International Journal of General Systems
Vol. 00, No. 00, October 2008, 1–30

Automated Formal Analysis and Verification: An Overview

Bohuslav Křena[†] and Tomás Vojnar[‡]

Brno University of Technology, FIT, IT4Innovations Centre of Excellence
Božetěchova 2, CZ-61266 Brno, Czech Republic

(*Received 00 Month 200x; final version received 00 Month 200x*)

This paper provides an overview of various existing approaches to automated formal analysis and verification. The most space is devoted to the approach of model checking, including its basic principles as well as the different techniques that have been proposed for dealing with the state space explosion problem in model checking. This paper, however, includes a brief discussion of theorem proving and static analysis too. All of the discussed approaches are introduced mostly on an informal level, with an attempt to provide the reader with their basic ideas and references to works where more details can be found.

Keywords: formal analysis and verification, model checking, state space explosion, theorem proving, static analysis

1. Introduction

Computer-aided technologies have become ubiquitous in most areas of our lives. The services they implement are continuously getting more complex due to supporting new hardware technologies, providing new features, or integrating different, originally stand-alone systems. Failures of computer systems may have a broad range of consequences, ranging from disappointment of customers (and their move to another producer) to more costly and even tragic situations when the system that fails is a control system of an aircraft, a space device, or an industrially critical system (such as a nuclear power plant or an electricity distribution network). The fact that such faults do happen may be illustrated by a number of real accidents (e.g., the failure of the maiden flight of Ariane 5 in 1996, four NASA Mars missions failing between 1997 and 2004, the US Northeast blackout in 2003, or the "mutiny" of the autopilot of a Boeing 777 aircraft during a regular passenger flight in 2005). Moreover, even if a computer-based system does not fail itself, it may contain weak points which may be used for a successful intentional attack on the system by human attackers, which is also considered increasingly dangerous.

Correspondingly, a significant stress is put on the use of various methods of discovering errors in computer-based systems. The prevailing approach is to use different forms of testing, simulation, or code inspection. Despite that these techniques have been in use for a long time, research and development devoted to their further improvements is still quite active, producing new methods and methodologies such as search-based testing, model-based design, agile testing, or extreme programming. However, these—let us say "traditional"—methods suffer one important deficiency: *They cannot prove a system correct*, i.e., they cannot prove it to be free of errors with respect to some specification. That is why one can also

[†]Email: krena@fit.vutbr.cz [‡]Email: vojnar@fit.vutbr.cz

ISSN: 0308-1079 print/ISSN 1563-5104 online
© 2008 Taylor & Francis
DOI: 10.1080/0308107YYxxxxxxxx
http://www.informaworld.com

witness a strong and ever rising interest in the development and applications of *formal verification* methods that can remove this constraint. Moreover, it turns out that even in cases in which the formal verification process is not completely finished due to its high cost, or when it is intentionally restricted in some way (e.g., by restricting the depth of the state space of the verified system to be traversed or by intentionally suppressing some generated warnings when it is not sure whether they correspond to real errors or not), it may still be quite valuable. This is because even in such cases, it may find a number of errors that are often different from those found by traditional methods, which is due to the very different principles on which these methods work.

A large interest of the industry in formal verification may be documented by the existence of groups specializing in research as well as applications of formal verification within various leading industrial companies and organisations, such as Microsoft, Intel, IBM, NASA, Airbus, CEA, Cadence, Mentor Graphics, etc. Another indication is the emergence of spin-offs working in the area of formal verification or at least restricted formal verification (such as Coverity, GrammaTech, AbsInt, Prover, Monoidics, and others). On the other hand, an intense interest of academia in formal verification can be documented by the existence of many live and very competitive conferences (often sponsored by the industry) as well as by many publications, academic tools, and projects arising in the area. An interesting trend is also the emergence of various verification competitions—such as the Hardware Model Checking Competition (HWMCC), the Competition on Software Verification (SV-COMP), or the VSTTE Software Verification Competition—intended to compare the various existing verification tools and to stimulate their further development.

1.1 *Formal Verification and Analysis*

We use the term *formal verification* to denote verification methods based on formal, mathematical roots and (at least potentially) capable of proving error freeness of systems with respect to some correctness specification. The potential to detect all errors with respect to a given specification is called *soundness* of a method. It means that if such a method terminates and claims a system correct with respect to a certain specification, the system is indeed correct. On the other hand, we call a method *complete* if it does not raise false alarms, i.e., if it does not report spurious errors. As we have already indicated above, sometimes, the potential of a method to be sound is sacrificed in order to increase efficiency of the approach, leading to an error detection approach with formal verification roots.

By *formal analysis*, we denote approaches that can answer in a sound way questions other than whether the given system is correct with respect to some specification. Examples of such questions may be checking which variables are aliased at a given program point, how many elements can appear in some buffer, what the shape of dynamic linked data structures that can appear in a program is, and so on. Answers to such questions may be used for verification (which may require some further automated or manual reasoning), but also for optimisation, code generation, parallelisation, complexity analysis, etc.

The most common approaches used in computer-aided formal analysis and verification include *theorem proving*, *model checking*, and various forms of *static analysis*. We have to, however, note that the meaning of these terms is sometimes not completely sharp and not always understood in exactly the same way. Moreover, the approaches may be used in various combinations and/or non-standard extensions.

In this survey, we devote the most space to the area of model checking. However, for a better orientation in the subject, we briefly introduce all the mentioned approaches and characterise their relations. In the description, we stay on an informal level only, trying to grasp the intuition and main ideas behind the discussed approaches: their formal details can be found in the referenced works.

1.2 *A Note on Testing and Dynamic Analysis*

Despite this survey is focusing on formal verification, we note that many interesting advances are constantly happening in the area of traditional *testing* too. A lot of effort is, for instance, invested into testing of concurrent software that is very difficult due to many errors manifest only very rarely when the concurrently running processes or threads are scheduled in a very specific way. In order to increase chances to spot such errors, techniques like *injection of noise* into the scheduling of concurrent processes have been proposed and supported by tools such as IBM ConTest (Edelstein *et al.* 2002) or ANaConDA (Fiedor and Vojnar 2012). Further, there have been proposed various *dynamic analyses* that analyse the behaviour seen in a testing run (tracking, e.g., the order in which locks are taken or the way shared variables are protected when accessed). Consequences of the analysed behaviour can then be extrapolated leading to warnings about possible errors even when such an error is not witnessed in the given testing run. In particular, many dynamic analyses have been proposed for detection of data races and deadlocks including, e.g., the Eraser (Savage *et al.* 1997), GoldiLocks (Elmas *et al.* 2007), FastTrack (Flanagan and Freund 2009), or GoodLock (Havelund 2000) detectors. Other approaches to improve the quality of testing include *combinations* with various (restricted) formal verification techniques, which we will briefly tackle later on in this paper, or using advanced *search-based techniques* in order to generate test data or parameters (McMinn 2004, Hrubá *et al.* 2012).

A lot of effort has also been devoted into automation of *debugging* leading to techniques such as *delta debugging* (Zeller and Hildebrandt 2002). Delta debugging is in particular based on automatically pruning the set of possible circumstances leading to a failure until a minimal set remains. Finally, techniques of *self-healing* have also appeared that not only try to automatically detect an error, but also to prevent it from manifesting or at least to reduce its appearance—cf., e.g., (Nagpaly *et al.* 2007, Křena *et al.* 2007, Jin *et al.* 2011, Ratanaworabhan *et al.* 2011). These areas are, however, beyond the scope of this survey.

Plan of the Paper. The rest of the paper is structured as follows. Section 2 provides a brief overview of the fields of theorem proving, static analysis, and model checking. Section 3 provides a more detailed introduction to the approach of model checking. Section 4 discusses the main principles of various existing approaches to fighting the state explosion problem arising in model checking of finite-state systems. Finally, Section 5 discusses possibilities of applying model checking to infinite-state systems.

2. Common Approaches to Formal Analysis and Verification

In this section, we characterise the main approaches to formal analysis and verification, namely, theorem proving, static analysis (we will, in particular, mention data flow analysis, constraint-based analysis, type-based analysis, and abstract interpretation), and model checking.

2.1 *Theorem Proving*

Theorem proving—also called *deductive verification*—is usually a semi-automated approach using some inference system for deducing various theorems about the examined system from the facts known about the system and from general theorems of various logical theories. This approach is quite close to classical mathematical reasoning, but it is supported by computer-aided tools, the so-called *theorem provers* (or *proof assistants*), such as PVS (Owre *et al.* 2001), Isabelle (Nipkow *et al.* 2005), ACL2 (Kaufmann *et al.* 2000b,a), Coq (Bertot and Castéran 2004), and many others. These tools take care of remembering all of the so-far deduced facts and of correctly applying inference rules. The inference process is, however, usually guided by the user. The approach is very general but often very hard to use. The approach is sometimes also weak in generating counterexamples (diagnostic information) to correctness specifications in faulty systems—one may have troubles to distinguish whether the effort to prove some property is failing because there is an error in the system being examined, or because the user of the method is not bright enough to prove it correct.

On the other hand, there has also been a lot of progress in developing automated *decision procedures* (or *satisfiability solvers*) for different logics and logical theories. These solvers are used as a building block within various higher-level verification methods. Among the solvers, an important position belongs to the so-called *SAT solvers*, such as Glucose (Audemard and Simon 2009), deciding satisfiability of propositional formulae, and hence solving the classical Boolean satisfiability problem, i.e., the *SAT problem*. Another important category of the solvers is then the category of the so-called *SMT solvers*[1], like Z3 (de Moura and Bjørner 2008), which support various first-order logical theories with equality (such as linear arithmetic, linear real arithmetic, uninterpreted functions, theory of arrays, etc.). Finally, one can also find solvers for various higher-order theories, such as the Presburger arithmetic, supported, e.g., by the Omega tool set (Kelly *et al.* 1996), or WSkS, i.e., the weak second order theory of k successors, supported by MONA (Klarlund and Møller 2001).

The Boolean satisfiability problem is traditionally considered computationally intractable due to the fact that it is an NP-complete problem. However, various heuristics, like the DPLL algorithm (Davis *et al.* 1962), which have been proposed in the past years, make SAT solving very efficient in many practical cases. The same holds for the various more complex logical theories mentioned above whose worst case complexity can be much worse than that of the SAT problem (e.g., it is non-elementary in the case of WSkS). Moreover, the efficiency of the solvers is being constantly improved. This process is greatly aided by regular competitions among the solvers such as the SAT competition and the SMT competition[2].

Decision procedures can, for instance, be used as a support for different kinds of automated abstraction, such as the *predicate abstraction* (Graf and Saïdi 1997) used in model checking, which we will discuss later on. Another approach is to *annotate the verified program* by loop invariants, procedure pre- and post-conditions, and assertions to be checked, expressed in a suitable logic. This way the program is cut into *loop-free code fragments* that together with the conditions assumed to hold before and after these code fragments form (loop-free) *Hoare triples* (Hoare 1969). Provided that the effect of the loop-free code can be effectively expressed in

[1]The abbreviation comes from the words "Satisfiability Modulo Theories".

[2]There, of course, exist many more SMT and SAT solvers than the two above mentioned ones. However, giving their overview is beyond the scope of this article. An interested reader is referred to the web pages of the SAT and SMT competitions for an overview of the currently existing solvers.

the chosen logical fragment, the verification problem reduces as follows. First, the result of applying loop free code fragments on their pre-conditions can be automatically computed. Then, it is possible to automatically check whether the resulting formulae imply the post-conditions of the code fragments (such implications are called *verification conditions*).[1] This approach is used, e.g., in the VCC (Cohen *et al.* 2009) and ESC/Java2 (Cok and Kiniry 2005) tools. In order to reduce the burden of a user to provide the annotations manually, various heuristics for their discovery have been proposed—cf., e.g., the VS3 tool (Srivastava *et al.* 2009).

2.2 *Static Analysis*

Static analysis is usually characterised as analysis that collects some information about the behaviour of a system without actually executing it under its original semantics. Such a characterisation is, of course, rather broad, and it may be viewed to include even theorem proving or model checking (at least in some of its forms), which is, indeed, sometimes done by some authors. However, when setting these approaches apart (together with their various derivatives, such as model checking combined with predicate abstraction as discussed later on) and giving up attempts to cover all existing approaches, the perhaps most common approaches to static analysis include syntactic checks looking for various *error patterns* (as implemented, e.g., in the Lint tool already in the late 70's), *data flow analysis*, *constraint-based analysis*, *type-based analysis*, and *abstract interpretation*. In many cases, static analyses are not designed for checking correctness of programs but to be used within compiling, optimisation, code generation, etc. Static analyses are often highly specialised. On the other hand, they sometimes just collect some information about the system, and it is up to the user to exploit it for a given verification task.

