Front. Comput. Sci., 2018, 12(4): 642–668
https://doi.org/10.1007/s11704-016-6192-0

REVIEW ARTICLE

The role of model checking in software engineering

Anil Kumar KARNA (✉)[1], Yuting CHEN[1], Haibo YU[2], Hao ZHONG[1], Jianjun ZHAO[3]

1 Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China
2 School of Software, Shanghai Jiao Tong University, Shanghai 200240, China
3 Department of Advanced Information Technology, Kyushu University, Fukuoka 819-0395, Japan

Abstract Model checking is a formal verification technique. It takes an exhaustively strategy to check hardware circuits and network protocols against desired properties. Having been developed for more than three decades, model checking is now playing an important role in software engineering for verifying rather complicated software artifacts.

This paper surveys the role of model checking in software engineering. In particular, we searched for the related literatures published at reputed conferences, symposiums, workshops, and journals, and took a survey of (1) various model checking techniques that can be adapted to software development and their implementations, and (2) the use of model checking at different stages of a software development life cycle. We observed that model checking is useful for software debugging, constraint solving, and malware detection, and it can help verify different types of software systems, such as object- and aspect-oriented systems, service-oriented applications, web-based applications, and GUI applications including safety- and mission-critical systems.

The survey is expected to help human engineers understand the role of model checking in software engineering, and as well decide which model checking technique(s) and/or tool(s) are applicable for developing, analyzing and verifying a practical software system. For researchers, the survey also points out how model checking has been adapted to their research topics on software engineering and its challenges.

Keywords software engineering, model checking, state-explosion

Received March 31, 2016; accepted December 8, 2016

E-mail: anil.karna@gmail.com

1 Introduction

Model checking is a formal verification technique that exhaustively checks hardware circuits and network protocols against desired properties [1]. For instance, if a desired property is expressed as a temporal logic p to verify the machine M with initial state s, model checking decides whether $M, s \models p$. As Fig. 1 shows, given an objective system, model checking checks whether an abstract model of the system holds some critical properties [2]: if the system definitely holds the properties, the system passes the verification; if the system fails, a counterexample needs to be produced; if the time budget or memory is used up during verification, a *state-explosion* problem appears.

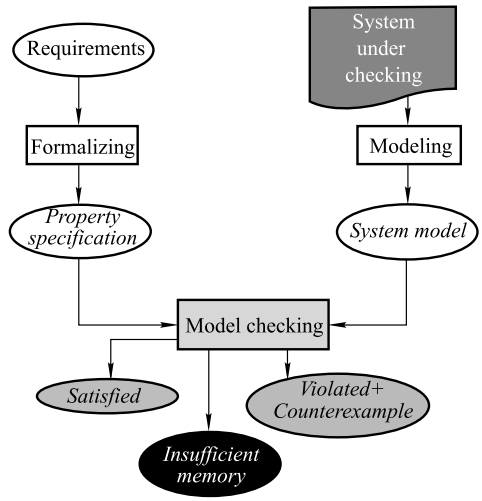


Fig. 1 Basic principle of model checking

Model checking was first proposed in the early of 1980s. In the past three decades, model checking has been playing an important role in software engineering for verifying rather complicated software artifacts. As what will be explained in the following sections, model checking is useful for software debugging, constraint solving, and malware detection; it can also be employed to verify different types of software systems, such as object- and aspect-oriented systems, service-oriented applications, web-based applications, safety- and mission-critical systems, and GUI applications.

In literature, researchers [3–10] have conducted surveys on model checking. Garcia-Ferreira et al. [3] briefly introduce the techniques of model checking. Grumberg et al. [11] introduce the history of model checking. Other researchers focus on specific techniques (e.g., partial model checking [4], regular model checking [5], and directed model checking [6]) or specific systems (e.g., state charts [7], parameterized systems [8], and test derivation [10]). However, to the best of our knowledge, no previous researchers conduct a comprehensive survey on model checking, especially on its applications in software engineering tasks. As a result, it is challenging for practitioners and even researchers to select proper model checking techniques, since they often do not fully understand the state of the art.

This paper surveys the role of model checking in software engineering. In particular, we searched for the related literatures published at reputed conferences, symposiums, workshops, journals, and transactions, and took a survey of (1) various model checking techniques that can be adapted to software development and their implementations, and (2) the use of model checking at different stages of a software development life cycle. We thus draw out the role of model checking in software engineering and also discuss some challenging problems and solutions. Some observations during the survey are explained throughout this paper.

This survey makes four contributions:

1) A large scale survey We collected 236 model checking-associated publications accepted at some reputed conferences, symposiums, workshops, journals and transactions, in which we carefully selected 173 ones to survey the state-of-the-art model checking techniques and tools that are prevalent in software development.
2) A novel perspective from software engineering Our survey analyzes the application of model checking in software engineering, with an emphasis on its capability of verifying real software artifacts.
3) Challenges, difficulties, and solutions We address some typical challenges and difficulties of employing model checking in software development. In particular, the state-explosion problem is addressed and the solutions are surveyed.
4) Integration with other existing techniques We briefly review some of the other existing code analysis and verification technique. The methods, comparison and integration with model checking are also surveyed.

We believe that the survey can help human engineers decide which model checking technique(s) and/or tool(s) are applicable for their needs. For researchers, the survey also points out how model checking can be adapted to the research topics on software engineering.

The rest of the paper is organized as follows. Section 2 explains how we select papers. Section 3 presents the role of model checking in verifying software products. Section 4 presents how model checking can supplement or be supplemented by other techniques for developing software systems. Section 5 addresses the state-explosion problem and its solutions. Section 6 surveys the various model checkers and their uses in software development. Section 7 studies the integration of model checking techniques with the other code analysis and verification techniques. Section 8 concludes this paper.

2 Paper selection

Clarke and Emerson [12] are the pioneers who propose the concept of model checking in 1981. Queille and Sifakis proposed a similar idea of verifying finite-state systems in 1982 [13]. Clarke, Emerson, and Sifakis accepted the ACM's Turing Awards "for [their roles] in developing model checking into a highly effective verification technology, widely adopted in the hardware and software industries" [14].

2.1 Publications on model checking

A number of literatures on model checking are available, as it can be applied in several aspects of computing. We used the keywords *model checking* and *model checker* to search the literatures from the *dblp* (see dblp official site) database (on October 10, 2016); 4,973 and 361 publications were retrieved when the two keywords were employed, respectively. Figure 2 shows the numbers of publications related to *model check+model checking+model checker*. Up to 1990, less than ten papers appeared per year, and consequently 10–50 after

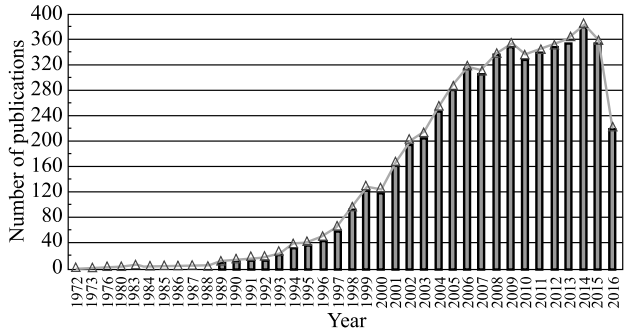


Fig. 2 Year-wise publications on model checking (The number of publications are less than 300 in 2016 as many publications have not yet been recorded by the searching date)

1990, 50+ after 1996, 100+ after 1998, 200+ after 2003, 250+ after 2004, and 300+ after 2007.

Figure 3 shows the conferences including workshops and symposiums, and journals including transactions that have 20+ publications on model checking. It clearly indicates that papers on model checking are prevalent in *SPIN*, *CAV*, *CHARME*, *TACAS*, *NASA FM*, *ATVA*, *STTT*, and *FM in System Design*.

By observing the searching results, we draw out the next observation:

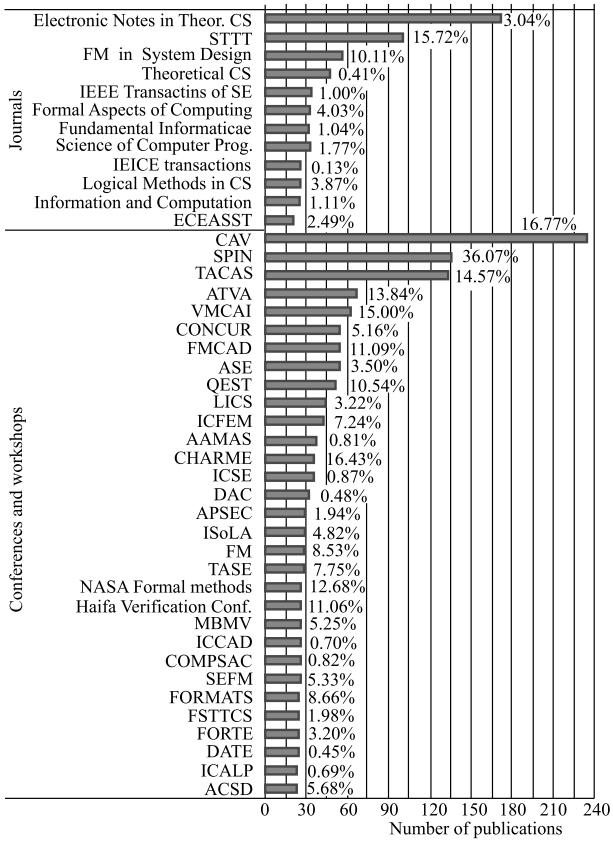


Fig. 3 Journals/conferences with 20+ publications on model checking (Each data label shows a percentage of publications which are related to model checking for a particular conference or journal)

Observation 1 Model checking has become a popular research topic since 1998/1999; publications on model checking continuously increase.

2.2 Publications at reputed SE conferences and journals

We select the publications from some reputed conferences (including symposiums and workshops) and journals (including transactions). The conferences and journals mainly belong to the category "Software engineering/System software/Programming languages" of a ranking list directed by the China Computer Federation (CCF) (see CCF official ranking site). In the list, the conferences (e.g., ESEC/FSE, ICSE, and ASE) are recommended on the basis of their acceptance rates, reputations and citations; the journal (e.g., TSE and TOSEM) are recommended on the basis of their SCI factors and citations.

As a result, 166, 711, and 625 papers are available at the A, B, and C level conferences, respectively, and 92, 76, and 216 papers are at the A, B, and C level journals, respectively. Meanwhile, we only focus on some reputed, SE-related conferences and journals shown in the first column of Table 1; publications at CAV (235 publications) and TACAS (134 publications) are not considered as they are mainly related to the theoretical aspects of model checking, rather than its role in software engineering.

Table 1 Papers for further review

Venue	NAP	NCP	References
ASE	54	49	[15–63]
ICSE	35	28	[64–91]
IEEE Transactions on SE (TSE)	34	19	[92–110]
SAS	13	11	[111–121]
FASE	14	10	[122–131]
ESEC/FSE	13	09	[132–140]
TOPLAS	13	07	[141–147]
TOSEM	12	07	[148–154]
ISSTA	10	06	[155–160]
POPL	10	06	[161–166]
IEEE Transaction on Computers (TC)	10	05	[167–171]
OOPSLA	05	05	[172–176]
PLDI	06	04	[177–180]
ISSRE	03	03	[181–183]
ICSM	02	02	[184,185]
RE	02	02	[186,187]
Total	236	173	

Only 173 publications are selected for further reviewing. Table 1 summarizes the publications selected for this survey. The columns NAP and NCP correspond to the numbers of *available papers* on model checking and the numbers of *se-*

lected papers for survey, respectively. 63 (i.e., 236–173) papers are not surveyed because some are irrelative to model checking (e.g., ICSE 1976: *rollback recovery model*), while the others focus on topics beyond the scope of this paper (e.g., definitions and proofs w.r.t. model checking).

