, J. S. (1988): 'Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic', *Machine Intelligence* **11**, 83–124.

[14] Bradfield, J. C., Stirling, C. (2001): Modal logics and μ-calculi: An introduction, *in* J. A. Bergstra, A. Ponse, S. A. Smolka, eds, 'Handbook of Process Algebra', Elsevier Science, chapter 4, pp. 293–330.

[15] Brock, B., Hunt, W. (1997): Formally specifying and mechanically verifying programs for the motorola complex arithmetic processor dsp, *in* 'Proceedings of the IEEE International Conference on Computer Design (ICCD'97)', pp. 31—-36.

[16] Cengarle, M. V., Haeberer, A. M. (2000): Towards an epistemology-based methodology for verification and validation testing, Technical report 0001, Ludwig-Maximilian's Universität, Institut für Informatik, München, Oettingenstr. 67. 71 pages.

[17] Cimatti, A., Clarke, E., Giunchig lia, E., Giunchig lia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A. (2002): NuSMV2: A new opensource tool for symbolic model checking, *in* E. Brinksma, K. Larsen, eds, 'Proceedings of the 14th International Conference on Computer-Aided Verification (CAV 2002)', Vol. 2404 of *Lecture Notes in Computer Science*, Springer-Verlag, Copenhagen, Denmark, pp. 359—-364.

[18] Cimatti, A., Clarke, E., Giunchig lia, F., Roveri, M. (1999): NuSMV2: A new symbolic model verifier, *in* N. Halbwachs, D. Peled, eds, 'Proceedings of the 11th International Conference on Computer-Aided Verification (CAV '99)', Vol. 1633 of *Lecture Notes in Computer Science*, Springer-Verlag, Trento, Italy, pp. 495—499.

[19] Cimatti, A., Clarke, E. M., Giunchig lia, F., Roveri, M. (2000): 'NuSMV: A new symbolic model checker', *International Journal on Software Tools for Technology Transfer* **2**(4), 410–425.

[20] Clarke, E. M. (2008): The birth of model checking, *in* DBLP:conf/spin/5000, pp. 1–26.

[21] Clarke, E. M., E., E. A. (1982): Design and synthesis of synchronization skeletons using branching-time temporal logic, *in* 'Logic of Programs, Workshop', Vol. 131 of *Lecture Notes in Computer Science*, Springer-Verlag, London, UK, pp. 52—-71.

[22] Clarke, E. M., Emerson, E. A., Sistla, A. P. (1983): Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach, *in* 'Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages', Austin, Texas, pp. 117–126.

[23] Clarke, E. M., Emerson, E. A., Sistla, A. P. (1986): 'Automatic verification of finite-state concurrent systems using temporal logic specifications', *ACM Transactions on Programming Languages and Systems* **8**(2), 244–263.

[24] Clarke, E. M., Grumberg, J. O., Peled, D. A. (1999): *Model Checking*, The MIT Press.

[25] Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., Veith, H. (2001): Progress on the state explosion problem in model checking, *in* 'Informatics — 10 Years Back. 10 Years Ahead.', Vol. 2000 of *Lecture Notes in Computer Science*, Springer-Verlag, London, UK, pp. 176–194.

[26] Clarke, E. M., Wing, J. M., Alur, R., Cleaveland, R., Dill, D., Emerson, A., Garland, S., German, S., Guttag, J., Hall, A., Henzinger, T., Holzmann, G., Jones, C., Kurshan, R., Leveson, N., McMillan, K., Moore, J., Peled, D., Pnueli, A., Rushby, J., Shankar, N., Sifakis, J., Sistla, P., Steffen, B., Wolper, P., Woodcock, J., Zave, P. (1996): 'Formal methods: state of the art and future directions', *ACM Computing Surveys* **28**(4), 626–643.

[27] Clarke, E., Wing, J. M. (1996): 'Formal methods: State-of-the-art and future directions', *ACM Comput. Surv.* **28**(4), 626—-643. Report by the Working Group on Formal Methods for the ACM Workshop on Strategic Directions in Computing Research.

[28] Coe, T., Mathisen, T., Moler, C., Pratt, V. (1995): 'Computational aspects of the pentium affair', *IEEE Comput. Sci. Eng.* **2**(1), 18–31.

[29] Connelly, R., Gousie, M. B., Hadimioglu, H., Ivanov, L., Hoffman, M. (2004): 'The role of digital logic in the computer science curriculum', *Journal of Computing Sciences in Colleges* **19**, 5–8.

[30] Courcoubetis, C., M. Yannakakis, M. (1988): Verifying temporal properties of finite state probabilistic programs, *in* 'Proc. 29th Annual Symposium on Foundations of Computer Science (FOCS'88)', IEEE Computer Society Press, pp. 338—-345.