Compared to model checking, static analyses have often—though not always—the advantage of not needing any model of the environment in which the system should run and of being able to handle very big code bases (even tens of millions of lines of entire operating system kernels, such as Linux or Windows). The need to model the environment and usually also parts of the system being examined (which would otherwise be too big to be handled) may be quite expensive and may also hide some errors, which may be ruled out by the manual modelling (Engler and Musuvathi 2004). On the other hand, not tracking the (exact) values that particular system variables may get can lead to a vast number of false alarms raised by static analysis[2]. Moreover, some kinds of errors may be difficult or impossible to discover via certain static analyses. For instance, it may be difficult or impossible to identify all possible "syntactic patterns" that could lead to certain errors, and then the otherwise very efficient methods like those mentioned in (Engler and Musuvathi 2004) may be hard to use.

2.2.1 Data Flow Analysis

Given a pre-defined set of properties of program states, which are of interest for some particular reason and which can be denoted as the so-called *data flow facts*, a *data flow analysis* tracks how the data flow facts propagate in between of

[1]Likewise, one can, of course, start from the post-condition and compute backwards.

[2]Sometimes, the tools developed in this area ignore much of the detected potential errors in order not to overwhelm the user. Then, however, the approach becomes unsound—though it may still be very valuable. For this price and due to using additional techniques to prune infeasible program paths based, e.g., on SAT/SMT solving, some of the current commercial static analysers, such as those mentioned in Section 2.2.1, can get down to low tens of per cents of false alarms.

neighbouring control points of the *control flow graph* (CFG) of a program[1]. The propagation of data flow facts is tracked in a way consistent with all feasible paths through the CFG but without directly executing the program. The most common approach to data flow analysis is the lattice-theoretic *iterative data flow analysis* pioneered in (Kildall 1973, Kam and Ullman 1976, 1977).

An iterative data flow analysis is typically built on a set of data flow facts that forms a *complete lattice*[2]. The *meet operation* is used to merge analysis results flowing to a certain control point through several different controls paths. The effect of program statements is modelled using *monotone transfer functions* defined on the carrier set of the analysis (that is why such analyses are also often called as *monotone data flow analyses*). The analysis is performed by iteratively evaluating the transfer functions and meet operations starting from some defined initial data flow fact. This fact is associated with either the entry or exit point of the analysed code depending on whether the code is analysed forward or backward[3]. The analysis stops when the data flow facts computed for particular control points stop changing. The monotonicity of transfer functions is needed together with a restriction to lattices with no infinite descending chains in order to guarantee termination of the analysis. In many cases, the data flow facts have the form of sets that are subsets of some universe of elementary data flow facts[4]. The transfer functions are then often defined using the so-called *Gen* and *Kill sets* that define which elementary data flow facts are killed and which generated (i.e., removed from or added to the input data flow fact, respectively) when executing a certain statement.

Data flow analysis can be performed in an *intra-procedural way*, i.e., within a single function, or in an *inter-procedural way*, i.e., across function boundaries (without in-lining functions into a single function). The latter can be implemented by using functional data flow facts or incrementally built *function summaries*. The summaries say that a certain data flow fact at the input leads to a certain data flow fact at the output (allowing one to avoid a repeated analysis of a function for the same arguments). Such analyses can then be further distinguished according to whether or not they track also the relevant *context* of calling a function with certain arguments.

Many specific data flow analyses have been defined to date, ranging from analyses computing information commonly used in optimising compilers (such as live variables, reaching definitions, available expressions, etc.) to analyses used for verification purposes. Data flow analysis is used in many currently leading commercial tools for static analysis (such as the static analyser of Coverity, CodeSonar from GrammaTech, or TruePath from Klocwork) as well as advanced open-source tools, such as FindBugs (Hovemeyer and Pugh 2004). In these tools, data flow analysis is either directly used to find bugs or to characterise properties of the functions into which the given code is structured before actually looking for occurrences of some bug patterns. Using the information obtained this way, the number of infeasible paths considered by the analysis may be significantly reduced, which may in turn significantly reduce the number of false alarms which would otherwise be reported.

[1]Informally, a control flow graph is a directed graph whose nodes represent basic blocks of the program, and edges express the transfer of control among them. A basic block is a maximal sequence of statements that is always entered via its first statement and always left via its last statement. Basic blocks are used to decrease the granularity of the code to be analysed.

[2]Sometimes, an equivalent notion of a complete meet semi-lattice is used.

[3]A generalisation of data flow analysis to bidirectional analysis has also been proposed (Khedker and Dhamdhere 1994).

[4]Data flow facts having the form of sets of elementary data flow facts, such as sets of live variables, available expressions, and the like, are often encoded in the form of bit-vectors having a Boolean entry for each elementary data flow fact, which encodes its membership in the data flow fact. Analyses using such data flow facts are usually called *bit-vector analyses*.

A more exact description of the use of data flow analysis in leading commercial tools is, unfortunately, difficult due to their producers keeping all details of their tools secret. According to the limited information available, the use of data flow analysis and bug patterns is often combined, e.g., with further pruning of infeasible paths using various logical solvers to check satisfiability of the conditions that appear on the concerned program paths. Further, the tools also exploit statistical methods to distinguish common and less common coding styles used in a program in order to better identify suspicious parts of the code. The tools usually ship with a set of bug detectors (checkers) developed by the tool developers for discovering the most frequent kinds of bugs. Apart from that, it is often the case that the users are offered an interface that allows them to create custom data flow analyses based on the monotone framework (which requires them to specify the data flow facts, the transfer functions, the meet operator, etc.) and to use them in custom bug detectors.

2.2.2 Constraint-based Analysis

In constraint-based analysis, a set of constraints is derived from the analysed program such that when the constraints are solved, the solution provides the needed information about the program (Aiken 1999, Nielson *et al.* 2005, Sipma *et al.* 2006). Various kinds of constraints can be used such as conditional set constraints, linear arithmetic constraints, polynomial arithmetic constraints, etc.

Constraint-based analysis is more general than data flow analysis, at least when classical one-pass unidirectional data flow analysis with no unbounded auxiliary storage is used (Khedker and Dhamdhere 1994). Constraint-based analysis can provide all solutions to the given analysis problem (not just one), it naturally allows for a bi-directional flow of information, and it can work in a better way with infinite domains. Of course, solving the generated constraints needs not be easy, but one can leverage the constantly improving results of the constraint solving community.

Typical applications of constraint-based analysis include control flow analysis (i.e., analysis deriving the possible flow of control in programs written in languages where the control flow is not obvious, which is the case, e.g., in functional and object-oriented languages), points-to analysis, derivation of linear as well as non-linear loop invariants, or derivation of ranking functions to be used for verification of program termination.

2.2.3 Type-based Analysis

Type-based analyses may be viewed to include any static analyses that make use of type information as a basis of the analysis or as a way how to make it more precise and/or more efficient. One of the most common approaches to typed-based analysis is the use of the so-called *type and effect systems* (Nielson and Nielson 1999, Palsberg 2001).

Type and effect systems extend the basic type systems of programming languages to take into account various *semantic effects* of the allowed programming constructions. One can, for instance, track how the memory is accessed (reading, writing, allocation, de-allocation), whether and how synchronisation is used (locking and unlocking of mutexes), whether and how various other resources are used (such as reading and writing of files), whether some exceptions can be generated, etc. Apart from the basic typing information and the effect information (essentially saying *what* is being done), the so-called region information is sometimes also tracked, giving information about *which resource* is affected (i.e., which file, memory location, lock, etc.). The regions have to be identified using static information only, hence they can be, e.g., associated with the program locations where some memory was allocated, a file opened, a lock created, etc.

Type and effect systems are typically specified in terms of rules similar to inference rules in logics. They can involve complex types and effects based on notions such as sub-typing/sub-effecting and polymorphism allowing quantification over type, effect, and region variables. For a new type and effect system, an appropriate inference algorithm has to be provided (and proved correct), possibly based on the algorithms already published in the literature. Alternatively, one can try to implement a desired type and effect system as an instantiation of a generic type and effect system and use its generic inference algorithm (Marino and Millstein 2009).

Many different applications of type and effect systems have been proposed, including side effect analysis, control flow analysis, binding type analysis (distinguishing data available at compile time and run time), security analyses (distinguishing secret and public information), callability analysis, strictness analysis of functions, and so on. The information obtained from type and effect systems can be used both for optimisation as well as verification (correctness of memory accesses, absence of data races, etc.).

2.2.4 Abstract Interpretation

Abstract interpretation (Cousot and Cousot 1977) is a theory of a sound approximation of the semantics of computer programs that, among other applications, allows for constructing static analyses sound by construction. Abstract interpretation consists in giving a class of programs a concrete and abstract semantics defined on suitable concrete and abstract lattice-based domains. These domains are usually linked by a pair of monotone functions—the so-called *abstraction* and *concretisation*, traditionally denoted α and γ, respectively—that form a Galois connection[1]. Program statements are modelled as monotone functions, often called as concrete and abstract transformers, on the concrete and abstract domains, respectively. The abstraction is sound if the concretisation of the result of applying any abstract transformer on any abstract value gives a larger concrete value than the value that is obtained by first concretising the abstract value and then applying the corresponding concrete transformer[2].

In a more general formulation of abstract interpretation (Cousot and Cousot 1992), the requirement of dealing with a Galois connection is lifted, and the analysis is defined in terms of a concretisation (or, dually, abstraction) function only. This, however, excludes the possibility of defining best abstract transformers, which can be defined when using Galois connections (by simply first concretising the input abstract value, then using the concrete transformer, and finally abstracting the result). Another consequence of using the more general setting is that there is no easy way of comparing the precision of abstractions.

In order to be able to use abstract interpretation for analysing programs, one further needs an operator for *accumulation* of abstract values computed for a single program point via multiple program paths. Moreover, since the abstract domain is often infinite, one needs the so-called *widening* operator ∇ that over-approximates the accumulation operator and that has the property that for any infinite sequence of abstract values $x_0, x_1, ...$, the sequence $y_0, y_1, ...$ where $y_0 = x_0$ and $y_{i+1} = y_i \nabla x_{i+1}$ eventually stabilises. The analysis is then performed by iterating the abstract transformers over the control flow graph, using the accumulation operator at program points where several program paths meet, and applying the widening operator at loop junctions to make the analysis terminate. Sometimes, the so-called *narrowing* operator $\triangle$ is also used after widening to refine its effect.

[1] A Galois connection between two partially ordered sets $(A, \leq_A)$ and $(B, \leq_B)$ consists of two monotone functions $f : A \to B$ and $g : B \to A$ such that $\forall a \in A\ \forall b \in B : f(a) \leq_B b \iff a \leq_A g(b)$.

[2] This is, if $(D_c, \leq_c)$ is a concrete domain, $(D_a, \leq_a)$ an abstract domain, $f_c : D_c \to D_c$ a concrete transformer, and $f_a : D_a \to D_a$ an abstract transformer, then $\forall x \in D_a :\ f_c(\gamma(x)) \leq_c \gamma(f_a(x))$.

The notion of abstract interpretation is quite flexible and can be instantiated in a number of ways significantly differing in their precision (basically ranging from the precision of simple, purely syntactic static analyses to full model checking). Abstract interpretation is also sometimes used as a formal framework in which abstractions to be used together with model checking are defined, as in the case of the so-called predicate abstraction (Graf and Saïdi 1997, Ball *et al.* 2001) that we will briefly discuss in Section 4.3. Abstract interpretation can, however, be used even for other purposes, such as program transformation, watermarking, information hiding, code obfuscation, and so on.

Many abstract domains suitable for practical program analysis have been defined to date and implemented in libraries such as Apron (Jeannet and Min 2009). The most domains have probably been defined for analysis of numerical programs, including domains such as intervals, octagons, or convex polyhedra. However, abstract domains suitable for many other purposes, e.g., analysis of heap, string, or array manipulation, have also been proposed. As examples of commercial tools based on abstract interpretation, one can mention tools from AbsInt including the worst-case execution time analyser aiT (Ferdinand *et al.* 2007) and the Astrée analyser checking for run-time errors in safety-critical embedded applications written or automatically generated in C (Cousot *et al.* 2005). Both of these tools have been applied, for instance, to analyse flight software of several Airbus aircraft. Another example is MathWorks with its PolySpace analyser (Deutsch 2003) aimed at run-time errors such as overflows, division by zero, or out-of-bounds array accesses in C, C++, as well as Ada.