Figure 4 classifies the topics of the 173 publications: 17% of the papers discuss how to perform model checking of software; 29% of the papers explain how to perform model checking for code analysis or debugging; 26% of the papers present how to check domain-specific real-world applications. Besides, model checking of software requirements, design models, and software product-lines (SPLs) cover 4%, 9%, and 5% of the papers, respectively. Other topics are studied in 10% of the papers.

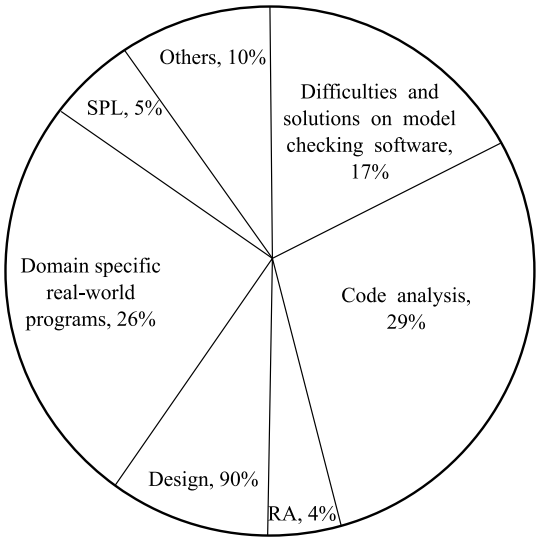


Fig. 4 Categorizing the reviewed papers

For example, Table 2 shows more details of 35 sampling papers; the other 138 papers are not explained for the sake of space. The columns "Major issue", "Solution", "SCL", "LOC", "NS", and "NT" represent the main problem discussed in the paper, the solution implemented, the modelling languages, the size of the largest application in verification, the maximum number of states visited successfully, and the maximum number of transitions during the evaluation, respectively. A detailed explanation about the techniques, tools, and applications will be given in the next sections.

We can understand conferences and journals have their specific tastes to their papers. The selected papers at "IEEE TSE" are more relative to the verification languages and specific theories, definitions, and proofs for model checking. They are also strongly related to specific model checking techniques (e.g., symbolic model checking, bounded model checking, probabilistic model checking) and tools (e.g., Spin, JPF, BMC [188], SMV [189], and Prism.

Many selected papers published at ICSE before 2009 are related to verification of software designs (eight publications) and empirical studies (four publications). After 2009, six publications are on probabilistic model checking, three on bounded model checking, and one on model checking of Java programs, one on abstraction for model checking, one on model checking using petri net, and another on combination of symbolic execution and model checking. In addition, six publications are on model checking of software product-lines.

Selected papers at the ASE conference are related to nearly all respects of model checking. Most papers before 2005 are related to its uses at some early stages of a software life cycle. After 2005, papers are on model checking of different types of applications and software systems. Papers are also concerned with the topics of model checking for static analysis, verifying software product-lines, and combining different model checking tools for optimizing verification process.

Nine out of eleven selected papers at SAS are closely related to the abstraction-based model checking.

Papers at FASE before 2010 are closely related to verification of software system designs (six out of seven publications); after 2010, they are more relevant to topics such as bounded model checking, probabilistic model checking, CEGAR based model checking.

Papers at RE are more relative to model checking of software requirements. Papers at POPL are more relative to model abstraction. Papers at PLDI are relative to concurrency bugs and stateless model checking. Papers at TOPLAS are more related to type checking and property checking. Papers at TOSEM are more relative to the model checking techniques for software development and their implementations.

Papers at TC, FSE, ISSTA, ISSRE, OOPSLA, and ICSM are more relative to implementations of model checking techniques for testing and verification. Some selected papers at FSE also discuss how to conduct model checking of graphical user interface (GUI) applications and how to use model checking to detect malware from a final software product.

Observation 2 Before 2005, model checking is mainly for verifying requirements and design models. After 2005, model checking is more towards verifying real-world applications.

3 Model checking of software

3.1 Background

The main role of model checking in software engineering is

Table 2 Details of 35 sampling papers

Year	Ref.	Major issue	Solution	Tool	SCL	LOC	NS	NT
1997	[100]	Design errors in software	Applying Spin tool	Spin	Promela	-	4,790,030	-
1997	[165]	Verifying abstracted models	Verifying source code directly	Verisoft	C	2,500	-	-
1998	[104]	Large software specification	Using symbolic model checking	SMV	SMV	-	1.4×10^{65}	-
2000	[26]	Abstracting models	Using C++/Java as models	JPF	C++/Java	1K/1.43K	10^{60}+	-
2000	[17]	Multithreaded Java program	Checking dynamic data structures	SAL	Java	4×100	3,990,883	-
2001	[18]	Symmetries (sym.) in codes	Reducing sym. of dynamic objects	dSpin	Promela	-	4.62×10^6	5.67×10^6
2001	[27]	Reduction of states and deadlocks	Static analysis+model checking	JPF	Java	60	866	1,489
2002	[160]	Direct model check Java bytecode	Heuristics coverage measurement	JPF	Java	-	1,836,675	-
2003	[24]	Verifying web based applications	Using Bandera to verify GUI	Bandera	Java	3,600	-	-
2003	[31]	Verifying large domain	Using domain reduction techniques	NuSMV	SMV	2,953	-	-
2003	[123]	NASA robot control software	Using loop abstraction	Cospan	S/R	-	6×10^{24}	-
2004	[21]	User interaction properties of GUI	Static analysis of SWING	Bandera	Java	-	84,439	84,493
2004	[126]	Communication (com.) processes	Optimized com. structure of tool	SPIN	Python	8,000	-	-
2007	[25]	Reduction of state space	Slicing technique	Bandera/JPF	Java	-	471,124	1,222,986
2007	[102]	Model checking for multi-core	Using Swarm tool	Swarm	Promela	6,635	22×10^6	601×10^6
2007	[151]	Java Metalocking	Java Metalocking Algorithm	XMC/Spin	Promela	-	198,901	987,009
2008	[56]	Hidden bugs in device drivers	SAT technique for unit testing	CBMC	C	30K	-	-
2008	[36]	Safety of array accesses	Using proof template, supports arrays	PTYASM	C	2K	-	-
2009	[45]	Accelerating computation time	Extended I/O efficient technique	-	DVE	-	2.3×10^{10}	-
2009	[28]	Race detection	Data-race specific heuristic search	JRF	Java	-	1,884	-
2009	[42]	Verifying networked applications	Cache based technique	JPF	Java	-	13,659,700	-
2009	[51]	Model checking embedded system	Combining different solvers	ESBMC	C	1,432	-	-
2009	[121]	State explosion problem	Directed model checking	MCTA	UPPAAL	-	4.56×10^6	-
2010	[16]	Inter-procedural analysis	Using control flow graph	-	Php/Java	34,101	296,340	406,304
2010	[97]	Finding web script crashes.	Dynamic test generat^n+model check	Apolo	PHP	16,993	-	-
2011	[52]	Hierarchical system	Library for any modelling language	PAT	C#	3,248	-	-
2011	[75]	Multi-threaded software	Checking shared variables and locks	ESBMC	C	6,366	-	-
2012	[106]	Embedded C software	Combining different solvers	ESBMC	C	258	-	-
2013	[50]	Verifying large programs	Compositional bounded model check	BLITZ	C	114.5K	-	-
2013	[94]	UML 2 design	MDD of embedded systems	JPF	Java	799,354	-	
2013	[139]	UML state machine (SM)	Self-contained tool for UML SM	USMMC	UML	-	398,101	2,385,361
2014	[96]	Larger class of distribute system	Using check-pointing tool	JPF	Java	-	5,507,260	-
2014	[93]	Check spurious counterexample	Enabling multi-core processors	CPA-Checker	C	-	50,000	18×10^7
2015	[65]	Data flow testing	Dynamic symbolic execut^n+CEGAR	CAUT	C	8,763	-	-
2015	[178]	Concurrency bugs	Maximal causality reduction	ASER	Java	380K	-	-

to verify software, i.e., to check whether a software system meets some desired properties. A typical model checking process includes four main steps:

1) Constructing a model for the system under verification

In order to verify a software system, an engineer first takes a formal modelling language to construct a model that can represent the system and as well be suitable for formal verification. Meanwhile, it will be time consuming to learn a formal language and as well construct an abstract model (see modex manual at spinroot official site) for a system without losing much information. Recent advances have thus been proposed auto-generation of models from programs in high level languages [55,190–195], which alleviates the cost of learning formal modelling languages. Researches have also been conducted on model checking of programs in high level languages or even binaries, where programs or binaries are directly taken as models.

2) Defining the properties that the system should satisfy

A model checker checks whether a model satisfies specific properties (e.g., functional correctness, safety, liveness, and fairness). Some properties need to be explicitly defined, usually in formal languages (e.g., LTL, CTL, and their extensions), while the others are not. The languages for defining the properties can be same as the modelling languages.

3) Performing model checking

Model checkers take specific algorithms to verify models against properties. A model checking algorithm mainly searches for the state space until the whole state space has

been explored or a counterexample is found. Typically, there are two model checking algorithms, both of which start at the root node of the model, but take their respective strategies to explore the nodes and branches [2]: the depth-first search (DFS) algorithm explores as far as possible along one branch before backtracking; the breadth-first search (BFS) algorithm explores the neighbor nodes first, before moving to the next level neighbors.

4) Generating results

Correspondingly, model checkers generate one of the three results.

- *Satisfied* The model meets the desired properties;
- *Unsatisfied* Some property is violated and thus one or more counterexamples are produced;
- *Unfinished* Verification is terminated due to some state-explosion problem.

In order to facilitate the understanding of a model checking process, we next show how a model checker, *Spin*, verifies a simple Java program. Figure 5(a) shows a Java program of two threads. The property that the program should satisfy is: *The program never reaches a state (x = N) at runtime*.

Figure 5(b) shows an abstract model in PROMELA for the program in Fig. 5(a). During model checking, Spin checks whether x equals to N at any transition.

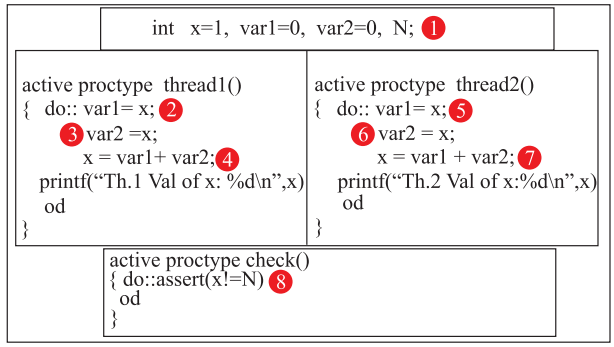
- Let N be 1 Spin can quickly catch a violation to the assertion at the step ⑧, when a counterexample is found at a depth of 0.
- Let N be 2 Spin catches a violation to the assertion at a depth of 3 by traversing through ② → ③ → ④ (see Fig. 6(a)) or ② → ③ → ⑤ → ⑥ → ⑦.
- Let N be 3 The assertion can also be violated by following a trace ② → ⑤ → ⑥ → ⑦ → ③ → ④. Meanwhile, even Spin of the latest release (released on Nov. 01, 2015) may not easily produce such a trace due to the state-explosion problem (see Fig. 6(b)).

A more detailed survey of model checkers and their applications will be given in Section 6, and the solutions to the state-explosion problem will be explained in Section 7.

3.2 Extending scalability of model checking

Model checking of real-world software systems is significantly different from model checking of hardware or network circuits. It is also different from model checking of the toy program in Fig. 5(a). Comparatively, a real-world software system is much more complicated, consisting of lots of logics, interactions, and/or software behaviors. The capability of model checking must be extended such that it can verify real-world software systems.