[31] Daskalopulu, A. (2000): Model checking contractual protocols, *in* J. Breuker, R. Leenes, R. Winkels, eds, 'Legal Knowledge and Information Systems', JURIX 2000: The 13th Annual Conference, IOS Press, Amsterdam, pp. 35–47.

[32] Dijkstra, E. W. (1968): Notes on structured programming, *in* E. W. D. O.-J. Dahl, C. A. R. Hoare, eds, 'Structured Programming', Academic Press, London, pp. 1–82.

[33] Dijkstra, E. W. (1989): In reply to comments. EWD1058.

[34] Dill, D. L. (1996): The murϕ verification system, *in* R. Alur, T. Henzinger, eds, 'Proceedings of the 8th International Conference on Computer Aided Verification (CAV '96)', Vol. 1102 of *Lecture Notes in Computer Science*, Springer-Verlag, New Brunswick, NJ, pp. 390—-393.

[35] Dill, D. L., Drexler, A. L., Hu, A. J., Yang, C. H. (1992): Protocol verification as a hardware design aid, *in* 'Proceedings of the 1992 IEEE International Conference on Computer Design: VLSI in Computer and Processors (ICCD'92)', IEEE Computer Society, Cambridge, MA, pp. 522–525.

[36] Emerson, E. A. (2008): The beginning of model checking: A personal perspective, *in* DBLP:conf/spin/5000, pp. 27–45.

[37] Floyd, R. W. (1967): Assigning meanings to programs, *in* J. T. Schwartz, ed., 'Mathematical Aspects of Computer Science. Proceedings of Symposia in Applied Mathematics', Vol. 19, American Mathematical Society, Providence, pp. 19–32.

[38] Giunchig lia, F., Traverso, P. (1999): Planning as model checking, *in* 'Proceedings of the Fifth European Workshop on Planning, (ECP'99)', Springer, pp. 1–20.

[39] Gordon, M., Melham, T. (1993): *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, Cambridge University Press.

[40] Grumberg, O., Veith, H., eds (2008): *25 Years of Model Checking - History, Achievements, Perspectives*, Vol. 5000 of *Lecture Notes in Computer Science*, Springer.

[41] Hansson, H., Jonsson, B. (1994): 'A logic for reasoning about time and reliability', *Formal Aspects of Computing* **6**, 512–535.

[42] Hasle, P. F. V., Øhrstrøm, P. (2004): Foundations of temporal logic. The WWW-site for Arthur Prior, `http://www.kommunikation.aau.dk/prior/index2.htm`.

[43] Heath, J., Kwiatowska, M., Norman, G., Parker, D., Tymchysyn, O. (2006): Probabalistic model checking of complex biological pathways, *in* C. Priami, ed., 'Proc. Comp. Methods in Systems Biology, (CSMB'06)', Vol. 4210 of *Lecture Notes in Bioinformatics*, Springer, pp. 32–47.

[44] Henzinger, T., Ho, P., Wong-Toi, H. (1997): 'A model checker for hybrid systems', *Int. J. Softw. Tools Technol. Transfer* **1**(1/2), 110–122.

[45] Henzinger, T., Jhala, R., Majumdar, R., Sutre, G. (2003): Software verification with BLAST, *in* T. Ball, S. Rajamani, eds, 'Model Checking Software: Proceedings of the 10th International SPIN Workshop (SPIN 2003)', Vol. 2648 of *Lecture Notes in Computer Science*, Springer-Verlag, Portland, OR, pp. 235–239.

[46] Hermanns, H., Katoen, J.-P., Meyer-Kayser, J., Siegle, M. (2000): A Markov chain model checker, *in* 'Tools and Algorithms for Construction and Analysis of Systems', pp. 347—-362.

[47] Hillston, J. (1996): *A Compositional Approach to Performance Modeling*, Distinguished Dissertations in Computer Science, Cambridge University Press, Cambridge, UK.

[48] Hoare, C. A. R. (1969): 'An axiomatic basis for computer programming', *Communications of the ACM* **12**(10), 576–580,583. Również w: [49, 45–58].

[49] Hoare, C. A. R., Jones, C. B. (1989): *Essays in Computing Science*, Prentice Hall.

[50] Holzmann, G. (1991): *Design and validation of computer protocols*, Prentice Hall, New Jersey.

[51] Holzmann, G. J. (2002): Software analysis and model checking, *in* 'CAV', pp. 1–16.