2.3 *Model Checking*

Model checking (Clarke *et al.* 1999, Baier and Katoen 2008) is an approach of automated checking whether a system (or a model of a system) satisfies a certain correctness specification based on a *systematic exploration of the state space* of the system. The system or model to be verified can be described in a variety of different languages ranging from real-life hardware description languages (such as VHDL or Verilog), common programming languages (like C or Java), various kinds of Petri nets or process algebras to specialised modelling languages (such as the Promela language of the Spin model checker (Holzmann 1997)). The specification is typically written in some temporal logic like LTL (Pnueli 1977), CTL (Clarke and Emerson 1981), CTL* (Emerson and Halpern 1986), or μ-calculus (Kozen 1983), but some simpler specification means such as C-like assertions or end-state and progress labels known from Promela can also be used. Model checking has originally been proposed for verification of *finite-state systems*. Its roots can be traced back to the works (Clarke and Emerson 1981, Queille and Sifakis 1982)[1]. Model checking can usually be fully automated and can generate error traces explaining why a certain property does not hold in a given system. We provide a more detailed description of the basic ideas behind CTL and LTL model checking in Section 3.

The main problem to cope with when model checking finite-state systems is the *state space explosion* problem, i.e., the need to cope with the exponential growth of the number of reachable states in the size of the examined systems. To cope with the state space explosion problem, many different heuristics have been proposed. We provide an overview of some of these techniques in Section 4.

[1]E.M. Clarke, E.A. Emerson, and J. Sifakis were awarded the 2007 ACM Turing prize as a recognition of the impact of their work.

An even more complicated situation arises when one wants to apply model checking to *infinite-state systems* (such as systems with unrestricted parameters and/or various unbounded data and control structures like communication queues, dynamic linked data structures, stacks, counters, and so on). Clearly, in this case, one cannot simply enumerate reachable states to verify a given system. Indeed, most interesting verification problems over infinite-state systems are undecidable. However, there have been proposed approaches for applying model checking even in this context, and we briefly discuss some of them in Section 5.

Yet another problem—which, however, may arise with some forms of static analysis too—is the need to *model the environment* of the system being verified. This modelling task is usually quite tedious and may hide some errors if the model of the environment ignores some behaviour that is possible in practice (on the other hand, if the model of the environment enables some behaviours that are not possible in reality, many false alarms may be obtained).

Model checking has found many successful applications, including commercial ones, especially in the area of hardware verification. However, it has been successfully applied in verification of concurrent and distributed systems (such as communication and synchronisation protocols or security protocols), software (e.g., device drivers), real-time systems, probabilistic systems, as well as other kinds of systems (e.g., UML models) too. Moreover, model checking has been applied not only to computer systems but also other kinds of systems such as models from the area of systems biology, work flow, etc.

Commercial *hardware model checkers* include RuleBase from IBM, Incisive Verifier from Cadence, Magellan from Synopsys, the JasperGold Formal Property Verification App from Jasper, or the Questa Formal Verification from Mentor Graphics[1]. A different approach is taken by companies such as the Oski company which does not sell its own tool, but rather offers RTL formal verification services, including development of abstractions of the verified designs needed to overcome the state explosion problem as well as computation of the achieved coverage of the behaviour of the verified system (Aggarwal *et al.* 2011). Freely available tools supporting hardware model checking include Cadence SMV (McMillan 2000), NuSMV (Cavada *et al.* 2010), Uclid (Seshia *et al.* 2003), or the model checking engines available within the ABC framework (Brayton and Mishchenko 2010).

Model checkers for *concurrent and distributed systems* include, e.g., Spin (Holzmann 1997) and DiViNe (Barnat *et al.* 2010b) using specialised input languages (Promela, DVE) for describing models of the verified systems. One of the most successful *software model checkers* is the Static Driver Verifier (Ball *et al.* 2004) from Microsoft that is successfully used for verifying selected critical properties of Windows drivers. Apart from it, there exist many more experimental academic software model checkers such as the Java PathFinder from NASA for model checking Java programs on the byte code level (Visser *et al.* 2003); Blast (Henzinger *et al.* 2003), CPAChecker (Beyer and Keremoglu 2011), SatAbs (Clarke *et al.* 2005), Wolverine (Kroening and Weissenbacher 2011) for model checking C programs using predicate abstraction (briefly discussed in Section 4.3); or bounded model checkers for the C language such as CBMC (Clarke *et al.* 2004a) or LLBMC (Merz *et al.* 2012). Finally, to give some representatives of tools for model checking of *real-time* and *probabilistic systems* too, we can mention Uppaal (Behrmann *et al.* 2004) and Prism (Kwiatkowska *et al.* 2011).

[1]The producers often speak about formal verification instead of model checking. Moreover, instead of full model checking, they often successfully use bounded model checking that explores the state space up to some given depth only, which we will briefly discuss in Section 4.4. As for commercial static analysers, the details of the techniques used in the tools are usually unpublished.

3. Basics of CTL and LTL Model Checking

In the previous section, we have introduced model checking as an automated technique that verifies a system (or its model) against a property specified in some suitable way through a systematic exploration of the state space of the system (or its model). We have said that many different languages can be used for specifying the properties to be verified. We now concentrate on two of these languages that belong among the most often used, namely, the Linear Temporal Logic (Pnueli 1977), abbreviated as LTL, and the Computation Tree Logic (Clarke and Emerson 1981), abbreviated as CTL. We will briefly introduce both of these logics, and we will also mention the main ideas behind the classical LTL and CTL model checking algorithms.

LTL and CTL differ primarily in the underlying notion of *time*. LTL considers time to be *linear*, and it views the behaviour of a system as a set of linear executions. On the other hand, CTL considers time to be *branching*, and it views the behaviour of a system as a tree of gradually branching executions. Both of the logics, however, work with *logical time* only, which allows one to express requirements on the order in which certain states (events) should occur in the system, but unlike *physical time*, it does not allow one to measure how much time elapses between two states (events).

Safety, Liveness, and Fairness. Both LTL and CTL allow one to express various safety as well as liveness properties. Intuitively, *safety properties* state that nothing bad ever happens (and counterexamples to them are finite executions) whereas *liveness properties* state that something good eventually happens (and counterexamples to them are infinite or at least complete executions, i.e., executions that cannot be extended any more). Examples of safety properties can be the following: “A deadlock never happens.” or “The length of a certain buffer never exceeds 5 elements.”. Examples of liveness properties can be: “The program eventually terminates.” or “Each incoming request is eventually handled by a server.”.

In order to verify liveness properties, one usually has to provide some *fairness assumptions* limiting the non-determinism in the system to be verified in order to avoid artificial counterexamples that never happen in practice. This especially concerns the scheduling of concurrently running processes for which one typically wants to state, without having to precisely describe the scheduler used, that a process that is ready to run will not wait for ever.

Two most common notions of fairness are the so-called *weak* and *strong fairness.* Weak fairness assumes that an action that is eventually always enabled must always eventually be taken. Strong fairness assumes that an action that is always eventually enabled must always eventually be taken. The fairness requirements may be stated as a part of the formula to be verified provided that the logic used is powerful enough, which is the case of LTL but not CTL for the above mentioned notions of weak and strong fairness. Alternatively, the verification tool may provide a specialised means for stating whether and which notion of fairness should be used (possibly complemented with specially optimised algorithms for dealing with fairness assumptions).

Kripke Structures. In the section, we assume working with finite-state systems only. Such systems can be expressed or modelled in many different ways. For the purpose of explaining the semantics of LTL and CTL and for describing basics of their model checking algorithms, we will, however, work on the level of the so-called *Kripke structures*, which provide a uniform description of the state spaces of systems to be verified. Let AP be a finite non-empty set of atomic propositions. A (finite) Kripke structure is a 4-tuple $M = (S, S_0, R, L)$ where S is a (finite) set of states, $S_0 \subseteq S$ is a non-empty set of initial states, $R \subseteq S \times S$ is a transition relation between the states, and $L : S \to 2^{AP}$ is a labelling function mapping each state $s \in S$ to the set of atomic propositions $L(s) \subseteq AP$ that are true in s. The atomic propositions are observations about particular states of the system being verified (such as program locations, values of variables, etc.) that a property specification may refer to. For convenience, we assume the transition relation to be such that each state has some successor, i.e., $\forall s \in S\ \exists s' \in S :\ R(s, s')$.

An *execution path* in a Kripke structure $M = (S, S_0, R, L)$ is an infinite sequence of states $\pi = s_0, s_1, \ldots$ such that $s_i \in S$ and $R(s_i, s_{i+1})$ for each $i \geq 0$. For each $s \in S$, we denote by $\Pi(s)$ the set of all execution paths leading from s. For an execution path $\pi = s_0, s_1, s_2, \ldots$ and $i \geq 0$, let $\pi(i) = s_i$. Finally, we define the *suffix* π^i of π starting from the state s_i to be the path $\pi^i = s_i, s_{i+1}, s_{i+2}, \ldots$ for any $i \geq 0$.

3.1 *Linear Temporal Logic (LTL)*

Formulae of LTL (Pnueli 1977) are built from atomic propositions allowing one to refer to relevant aspects of the particular states of the system being verified[1], Boolean constants and connectives (coming from and used in the same way as in propositional logic), and, most importantly, *temporal operators* that allow one to express logical timing requirements. In particular, LTL comes with five temporal operators[2] allowing one to reason about execution paths:

(1) The "*next time*" operator, denoted $\mathbf{X}$ or $\bigcirc$, can be used in a formula $\mathbf{X}\,\varphi$ to stipulate that the property φ should hold for the given path starting from its second state.
(2) The "*finally*" operator, sometimes also pronounced as "eventually" and denoted as $\mathbf{F}$ or $\Diamond$, can be used in a formula $\mathbf{F}\,\varphi$ to state that φ should hold for the given path starting from some of its states.
(3) The "*globally*" operator, sometimes also pronounced as "always" and denoted as $\mathbf{G}$ or $\Box$, can be used in a formula $\mathbf{G}\,\varphi$ to state that φ should hold for the given path invariantly starting from any of its states.
(4) The "*until*" operator, denoted $\mathbf{U}$, can be used in a formula $\varphi\mathbf{U}\psi$ to state that ψ should eventually hold at the given path, and φ should hold until then.
(5) The "*release*" operator, denoted $\mathbf{R}$, can be used in a formula $\varphi\mathbf{R}\psi$ to state that ψ should hold for all prefixes of the given path starting from states prior to which no state at which φ holds has appeared (with no requirement for such a state to appear).

Thus, given a finite non-empty set AP of atomic propositions, assuming $p \in AP$, and using $\top$ and $\bot$ to denote true and false, respectively, LTL formulae can be generated by the following grammar:

$$\varphi ::= \top \mid \bot \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \to \varphi \mid \varphi \leftrightarrow \varphi \mid \mathbf{X}\varphi \mid \mathbf{F}\varphi \mid \mathbf{G}\varphi \mid \varphi\mathbf{U}\varphi \mid \varphi\mathbf{R}\varphi$$

[1]There are, of course, works that concentrate on events instead of states or even both on states as well as events.
[2]Sometimes, more operators are defined, which can, however, be seen as syntactic sugar only. Moreover, as we shall see, three of the five classical operators have the form of syntactic sugar too.