(a)



(b)

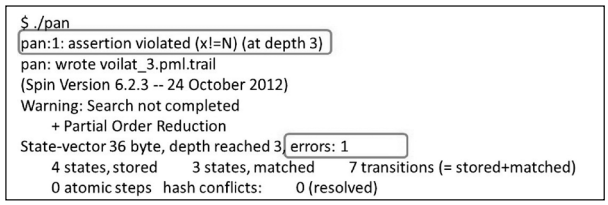
Fig. 5 An illustrative example. (a) Sampled Java source code; (b) abstracted promela code of (a)

3.2.1 Abstracting complicated software systems

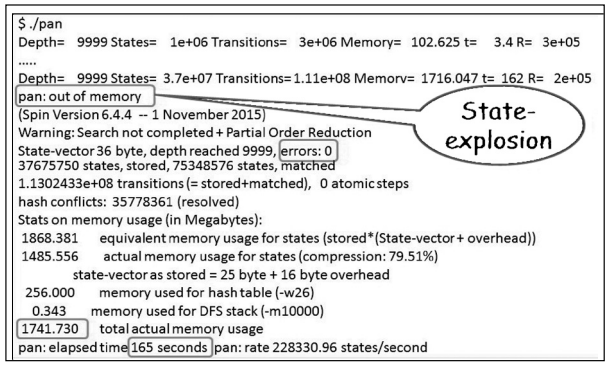
Abstraction is a technique for easing model checking of complicated systems [114,115]. Several abstraction techniques (e.g., abstract interpretation, control flow graph abstraction, use-mod abstraction) have been discussed in 1998 [112]. They are mainly used to extract a controllable model for a software system on which model checking can be performed. Domain reduction abstraction [31] and abstracting dynamic view model [19] have also been proposed to enhance the scalability of model checking. Some other typical modelling

and abstraction techniques that may be applicable to model checking of large-scale software systems are:

- Levi [116] has integrated abstract interpretation with the μ-calculus model checking technique for verifying the value-passing concurrent systems.
- Sharygina and Browne [123] have used loop abstraction to reduce the loops in software models.
- Choi and Heimdahl [31] have designed the domain reduction abstraction technique for model checking. This abstraction analyzes the specifications (of a system) statically and automatically produces an abstracted model that can be reused over time for regression verification.
- Thachuk and Dwyer [134] have adapted (flow- and context-sensitive) points-to analysis and side-effect analysis to generate refined abstract models for model checking.
- He et al. [168] have integrated an evolutionary algorithm and another abstraction refinement method for generating models for model checking.
- Specifying and modeling data structures, concurrent and real-time behaviors of some concurrent system relies a strong mechanism. Thus a specification language of reactive system *Action* [70] has been proposed.



(a)



(b)

Fig. 6 Results generated by Spin. (a) Results generated when $N = 2$; (b) encountered a state-explosion problem when $N = 3$

Meanwhile, abstracting complicated software systems for model checking still faces two challenges. First, it can be efficient to perform model checking on an abstract model, while abstraction always leads to a problem of "information loss": some features of the original system may be missing in the abstract model. So far there does not exist a solution completely addressing this problem, although it is partially solved by *domain abstraction*, an abstraction technique based on data equivalence and trajectory reduction [135], and *abstract satisfiability relation* which analyzes temporal properties with different levels of precision [117].

Second, the existence of counterexamples *w.r.t.* an abstract model does not indicate that these counterexamples will be adapted to a concrete model (or the objective system). Thereafter, algorithms have been developed to check whether a counterexample is spurious [66,92,93,119,120,169].

3.2.2 Verifying real-world software applications

Model checking has been applied to verify many real-world applications such as those for medical equipments, safety-critical systems [167] and mission critical systems [89,104,105,159].

• **Verifying device drivers** Model checking is employed to detect bugs in device drivers, one main cause of system crashes. Witkowski et al. [37] have enabled the automated verification of Linux device drivers and provided an accurate model of the relevant parts. At Verified Software Summer School in 2012, Patrice Godefroid has shown how model checking detects the Blue Screen of Death (BSOD) errors in Windows 7. Similarly, model checking has been employed to find latent bugs in Samsung's OneNAND flash memory [56,103].

• **Verifying distributed programs** Model checking has been extensively applied to verify distributed programs. During verification, some issues need to be addressed: the scheduling and non-deterministic asynchronous communications may introduce redundancy into model checking; communications among processes also need to be carefully processed [41].

For tackling these challenges, *process centralization* is developed which converts processes into threads followed by merging them such that model checking can be performed on a non-distributed application [40]. A cache-based approach has been developed to hide redundant communication operations of networked applications [42]. This special cache manages socket-based TCP/IP communications between model checkers and applications, allowing one client connection using blocking input/output per thread. Since most of the recent servers use non-blocking, selector-based input/output, an im-

proved cache-based approach is proposed for model checking [41,43,96].

• **Verifying GUI applications** A GUI application can be associated with many events and event-driven behaviors. Model checking a GUI application needs to check the software behaviors under all potential interaction orderings, which can be cumbersome. Instead, components of GUI applications can be model checked by leveraging domain specific abstractions and environment modelling [133].

Many GUI applications heavily rely on libraries such as AWT/Swing in Java. A static analysis of user-interaction properties of Swing-based GUI applications has been presented in [21]. It facilitates model construction and model checking. Formal notations to specify interaction sequence (in Java AWT) have also been developed for verifying all potential thread interleavings and input sequences.

• **Verifying multi-agent applications** A multi-agent system is composed of multiple autonomous agents. In order to support the automatic verification of such system, a programming language has been proposed [196]. Likewise, a probabilistic model checking approach has been applied to verify multi-agent systems [71].

• **Verifying service-oriented applications** Model checking of service-oriented applications has also been studied. A framework for checking functional properties of service-oriented applications specified using COWS (Calculus for Orchestration of web services) has been proposed in [128]. It illustrates a bank service scenario specified in COWS.

• **Verifying embedded systems** Model checking techniques have been implemented to analyze and verify reactive embedded system [197]. Eisler et al. [198] present the preliminary result of an automotive case study in the context of the European project EASIS.

• **Verifying aspect-oriented programs** CTL-based model checking has been developed to verify aspect-oriented programs which are abstracted as state machines [149].

• **Verifying mobile applications** Model checking techniques are useful to generate test case for mobile applications. Espada et al. [199] have presented the use of model-based testing to explain the potential human interaction with android applications. Aceto et al. [200] have proposed a method to provide run-time decision support for mobile cloud computing application using model checking.

• **Verifying web applications** Model checking has been performed to verify the key components of some web application supporting XML data manipulation operation [157]. Haydar et al. [44] have verified web applications using model checking techniques; the stable and unstable states are identified during verification. Artzi et al. [97] have combined concrete and symbolic execution with model checking in order to generate test inputs for web applications and validate the conformance of the outputs to the specification in HTML. Halle et al. [46] have proposed a runtime enforcement to restrict the control flow of a web application. Model checking is employed to reduce server processing time while handling unexpected requests.

Safety testing of web-application using model checking has been studied widely. Huang et al. [201] have explained the benefit of using counterexample traces for safety verification of web applications. Automatic derivation and generation of test cases from counterexample obtained through model checking for testing (security testing) of web-based applications is available in literature [202,203]. Similarly, model checking techniques have been implemented to verify class specification [204], UML design [205], BPEL (Business Process Execution Language) [206] and REST (Representational state transfer) web applications [207].

• **Verifying database applications** Model checking techniques have been implemented to verify database application recently. Gligoric and Majumdar [208] propose model checker for database-backed web applications. The model checker interposes between program code and database layer, and tracks the effects of queries made to the database precisely.

• **Verifying safety and mission-critical systems** Model checking techniques have been applied successfully in various safety and mission-critical physical, chemical, and biological systems as well as transportation (ground, water, pipeline, cable, and aviation), oceaonography, and astronomy. Using model checking techniques, Honeywell has analyzed the behavior and correctness of (a) automatic synthesis of real-time controllers, (b) real-time scheduler of the MetaH executive, (c) fault-tolerant ethernet protocol, (d) time partitioning in integrated modular avionics, and (e) synchronization protocol for avionics communication bus [209]. Cimatti [210] has used model checking techniques to verify safety layer of communication protocol used in several distributed safety-critical products and embedded control system. Likewise, Hoque et al. [211] have proposed a probabilistic model checking method to analyze the performability and dependability properties of safety-critical systems such as aerospace applications, and satellite system [212].

Besides, model checking techniques have been broadly used for verifying other systems such as verifying spacecraft controllers and real-time OS systems [26], analyzing aircraft collision avoidance system [89,104,105,159], human inter-

face issues in complicated, critical systems [23], security-critical system [213], crowd protocols [92], UAV mission plans [137], applications in railway domain [214], railway signaling system, the hydraulic system in Airbus A320, nuclear reactor of the TMI accident [167], and the traffic light controlling system [170].

• **Verifying biological applications** Model checking techniques have been implemented to verify biological applications such as biological models, reaction system [215], and bio-chemical reaction models [216]. Clarke et al. [217] present an algorithm for statistical model checking of temporal properties and automated analysis of T-cell receptor signalling pathway.

• **Verifying binaries** Blackham and Heiser [218] have proposed a framework for model checking of binary code. During model checking, it determines the loop counts and infeasible paths in binaries.

• **Detecting malware** Model checking is useful for detecting malicious behaviors from source programs [219]. A binary can also be modelled as a pushdown automata and model checking is then performed to detect malicious behaviors of the system under test [138].

3.3 Verifying other software artefacts

Model checking can be used to verify software requirements [95,99,136,187]. Barber et al. [34] have combined simulation and model checking to automate the evaluation of software architecture safety and liveness during the requirements collection and analysis phase. Giannakopoulou and Magee [132] have introduced the *fluent* property, an property naturally expressing the properties w.r.t. *state* for *normal system* and *action* for *event-based system*, for model checking of software requirements.

The idea of verifying software design models has been discussed by Visser et al. [26] and was popular during 2000–2005. Model checking can be performed on verifying design models specified in *xUML* (an executable subset of UML) [32,124]. It can also integrate state space reduction [125] for verifying communication structure [126]. Similarly, many approaches have been proposed on model checking of UML design models [67,69,139,152], consistency checking of UML design models [33], verifying protocol conformance of embedded systems based on UML 2 [94], checking consistency between UML class model and its Java implementation [110], merging design models [186], specifying metamodels [150], and checking distributed models [68]. It also benefits to software maintenance (i.e., reducing the future maintenance cost).

The dynamic behaviors of a software system may also be associated with its environment(s). However, environments are hard to describe since their behaviors depend on invocation of system components. Auto-generating environments for a Java program fragment is presented in [24], where an environment model is abstracted by assuming environment behaviors. Environment generation is also integrated with model slicing and error reduction for proving the absence of errors in [25].

3.4 Verifying software product-lines

Software product-line (SPL) is a collection of methods, tools, and techniques for producing similar software-based systems. Dynamic SPL (DSPL) produces products which can adapt proactively to environment changes.

Model checking has been used for verifying SPLs and DSPLs. Lauenroth et al. [54] have performed model checking of domain artifacts in SPLs, which aims to verify all permissible products (specified with I/O automata) from a product line fulfill the specified properties (specified in CTL).

However, performing model checking on SPLs and DSPLs has several challenges. First, it is hard to maintain the consistency of the product line variability models over time. Vierhauser et al. [53] have presented how incremental checker can help to maintain consistency on product-line variability models. Indeed, the checker can work across different levels of variability models and can even check consistency between variability models and source code. Consistency checking in DSPLs has also been performed on UML profiles [81].

Second, the number of states can grow rapidly when SPLs behaviors are model checked against temporal properties. Thus Classen et al. [76] have transformed the problem of model checking of an SPL to verifying a feature transition system that describes the combined behaviors of all the software systems w.r.t. an SPL. Model checking using symbolic representations [77] and symbolic checking with SAT-solver [78] also tackle the same problem.