[52] Holzmann, G. J., Smith, M. H. (2002): FEAVER 1.0 user guide, Technical report, Bell Labs. 64 pgs.

[53] Holzmann, G., Smith, M. (1999): A practical method for the verification of event-driven software, *in* 'Proceedings of the 21st International Conference on Software engineering (ICSE '99), Los Angeles, CA', ACM Press, New York, pp. 597–607.

[54] Holzmann, G., Smith, M. (1999): Software model checking. Extracting verification models from source code, *in* J. W. et al., ed., 'Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification (FORTE/PSTV '99)', Vol. 156, International Federation for Information Processing, Kluwer, Beijing, China, pp. 481–497.

[55] Hughes, G. E., Cresswell, M. J. (1968): *An Introduction to Modal Logic*, Methuen and Co., London.

[56] Huth, M. R. A., D., R. M. (2000): *Logic in Computer Science: Modelling and Reasoning about Systems*, Cambridge University Press.

[57] Iosif, R., Sisto, R. (1999): dspin: A dynamic extension of spin, *in* D. D. et al., ed., 'Proceedings of the 5th and 6th International SPIN Workshops', Vol.

1680 of *Lecture Notes in Computer Science*, Springer-Verlag, Trento, Italy and Toulouse, France, pp. 20–33.

[58] Katoen, J.-P., Khattri, M., Zapreev, I. S. (2005): A Markov reward model checker, *in* 'Quantitative Evaluation of Systems (QEST)', pp. 243—-244.

[59] Kaufmann, M., Moore, J. S. (2004): 'Some key research problems in automated theorem proving for hardware and software verification', *Rev. R. Acad. Cien. Serie A. Mat.* **98**(1), 181—-196.

[60] Kautz, H., Selman, B. (1992): Planning as satisfiability, *in* 'ECAI '92: Proceedings of the 10th European conference on Artificial intelligence', John Wiley & Sons, Inc., New York, NY, USA, pp. 359–363.

[61] Kröger, F. (1977): 'A logic of algorithmic reasoning', *Acta Informatica* **8**(3), 243–266.

[62] Kröger, F. (1987): *Temporal Logic of Programs*, Springer-Verlag New York, Inc., New York, NY, USA.

[63] Kröger, F., Merz, S. (1991): 'Temporal logic and recursion', *Fundam. Inform.* **14**(2), 261–281.

[64] Kröger, F., Merz, S. (2008): *Temporal Logic and State Systems*, Springer.

[65] Kuntz, M., Siegle, M., Werner, E. (2004): *Symbolic performance and dependability evaluation with the tool CASPA*.

[66] Kurshan, R. (1995): *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*, Princeton Series in Computer Science, Princeton University Press, Princeton, NJ.

[67] Kwiatkowska, M., Norman, G., Parker, D. (2001): *PRISM*: Probabilistic symbolic model checker, *in* P. Kemper, ed., 'Proc. Tools Session of Aachen 2001', International Multiconference on Measurement, Modelling and Evaluation of Computer-Communication Systems, Dortmund, pp. 7–12. Available as Technical Report 760/2001, University of Dortmund.

[68] Kwiatkowska, M., Norman, G., Parker, D. (2002): Probabilistic symbolic model checking with *PRISM*: A hybrid approach, *in* J.-P. Katoen, P. Stevens, eds, 'Proc. 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)', Vol. 2280 of *Lecture Notes in Computer Science*, Springer, pp. 56–66.

[69] Kwiatkowska, M., Norman, G., Parker, D. (2002): Probabilistic symbolic model checking with PRISM, *in* J. Katoen, P. Stevens, eds, 'Proceedings of the 8th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2002)', Vol. 2280 of *Lecture Notes in Computer Science*, Springer-Verlag, Grenoble, France, pp. 52–66. Held as part of the Joint European Conference on Theory and Practice of Software (ETAPS 2002).

[70] Larson, K., Pettersson, P., Yi, W. (1997): 'Uppaal in a nutshell', *Int. J. Softw. Tools. Technol. Transfer* **1**(1/2), 134–152.

[71] M. Kaufmann, M., Manolios, P., Moore, J. S. (2000): *Computer-Aided Reasoning: An Approach*, Kluwer Academic Press, Boston.

[72] Manna, Z., A. Pnueli, A. (1992, 1995): *The Temporal Logic of Reactive and Concurrent Systems*, Vol. 1: Specification, 2: Safety, Springer-Verlag, New York.

[73] Manna, Z., Bjørner, N., Browne, A., Chang, E., Alfaro, L. D., Devarajan, H., Kapur, A., Lee, J., Sipma, H. (1994): STeP: The stanford temporal prover, Technical report, Computer Science Department, Stanford University Stanford, CA.