We can now formalise the semantics of LTL formulae. Let M be a Kripke structure over a finite non-empty set of atomic propositions AP. The satisfaction relation $\models$ between execution paths $\pi = s_0, s_1, s_2, \ldots$ of M and LTL formulae is defined as follows (we skip the definition of the meaning of some of the propositional constants and connectives):

$$\begin{array}{lll}
\pi \models p & \textbf{iff} & p \in L(s_0) \\
\pi \models \neg\varphi & \textbf{iff} & \pi \not\models \varphi \\
\pi \models \varphi \vee \psi & \textbf{iff} & \pi \models \varphi \textbf{ or } \pi \models \psi \\
\pi \models \mathbf{X}\,\varphi & \textbf{iff} & \pi^1 \models \varphi \\
\pi \models \mathbf{F}\,\varphi & \textbf{iff} & \exists i \geq 0 :\ \pi^i \models \varphi \\
\pi \models \mathbf{G}\,\varphi & \textbf{iff} & \forall i \geq 0 :\ \pi^i \models \varphi \\
\pi \models \varphi\,\mathbf{U}\,\psi & \textbf{iff} & \exists\, i \geq 0 : (\pi^i \models \psi \ \wedge\ \forall\, 0 \leq j < i : \pi^j \models \varphi) \\
\pi \models \varphi\,\mathbf{R}\,\psi & \textbf{iff} & \forall i \geq 0 : (\forall\, 0 \leq j < i : \pi^j \not\models \varphi) \Rightarrow \pi^i \models \psi
\end{array}$$

Not all of the above introduced LTL operators are, however, needed to achieve the expressive power of LTL. In particular, the expressive power stays the same if one works with the atomic propositions, negation, disjunction, the next time operator, and the until operator only. All the rest can be obtained as syntactic sugar as follows (we again skip some of the propositional constants and connectives that can be expressed as usual):

$$\begin{array}{lll}
\top & \equiv & p \vee \neg p \quad \text{for any } p \in AP \\
\varphi \wedge \psi & \equiv & \neg(\neg\varphi \vee \neg\psi) \\
\mathbf{F}\,\varphi & \equiv & \top\,\mathbf{U}\,\varphi \\
\mathbf{G}\,\varphi & \equiv & \neg\mathbf{F}\,\neg\varphi \\
\varphi\,\mathbf{R}\,\psi & \equiv & \neg(\neg\varphi\,\mathbf{U}\,\neg\psi)
\end{array}$$

So far, we have dealt with satisfaction of LTL formulae with respect to a single execution path only. A given LTL formula φ is defined to be satisfied by a Kripke structure $M = (S, S_0, R, L)$ as a whole, denoted $M \models \varphi$, iff it is satisfied by each execution path starting from any of the initial states of M, i.e., $M \models \varphi$ iff $\forall s_0 \in S_0\ \forall \pi \in \Pi(s_0) :\ \pi \models \varphi$.[1]

3.2 *Automata-Theoretic LTL Model Checking*

One of the most common approaches to LTL model checking is the *automata-theoretic approach* (Vardi and Wolper 1986, Vardi 2007) whose basic idea we will now describe. Assume that we are given a Kripke structure M describing the state space of the system to be verified and an LTL formula φ describing a correctness requirement on the system. We need to check whether any execution path π starting from any initial state of M satisfies φ. To check this using the automata-theoretic framework, one can use *Büchi automata* and/or various other finite automata accepting *infinite words* as follows[2].

First, one constructs a non-deterministic Büchi automaton $A_{\neg\varphi}$ that accepts infinite words corresponding exactly to those execution paths that do *not* satisfy φ.

[1]This definition assumes an implicit universal quantifier over all execution paths starting from all initial states. Sometimes, a dual notion of existential LTL with an implicit existential quantifier over the execution paths starting from initial states is used too.

[2]A Büchi automaton is essentially a finite automaton that accepts infinite words by *looping* through an accepting state. For more information on Büchi automata and various other finite automata accepting infinite words, see, e.g., (Perrin and Pin 2003). In LTL model checking, automata on infinite words are needed to cope with liveness properties. For the case of safety properties, the verification can be simplified, but we prefer a uniform presentation here.

Second, the Kripke structure is converted to a non-deterministic Büchi automaton A_M accepting infinite words corresponding exactly to all the execution paths of M. Third, the product automaton $A_M \otimes A_{\neg\varphi}$, which accepts the intersection of the languages of A_M and $A_{\neg\varphi}$, is constructed. Finally, emptiness of the language of $A_M \otimes A_{\neg\varphi}$ is checked. If the language of the product automaton is empty, the system modelled by M satisfies φ. Otherwise, the language of the product automaton consists of words corresponding exactly to those executions of M that break the specification φ.

The translation from a Kripke structure to a non-deterministic Büchi automaton is easy. Essentially, the Kripke structure M is converted to a non-deterministic Büchi automaton A_M by using 2^{AP} as the alphabet, adding a new initial state with outgoing transitions leading to all initial states of M, moving the labelling of the states of M to their incoming transitions, and making all states accepting. On the other hand, the construction of the non-deterministic Büchi automaton $A_{\neg\varphi}$ is far beyond the scope of this article. We just note that its size may be exponential in the size of the formula: Indeed, LTL model checking is linear in the size of M but PSPACE-complete in the size of φ. However, various heuristics for keeping the size of $A_{\neg\varphi}$ as small as possible have been proposed and implemented in freely available translators from LTL to Büchi automata, such as the LTL2BA tool implementing the translation proposed in (Gastin and Oddoux 2001). Note also that the non-deterministic Büchi automaton is constructed for the *negation* of φ. A construction of the automaton for φ, followed by complementing the automaton, is possible, but it is typically avoided since complementation of non-deterministic Büchi automata is quite complicated and costly.

Construction of the product Büchi automaton is easy—one just has to take care of the fact that the accepting states need not be reached at the same time when looping through A_M and $A_{\neg\varphi}$. It remains to discuss how the emptiness of the product Büchi automaton may be checked. This emptiness check amounts to checking that there is no reachable *accepting loop* in the automaton. For that purpose, one can use various versions of the *Tarjan's algorithm* for computing *strongly connected components* or the *nested depth-first search* (nested DFS), in which the outer DFS is used for finding accepting states while the inner DFS for checking whether a loop exists on the state identified by the outer loop (Holzmann *et al.* 1996, Gaiser and Schwoon 2009).

Finally, let us note that despite LTL model checking is linear in the size of the Kripke structure and exponential in the size of the formula being checked, it is usually the size of the Kripke structure that is problematic. This comes from that the formulae of interest are often very small, but the Kripke structures tend to be very large. Indeed, they represent the state space of the system being verified, which is often exponentially larger than the system itself. To cope with this situation, various heuristics are used.

First of all, one can use the so-called *on-the-fly model checking* (Holzmann 1996) in which the Kripke structure is not generated first and only then explored. Instead, the property Büchi automaton is composed with the Kripke structure during the generation of the latter. This has two advantages. First, one can stop the state space generation as soon as an error is found, possibly avoiding generation of many further states (which can be quite useful since erroneous systems tend to have more states due to they do not observe various invariants they should otherwise observe). Moreover, one can avoid generation of parts of the state space for which it is clear that they do not compose with the property automaton (and hence cannot lead to any error). The state space generation is thus *property driven.* Next, one can use various techniques to store and explore the generated state space efficiently as well as to reduce it by not exploring some of its states whose exploration can be seen redundant with respect to the property being checked and with respect to the other explored states. Many different heuristics have been proposed for this purpose, and we will briefly discuss some of them in Section 4.

3.3 *Computation Tree Logic (CTL)*

Formulae of CTL (Clarke and Emerson 1981) are built from atomic propositions, Boolean constants and connectives, temporal operators, and universal and existential *path quantifiers* **A** and **E**. Compared to LTL, the path quantifiers are added, which is consistent with CTL being a branching time logic. The use of the path quantifiers is, however, restricted in CTL in that the temporal operators and path quantifiers must regularly interleave, leading to ten non-propositional connectives: **AX**, **EX**, **AF**, **EF**, **AG**, **EG**, **AU**, **EU**, **AR**, and **ER**. Thus, provided that $p \in AP$ where AP is a finite non-empty set of atomic propositions, CTL formulae are generated by the following grammar, in which an infix notation is used for **AU**, **EU**, **AR**, and **ER**:

$$\begin{aligned}\Phi ::= {} & \top \mid \bot \mid p \mid \neg\Phi \mid \Phi \vee \Phi \mid \Phi \wedge \Phi \mid \Phi \rightarrow \Phi \mid \Phi \leftrightarrow \Phi \mid \mathbf{AX}\,\Phi \mid \mathbf{EX}\,\Phi \mid \mathbf{AF}\,\Phi \mid \\ & \mathbf{EF}\,\Phi \mid \mathbf{AG}\,\Phi \mid \mathbf{EG}\,\Phi \mid \mathbf{A}\,[\,\Phi\,\mathbf{U}\,\Phi\,] \mid \mathbf{E}\,[\,\Phi\,\mathbf{U}\,\Phi\,] \mid \mathbf{A}\,[\,\Phi\,\mathbf{R}\,\Phi\,] \mid \mathbf{E}\,[\,\Phi\,\mathbf{R}\,\Phi\,]\end{aligned}$$

The satisfaction relation of CTL formulae is defined for each state of a Kripke structure, taking into account all execution paths that originate from that state. Let $M = (S, S_0, R, L)$ be a Kripke structure over a non-empty finite set of atomic propositions AP. The satisfaction relation $\models$ between states $s \in S$ and CTL formulae is defined as follows (for brevity, we skip the definition of the meaning of some of the propositional constants and connectives):

$$\begin{array}{lll}
s \models p & \textbf{iff} & p \in L(s) \\
s \models \neg\Phi & \textbf{iff} & s \not\models \Phi \\
s \models \Phi \vee \Psi & \textbf{iff} & s \models \Phi \textbf{ or } s \models \Psi \\
s \models \mathbf{AX}\,\Phi & \textbf{iff} & \forall \pi \in \Pi(s) : \pi(1) \models \Phi \\
s \models \mathbf{EX}\,\Phi & \textbf{iff} & \exists \pi \in \Pi(s) : \pi(1) \models \Phi \\
s \models \mathbf{AF}\,\Phi & \textbf{iff} & \forall \pi \in \Pi(s)\, \exists i \geq 0 : \pi(i) \models \Phi \\
s \models \mathbf{EF}\,\Phi & \textbf{iff} & \exists \pi \in \Pi(s)\, \exists i \geq 0 : \pi(i) \models \Phi \\
s \models \mathbf{AG}\,\Phi & \textbf{iff} & \forall \pi \in \Pi(s)\, \forall i \geq 0 : \pi(i) \models \Phi \\
s \models \mathbf{EG}\,\Phi & \textbf{iff} & \exists \pi \in \Pi(s)\, \forall i \geq 0 : \pi(i) \models \Phi \\
s \models \mathbf{A}\,[\,\Phi\,\mathbf{U}\,\Psi\,] & \textbf{iff} & \forall \pi \in \Pi(s)\, \exists i \geq 0 : (\pi(i) \models \Psi \ \wedge\ \forall\, 0 \leq j < i : \pi(j) \models \Phi) \\
s \models \mathbf{E}\,[\,\Phi\,\mathbf{U}\,\Psi\,] & \textbf{iff} & \exists \pi \in \Pi(s)\, \exists i \geq 0 : (\pi(i) \models \Psi \ \wedge\ \forall\, 0 \leq j < i : \pi(j) \models \Phi) \\
s \models \mathbf{A}\,[\,\Phi\,\mathbf{R}\,\Psi\,] & \textbf{iff} & \forall \pi \in \Pi(s) :\ \forall i \geq 0 : (\forall\, 0 \leq j < i : \pi(j) \not\models \Phi) \rightarrow \pi(i) \models \Psi \\
s \models \mathbf{E}\,[\,\Phi\,\mathbf{R}\,\Psi\,] & \textbf{iff} & \exists \pi \in \Pi(s) :\ \forall i \geq 0 : (\forall\, 0 \leq j < i : \pi(j) \not\models \Phi) \rightarrow \pi(i) \models \Psi
\end{array}$$

As in the case of LTL, the above introduced set of connectives is not minimal. In particular, to preserve the expressive power of CTL, it is enough to use atomic propositions, negation, disjunction, and the following three non-propositional connectives: **EX**, **EG**, and **EU**. The meaning of the other non-propositional connectives can be obtained as follows:

$$\begin{array}{lll}
\mathbf{AX}\,\Phi & \equiv & \neg\mathbf{EX}\,\neg\Phi \\
\mathbf{EF}\,\Phi & \equiv & \mathbf{E}\,[\,\top\,\mathbf{U}\,\Phi\,] \\
\mathbf{AG}\,\Phi & \equiv & \neg\mathbf{EF}\,\neg\Phi \\
\mathbf{AF}\,\Phi & \equiv & \neg\mathbf{EG}\,\neg\Phi \\
\mathbf{A}\,[\,\Phi\,\mathbf{U}\,\Psi\,] & \equiv & \neg(\mathbf{E}\,[\,\neg\Psi\,\mathbf{U}\,\neg(\Phi \vee \Psi)\,] \ \vee\ \mathbf{EG}\neg\Psi) \\
\mathbf{A}\,[\,\Phi\,\mathbf{R}\,\Psi\,] & \equiv & \neg\mathbf{E}\,[\,\neg\Phi\,\mathbf{U}\,\neg\Psi\,] \\
\mathbf{E}\,[\,\Phi\,\mathbf{R}\,\Psi\,] & \equiv & \neg\mathbf{A}\,[\,\neg\Phi\,\mathbf{U}\,\neg\Psi\,]
\end{array}$$

For a given Kripke structure M and a CTL formula Φ, we can now define the so-called *satisfaction set* $Sat(\Phi) = \{s \in S \mid s \models \Phi\}$, i.e., the set of all the states of M that satisfy Φ. Then, we say that M satisfies Φ, denoted $M \models \Phi$, iff $S_0 \subseteq Sat(\Phi)$, i.e., all initial states of M satisfy Φ.