Furthermore, SPL has limitations on dealing with non-boolean and some features. A solution is proposed in [80] which integrates software constructs with SPL behavioural specifications for model checking.

Observation 3 Model checking has been extended to verify real-world software applications: it benefits to detecting bugs and malicious behaviors from software systems and as well software requirements and designs; it has also been successfully applied to verify different types of software applications.

4 Model checking for software testing and program analysis

Model checking can also be integrated with software testing and program analysis such that software defects can be more effectively detected.

4.1 Model checking for software testing and debugging

Model checking is sometimes taken as a testing and debugging technique [220], as both model checking and software testing can be used for verifying a system under checking: model checking formally verifies the abstract model of program code, whereas software testing informally verifies program code itself. Model checking differs from testing in that model checking can approve or disapprove some properties of the system, while software testing cannot prove the absence of bugs.

Meanwhile, model checking can still supplement software testing in the sense of reducing test cases generated. Specifically, a counterexample generated by model checking can be used as a test for detecting some defect or locating the possible causes of some error [221]. For instance, let N be 5 in our example (see Fig. 5(b)). Spin detects an error when checking this code (we set the bound for model checking to avoid the state-explosion problem). The counterexample found during model checking can be traced as Fig. 7 shows. This can be used as a test for software testing. Hong et al. [64] have integrated dataflow analysis with model checking to automate test generation.

```
$ spin  –p –t  voilat_5.pml
using  statement  merging
  1:    proc  1  (thread2:1)  voilat_5.pml:12 (state 1)  [t1 = x]
  2:    proc  1  (thread2:1)  voilat_5.pml:13 (state 2)  [t2 = x]
  3:    proc  0  (thread1:1)  voilat_5.pml:4 (state 1)  [t2 = x]
  4:    proc  0  (thread1:1)  voilat_5.pml:5 (state 2)  [t2 = x]
  5:    proc  1  (thread2:1)  voilat_5.pml:14 (state 3)  [x (t1 + t2]
        Th.2 Val  of  X:2
  6:    proc  1  (thread2:1)  voilat_5.pml:15  (state 4)
        [printf (′Th. 2 Val of X : %D\\n′,x)]
  7:    proc  1  (thread2:1)  voilat_5.pml:12  (state 1)  [t1 = x)]
  8:    proc  0  (thread1:1)  voilat_5.pml:6  (state 3)  [x = (t1 + t2)]
  9:    proc  1  (thread2:1)  voilat_5.pml:13  (state 2)  [(t2 = x)]
 10:    proc  1  (thread2:1)  voilat_5.pml:14  (state 3)  [ x = (t1 + t2)]
spin: voilat_5.pml:20, Error: assertion violated
spin: text  of failed  assertion: assert ((x!=N))
 11 :  proc   2 (check:1) voilat_5.pml:20 (state  1)   [assert((x!=N))]
spin :  trail  ends  after  11  steps
 #processes: 3
              x = 5
              t1 = 2
              t2 = 3
              N  =  5
 11:     proc  2 (check:1) voilat_5.pml: 20(state  2)
 11:     proc  1 (thread2:1) voilat_5.pml: 15(state  4)
 11:     proc  0 (thread1:1) voilat_5.pml: 7(state  4)
3  processes   created
```

Fig. 7 Tracing counterexample when N is 5

Model checking is also benefited from some techniques typically used in software testing.

- Krishnamurthi and Fisler [149] have taken incremental analysis during model checking for verifying some changed components instead of verifying the entire system.
- Yang et al. [185] have also proposed the idea of *regression model checking*. Regression model checking incrementally checks a new version of a system, and thus the performance of regression model checking surpasses those of the traditional model checking techniques significantly.
- Chen et al. [170] have proposed mutation-based model checking, focusing only on checking the parts affected by the mutants. It accelerates coverage estimation.
- Gui et al. [156] have proposed a combination of hypothesis testing (to deterministic system components) and probabilistic model checking (to lift the results through non-determinism) to provide assurance and quantify the error bounds.

4.2 Model checking and code analysis

Model checking can be taken as a dataflow analysis [164,222] since they are both concerned with properties of program points or states. The connection between model checking and dataflow analysis can be precisely established through abstract interpretation [112].

Furthermore, model checking and static analysis can be closely related with each other [26,27]. Static analysis can help reduce the size of the program to verify. Model checkers such as JPF allow model checking and static analysis to be integrated with each other: static analysis computes partial order information which can be used by model checking to reduce the state space to explore; the partial order information can be refined by some static analyzer, and JPF can use this refined information for partial order reduction during verification.

Model checking has been applied to software analysis, such as analyzing software deviation [30], reducing auto-generated redundant assertions [122], and solving constraints for infinite-state systems [113].

Model checking can help discover some of the aforementioned bugs by statically analyzing the program code. Null pointer and double free can be detected by model checking.

- Park et al. [17] have explained how model checking can be used for static analysis of dynamically changed data structures (e.g., object creation and growing/shrinking call stacks) in multi-threaded Java programs. Both Java

source code or bytecode can be directly taken as the input of this model checking framework.

- Iosif [18] has presented an object-based model checker which can statically analyze a program and reduce the state space to explore by discovering symmetries that are induced by dynamically-created objects.
- Rungta and Mercer [15] have presented an algorithm of reconstructing calling context using the runtime stack with more accurate control flow graph. This algorithm helps estimate the distance to an error state and also can detect the errors in the program with function calls.
- Bouajjani et al. [111] have presented how model checking can be used for statically analyzing C code for manipulating dynamically linked data structures with pointer selectors and finite domain non-pointer data.
- Darga and Boyapati [175] have proposed analysis techniques to optimize and speed up model checking the properties of data structures by pruning redundant states and operation without checking them.
- Ku et al. [58] have developed benchmarks consisting of 298 code fragments. They have shown that a model checker (SetAbs) can discover buffer overflows in the benchmark.

Static analysis faces difficulties created by inter-procedural function calls. The transformation of model checking of inter-procedural aspects to an equivalent model checking task has been studied in 2010 [184]. Graph rewriting rules for producing a model checking automaton from a control flow graph are designed. The representation for inter-procedural analysis, limited to binary lattice inter-procedural analysis, has been presented in [16]. It uses the regular graph theory for easing its integration with model checkers.

Model checking has been implemented to precisely detect data races of Java bytecode by incorporating the knowledge of the Java Memory Model [28]. It helps verify the program satisfy the sequential consistency. This framework also implements some data-race-specific heuristic searching algorithms to find short counterexample-indicative paths. The detected data races can also be debugged and eliminated by analyzing the counterexample traces and histories [29]. Leesatapornwongsa and Gunawi [158] have proposed an open source model checking framework which can be adapted to many distributed cloud systems to find concurrent bugs raised by non-deterministic distributed events. Verification of safety and liveness properties of unbounded integer variables on a concurrent systems has been proposed by Bultan et al. [145].

Huang [178] has proposed the stateless model checking method to detect the concurrency bugs of Java programs with maximal causality reduction. Burckhardt et al. [177] have presented how concurrent data types on relaxed memory model can be checked using model checking. Similarly, symbolic model checking has been implemented to verify event-driven real-time systems [142].

Observation 4 Model checking can supplement or be supplemented by software testing and program analysis.

5 State-explosion problem and solutions

State-explosion problem affects the scalability of model checking in verifying real-world software systems. This section briefly describes the state-explosion problem and some solutions provided in model checking tools.

5.1 Problem description

A state-explosion problem occurs when memory is exhausted without completing searching the state space. When verifying software, a model checker usually needs to solve the state-explosion problem as the number of states grows during state exploration. As what we have explained in Section 3, a model checker usually takes some algorithm (DFS or BFS) to explore the state space. When DFS is taken, the state-explosion problem can occur if it has potential counterexamples at whatever depth of another branches. When BFS is taken, the problem can occur if the counterexample lies deeply in one branch.

The state-explosion problem can occur even in a few lines of code (e.g., the program in Fig. 5(b)). Figure 8 shows a path explored by Spin. As Fig. 8 illustrates, Spin exhaustively searches the state-space until the problem is detected.

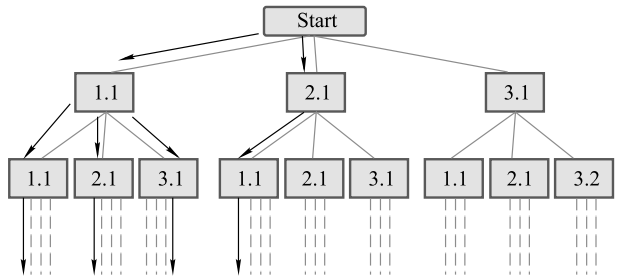


Fig. 8 Example of state exploration

5.2 Solutions

Solutions have been extensively proposed to address the state-explosion problem such that model checking can be

much more competitive and practical for verifying real-world software [11]. The solutions can be classified into four categories.

Solution 1: reducing the number of states

Symbolic model checking represents models by sets of states and transitions instead of individual states and transitions. Its efficiency can be improved by reducing the size of Binary Decision Diagram (BDD) using ordered BDD (OBDD) [223], CTL algorithm using BDD [189] and some other methods [89,104,105,159].

Partial order reduction reduces the state-space to search and proves that the reduced space is sufficient for checking w.r.t. a given property. Various partial order reduction techniques have been proposed [162,224–230]. In addition, since the state execution order does not affect the result in the independent concurrent actions, partial order reduction reduces independent interleavings of concurrent processes to reduced transitions. However, this technique only reduces equivalent independent states. If the states are dependent, partial order reduction is not capable of solving the state-explosion problem (see the example in Fig. 6(b)).

Bounded model checking is a symbolic model checking technique using SAT solvers, rather than using BDD [188]. Basically, it unrolls the transition relation upto a fixed k steps for checking violations; it repeats the above process with larger k if no counterexample can be found within k steps. Hence it is an incomplete checking method. Some methods have been proposed to estimate k such that bounded model checking can be complete [188,231–234]. Besides, *incremental bounded model checking* increases k without much extra cost of restarting [235]. Bounded model checking has been updated by replacing SAT solvers with SMT solvers, as SMT solvers have extra advantages in constraint solving [106,118,236]. Bounded model checking has also been tested using non-linear program solver [62]. Furthermore, it can be implemented in order to model check multi-threaded C-programs [63] and Qt applications [237].

Probabilistic model checking is applied for verifying the qualitative, quantitative and stochastic behaviors of a probabilistic system [131,238] and regenerative concurrent system [108]. It takes a property and a model as its input and delivers *yes*, *no*, or *some probabilities* as its outputs. *Statistical model checking* is performed by monitoring several runs of the system w.r.t. some property. The output is computed from the statistics [61,183]. It combines probabilistic simulation with some statistical method which provides clear error bounds [239]. Command-based importance sampling can reduce the cost of verification in statistical model checking [240]. *Parametric model checking* can support the computation of aggregate functions for a wide range of performance metrics to evaluate reliability of run-rime quality-of-service [90,109].

Besides, symmetry reduction [143,144], state merging, state caching, and stratified caching [241] are techniques for reducing the number of states.

Solution 2: partial model checking

Abstraction attempts to simplify model checking by abstracting away irrelevant properties from the concrete state transition system. A number of methods for obtaining useful abstract models to verify properties of the program are discussed by Clarke et al. [146,166]. Loop abstraction has been proposed to reduce the number of states to explore by minimizing loop executions [123]. Glass box software model checking has been proposed to prune numbers of redundant states without explicitly checking them [172]. Instead, in each steps, it checks a set of states together.

Counterexample guided abstraction refinement (CEGAR) is an abstraction method where precise and abstract representation of the system is verified by starting from verifying a relatively small representation of the system [242]. When a counterexample is found, the tool analyzes whether the violation is feasible: If the violation is feasible, it is reported; Otherwise a proof of the infeasibility is given and analysis is iteratively performed. CEGAR has also been used for higher order model checking to verify functional programs [179].