[74] Manna, Z., Waldinger, R. (1985): *The Logical Basis for Computer Programming*, Addison-Wesley.

[75] McMillan, K. L. (1993): *Symbolic Model Checking: An approach to the State Explosion Problem*, Kluwer Academic, Hingham, MA.

[76] Miller, A., Donaldson, A., Calder, M. (2006): 'Symmetry in temporal logic model checking', *ACM Computing Surveys* **38**(3).

[77] Naur, P. (1966): 'Proof of algorithms by general snapshots', *BIT* **6**(4), 310–316.

[78] Owicki, S. S., Lamport, L. (1982): 'Proving liveness properties of concurrent programs', *ACM Trans. Program. Lang. Syst.* **4**(3), 455–495.

[79] Pnueli, A. (1977): The temporal logic of programs, *in* 'Proceedings of the 18th IEEE-CS Symposium on Foundation of Computer Science (FOCS-77)', IEEE Computer Society Press, pp. 46–57.

[80] Pnueli, A. (1981): 'The temporal semantics of concurrent programs', *Theoretical Comput. Sci.* **13**, 45–60.

[81] Pratt, V. R. (1980): 'Applications of modal logic to programming', *Studia Logica* **9**, 257–274.

[82] Queille, J.-P., Sifakis, J. (1982): Specification and verification of concurrent systems in CESAR, *in* 'Proceedings 5th International Symposium on Programming', Vol. 137 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 337–351.

[83] Randell, B. (1973): *The Origin of Digital Computers*, Springer Verlag.

[84] Rescher, N., Urquhart, A. (1971): *Temporal Logic*, Springer, Wien, New York.

[85] Rutten, J., Kwiatkowska, M., Norman, G., Parker, D. (2004): *Mathematical Techniques for Analysing Concurrent and Probabilisitic Systems*, Vol. 23 of *American Mathematical Society, CRM Monograph Series*, Centre de Recherches Mathématiques, Université de Montréal.

[86] Schneider, K. (2003): *Verification of Reactive Systems. Formal Methods and Algorithms*, Texts in Theoretical Computer Science (EATCS Series), Springer-Verlag.

[87] Schnoebelen, P. (2002): 'The complexity of temporal logic model checking', *Advances in Modal Logic* **4**, 1–44.

[88] Turing, A. M. (1936–37): 'On computable numbers, with an application to the Entscheidungsproblem', *Proceedings of the London Mathematical Society* **42**(Series 2), 230–265. Received May 25, 1936; Appendix added August 28; read November 12, 1936; corrections Ibid. vol. 43(1937), pp. 544–546. Turing's paper appeared in Part 2 of vol. 42 which was issued in December 1936 (Reprint in: [89]; 151–154). Online version: `http://www.abelard.org/turpap2/tp2-ie.asp`.

[89] Turing, A. M. (1965): On computable numbers, with an application to the Entscheidungsproblem, *in* M. Davis, ed., 'The Undecidable', Raven Press, Hewlett, NY, pp. 116–151.

[90] Vaandrager F. W.and De Nicola, R. (1990): Actions versus state based logics for transition systems, *in* 'Proc. Ecole de Printemps on Semantics of Concurrency', Vol. 469 of *Lecture Notes in Computer Science*, Springer, pp. 407—-419.

[91] Wang, W., Hidvegi, Z., Bailey, A., Whinston, A. (2000): 'E-process design and assurance using model checking', *IEEE Computer* **33**(10), 48–53.

[92] Yovine, S. (1997): 'Kronos: A verification tool for real-time systems', *Int. J. Softw. Tools Technol. Transfer* **1**(1/2), 123–133.

LOGIKA I FORMALNA WERYFIKACJA SYSTEMÓW KOMPUTEROWYCH. KILKA UWAG.

Streszczenie Do specyfikacji i weryfikacji zarówno sprzętu jak i programów stosowane są różne logiki. Główną wadą metody teorio-dowodowej weryfikacji jest problem znalezienia dowodu. Zastosowanie tej metody zakłada aksjomatyzację logiki. Własności systemu mogą być sprawdzane za pomocą jego modelu. Model jest zbudowany w języku specyfikacji i sprawdzany automatycznie. Zastosowanie sprawdzania za pomocą modelu zakłada rozstrzygalność zadania. Istnieje wielka różnorodność programów (model checker) do sprawdzania własności za pomocą modeli.

Słowa kluczowe: Logika, Wery?kacja, Metoda teorio-dowodowa, Sprawdzanie za pomoca modelu

Praca wspierana przez grant MNiSW nr 3 T11F 01130.