3.4 *Explicit-State CTL Model Checking*

Let $M = (S, S_0, R, L)$ be a Kripke structure over a non-empty finite set of atomic propositions AP, and let Φ be a CTL formula over AP. The basic algorithm of CTL model checking (i.e., checking whether $M \models \Phi$ holds) is rather straightforward. It consists in computing the satisfaction sets for all sub-formulae of Φ, followed by checking that the set S_0 of initial states of M is included in $Sat(\Phi)$. In particular, the algorithm starts by computing the satisfaction sets for the atomic propositions that appear in Φ and then proceeds in a bottom-up fashion through the structure of Φ, using the following rules (we provide the rules for the minimum needed set of CTL operators only):

- $Sat(p) = \{s \in S \mid p \in L(s)\}$ for any $p \in AP$,
- $Sat(\neg\Phi) = S \setminus Sat(\Phi)$,
- $Sat(\Phi \vee \Psi) = Sat(\Phi) \cup Sat(\Psi)$,
- $Sat(\mathbf{EX}\,\Phi) = \{s \in S \mid \exists s' \in Sat(\Phi) : R(s, s')\}$,
- $Sat(\mathbf{E}\,[\,\Phi\,\mathbf{U}\,\Psi\,])$ is the smallest set $T \subseteq S$ such that
  (1) $Sat(\Psi) \subseteq T$ and
  (2) $(s \in Sat(\Phi) \;\wedge\; \exists s' \in T : R(s, s')) \;\rightarrow\; s \in T$,
- $Sat(\mathbf{EG}\,\Phi)$ is the largest set $T \subseteq S$ such that
  (1) $T \subseteq Sat(\Phi)$ and
  (2) $s \in T \;\rightarrow\; \exists s' \in T : R(s, s')$.

Most of the above rules are rather straightforward: Atomic propositions hold in the states that are labelled by them, negation corresponds to complementing the satisfaction set, disjunction to taking the union, and $\mathbf{EX}\,\Phi$ to going one step back from the states satisfying Φ. The rules for handling **EU** and **EG** are a bit more involved, but not too much.

Indeed, $Sat(\mathbf{E}\,[\,\Phi\,\mathbf{U}\,\Psi\,])$ can be obtained by computing the smallest fixpoint of the equation $T = Sat(\Psi) \cup \{s \in Sat(\Phi) \mid \exists s' \in T : R(s, s')\}$, which can be obtained by letting $T := Sat(\Psi)$ and then iterating the transformation $T := T \cup \{s \in Sat(\Phi) \mid \exists s' \in T : R(s, s')\}$ until T stops changing. The final value of T then gives the desired satisfaction set.[1] Intuitively, this means that one starts from the states satisfying Ψ and then proceeds backwards through M while passing through states satisfying Φ only.

Similarly, $Sat(\mathbf{EG}\,\Phi)$ can also be obtained quite easily as the greatest fixpoint of the equation $T = \{s \in Sat(\Phi) \mid \exists s' \in T : R(s, s')\}$, which can be obtained by letting $T := Sat(\Phi)$ and then iterating the transformation $T := \{s \in T \mid \exists s' \in T : R(s, s')\}$ until the set T stops changing. The final value of T then gives the desired satisfaction set.[2]

Moreover, for computing $Sat(\mathbf{EG}\,\Phi)$, one can also build on the notion of non-trivial strongly connected components (containing more than one state or at least one state with a self-loop). In particular, $Sat(\mathbf{EG}\,\Phi)$ can then be computed by restricting M to its sub-structure consisting of states in $Sat(\Phi)$ only, applying the

[1]This computation is, in fact, based on the expansion law $\mathbf{E}\,[\,\Phi\,\mathbf{U}\,\Psi\,] \;\equiv\; \Psi \vee (\Phi \wedge \mathbf{EX}\,\mathbf{E}\,[\,\Phi\,\mathbf{U}\,\Psi\,])$.
[2]This computation is based on the expansion law $\mathbf{EG}\,\Phi \;\equiv\; \Phi \wedge \mathbf{EX}\,\mathbf{EG}\,\Phi$.

Tarjan's algorithm to compute non-trivial strongly-connected components of the sub-structure, using states of these components as the initial approximation of the sought satisfaction set, and then extending them by all the states that can be reached from them by going backwards through the sub-structure.

Clearly, the time complexity of the above described approach to CTL model checking is $\mathcal{O}((|S| + |R|) \cdot |\Phi|)$. However, since the state spaces of practical systems tend to be extremely large, one still has to fight with the state space explosion problem here too, using some of the heuristics for efficient state space generation and storage or state space reduction briefly discussed in Section 4.

3.5 *Expressiveness of LTL and CTL*

Although many relevant properties of systems to be verified can be specified both in LTL and CTL, the expressive power of these logics is, in fact, incomparable. Some properties expressible in LTL (e.g., $\mathbf{FG}\, p$) cannot be expressed in CTL while some properties expressible in CTL (e.g., $\mathbf{AG}\,\mathbf{EF}\, p$) cannot be expressed in LTL. The *CTL* logic* proposed in (Emerson and Halpern 1986) combines features of both LTL and CTL and is more expressive than any of them. The syntax of CTL* is easy to obtain from that of CTL by lifting the restriction on the regular interleaving of path quantifiers and temporal connectives. A commonly used temporal logic that is even more expressive than CTL* is then the *modal μ-calculus* (Kozen 1983), which is based on allowing one to explicitly use least/greatest fixpoint operators on sets of states (thus essentially allowing one to define new specialised modalities).

4. State Space Explosion in Model Checking

As we have already mentioned above, the *state space explosion problem*, which is one of the main problems to face in model checking, means the need to deal with an exponential growth of the number of reachable states in the size of the examined systems. The state explosion stems from a huge number of possible interleavings of concurrently running processes (a system with n processes, each having k states, may have up to k^n reachable states) and/or from a huge number of possible data values that may be handled by the system to be verified (indeed, a single 32-bit-wide integer variable can have 2^{32} possible values, n of such variables then $2^{32.n}$ possible values). To cope with the state space explosion problem, many different heuristics have been proposed (Valmari 1998, Clarke *et al.* 1999, Baier and Katoen 2008). We briefly overview some of the most important of them below.

4.1 *Compact State Space Storage*

The first possible approach to cope with the state space explosion problem is to store state spaces as *compactly* as possible. One way to achieve compactness is to store state spaces in a *hierarchical way*. This approach can be used in the quite frequent case when the particular states have some internal hierarchical structure. For instance, it may be the case that a state of a system consists of states of various processes existing in that state. The states of processes may in turn consist of states of threads running within these processes. Then, a lot of space is waisted when an entire new state vector is created whenever the state of a single thread changes. Instead, one can store state vectors as vectors of pointers to states of processes, which, similarly, can have the form of vectors of pointers to states of threads. Now, when the state of a single thread changes, one has to only create a new vector of pointers for the global state, a new vector of pointers for the state of the concerned process, and a new state vector for the thread whose state changed. This is much more space efficient than making a new copy of the unchanged states of all existing threads. A hierarchical storage of states has thus been used in many model checkers, such as Spin (Holzmann 1997) or Java PathFinder (Visser *et al.* 2003).

Another approach is to use the so-called *symbolic verification* that, instead of exploring individual states one-by-one, works with sets of states at the same time. These sets must, of course, be encoded in a way allowing for an efficient manipulation with them. The most famous symbolic verification method is probably the one based on *binary decision diagrams* (Bryant 1986, Burch *et al.* 1992), commonly abbreviated as BDDs. The use of BDDs is behind many successes of model checking, especially in hardware.

BDDs provide a (usually) very compact and canonical representation of Boolean functions (i.e., functions of the form $\{0,1\}^k \longrightarrow \{0,1\}$, $k \geq 0$), corresponding to propositional formulae (possibly representing finite sets or relations). BDDs have a form of rooted, directed, connected, acyclic graphs that consist of internal Boolean decision nodes and terminal Boolean result nodes. BDDs may be viewed to arise from Boolean decision trees by removing redundancies from them by merging isomorphic sub-trees and removing useless nodes with identical children. An illustration of the use of BDDs for representing a state space may be found in Fig. 1 where a, b, and c may be viewed to refer to particular bits of state vectors of some system being verified, and the leaves say whether the appropriate state vector is or is not in the reachable state space. When using BDDs, it may actually happen that larger state spaces require less storage space than smaller ones since they can contain more redundancies (which, however, depends on many factors, including the density of the state space and the choice of ordering of the particular bits of state vectors).

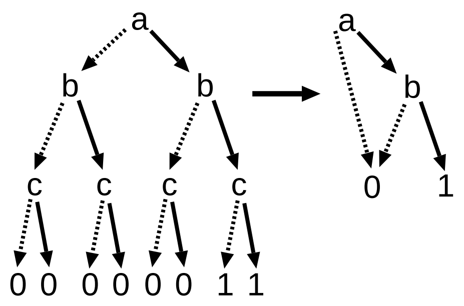

Figure 1. Binary decision diagrams

When using BDDs for symbolic model checking, one needs to use BDDs not only to represent sets of reachable states but also the transition relation of the system being examined. Moreover, one needs to be able to compute satisfaction sets of temporal formulae purely on the level of BDDs (and hence, equivalently, propositional formulae). For that, fixpoint characterisations of temporal operators, such as those used in Section 3.4, are useful since they can be easily written as (quantified) propositional formulae.

Sometimes, the compact representation of state spaces may disregard some information in which case soundness of model checking is deliberately sacrificed in order to achieve efficiency. This way, an error detection method with formal verification roots is obtained. An example of such an approach is the *bit-state hashing method* (Holzmann 1987) where different reachable states of the system being verified are not distinguished when they have the same hash value (alternatively, several different hash functions can be used, not distinguishing states only when they are hashed to the same value by all the functions). This approach is used, for instance, in the Spin model checker (Holzmann 1997). In particular, Spin uses two hash functions, and it also provides the user with an estimation of how well the state space was probably covered based on the so-called hash factor (computed as the size of the hash-table divided by the number of stored states).

4.2 *State Space Reductions*

The goal of *state space reduction techniques* is to avoid generation and exploration of states for which it is clear that their properties are not important with respect to the specification being checked, or their properties are covered by the properties of other explored states. We are by far, of course, not able to describe all the numerous state space reduction techniques studied in the literature, and so we will briefly present just three of them that are quite commonly used.

One of the commonly used ways to reduce the generated and explored portion of the state space of a system being verified is the *on-the-fly model checking* (Holzmann 1996) when the property to be examined is checked in parallel with the state space generation. As we have already mentioned in Section 3.2, this approach builds on the fact that the state space generation can sometimes be driven by the property being checked. For instance, when using an automata-theoretic approach to model checking, one can avoid generation and exploration of certain paths in the state space once it is clear from their already generated prefixes that they cannot be accepted by the automaton describing undesirable behaviours. Moreover, one can avoid not only the generation of some unimportant states with respect to the property being checked, but one can also stop the state space generation and exploration as soon as an error is found without having to generate many further reachable states.

Another commonly used state space reduction is the so-called *symmetry reduction*—cf., e.g., (Clarke *et al.* 1993, Ip and Dill 1996a, Sistla *et al.* 1997). Intuitively, this approach builds on identifying sets of the so-called *symmetrical states* that can be made identical by exchanging the roles of some of their components. Out of the sets of symmetrical states, it is then enough to explore just one representative state. One can, e.g., rotate the philosophers in the well-known dining philosophers problem since it is not really important whether the first one is eating and the others are thinking, or the third one is eating and the others thinking, and so on (cf. Fig. 2). Clearly, similar situations are likely to arise in many practical scenarios when dealing with replicated components, objects, processes, and the like. The use of symmetries is especially important in the context of verification of systems with dynamically instantiated processes or objects (Visser *et al.* 2003, Češka *et al.* 2001) where the use of fresh identifiers of processes or objects being regularly destroyed and created may cause a very significant state explosion. Of course, a problem that must be addressed when applying symmetries is how to detect symmetrical states efficiently, or, even better, how to transform states into a canonical form in order not to loose too much time by testing whether some states are or are not symmetrical (which is to be performed every time a potentially new state is generated).