Directed model checking guides exploring first the parts of the state-space which contains reachable error states based on specific criteria [121,160]. It can verify huge systems by only checking the specific parts of the state-spaces. A type-directed abstraction refinement to address the higher order model checking is proposed by Ramsay et al. [161].

Beside, slicing [243], dead variable detection [244], dynamic delayed duplicate detection [245], nevertrace claim [246], and local first search [247] are some of the other partial model checking techniques to reduce the state explosion.

Solution 3: compacting the state consumption

Storing compressed states in memory (e.g., run-length encoding, static Huffman compression, or byte masking [248]) is an another solution to the state-explosion problem which is mainly caused by the heavy consumption of memory on saving every visited states during state exploration.

Bit state hashing is used for deciding a visited state. Bloom filters based on such hashing can explore much larger state-space. The accuracy of coverage estimation can get improved by leveraging bit-state hashing and the statistics from multiple verification runs [107]. *Hash compact* stores a single hash value, facilitating solving tractable problems in verification.

Incremental hashing in [249] combines both bit state hashing and hash compaction.

Solution 4: utilizing the other hardware resources

A large system can be verified through distributing its searching tasks on number of computers [39,250,251]. Partitioning the program texts into sequentially composed subprograms and then examining each sub program separately and sequentially on one computer [252], or distributively on many computers [253] help alleviate the state-explosion problem.

Deploying multiple cores for model checking can reduce the cost of computing successful/unsuccessful executions. However, it does not directly solves the state-explosion problem. Several proposals are available for parallelizing model checking in order to speed up computing successful/unsuccessful executions on multi-core systems [102,254–258].

I/O efficient model checking uses external memory to save visited states during search. Typical algorithms include DAC [259], MAP [260], IDDFS [261], and IOEMC [262]. Barnat et al. [45] have extended this technique to search over a network of computers.

Stateless model checking avoids searching and storing unreachable states [173,178,263]. The states of the program under checking can be stored in different hardware and machines over the network in addition to the global memory [180]. Model checkers based on SAT and SMT solvers also implement the stateless model checking [263] such that they can verify large-scale programs. Stateless model checking has been used to verify event driven applications such as web pages and mobile applications [174]. It has also been implemented to model check C/C++11 code [147] and detects concurrent bugs under relaxed memory models [91].

Besides, the sweep-line method deletes non-prioritize states from internal memory during state space exploration [264]. Compositional model checking is also performed to solve the state-explosion problem [171]. State explosion problem is also handled by implementing simulation-based abstractions [79] and probabilistic model checking techniques [131] when SPLs are verified.

5.3 Discussions on the existing solutions

The solutions to reduce the state-explosion problem are mainly categorized in four independent major directions. Since the number of states grows exponentially, reducing the number of states can slow down the exponential growth to reach the state-explosion. On the other hand, since the state-explosion problem is mainly due to the consumption of memory of the system, compacting the space needed to save a state or saving the required information of the states to the other available hardware resources can somehow solve this state-explosion problem. In similar way, model checking can partially verify the decomposed partial abstraction of the real system. Hence, it can verify most of the properties of the abstracted code, but it is not exhaustive in nature. Moreover, partial model checking with reduced number of compacted states saved on the other available hardware resources beside RAM can be a higher level solution for this historical state-explosion problem. In order to make it exhaustive, such model checking tool (with reduced number of compacted states saved on other available hardware resources) can check the original or decomposed source code rather than decomposed partial abstraction.

Observation 5 Model checking faces a strong challenge in solving the state-explosion problem. Some solutions are: reducing number of states, compacting state consumption, performing partial model checking, and utilizing other hardware resources.

6 Model checkers

Many model checkers have been developed. Table 3 lists the model checkers that are referred in at least two papers out of the 173 papers. The column “Modeling Languages” lists the modelling languages taken by these checkers: some take specific modelling languages, some use high-level language such as C/C++ and Java, and some directly take binary code as their input. The column “NP” represents the number of publications in which the tool is selected.

Spin Spin is an open source model checker. It can efficiently verify multi-threaded software and also allow embedded C code to be part of the model to check [100,101].

• Extensions Several extensions to Spin are available: optimizing the communication structure and using parallel BFS LTL checking have been presented in [126] and [20], respectively; αSpin is an XML-based extension to Spin that refines the abstract model and properties for verification [117]; dSpin, a dynamic extension to Spin, exploits symmetries induces by dynamically created objects [18]; MPI-Spin, a parallel extension to Spin, can verify parallel numerical programs [148,155]; and Swarm, a parallel version of Spin for multi-core systems, is discussed in [102].

• Applications Spin has been used to trace logical design errors [87]. It has also been used to analyze and verify soft

Table 3 Popular model checkers used in the 173 papers

Tool	Modelling language	NP	Applications reference
Spin	Promela	15	[18–20,87,99–102,117,126,148,155,157,171,181]
JPF	Java	14	[22,26–29,40–43,94,96,134,160,185]
Bounded model checking tools	SMV, C, C++, Java	13	[47,48,50,51,56,58,60,62,63,74,75,103,106]
SMV	SMV	10	[23,38,64,89,104,105,133,136,142,159]
NuSMV	SMV	6	[30,31,47,79,152,187]
Bandera	Java	6	[21,24,25,35,82,134]
PAT	CSP	5	[52,71,88,156,182]
Cospan	S/R	5	[32,123–125,127]
Prism	Prism	2	[92,137]
BLAST	C	2	[55,103]
*X*Chek	*X*Kripke, SMV	2	[86,154]
SGM	VPL (Verification Procedure Language)	2	[167,170]
Verisoft	C, C++	2	[83,165]

ware architect model [19], detect safety violation in software requirements [99], and verify XML data manipulation operations [157], web application [44], Tempoline OS [181], and concurrent system models [171]. In addition, Spin has been widely presented in Spin workshop (see spinroot official site). Some of applications to verify are: RUBIS μ-Kernel, message flow graphs and message sequence charts generated graphs, scenarios based specifications, business process models, multi-threaded C and systemC programs, C code generated by domain specific language. Spin also takes behavioural analysis of the enterprise JavaBeans, checks the atomicity of codes, and performs runtime verification of energy consumption of smart phone.

JPF Java Pathfinder (JPF) is an open source tool which combines static analysis and model checking to verify Java programs [27,265]. JPF was initially designed as a translator from Java to PROMELA for Spin, while it can now verify Java bytecode directly: it can store, match, and restore program states of Java bytecode for verification.

• Extensions Some extensions to JPF are: *Java Race Finder (JRF)* is developed for detecting data races [28]; *JRF-Eliminator (JRF-E)* is an extension to JRF, which provides suggestions to eliminate data races [29]; *JPF-AWT* is for verifying GUI-based concurrent applications [22] by verifying all possible thread interleavings and input sequences.

• Applications JPF was designed for detecting defects (such as data races and deadlocks) in concurrent programs [28,29], while it is applicable to many complicated applications, such as distributed and networked applications [40–43,96], spacecraft controllers and realtime OS systems [26], embedded systems [94], and GUI applications [22]. In some cases, points-to and side-effect analyses help analyze Java environment [134]. JPF can be used for test case generation by means of symbolic execution, low level program inspection, program instrumentation and run-time monitoring. It has also been applied to regression model checking [185] and directed model checking based on coverage measurement [160].

Symbolic model checkers SMV is the first model checker to support symbolic model checking. It has been used in checking declarative system against static properties [38], and analyzing aircraft collision avoidance system [89,104,105,159], event driven real time system [142], facilitating dataflow testing [64], generating tests from requirements [136], and verifying GUI applications [133].

Model checkers often generate counterexamples which are not easy to understand and thus it hinders many software engineers from using them. A tool collection around *SMV* has been developed for facilitating engineers to verify, e.g., human interface issues in complicated, critical systems [23].

SMV's extension *NuSMV* [266] can verify requirement specifications [31,187], UML activity diagrams [152], software deviations [30], and abstract SPLs [79]. Similarly, *X*Chek, a multi-valued symbolic model checker, allows to check models that contain uncertainty, disagreements, or relative priorities [154]. Likewise, symbolic model checker *Eureka* is designed to check linear programs with arrays [59].

Bounded model checkers BMC, CBMC, and BLITZ are model checkers supporting bounded model checking techniques. *BMC* permits to find a large number of redundant assertions to speed up the verification process [122]; it has been applied, together with NuSMV, to verify a flight guidance system [47]. *CBMC* has been used to verify embedded software in multi-core systems [74] and verify, along with BLAST [103], flash driver [56]. It has also been used, combined with CEGAR, to reduce false positives in model checking [47]. *ESBMC* has integrated different solvers to analyze

C code and finds number of bugs (buffer overflow, invalid pointers, etc.) in the benchmarks related to arithmetic and and pointer arithmetic [51,75,106]. *ESBMC*^{QtOM} is developed to verify Qt-based applications [237]. *BLITZ* helps find bugs in large-scale programs [50], *LLBMC* finds bugs and runtime errors in C/C++ programs [60], and *Lazy-CSeq* model checks multi-threaded C-programs.

Probabilistic model checkers *PAT* supports various probabilistic model checking algorithms [182]. PAT implements model checking techniques for 20+ languages [88]. It integrates partial order reduction, symmetry reduction, and process counter abstraction [267], and has been used as symbolic model checker for verifying hierarchical system [52] and multi-agent systems [71]. *Rapid* has been used for reliability prediction and distribution [156]. *Prism* has been used for verifying algorithms, complex systems [268], crowd protocols [92], and UAV mission plans [137].

Dynamic model checkers Dynamic model checkers (e.g., Verisoft and Bandera) ease the verification by directly taking programs in high level languages as models. *Bandera* enables extraction of safe, compact finite-state models from JAVA code [82]. It has been used to analyze GUI libraries such as SWING [21], verify interstage business process management software [25], generate environments for Java program fragments [24], and perform points-to and side-effect analyses of Java environment [134]. *Bogor* [140] provides an Eclipse extension for verifying object-oriented designs [35]. *VeriSoft* systematically explores the state-spaces of systems composed of concurrent processes written in high level languages such as C/C++ [83,165]. *BLAST* has been used, combined with CBMC [103], to verify SSL and C libraries [55] and flash driver [56].

Domain-specific model checkers Domain specific model checkers may be effective in verifying some large-scale software systems. *Bogor* allows to extend and customize its modelling language and algorithms to create domain-specific model checking engines [35]. *PTYASM* uses proof templates in checking the domain of array bounds [36]. It can verify the safety of array accesses in majority of test cases which were not easy to verify by existing tools due to the loop unrolling.

Other model checkers *Cospan* has been used for analysis and verification of UML designs [32,124] or designs in other languages [127]. It has been successfully applied to verify the designs of online ticket sale system [125] and NASA robot control software [123]. *Rational Rose* allows UML models to be checked [69]. *ObjectCheck* analyzes xUML models [124]. *CASE* has been used to analyze UML models of middleware [67]. *MCC* allows consistency among UML models to be checked [33]. *USMMC* checks UML state machines [139]; it also provides editing and interactive simulation. *SGM* has been used to analyze railway signaling system, the hydraulic system in Airbus A320, nuclear reactor of the TMI accident [167], and the traffic light controlling system [170].

Other tools appearing in reviewed papers are summarized in Table 4. Besides, tools such as Magic, Pex, Yogi, Slam, Chess, SAGE [269], and Blast [270] have been frequently mentioned in the articles. Many famous model checkers, such as DIVINE, UPPAAL, Murphi, and CADP, are not included in this survey, as they are rarely used by the selected papers.

Table 4 Other tools used in selected 173 papers

Tool	Uses in selected papers
ABC toolset	Verifying software product-lines [78]
Apolo	Locating bugs in web-applications [97]
Arcade	Evaluating software architectures [34]
ASER	Concurrent programs with maximal causality reduction [178]
ASTRAL	Using SMC technique for theorem proving [84]
CAUT	Data-flow testing of programs [65]
CDSChecker	Model checking C/C++11 code [147]
CPA-checker	Checking spurious counterexamples [93]
CMC	Checking service-oriented applications [128]
DDVerify	Verifying Linux device drivers [37]
DOPLAR	Verifying software product-lines [53]
FDR	Analyzing CSP-Z deadlocks [130]
HighSpec	Checking realtime systems in OZTA models [72]
JMOCHA	Analyzing modular design structures [85]
LTSA	Analyzing *fluent* property of a model [132]
MCTA	A directed model checking tool [121]
PIPAL	Prune number of redundant spaces [172]
PIPER	Model checking message passing programs [163]
Plato	Verifying asynchronous designs [169]
Platu	Verifying non-trivial concurrent system models [171]
POMMADE	Detecting malware and tracking program stack [138]
PREFACE	Abstraction refinement for higher order model checking [161]
PuMoC	Analyzing sequential C/C++/Java programs [57]
R4	Stateless model checking of web applications [174]
SAL	Analyzing data structures changes [17]
SAMC	Detecting concurrency bugs [158]
SetAbs	Detecting buffer overflows [58]
SMC	Detecting symmetries [153]
TReMer+	Verifying models of distributed systems [68]
XMC	Verifies Java metalocking algorithms [151]

6.1 Compositional model checkers

BLITZ implements compositional and property-sensitive algorithms for finding bugs in large-scale systems [50]. It decomposes behaviors of a program to a sequence of bounded model checking instances preserving accuracy of bounded

model checking. It then uses control- and data-flow analyses to incrementally generate smaller bounded model checking instances. It has been used to evaluate vulnerability benchmarks containing real-world programs.

Platu has been applied to compose several model checking techniques for verifying concurrent systems [171].

6.2 Combined model checkers

Several verification languages can be combined together such that verification can be more effective. In such cases, models are usually written in a single or a mixture of different languages. The combination of first-order quantifier, relational operators, and temporal logic operators in a single framework in a BDD-based model checker has been introduced by Chang and Jackson [73].

Model checking techniques can be combined with each other for optimizing performance.

- Santone [98] has combined heuristic searches with model checking in order to overcome the state-explosion problem.
- Choi and Heimdahl [47] have combined symbolic and bounded model checking to accelerate the verification process and to generate much more effective counterexamples. Symbolic model checking verifies whether a system holds a property. If fails, bounded model checking generates counterexamples.
- Post et al. [48] have combined bounded model checking and abstract interpretation to reduce false positives: spurious errors of C code are emitted by an abstract interpretation and false error reports are reduced by bounded model checking.
- Predela et al. [49] have performed bounded satisfiability checking of realtime systems. It shows the feasibility of bounded satisfiability checking and refinement checking, with modest performance loss with respect to bounded model checking.
- Beyer and Lowe [129] have combined abstraction, CEGAR and interpolation together for explicit value analysis. It extracts information from infeasible paths where the resulting interpolants refine the abstract model.

Combining constraint solvers can also enhance model checkers. *CBMC* has integrated SAT solvers *CVC*3, *Boolector*, and *Z*3 to verify relative large problems in reduced time [51,106], compared with that using a single solver. This integration provides supports for variables of finite bit width, bit-vector operations, arrays, structures, unions, and pointers. *ESBMC* extends CBMC by taking benefits from multi-threaded software and multi-core systems [74], and outperforms existing tools [75].

Observation 6 SPIN, JPF, BMC, SMV, NuSMV, Bandera are popular model checking tools in SE community. Besides, different model checking and verification techniques can be combined together to overcome their limitations or increase their performance.

7 Other analysis and verification techniqes

Model checking is one of the formal verification techniques. There exists some other code analysis and verification techniques such as type checking (type system), symbolic execution, and theorem proving. Model checking can be integrated into many other analysis and verification techniques. This section briefly discusses the other techniques and the integration of model checking with the other techniques.

Type system Type system is an approach to program verification. The main purpose of a type system is to reduce the bugs existing in a computer program by checking the consistent and sound connection between different parts (variables, expressions, functions, and modules) of the programs. It can be executed statically at compile time, dynamically at run time, or both.

Typically, type systems are defined in a syntactic and modular style, unlike model checking which is performed in a semantic and whole program style [141]. This is why a type system can justify the program acceptance and model checking the program rejection [141].

Many researchers have taken advantage of the type systems to for a better, complete and sound model checking of a program.

- Chaki et al. [163] have focused on using types as models for model checking message passsing programs. The behavioral properties of system such as deadlock freedom, race conditions, and message understood properties are statically checked.
- Naik and Palsberg [141] have proposed a type system which is equivalent to model checking and verifying safety properties of imperative programs. It performs type checks of the programs which are already accepted by model checker. They also study the relative expressiveness between the type systems and model checking.
- Roberso et al. [176] have employed model checking to

automatically test the soundness of a type system. This reduces the state space of the software model checker, hence reduces the impending state-explosion.

Symbolic execution Symbolic execution is widely used in many code analysis, testing, and verification tools [271]. It executes a program using symbolic values rather than actual inputs as its inputs [272,273]. It tries to cover all feasible paths of a code snippet by exhaustive exploring path.

Both symbolic execution and model checking can be used for code analysis. However, symbolic execution suffers from the path-explosion and model checking suffers from the state-explosion problem. Researchers have tried to integrate the both techniques for code analysis, testing, and verification. JPF is an explicit state model checker and also a symbolic execution tool.

- Khurshid et al. [274] have generalized symbolic execution for model checking and testing by instrumenting a source to source translation of a program. They use JPF to perform symbolic execution of the program to handle dynamically allocated structures (e.g., lists and trees), method preconditions (e.g., acyclicity), data (e.g., integers and strings) and concurrency.
- Siegel et al. [155] have combined model checking and symbolic execution to verify the correctness of parallel programs *w.r.t.* floating-point numbers. The path condition from symbolic execution constrains the search in parallel [148].
- Su et al. [65] have proposed a hybrid dataflow testing framework, in which dynamic symbolic execution is applied for testing and model checking for reachability analysis.

Theorem proving Theorem proving is an approach to program verification. In theorem proving, a program under checking is modelled as a set of mathematical definitions in some formal mathematical logic [275]. The desired properties of the system are then proved from these definitions.

Model checking is a state-based automatic approach but cannot handle complex formalisms, whereas theorem proving is a proof-based manual approach which can handle complex formalism [275]. In other words, model checking is an algorithmic verification and theorem proving is a deductive verification. Model checking can only be applied on a finite-state system, but theorem proving can be applied on an infinite-state system [276]. Model checking can generate counterexample, but theorem proving cannot generate it.

Researchers have tried to combine both of the formal verification techniques to verify parallel processes [277], Haskell programs [278], and AMBA [279].

- Uribe [276] has presented several earlier work on combining these two techniques. It includes the important links (abstraction and invariant generation) between theorem proving and model checking and the combination method using loosely coupled combinations and tight combinations.
- Amjad [275] has implemented a model checker for the modal μ-calculas as a derived rule in a mechanical theorem prover. The tool does not cause an unacceptable performance penalty.
- Seidel [280] has proposed a multi-level combination of these two methods to verify a functional unit. It basically uses a theorem prover with the help of a model checking tool.

Observation 7 Model checking can be integrated with other code analyzing and verification techniques such as type systems, symbolic execution, and theorem proving for efficient analysis, testing, and verification.

8 Conclusion

Model checking is a formal verification technique which has been widely employed for past 35+ years in the field of computer science and engineering. We reviewed 173 articles on model checking published at reputed conferences and journals on software engineering and draw out its role, particularly in software engineering. When applied to software development, model checking can improve the reliability of software systems through verifying software artifacts, detecting software defects and malware, and/or supplementing or being supplemented by several other techniques (testing, software debugging, data-flow analysis, constraint solving, and deviation analysis). It has been used to verify various types of software systems such as service-oriented application, Web-based applications, banking applications, GUI-applications, multi-agent applications, mobile applications, biological applications, embedded system, object- and aspect-oriented system, database systems, safety- and mission-critical systems including traffic, transport, avionics, and space systems. It can also be integrated with other formal methods to optimize and magnify the analysis and verification process of properties to a rather big real-world program. Although

model checking has been used widely in different areas of software engineering, research on model checking parallel programs, real-world large system and application software, and mobile and embedded applications still has a long path to take in order to meet the specific requirements of the formal verification team.

Model checking is suffering from a severe state-explosion problem. Even though various methods have been proposed to tackle this historical problem in different directions, the state-explosion problem still persists. There still have some space to reduce this problem by integrating different solutions together for an optimized verification process.

We believe that the survey is comprehensive; it can help guide engineers and researchers to choose and apply model checking techniques and tools when developing their software systems, and also improve the capabilities of many software development and verification techniques.

Acknowledgements The authors are grateful to anonymous referee for their valuable comments and suggestions to improve the presentation and contents of this paper. This research was sponsored in part by the National Basic Research Program of China (2015CB352203) and the National Nature Science Foundation of China (Grant Nos. 61572312 and 61572313). Jianjun Zhao was supported in part by Japan Society for the Promotion of Science, Grant-in-Aid for Research Activity Start-up (16H07031). Hao Zhong was partially supported by Science and Technology Commission of Shanghai Municipality's Innovation Action Plan (15DZ1100305).

References

1. Ben-Ari M. A primer on model checking. Inroads, 2010, 1(1): 40–47
2. Baier C, Katoen J-P. Principles of Model Checking. Cambridge: The MIT Press, 2008
3. Garcia-Ferreira I, Laorden C, Santos I, Bringas P G. A survey on static analysis and model checking. In: Proceeding of International Joint Conference SOCO, CISIS & ICEUTE. 2014, 443–452
4. Andersen H R, Lind-Nielsen J. Partial model checking of modal equations: a survey. International Journal on Software Tools for Technology Transfer, 1999, 2(3): 242-259
5. Abdulla P A, Jonsson B, Nilsson M, Saksena M. A survey of regular-model checking. In: Proceedings of the 15th International Conference on Concurrency Theory. 2004, 35-48
6. Edelkamp S, Schuppan V, Bosnacki D, Wijs A, Fehnker A, Aljazzar H. Survey on directed model checking. In: Proceedings of the 5th International Workshop on Model Checking and Artificial Intelligence. 2008, 65-89
7. Bhaduri P, Ramesh S. Model checking of statechart models: survey and research directions. Computer Science, 2004
8. Zuck L D, Pnueli A. Model checking and abstraction to the aid of parameterized systems (a survey). Computer Languages, Systems & Structures, 2004, 30(3–4): 139–169
9. Eiter T, Gottlob G, Schwentick T. The model checking problem for prefix classes of second-order logic: a survey. In: Blass A, Dershowitz N, Reisig W, eds. Fields of Logic and Computation. Spinger International Publishing, 2010, 227–250
10. Fraser G, Wotawa F, Ammann P. Testing with model checkers: a survey. Software Testing, Verification & Reliability, 2009, 19(3): 215–261
11. Grumberg O, Veith H. 25 Years of Model Checking: History, Achievements, Perspectives. Berlin: Springer, 2008
12. Clarke E M, Emerson A E. Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Proceedings of Workshop on Logic of Programs. 1981, 52–71
13. Queille J-P, Sifakis J. Specification and verification of concurrent systems in CESAR. In: Proceedings of the International Symposium on Programming. 1982, 337–351
14. Clarke E M, Emerson E A, Sifakis J. Model checking: algorithmic verification and debugging. Communications of the ACM, 2009, 52(11): 74–84
15. Rungta N, Mercer E G. A context-sensitive structural heuristic for guided search model checking. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2005, 410–413
16. Letarte D. Model checking graph representation of precise Boolean inter-procedural flow analysis. In: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2010, 511–516
17. Park D Y W, Stern U, Skakkebak J U, Dill D L. Java model checking. In: Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE). 2000, 253–256
18. Iosif R. Exploiting heap symmetries in explicit-state model checking of software. In: Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE). 2001, 254–261
19. Inverardi P, Muccini H, Pelliccione P. Automated check of architectural models consistency using SPIN. In: Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE). 2001, 346–349
20. Barnat J, Brim L, Chaloupka J. Parallel breadth-first search LTL model-checking. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE). 2003, 106–115
21. Dwyer M B, Robby, Tkachuk O, Visser W. Analyzing interaction orderings with model checking. In: Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE). 2004, 154–163
22. Mehlitz P C, Tkachuk O, Ujma M. JPF-AWT: model checking GUI applications. In: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2011, 584–587
23. Loer K, Harrison M D. Towards usable and relevant model checking techniques for the analysis of dependable interactive systems. In: Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE). 2002, 223–226
24. Tkachuk O, Dwyer M B, Pasareanu C S. Automated environment generation for software model checking. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE). 2003, 116–129
25. Tkachuk O, Rajan S P. Combining environment generation and slicing for modular software model checking. In: Proceedings of the 22nd