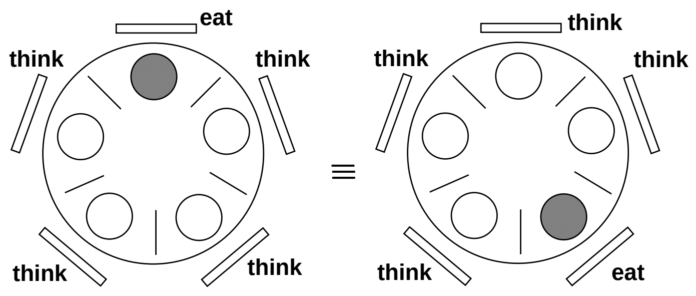

Figure 2. Symmetries and the dining philosophers

Another very important state space reduction, particularly successful in model checking of concurrent systems, is the so-called *partial-order reduction* or *commutativity-based reduction* (Valmari 1988, Katz and Peled 1988, Godefroid 1991). This reduction aims at reducing the amount of interleaving of concurrently enabled actions. In particular, if some concurrently enabled actions are *independent* (i.e., they do not mutually influence the enabledness and the effect of each other) and, moreover, the order of executing the actions is *not visible* through the property being checked (i.e., the order of executing the actions is not important for the validity of the property), the actions are fired in one of the possible orders only. This is illustrated in Fig. 3 where the independent, concurrently enabled actions correspond to the edges drawn using different styles of lines. The main practical issue with partial-order reduction is how to efficiently decide which actions are independent in order for the overhead of the reduction not to be higher than what it brings. For this purpose, various heuristics tailored for different classes of verified systems have been proposed and are in use in model checkers such as Spin (Holzmann 1997) or Java PathFinder (Visser *et al.* 2003).

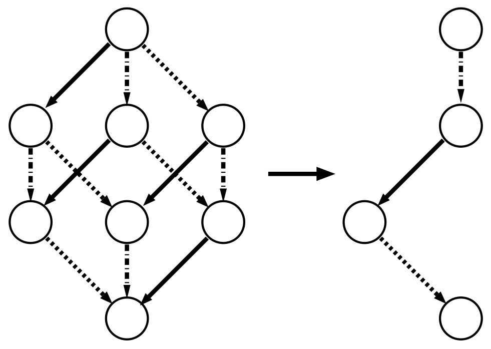
Figure 3. Partial-order reduction

4.3 *Abstraction*

When *abstraction* is combined with model checking (Clarke *et al.* 1994), only some selected aspects of states are tracked, hopefully yielding a smaller abstract state space to be explored. It is preferable that the abstraction is constructed automatically and possibly gradually refined in the so-called *counterexample-guided abstraction refinement* (CEGAR) loop (Clarke *et al.* 2000). CEGAR is based on excluding artificial counterexamples to the properties being verified, stemming from using a too coarse abstraction.[1]

Among the abstractions used in model checking, perhaps the most successful is the so-called *predicate abstraction* (Graf and Saïdi 1997), heavily used especially in software model checking. Intuitively, model checkers based on predicate abstraction do not track the exact reachable states of the system being verified, but only some *predicates* about the states. For instance, instead of remembering that a state in which the variable x has value 100 and the variable y has value 10 is reachable, they may remember that a state where $x > y$ is reachable only.

To find out how the validity of the predicates tracked by predicate abstraction changes in response to the transitions being fired, one can use specialised decision procedures or theorem provers operated in a fully automated way. The abstraction may either be applied before a state space exploration is started (Ball *et al.* 2001), or it is constructed on-the-fly during the state space exploration (Henzinger *et al.* 2002). An important issue in predicate abstraction is then how to choose the new predicates to be learnt from spurious counterexamples. For this, the so-called *Craig interpolation* known from logic is often used (McMillan 2005). Intuitively, Craig interpolation, when given a spurious counterexample path through a program (encoded as a formula), produces an abstract reason (encoded again as a formula) for the path not to be feasible at each particular line of the program through which the path is supposed to go through (which is then used as a source of new predicates to be tracked).

[1]The use of CEGAR, allowing the answers "yes, the property to be verified holds" or "no, the property does not hold" instead of "the property is possibly broken", often distinguishes approaches based on model checking from various static analyses using a fixed abstraction.

Predicate abstraction is at the core of many well-known software model checkers, such as Blast (Henzinger *et al.* 2003), CPAChecker (Beyer and Keremoglu 2011), SatAbs (Clarke *et al.* 2005), Wolverine (Kroening and Weissenbacher 2011), ARMC (Rybalchenko 2011), or the commercial Static Driver Verifier from Microsoft, developed on the basis of the SLAM model checker (Ball and Rajamani 2001). Although predicate abstraction is mostly used for checking safety properties, its variant for verifying liveness properties has also appeared (Podelski and Rybalchenko 2005).

4.4 *Bounded Model Checking*

A further way to cope with the state space explosion problem is to bound the state space exploration in some way. As in the case of the above mentioned bit-state hashing, this usually results in an unsound technique which can, however, be still very well used as a systematic testing approach.

One natural way how the state space exploration can be bounded is to restrict the depth of the state space exploration, leading to the classical notion of *bounded model checking* (Biere *et al.* 2003). In this case, the behaviour of the system is unfolded some given number of steps. An important observation is that the unfolded behaviour can easily be captured by a formula that can subsequently be conjoined with another formula describing undesirable behaviours, and the constantly increasing power of various SAT or SMT solvers can then be utilised to check satisfiability of the resulting formula. This way, one can reliably check whether an error can be reached within a given number of steps or not. Moreover, there are even ways how to make the approach complete (based, e.g., on automated induction (Sheeran *et al.* 2000)).

Bounded model checking in the form described above has become very popular and is widely applied even in the industry, especially in the area of hardware verification. It has, however, been applied even in software verification, e.g., in tools like CBMC (Clarke *et al.* 2004a) or LLBMC (Merz *et al.* 2012). A closely related approach is then used in tools such as KLEE (Cadar *et al.* 2008) under the name of *symbolic execution*[1]. The difference is that in bounded model checking, a single formula describing all runs up to some length is constructed. In symbolic execution, a separate *path formula* for each path in the program of a length up to some bound is examined. A path formula naturally describes in a chosen logic all the conditions that appear on the considered program path as well as the effect of each encountered statement.

A different approach of how to bound the state space exploration within model checking is to *bound the number of context switches* that are allowed to happen when verifying concurrent programs. This idea is motivated by the fact that many errors in concurrent programs do manifest already with a few processes (or threads) and a few context switches. The idea has been quite successfully implemented in tools for systematic (or deterministic) testing of concurrent programs such as CHESS (Musuvathi and Qadeer 2007). These tools try to record the already witnessed testing runs and then to force those that have not yet been seen until some given bound on the number of context switches is reached.

[1]In the literature, the term "symbolic execution" is used in other meanings than the above too. In particular, it is also used to denote execution of a system on the level of sets of its configurations described using a suitable formalism.

4.5 *Other Approaches to State Space Explosion*

Other approaches to the state space explosion problem in model checking include *modular verification* and the so-called *assume-guarantee reasoning* (Pnueli 1985, Clarke *et al.* 1989, Chaki *et al.* 2005, Alur *et al.* 2005) in which the system is verified part-by-part. The assumptions on the behaviour of various parts of a system needed for its modular verification can be provided manually or automatically learnt from sample behaviours of the system.

Yet another approach is to use *more computational power* in the form of distributed computing environments, external storage (instead of RAM), multi-processor systems, graphics processing units (GPUs), and so on. For examples of such approaches, see, e.g., (Holzmann and Bošnački 2007, Barnat *et al.* 2009, 2010a,b, Blom *et al.* 2010).

Finally, various *combinations* of (bounded) model checking and other approaches are possible. For instance, dynamic analysis may be used to detect possible defects in a program and to partially record a behaviour witnessing such a defect. The partially recorded behaviour can then be reconstructed in a model checker. Subsequently, bounded model checking in the neighbourhood of the behaviour can be used to check whether there is really an error in the system or not (Hrubá *et al.* 2009).

5. Model Checking and Infinite-State Systems

Model checking of systems with infinitely many states is usually even more demanding than that of systems with large but finite state spaces. Indeed, most verification problems for infinite-state systems are undecidable. However, dealing with infinite-state systems is common in practice due to using various *unbounded data and control structures* (such as queues, stacks, counters, dynamic linked data structures, or unrestricted dynamic creation of processes, objects, etc.) or due to using *parameterised* designs. That is why techniques for model checking of such systems are also highly needed and studied. In their case, however, one clearly cannot use a systematic enumeration of all individual reachable states.

One possibility of verifying infinite-state systems via model checking is to use the so-called *cut-offs*—cf., e.g., (German and Sistla 1992, Emerson and Namjoshi 1996, Emerson and Kahlon 2004, Clarke *et al.* 2004b, Emerson and Kahlon 2000, 2002, Kahlon *et al.* 2005, Bouajjani *et al.* 2006b, Kaiser *et al.* 2010). Cut-offs are such bounds on the various infinite resources present in the system being verified that once the property of interest is successfully verified with the sources of infinity limited by the cut-off bounds, the property is guaranteed to hold in general. If one can find a suitable cut-off, infinite-state model checking can be reduced to finite-state model checking with the possibility of using all the optimisations discussed in Section 4. On the other hand, cut-offs are usually highly specialised to only certain classes of systems and their properties. Moreover, sometimes, even if some cut-offs are known, they are so huge that they are not really applicable in practice.

Another possibility to apply model checking on infinite-state systems is to use various kinds of (finite-range) *abstractions*. The abstractions considered in the literature range from predicate abstraction (Graf and Saïdi 1997) discussed already in Section 4.3 (indeed, when we use a finite number of Boolean predicates, the abstract state space becomes finite regardless of the original domains of the concrete state variables) to various specialised abstractions proposed, for instance, for verifying parameterised networks of processes (Ip and Dill 1996b, Baukus *et al.* 2000, Pnueli *et al.* 2002).

Further, one may use *symbolic model checking* based on some kind of a finite representation of infinite sets of states by means of logics, automata, grammars, or another suitable formalism. An example of such an approach is the so-called *regular model checking*, cf., e.g., (Kesten *et al.* 1997, Pnueli and Shahar 2000, Bouajjani *et al.* 2000, Abdulla *et al.* 2002, Boigelot *et al.* 2003, Bouajjani *et al.* 2004, Abdulla 2012). Regular model checking is a generic automata-based framework for verification of infinite-state systems with linear or tree-like configurations that uses finite (word, tree, or infinite word) automata to finitely represent potentially infinite sets of reachable configurations of the systems being verified.[1] The technique has been successfully applied for verification of a wide range of infinite-state systems, including systems with unbounded counters, queues, stacks, parameters as well as complex dynamic linked data structures (Bouajjani *et al.* 2006a, Habermehl *et al.* 2011). Another successful symbolic verification is then, for instance, the symbolic model checking approach based on the so-called *zones* (Dill 1989, Henzinger *et al.* 1994) that has turned out to be very successful in the domain of model checking *real-time systems* modelled by *timed automata* (Alur and Dill 1994, Henzinger *et al.* 1994).

Yet another group of often studied approaches is based on various ways of *automated induction*—cf., e.g., (Wolper and Lovinfosse 1989, Kurshan and McMillan 1995, McMillan *et al.* 2000, Lesens *et al.* 1997, Creese and Roscoe 2000, Pnueli *et al.* 2001). Many of these works use the so-called *network invariants* which provide an abstraction of a composition of any number of processes. It then suffices to use model checking to verify that (1) the behaviour of a single process is covered by the network invariant, (2) a composition of the network invariant and a process is also covered by the invariant, and (3) the invariant satisfies the given property of interest.

Of course, when dealing with infinite-state systems, as we have already said, one very quickly reaches *undecidability* of most verification problems of interest. The same in particular holds for verification of parametric systems (Apt and Kozen 1986). Therefore, most of the model checking methods proposed for verification of infinite-state and parametric systems (including the methods discussed above) are either *not fully automated*, or they are *semi-algorithmic heuristics*, i.e., they do not guarantee termination, or they allow an indefinite answer of the type "don't know" to be returned. Note, however, that some verification problems are decidable even over infinite-state systems. This is, e.g., the case of many model checking problems over *push-down systems* (Walukiewicz 1996, Burkart and Steffen 1997, Bouajjani *et al.* 1997, Finkel *et al.* 1997, Esparza *et al.* 2000), *lossy FIFO channel systems* (Finkel 1994, Cécé *et al.* 1996, Abdulla and Jonsson 1996, Masson and Schnoebelen 2002), *timed automata* (Alur and Dill 1994, Alur *et al.* 1993), various *dynamic networks of concurrent processes with recursion* (Mayr 2000, Bouajjani *et al.* 2005), or the various scenarios in which some cut-offs have been found. However, it may sometimes be more advantageous to use semi-algorithmic heuristic approaches even in these cases since the heuristics may turn out to deliver more efficient results in practice.