IEEE/ACM International Conference on Automated Software Engineering (ASE). 2007, 401–404

26. Visser W, Havelund K, Brat G P, Park S. Model checking programs. In: Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE). 2000, 3–12
27. Brat G P, Visser W. Combining static analysis and model checking for software analysis. In: Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE). 2001, 262
28. Kim K, Yavuz-Kahveci T, Sanders B A. Precise data race detection in a relaxed memory model using heuristic-based model checking. In: Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE). 2009, 495–499
29. Kim K, Yavuz-Kahveci T, Sanders B A. JRF-E: using model checking to give advice on eliminating memory model-related bugs. In: Proceedings of the 25th IEEE International Conference on Automated Software Engineering (ASE). 2010, 215–224
30. Heimdahl M P E, Choi Y, Whalen M W. Deviation analysis through model checking. In: Proceedings of the 17th IEEE International Conference on Automated Software Engineering. 2002 (ASE), 37–46
31. Choi Y, Heimdahl M P E. Model checking software requirement specifications using domain reduction abstraction. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE). 2003, 314–317
32. Xie F, Levin V, Browne J C. Model checking for an executable subset of UML. In: Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE). 2001, 333–336
33. Simmonds J, Bastarrica C M. A tool for automatic UML model consistency checking. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2005, 431–432
34. Barber K S, Graser T J, Holt J. Providing early feedback in the development cycle through automated application of model checking to software architectures. In: Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE). 2001, 341–345
35. Robby, Dwyer M B, Hatcli J. Domain-specific model checking using the Bogor framework. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE). 2006, 369–370
36. Hart T E, Ku K, Gurfinkel A, Chechik M, Lie D. PtYasm: software model checking with proof templates. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE). 2008, 479–480
37. Witkowski T, Blanc N, Kroening D, Weissenbacher G. Model checking concurrent linux device drivers. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE). 2007, 501–504
38. Vakili A, Day N A. Using model checking to analyze static properties of declarative models. In: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2011, 428–431
39. Crhova J. Distributed modular model checking. In: Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE). 2002, 312
40. Artho C, Garoche P. Accurate centralization for applying model checking on networked applications. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE). 2006, 177–188
41. Leungwattanakit W, Artho C, Hagiya M, Tanabe Y, Yamamoto M.Model checking distributed systems by combining caching and process checkpointing. In: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2011, 103–112
42. Artho C, Leungwattanakit W, Hagiya M, Tanabe Y, Yamamoto M. Cache-based model checking of networked applications: from linear to branching time. In: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2009, 447–458
43. Artho C, Hagiya M, Potter R, Tanabe Y, Weitl F, Yamamoto M. Software model checking for distributed systems with selector-based, non-blocking communication. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2013, 169–179
44. Haydar M, Boroday S, PetrenkoA, Sahraoui H A. Properties and scopes in Web model checking. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2005, 400–404
45. Barnat J, Brim L, Simecek P. Cluster-based I/O-efficient LTL model checking. In: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2009, 635–639
46. Halle S, Ettema T, Bunch C, Bultan T. Eliminating navigation errors in Web applications via model checking and runtime enforcement of navigation state machines. In: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2010, 235–244
47. Choi Y, Heimdahl M P E. Combination model checking: approach and a case study. In: Proceedings of the 19th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2004, 354–357
48. Post H, Sinz C, Kaiser A, Gorges T. Reducing false positives by combining abstract interpretation and bounded model checking. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE). 2008, 188–197
49. Pradella M, Morzenti A, Pietro P S. Refining real-time system specifications through bounded model- and satisfiability-checking. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE). 2008, 119–127
50. Cho C Y, D'Silva V, Song D. BLITZ: Compositional bounded model checking for real-world programs. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2013, 136–146
51. Cordeiro L C, Fischer B, Marques-Silva J. SMT-based bounded model checking for embedded ANSI-C software. In: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2009, 137–148
52. Nguyen T K, Sun J, Liu Y, Dong J S. A model checking framework for hierarchical systems. In: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2011, 633–636
53. Vierhauser M, Grunbacher P, EgyedA, Rabiser R, Heider W. Flexible

and scalable consistency checking on product line variability models. In: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2010, 63–72

54. Lauenroth K, Pohl K, Toehning S. Model checking of domain artifacts in product line engineering. In: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2009, 269–280
55. Liu C, Ye E, Richardson D J. Software library usage pattern extraction using a software model checker. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE). 2006, 301–304
56. Kim M, Kim Y, Kim H. Unit testing of flash memory device driver through a SAT-based model checker. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE). 2008, 198–207
57. Song F, Touili T. PuMoC: a CTL model-checker for sequential programs. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2012, 346–349
58. Ku K, Hart T E, Chechik M, Lie D. A buffer overflow benchmark for software model checkers. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE). 2007, 389–392
59. Armando A, Benerecetti M, Carotenuto D, Mantovani J, Spica P. The EUREKA tool for software model checking. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE). 2007, 541–542
60. Falke S, Merz F, Sinz C. The bounded model checker LLBMC. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2013, 706–709.
61. He R, Jennings P, Basu S, Ghosh A P, Wu H. A bounded statistical approach for model checking of unbounded until properties. In: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2010, 225–234
62. Nishi M. Towards bounded model checking using nonlinear programming solver. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE). 2016, 560–565
63. Inverso O, Nguyen T L, Fischer B, Torre S L, Parlato G. Lazy-CSeq: a context-bounded model checking tool for multi-threaded C-programs. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2015, 807–812
64. Hong H S, Cha S D, Lee I, Sokolsky O, Ural H. Data flow testing as model checking. In: Proceedings of the 25th IEEE International Conference on Software Engineering (ICSE). 2003, 232–243
65. Su T, Fu Z, Pu G, He J, Su Z. Combining symbolic execution and model checking for data flow testing. In: Proceedings of the 37th IEEE International Conference on Software Engineering (ICSE). 2015, 654–665
66. Tian C, Duan Z. Detecting spurious counterexamples efficiently in abstract model checking. In: Proceedings of the 35th IEEE International Conference on Software Engineering (ICSE). 2013, 202–211
67. Kaveh N. Model checking distributd objects design. In: Proceedings of the 23rd IEEE International Conference on Software Engineering (ICSE). 2001, 793–794
68. Sabetzadeh M, Nejati S, Easterbrook S M, Chechik M. Global consistency checking of distributed models with TReMer+. In: Proceedings of the 30th IEEE International Conference on Software Engineering (ICSE). 2008, 815–818
69. Egyed A. UML/Analyzer: a tool for the instant consistency checking of UML models. In: Proceedings of the 29th IEEE International Conference on Software Engineering (ICSE). 2007, 793–796
70. Bultan T. Action language: a specification language for model checking reactive systems. In: Proceedings of the 22nd IEEE International Conference on Software Engineering (ICSE). 2000, 335–344
71. Song S, Hao J, Liu Y, Sun J, Leung H, Dong J S. Analyzing multi agent systems with probabilistic model checking approach. In: Proceedings of the 34th IEEE International Conference on Software Engineering (ICSE). 2012, 1337–1340
72. Dong J S, Hao P, Zhang X, Qin S. HighSpec: a tool for building and checking OZTA models. In: Proceedings of the 28th IEEE International Conference on Software Engineering (ICSE). 2006, 775–778
73. Chang F S-H, Jackson D. Symbolic model checking of declarative relational models. In: Proceedings of the 28th IEEE International Conference on Software Engineering (ICSE). 2006, 312–320
74. Cordeiro L C. SMT-based bounded model checking for multi threaded software in embedded systems. In: Proceedings of the 32nd IEEE International Conference on Software Engineering (ICSE). 2010, 373–376
75. Cordeiro L C, Fischer B. Verifying multi-threaded software using SMT-based context-bounded model checking. In: Proceedings of the 33rd IEEE International Conference on Software Engineering (ICSE). 2011, 331–340
76. Classen A, Heymans P, Schobbens P-Y, Legay A, Raskin J-F. Model checking lots of systems: efficient verification of temporal properties in software product lines. In: Proceedings of the 32nd IEEE International Conference on Software Engineering (ICSE). 2010, 335–344
77. Classen A, Heymans P, Schobbens P-Y, Legay A. Symbolic model checking of software product lines. In: Proceedings of the 33rd IEEE International Conference on Software Engineering (ICSE). 2011, 321–330
78. Ben-David S, Sterin B, Atlee J M, Beidu S. Symbolic model checking of product-line requirements using SAT-based methods. In: Proceedings of the 37th IEEE International Conference on Software Engineering (ICSE). 2015, 189–199
79. Cordy M, Classen A, Perrouin G, Schobbens P-Y, Heymans P, Legay A. Simulation-based abstractions for software product-line model checking. In: Proceedings of the 34th IEEE International Conference on Software Engineering (ICSE). 2012, 672–682
80. Cordy M, Schobbens P-Y, Heymans P, Legay A. Beyond Boolean product-line model checking: dealing with feature attributes and multi-features. In: Proceedings of the 35th IEEE International Conference on Software Engineering (ICSE). 2013, 472–481
81. Marinho F G. A proposal for consistency checking in dynamic software product line models using OCL. In: Proceedings of the 32nd IEEE International Conference on Software Engineering (ICSE). 2010, 333–334
82. Corbett J C, Dwyer M B, Hatcli J, Robby. Bandera: extracting finite-state models from Java source code. In: Proceedings of the 22nd IEEE International Conference on Software Engineering (ICSE). 2000, 439–448