6. Conclusion

We have presented multiple techniques for formal analysis and verification, including, in particular, various forms of theorem proving, static analysis, as well as model checking. The description was mostly informal, and, given the width of the

[1]Sometimes even systems with more general topologies of states may be handled provided that the non-word/non-tree links present in the states can be encoded over words or trees using suitable alphabet symbols (such as pairs of from/to marker symbols for encoding loops).

subject and the space of a typical article, omitted many details and alternatives of the presented approaches. Nevertheless, many references have been provided for works where further details can be found.

Despite the immense amount of work that has been invested into the development of the different techniques of formal analysis and verification, there is still a lot of space for improving their generality, efficiency, and degree of automation. Works going in these directions are constantly appearing and are likely to appear in the future too since the interest in all forms of automated verification is currently very high, and it is likely that the interest will further grow due to the ever increasing impact of computer systems on human lives.

Acknowledgements

This work was supported by the Czech Science Foundation (project No. P103/10/0306), the Czech Ministry of Education (projects COST OC10009 and MSM 0021630528), the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070, and Brno University of Technology (project FIT-S-12-1).

Notes on Contributors

Bohuslav Křena is an assistant professor at the Faculty of Information Technology of the Brno University of Technology. His research focuses on formal analysis and verification, especially on exploiting symbolic execution and on analysis of concurrent programs. He received his Ph.D. at the Faculty of Information Technology of the Brno University of Technology in 2004. In 2002, he visited the Central Laboratory for Parallel Processing of the Bulgarian Academy of Science, Sofia, Bulgaria; in 2003, he visited the Edinburgh Parallel Computing Centre at the University of Edinburgh in Scotland; and in 2004 and 2005, he worked as a researcher at the Software Testing and Analysis Laboratory of the Università degli Studi di Milano-Bicocca, Milano, Italy. Since 2004, he works at FIT BUT.

Tomáš Vojnar is a full professor at the Faculty of Information Technology of the Brno University of Technology (FIT BUT). His research focuses on computer-aided verification, including, in particular, automata-based and logic-based symbolic formal verification of infinite-state systems (especially programs with dynamic linked data structures); model checking, dynamic analysis, and intelligent testing of concurrent programs; as well as formal verification of modern hardware designs. He received his Ph.D. at the Faculty of Electrical Engineering and Computer Science of the Brno University of Technology in 2001. In 2001–03, he worked as a post-doc researcher at LIAFA, Université Paris Diderot/CNRS in France. Since 2003, he works at FIT BUT. He defended his habilitation thesis in 2007 and became a full professor of computer science and engineering in 2012.

References

Abdulla, P.A., (ed.), A Special Issue of the STTT Journal on Regular Model Checking. International Journal on Software Tools for Technology Transfer (STTT), 14(2), Springer-Verlag, 2012.

Abdulla, P.A., d'Orso, J., Jonsson, B. and Nilsson, M., 2002. Regular Model Checking Made Simple and Efficient. Vol. 2421 of *LNCS* Springer-Verlag.

Abdulla, P.A. and Jonsson, B., 1996. Verifying Programs with Unreliable Channels. *Information and Computation*, 127 (2).

Aggarwal, P., Chu, D., Kadamby, V. and Singha, V., 2011. End-to-End Formal using Abstractions to Maximize Coverage (Invited Tutorial). ACM and IEEE.

Aiken, A., 1999. Introduction to Set Constraint-based Program Analysis. *Science of Computer Programming*, 35 (2-3).

Alur, R., Courcoubetis, C. and Dill, D.L., 1993. Model-Checking in Dense Real-Time. *Information and Computation*, 104 First appeared in Proc. of LICS'90.

Alur, R. and Dill, D.L., 1994. A Theory of Timed Automata. *Theoretical Computer Science*, 126 First appeared in Proc. of ICALP'90.

Alur, R., Madhusudan, P. and Nam, W., 2005. Symbolic Compositional Verification by Learning Assumptions. Vol. 3576 of *LNCS* Springer-Verlag.

Apt, K. and Kozen, D., 1986. Limits for Automatic Verification of Finite-State Concurrent Systems. *Information Processing Letters*, 22 (6).

Audemard, G. and Simon, L., 2009. Predicting Learnt Clauses Quality in Modern SAT Solver. IJCAI and AAAI.

Baier, C. and Katoen, J.P., 2008. *Principles of Model Checking*. MIT Press.

Ball, T., Cook, B., Levin, V. and Rajamani, S.K., SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft, 2004, Technical report MSR-TR-2004-08, Microsoft.

Ball, T., Podelski, A. and Rajamani, S.K., 2001. Boolean and Cartesian Abstractions for Model Checking C Programs. Vol. 2031 of *LNCS* Springer-Verlag.

Ball, T. and Rajamani, S.K., 2001. The SLAM Toolkit. Vol. 2102 of *LNCS* Springer-Verlag.

Barnat, J., Bauch, P., Brim, L. and Češka, M., 2010a. Employing Multiple CUDA Devices to Accelerate LTL Model Checking. IEEE CS.

Barnat, J., Brim, L., Češka, M. and Ročkai, P., 2010b. DiVinE: Parallel Distributed Model Checker. IEEE CS.

Barnat, J., Brim, L. and Šimeček, P., 2009. Cluster-Based I/O-Efficient LTL Model Checking. IEEE CS.

Baukus, K., Bensalem, S., Lakhnech, Y. and Stahl, K., 2000. Abstracting WS1S Systems to Verify Prameterized Networks. Vol. 1785 of *LNCS* Springer-Verlag.

Behrmann, G., David, A. and Larsen, K.G., 2004. A Tutorial on Uppaal. Vol. 3185 of *LNCS* Springer-Verlag.

Bertot, Y. and Castéran, P., 2004. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer-Verlag.

Beyer, D. and Keremoglu, M.E., 2011. CPAchecker: A Tool for Configurable Software Verification. Vol. 6806 of *LNCS* Springer-Verlag.

Biere, A., *et al.*, 2003. Bounded Model Checking. *Advances in Computers*, 58.

Blom, S., van de Pol, J. and Weber, M., 2010. LTSMIN: Distributed and Symbolic Reachability. Vol. 6174 of *LNCS* Springer-Verlag.

Boigelot, B., Legay, A. and Wolper, P., 2003. Iterating Transducers in the Large. Vol. 2725 of *LNCS* Springer-Verlag.

Bouajjani, A., Esparza, J. and Maler, O., 1997. Reachability Analysis of Pushdown Automata: Application to Model-Checking. LNCS Springer.

Bouajjani, A., Habermehl, P., Rogalewicz, A. and Vojnar, T., 2006a. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. Vol. 4134 of *LNCS* Springer-Verlag.

Bouajjani, A., Habermehl, P. and Vojnar, T., 2004. Abstract Regular Model Checking. Vol. 3114 of *LNCS* Springer-Verlag.

Bouajjani, A., Habermehl, P. and Vojnar, T., 2006b. Verification of Parametric Concurrent Systems with Prioritized FIFO Resource Management. *Formal Methods in Systems Design*, 32.

Bouajjani, A., Jonsson, B., Nilsson, M. and Touili, T., 2000. Regular Model Checking. Vol. 1855 of *LNCS* Springer-Verlag.

Bouajjani, A., Mueller-Olm, M. and Touili, T., 2005. Regular Symbolic Analysis of Dynamic Networks of Pushdown Systems. Vol. 3653 of *LNCS* Springer-Verlag.

Brayton, R. and Mishchenko, A., 2010. ABC: An Academic Industrial-Strength Verification Tool. Vol. 6174 of *LNCS* Springer-Verlag.

Bryant, R.E., 1986. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35 (8).

Burch, J.R., *et al.*, 1992. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98 (2).

Burkart, O. and Steffen, B., 1997. Model Checking the Full Modal mu-Calculus for Infinite Sequential Processes. Vol. 1256 of *LNCS* Springer-Verlag.

Cadar, C., Dunbar, D. and Engler, D., 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. USENIX Association.

Cavada, R., *et al.*, Chapter title. *NuSMV 2.5 User Manual*, Fondazione Bruno Kessler, 2010.

Cécé, G., Finkel, A. and Iyer, S.P., 1996. Unreliable Channels Are Easier to Verify Than Perfect Channels. *Information and Computation*, 141 (1).

Češka, M., Janoušek, V. and Vojnar, T., 2001. Generating and Using State Spaces of Object-Oriented Petri Nets. *International Journal of Computer Systems Science and Engineering*, 16 (3).

Chaki, S., Clarke, E., Sinha, N. and Thati, P., 2005. Automated Assume-Guarantee Reasoning for Simulation Conformance. Vol. 3576 of *LNCS* Springer-Verlag.

Clarke, E.M. and Emerson, E.A., 1981. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. Vol. 131 of *LNCS* Springer-Verlag.

Clarke, E.M., Filkorn, T. and Jha, S., 1993. Exploiting Symmetry In Temporal Logic Model Checking. Vol. 697 of *LNCS* Springer-Verlag.

Clarke, E.M., *et al.*, 2000. Counterexample-Guided Abstraction Refinement. Vol. 1855 of *LNCS* Springer-Verlag.

Clarke, E.M., Grumberg, O. and Long, D.E., 1994. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16 (5).

Clarke, E.M., Grumberg, O. and Peled, D., 1999. *Model Checking*. MIT Press.

Clarke, E.M., Kroening, D. and Lerda, F., 2004a. A Tool for Checking ANSI-C Programs. Vol. 2988 of *LNCS* Springer-Verlag.

Clarke, E.M., Kroening, D., Sharygina, N. and Yorav, K., 2005. SATABS: SAT-based Predicate Abstraction for ANSI-C. Vol. 3440 of *LNCS* Springer-Verlag.

Clarke, E.M., Long, D. and McMillan, K.L., 1989. Compositional Model Checking. IEEE Press.

Clarke, E.M., Talupur, M., Touili, T. and Veith, H., 2004b. Verification by Network Decomposition. Vol. 3170 of *LNCS* Springer-Verlag.

Cohen, E., *et al.*, 2009. VCC: A Practical System for Verifying Concurrent C. Vol. 5674 of *LNCS* Springer-Verlag.

Cok, D.R. and Kiniry, J., 2005. ESC/Java2: Uniting ESC/Java and JML. Vol. 3362 of *LNCS* Springer-Verlag.

Cousot, P. and Cousot, R., 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. ACM Press.

Cousot, P. and Cousot, R., 1992. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2 (4).

Cousot, P., *et al.*, 2005. The Astre analyser. Vol. 3444 of *LNCS* Springer-Verlag.

Creese, S.J. and Roscoe, A.W., 2000. Data Independent Induction over Structured Networks. CSREA Press.

Davis, M., Logemann, G. and Loveland, D., 1962. A Machine Program for Theorem Proving. *Communications of the ACM*, 5.

de Moura, L.M. and Bjørner, N., 2008. Z3: An Efficient SMT Solver. Vol. 4963 of *LNCS* Springer-Verlag.

Deutsch, A., Static Verification of Dynamic Properties. SIGADA 2003. White paper of PolySpace Technologies., 2003.

Dill, D.L., 1989. Timing Assumptions and Verification of Finite-state Concurrent Systems. Vol. 407 of *LNCS* Springer-Verlag.

Edelstein, O., *et al.*, 2002. Multithreaded Java Program Test Generation. *IBM Systems Journal*, 41.

Elmas, T., Qadeer, S. and Tasiran, S., 2007. Goldilocks: A Race and Transaction-aware Java Runtime. ACM Press.

Emerson, E.A. and Halpern, J.Y., 1986. 'Sometimes' and 'Not Never' Revisited: On Branching Versus Linear Time Temporal Logic. *Journal of the ACM*, 33 (1).

Emerson, E.A. and Kahlon, V., 2000. Reducing Model Checking of the Many to the Few. Vol. 1831 of *LNCS* Springer-Verlag.

Emerson, E.A. and Kahlon, V., 2002. Model Checking Large-Scale and Parameterized Resource Allocation Systems. Vol. 2280 of *LNCS* Springer-Verlag.

Emerson, E.A. and Kahlon, V., 2004. Parameterized Model Checking of Ring-based Message Passing Systems. Vol. 3210 of *LNCS* Springer-Verlag.

Emerson, E.A. and Namjoshi, K.S., 1996. Automatic Verification of Parameterized Synchronous Systems. Vol. 1102 of *LNCS* Springer-Verlag.

Engler, D. and Musuvathi, M., 2004. Static Analysis versus Software Model Checking for Bug Finding. Vol. 2937 of *LNCS* Springer-Verlag.

Esparza, J., Hansel, D., Rossmanith, P. and Schwoon, S., 2000. Efficient Algorithms for Model Checking Pushdown Systems. Vol. 1855 of *LNCS* Springer.

Ferdinand, C., *et al.*, 2007. New Developments in WCET Analysis. *In*: T. Reps, M. Sagiv and J. Bauer, eds. *Program Analysis and Compilation, Theory and Practice.*, Vol. 4444 of *LNCS* Springer-Verlag.

Fiedor, J. and Vojnar, T., 2012. ANaConDA: A Framework for Analysing Multithreaded C/C++ Programs on the Binary Level. LNCS Springer-Verlag.

Finkel, A., 1994. Decidability of the Termination Problem for Completely Specified Protocols. *Distributed Computing*, 7 (3).

Finkel, A., Willems, B. and Wolper, P., 1997. A Direct Symbolic Approach to Model Checking Pushdown Systems. *ENTCS*, 9 A preliminary version was presented at Infinity'97.

Flanagan, C. and Freund, S., 2009. FastTrack: Efficient and Precise Dynamic Race Detection. ACM Press.

Gaiser, A. and Schwoon, S., 2009. Comparison of Algorithms for Checking Emptiness on Büchi Automata. FIT BUT.

Gastin, P. and Oddoux, D., 2001. Fast LTL to Büchi Automata Translation. Vol. 2102 of *LNCS* Springer-Verlag.

German, S.M. and Sistla, A.P., 1992. Reasoning about Systems with Many Processes. *Journal of the ACM*, 39 (3).

Godefroid, P., 1991. Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. Vol. 575 of *LNCS* Springer-Verlag.

Graf, S. and Saïdi, H., 1997. Construction of Abstract State Graphs with PVS. Vol. 1254 of *LNCS* Springer-Verlag.

Habermehl, P., *et al.*, 2011. Forest Automata for Verification of Heap Manipulation. Vol. 6806 of *LNCS* Springer-Verlag.

Havelund, K., 2000. Using Runtime Analysis to Guide Model Checking of Java Programs. Vol. 1885 of *LNCS* Springer-Verlag.

Henzinger, T.A., Jhala, R., Majumdar, R. and Sutre, G., 2002. Lazy Abstraction. ACM Press.

Henzinger, T.A., Jhala, R., Majumdar, R. and Sutre, G., 2003. Software Verification with Blast. Vol. 2648 of *LNCS* Springer-Verlag.

Henzinger, T.A., Nicollin, X., Sifakis, J. and Yovine, S., 1994. Symbolic Model Checking for Real-time Systems. *Information and Computation*, 111 First appeared in Proc. of LICS'92.

Hoare, C.A., 1969. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12.

Holzmann, G.J., 1987. On Limits and Possibilities of Automated Protocol Analysis. North-Holland Publishing Co.

Holzmann, G.J., 1997. The Model Checker Spin. *IEEE Transactions on Software Engineering*, 23 (5).

Holzmann, G., 1996. On-the-Fly Model Checking. *ACM Computing Surveys*, 28 (4es).

Holzmann, G. and Bošnački, D., 2007. The Design of a Multicore Extension of the SPIN Model Checker. *IEEE Transactions on Software Engineering*, 33 (10).

Holzmann, G., Peled, D. and Yannakakis, M., 1996. On Nested Depth First Search. American Mathematical Society.

Hovemeyer, D. and Pugh, W., 2004. Finding Bugs Is Easy. ACM Press.

Hrubá, V., *et al.*, 2012. Testing of Concurrent Programs Using Genetic Algorithms. Vol. 7515 of *LNCS* Springer-Verlag.

Hrubá, V., Křena, B. and Vojnar, T., 2009. Self-Healing Assurance Using Bounded Model Checking. Vol. 5717 of *LNCS* Springer-Verlag.

Ip, C.N. and Dill, D.L., 1996a. Better Verification Through Symmetry. *Journal of Formal Methods in System Design*, 9 (1/2).

Ip, C.N. and Dill, D.L., 1996b. Verifying Systems with Replicated Components in Murφ. Vol. 1102 of *LNCS* Springer-Verlag.

Jeannet, B. and Min, A., 2009. APRON: A Library of Numerical Abstract Domains for Static Analysis. Vol. 5643 of *LNCS* Springer-Verlag.

Jin, G., *et al.*, 2011. Automated Atomicity-Violation Fixing. ACM Press.

Kahlon, V., Ivančić, F. and Gupta, A., 2005. Reasoning about Threads Communicating via Locks. Vol. 3576 of *LNCS* Springer-Verlag.

Kaiser, A., Kroening, D. and Wahl, T., 2010. Dynamic Cutoff Detection in Parameterized Concurrent Programs. Vol. 6174 of *LNCS* Springer-Verlag.

Kam, J.B. and Ullman, J.D., 1976. Global Data Flow Analysis and Iterative Algorithms. *Journal of the ACM*, 23.

Kam, J.B. and Ullman, J.D., 1977. Monotone Data Flow Analysis Frameworks. *Acta Informatica*, 7.

Katz, S. and Peled, D., 1988. An Efficient Verification Method for Parallel and Distributed Programs. Vol. 354 of *LNCS* Springer-Verlag.

Kaufmann, M., Manolios, P. and (eds.), J.S.M., 2000a. *Computer-Aided Reasoning:*

ACL2 Case Studies. Kluwer Academic Publishers.

Kaufmann, M., Manolios, P. and (eds.), J.S.M., 2000b. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers.

Kelly, W., *et al.*, The Omega Calculator and Library, version 1. 1.0. , 1996.

Kesten, Y., *et al.*, 1997. Symbolic Model Checking with Rich Assertional Languages. Vol. 1254 of *LNCS* Springer-Verlag.

Khedker, U.P. and Dhamdhere, D.M., 1994. A Generalized Theory of Bit Vector Data Flow Analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16 (5).

Kildall, G.A., 1973. A Unified Approach to Global Program Optimization. ACM Press.

Klarlund, N. and Møller, A., MONA Version 1.4 User Manual. BRICS, Department of Computer Science, University of Aarhus, Denmark, 2001.

Kozen, D., 1983. Results on the Propositional μ-Calculus. *Theoretical Computer Science*, 27.

Kroening, D. and Weissenbacher, G., 2011. Interpolation-Based Software Verification with Wolverine. Vol. 6806 of *LNCS* Springer-Verlag.

Kurshan, R.P. and McMillan, K.L., 1995. A Structural Induction Theorem for Processes. *Information and Computation*, 117 (1).

Křena, B., *et al.*, 2007. Healing Data Races On-The-Fly. ACM Press.

Kwiatkowska, M., Norman, G. and Parker, D., 2011. PRISM 4.0: Verification of Probabilistic Real-time Systems. Vol. 6806 of *LNCS* Springer-Verlag.

Lesens, D., Halbwachs, N. and Raymond, P., 1997. Automatic Verification of Parameterized Linear Networks of Processes. ACM Press.

Marino, D. and Millstein, T., 2009. A Generic Type-and-Effect System. ACM Press.

Masson, B. and Schnoebelen, P., 2002. On Verifying Fair Lossy Channel Systems. Vol. 2420 of *LNCS* Springer-Verlag.

Mayr, R., 2000. Process Rewrite Systems. *Information and Computation*, 156 (1).

McMillan, K.L., Chapter title. *The SMV System*, Carnegie Mellon University and Cadence, 2000.

McMillan, K.L., 2005. Applications of Craig Interpolants in Model Checking. Vol. 3440 of *LNCS* Springer.

McMillan, K.L., Qadeer, S. and Saxe, J.B., 2000. Induction in Compositional Model Checking. Vol. 1855 of *LNCS* Springer.

McMinn, P., 2004. Search-based Software Test Data Generation: A Survey. *Software Testing, Verification and Reliability*, 14.

Merz, F., Falke, S. and Sinz, C., 2012. LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. Vol. 7152 of *LNCS* Springer-Verlag.

Musuvathi, M. and Qadeer, S., 2007. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. ACM.

Nagpaly, R., Pattabiramanz, K., Kirovski, D. and Zorn, B., 2007. Tolerace: Tolerating and Detecting Races.

Nielson, F. and Nielson, H.R., 1999. Type and Effect Systems. Vol. 1710 of *LNCS* Springer-Verlag.

Nielson, F., Nielson, H.R. and Hankin, C., 2005. *Principles of Program Analysis*. Springer-Verlag.

Nipkow, T., Paulson, L.C. and Wenzel, M., 2005. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag.

Owre, S., Shankar, N., Rushby, J.M. and Stringer-Calvert, D.W., Chapter title. *PVS System Guide*, Computer Science Laboratory, SRI International, Menlo Park, California, USA URL: `http://pvs.csl.sri.com/`, 2001.

Palsberg, J., 2001. Type-based Analysis and Applications. ACM Press.

Perrin, D. and Pin, J.E., 2003. *Infinite Words: Automata, Semigroups, Logic and Games*. Academic Press.

Pnueli, A., 1977. The Temporal Logic of Programs. IEEE.

Pnueli, A., 1985. In Transition from Global to Modular Temporal Reasoning about Programs. NATO Asi Series F: Computer And Systems Sciences, *In*: K. Apt, ed. *Logics and Models of Concurrent Systems*. Springer-Verlag.

Pnueli, A., Ruah, S. and Zuck, L., 2001. Automatic Deductive Verification with Invisible Invariants. Vol. 2031 of *LNCS* Springer-Verlag.

Pnueli, A. and Shahar, E., 2000. Liveness and Acceleration in Parameterized Verification. Vol. 1855 of *LNCS* Springer-Verlag.

Pnueli, A., Xu, J. and Zuck, L., 2002. Liveness with (0,1,infinity)-Counter Abstraction. Vol. 2404 of *LNCS* Springer-Verlag.

Podelski, A. and Rybalchenko, A., 2005. Transition Predicate Abstraction and Fair Termination. ACM Press.

Queille, J.P. and Sifakis, J., 1982. Specification and Verification of Concurrent Systems in CESAR. Vol. 137 of *LNCS* Springer-Verlag.

Ratanaworabhan, P., *et al.*, 2011. Efficient Runtime Detection and Toleration of Asymmetric Races. *IEEE Transactions on Computers*, 99.

Rybalchenko, A., ARMC: Abstraction Refinement Model Checker. URL: `http://www.mpi-inf.mpg.de/~rybal/armc/`, 2011.

Savage, S., *et al.*, 1997. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. ACM Press.

Seshia, S.A., Lahiri, S.K. and Bryant, R.E., Chapter title. *A User's Guide to UCLID version 1.0*, Carnegie Mellon University, 2003.

Sheeran, M., Singh, S. and Stålmarck, G., 2000. Checking Safety Properties Using Induction and a SAT-Solver. Vol. 1954 of *LNCS* Springer-Verlag.

Sipma, H., *et al.*, Constraint-Based Static Analysis of Programs. Master Class Seminar at Washington University at St Louis, 2006.

Sistla, A.P., Miliades, L. and Gyuris, V., 1997. SMC: A Symmetry Based Model Checker for Verification of Liveness Properties. Vol. 1254 of *LNCS* Springer-Verlag.

Srivastava, S., Gulwani, S. and Foster, J.S., 2009. VS3: SMT Solvers for Program Verification. Vol. 5643 of *LNCS* Springer-Verlag.

Valmari, A., 1988. State Space Generation: Efficiency and Practicality. Thesis (PhD). Tampere University of Technology, Tampere, Finland.

Valmari, A., 1998. The State Explosion Problem. *In*: W. Reisig and G. Rozenberg, eds. *Lectures on Petri Nets I: Basic Models.*, Vol. 1491 of *LNCS* Springer-Verlag.

Vardi, M., 2007. Automata-Theoretic Model Checking Revisited. Vol. 4349 of *LNCS* Springer-Verlag.

Vardi, M. and Wolper, P., 1986. An Automata-Theoretic Approach to Automatic Program Verification. IEEE CS.

Visser, W., *et al.*, 2003. Model Checking Programs. *Automated Software Engineering Journal*, 10 (2).

Walukiewicz, I., 1996. Pushdown Processes: Games and Model Checking. Vol. 1102 of *LNCS* Springer-Verlag.

Wolper, P. and Lovinfosse, V., 1989. Verifying Properties of Large Sets of Processes with Network Invariants. Vol. 407 of *LNCS* Springer-Verlag.

Zeller, A. and Hildebrandt, R., 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering*, 28.