83. Chandra S, Godefroid P, Palm C. Software model checking in practice: an industrial case study. In: Proceedings of the 24th IEEE International Conference on Software Engineering (ICSE). 2002, 431–441
84. Dang Z, Kemmerer R A. Three approximation techniques for ASTRAL symbolic model checking of infinite state real-time systems. In: Proceedings of the 22nd IEEE International Conference on Software Engineering (ICSE). 2000, 345–354
85. Alur R, Alfaro L, Grosu R, Henzinger T A, Kang M, Kirsch C M, Majumdar R, Mang F Y C, Wang B-Y. JMOCHA: a model checking tool that exploits design structure. In: Proceedings of the 23rd IEEE International Conference on Software Engineering (ICSE). 2001, 835–836
86. Easterbrook S M, Chechik M, Devereux B, Gurfinkel A, Lai A Y C, Petrovykh V, Tafliovich A, Thompson-Walsh C D. XChek: a model checker for multi-valued reasoning. In: Proceedings of the 25th IEEE International Conference on Software Engineering (ICSE). 2003, 804–805
87. Long B, Dingel J, Graham T C N. Experience applying the SPIN model checker to an industrial telecommunications system. In: Proceedings of the 30th IEEE International Conference on Software Engineering (ICSE). 2008: 693–702
88. Dong J S, Sun J, Liu Y. Build your own model checker in one month. In: Proceedings of the 35th IEEE International Conference on Software Engineering (ICSE). 2013, 1481–1483
89. Chan W, Anderson R J, Beame P, Jones D H, Notkin D, Warner W E. Decoupling synchronization from local control for efficient symbolic model checking of state charts. In: Proceedings of the 21st IEEE International Conference on Software Engineering (ICSE). 1999, 142–151
90. Su G, Rosenblum D S, Tamburrelli G. Reliability of run-time quality-of-service evaluation using parametric model checking. In: Proceedings of the 38th IEEE International Conference on Software Engineering (ICSE). 2016, 73–84
91. Huang A. Maximally stateless model checking for concurrent bugs under relaxed memory models. In: Proceedings of the 38th IEEE International Conference on Software Engineering (ICSE). 2016, 686–688
92. Han T, Katoen J-P, Damman B. Counterexample generation in probabilistic model checking. IEEE Transactions on Software Engineering, 2009, 35(2): 241–257
93. Tian C, Duan Z, Duan Z. Making CEGAR more efficient in software model checking. IEEE Transactions on Software Engineering, 2014, 40(12): 1206–1223
94. Moffett Y, Dingel J, Beaulieu A. Verifying protocol conformance using software model checking for the model-driven development of embedded systems. IEEE Transactions on Software Engineering, 2013, 39(9): 1307–1325
95. Alrajeh D, Kramer J, Russo A, Uchitel S. Elaborating requirements using model checking and inductive learning. IEEE Transactions on Software Engineering, 2013, 39(3): 361–383
96. Leungwattanakit W, Artho C, Hagiya M, Tanabe Y, Yamamoto M, Takahashi K. Modular software model checking for distributed systems. IEEE Transactions on Software Engineering, 2014, 40(5): 483–501
97. Artzi S, Kiezun A, Dolby J, Tip F, Dig D, Paradkar A M, Ernst M D. Finding bugs in web applications using dynamic test generation and explicit-state model checking. IEEE Transactions on Software Engineering, 2010, 36(4): 474–494
98. Santone A. Heuristic search + local model checking in selective mu-calculus. IEEE Transactions on Software Engineering, 2003, 29(6): 510–523
99. Heitmeyer C L, Kirby J, Labaw B G, Archer M, Bharadwaj R. Using abstraction and model checking to detect safety violations in requirements specifications. IEEE Transactions on Software Engineering, 1998, 24(11): 927–948
100. Holzmann G J. The model checker SPIN. IEEE Transactions on Software Engineering, 1997, 23(5): 279–295
101. Bang K-S, Choi J-Y, Yoo C. Comments on "The Model Checker SPIN". IEEE Transactions on Software Engineering, 2001, 27(6): 573–576
102. Holzmann G J, Bosnacki D. The design of a multi core extension of the SPIN model checker. IEEE Transactions on Software Engineering, 2007, 33(10): 659–674
103. Kim M, Kim Y, Kim H. A comparative study of software model checkers as unit testing tools: an industrial case study. IEEE Transactions on Software Engineering, 2011, 37(2): 146–160
104. Chan W, Anderson R J, Beame P, Burns S, Modugno F, Notkin D, Reese J D. Model checking large software specifications. IEEE Transactions on Software Engineering, 1998, 24(7): 498–520
105. Chan W, Anderson R J, Beame P, Jones D H, Notkin D, Warner W E. Optimizing symbolic model checking for state charts. IEEE Transactions on Software Engineering, 2001, 27(2): 170–190
106. Cordeiro L C, Fischer B, Marques-Silva J. SMT-based bounded model checking for embedded ANSI-C software. IEEE Transactions on Software Engineering, 2012, 38(4): 957–974
107. Ikeda S, Jibiki M, Kuno Y. Coverage estimation in model checking with bits tate hashing. IEEE Transactions on Software Engineering, 2013, 39(4): 477–486
108. Paolieri M, Horvath A, Vicario E. Probabilistic model checking of regenerative concurrent systems. IEEE Transactions on Software Engineering, 2016, 42(2): 153–169
109. Su G, Feng Y, Chen T, Rosenblum D S. Asymptotic perturbation bounds for probabilistic model checking with empirically determined probability parameters. IEEE Transactions on Software Engineering, 2016, 42(7): 623–639
110. Chavez H M, Shen W, France R B, Mechling B A, Li G. An approach to checking consistency between UML class model and its Java implementation. IEEE Transactions on Software Engineering, 2016, 42(4): 322–344
111. Bouajjani A, Habermehl P, Rogalewicz A, Vojnar T. Abstract regular tree model checking of complex dynamic data Structures. In: Proceedings of the 13th International Static Analysis Symposium (SAS). 2006, 52–70
112. Schmidt D A, Steen B. Program analysis as model checking of abstract interpretations. In: Proceedings of the 5th International Static Analysis Symposium (SAS). 1998, 351–380
113. Podelski A. Model checking as constraint solving. In: Proceedings of the 7th International Static Analysis Symposium (SAS). 2000, 22–37
114. Cleaveland R, Iyer S P, Yankelevich D. Optimality in abstractions of model checking. In: Proceedings of the 2nd International Static Analysis Symposium (SAS). 1995, 51–63

115. Saidi H. Model checking guided abstraction and analysis. In: Proceedings of the 7th International Static Analysis Symposium (SAS). 2000, 377–396
116. Levi F. A symbolic semantics for abstract model checking. In: Proceedings of the 5th International Static Analysis Symposium (SAS). 1998, 134–151
117. Gallardo M, Merino P, Pimentel E. Refinement of LTL formulas for abstract model checking. In: Proceedings of the 9th International Static Analysis Symposium (SAS). 2002, 395–410
118. Monniaux D, Gonnord L. Using bounded model checking to focus fix point iterations. In: Proceedings of the 18th International Static Analysis Symposium (SAS). 2011, 369–385
119. Giacobazzi R, Quintarelli E. Incompleteness, counterexamples, and refinements in abstract model-checking. In: Proceedings of the 8th International Static Analysis Symposium (SAS). 2001, 356–373
120. Ranzato F, Tapparo F. Making abstract model checking strongly preserving. In: Proceedings of the 9th International Static Analysis Symposium (SAS). 2002, 411–427
121. Wehrle M, Helmert M. The causal graph revisited for directed model checking. In: Proceedings of the 16th International Static Analysis Symposium (SAS). 2009, 86–101
122. Fedyukovich G, D'IddioA C, Hyvarinen A E J, Sharygina N. Symbolic detection of assertion dependencies for bounded model checking. In: Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering (FASE). 2015, 186–201
123. Sharygina N, Browne J C. Model checking software via abstraction of loop transitions. In: Proceedings of the 6th International Conference on Fundamental Approaches to Software Engineering (FASE). 2003, 325–340
124. Xie F, Browne J C. ObjectCheck: a model checking tool for executable object-oriented software system designs. In: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering (FASE). 2002, 331–335
125. Xie F, Levin V, Browne J C. Integrated state space reduction for model checking executable object-oriented software system designs. In: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering (FASE). 2002, 64–79
126. Saffrey P, Calder M. Optimising communication structure for model checking. In: Proceedings of the 7th International Conference on Fundamental Approaches to Software Engineering (FASE). 2004, 310–323
127. Xie F, Levin V, Kurshan R P, Browne J C. Translating software designs for model checking. In: Proceedings of the 7th International Conference on Fundamental Approaches to Software Engineering (FASE). 2004, 324–338
128. Fantechi A, Gnesi S, Lapadula A, Mazzanti F, Pugliese R, Tiezzi F. A model checking approach for verifying COWS specifications. In: Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering (FASE). 2008, 230–245
129. Beyer D, Lowe S. Explicit-state software model checking based on CEGAR and interpolation. In: Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE). 2013, 146–162
130. Mota A, Sampaio A. Model-checking CSP-Z. In: Proceedings of the 1st International Conference on Fundamental Approaches to Software Engineering (FASE). 1998, 205–220
131. Baier C, Dubslaff C, Kluppelholz S, Daum M, Klein J, Marcker S, Wunderlich S. Probabilistic model checking and non-standard multi-objective reasoning. In: Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering (FASE). 2014, 1–16
132. Giannakopoulou D, Magee J. Fluent model checking for event-based systems. In: Proceedings of the 11th ESEC/FSE. 2003, 257–266
133. Dwyer M B, Carr V, Hines L. Model checking graphical user interfaces using abstractions. In: Proceedings of the 5th ESEC/FSE. 1997, 244–261
134. Tkachuk O, Dwyer M B. Adapting side effects analysis for modular program model checking. In: Proceedings of the 11th ESEC/FSE. 2003, 188–197
135. Choi Y, Rayadurgam S, Heimdahl M P E. Automatic abstraction for model checking software systems with interrelated numeric constraints. In: Proceedings of the 9th ESEC/FSE. 2001, 164–174
136. Gargantini A, Heitmeyer C L. Using model checking to generate tests from requirements specifications. In: Proceedings of the 7th ESEC/FSE. 1999, 146–162
137. Zervoudakis F, Rosenblum D S, Elbaum S G, Finkelstein A. Cascading verification: an integrated method for domain-specific model checking. In: Proceedings of the 21st ESEC/FSE. 2013, 400–410
138. Song F, Touili T. PoMMaDe: pushdown model-checking for mal ware detection. In: Proceedings of the 21st ESEC/FSE. 2013, 607–610
139. Liu S, Liu Y, Sun J, Zheng M, Wadhwa B, Dong J S. USMMC: a self contained model checker for UML state machines. In: Proceedings of the 21st ESEC/FSE. 2013, 623–626
140. Robby, Dwyer M B, Hatcliff J. Bogor: an extensible and highly modular software model checking framework. In: Proceedings of the 11th ESEC/FSE. 2003, 267–276
141. Naik M, Palsberg J. A type system equivalent to a model checker. ACM Transactions of Programming Languages and Systems, 2008, 30(5): 29
142. Yang J, Mok A K, Wang F. Symbolic model checking for event-driven real-time systems. ACM Transactions of Programming Languages and Systems, 1997, 19(2): 386–412
143. Emerson E A, Sistla A P. Utilizing symmetry when model-checking under fairness assumptions: an automata-theoretic approach. ACM Transactions of Programming Languages and Systems, 1997, 19(4): 617–638
144. Sistla A P, Godefroid P. Symmetry and reduced symmetry in model checking. ACM Transactions of Programming Languages and Systems, 2004, 26(4): 702–734
145. Bultan T, Gerber R, Pugh W. Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations,and experimental results. ACM Transactions of Programming Languages and Systems, 1997, 21(4): 747–789
146. Clarke E M, Grumberg O, Long D E. Model checking and abstraction. ACM Transactions of Programming Languages and Systems, 1994, 16(5): 1512–1542
147. Norris B, Demsky B. A practical approach for model checking C/C++11 code. ACM Transactions of Programming Languages and Systems, 2016, 38(3): 10

148. Siegel S F, Mironova A, Avrunin G S, Clarke L A. Combining symbolic execution with model checking to verify parallel numerical programs. ACM Transactions on Software Engineering and Methodology, 2008, 17(2): 10
149. Krishnamurthi S, Fisler K. Foundations of incremental aspect model-checking. ACM Transactions on Software Engineering and Methodology, 2007, 16(2): 7
150. Paige R F, Brooke P J, Ostroff J S. Metamodel-based model conformance and multiview consistency checking. ACM Transactions on Software Engineering and Methodology, 2007, 16(3): 11
151. Basu S, Smolka S A. Model checking the Java metalocking algorithm. ACM Transactions on Software Engineering and Methodology, 2007, 16(3): 12
152. Eshuis R. Symbolic model checking of UML activity diagrams. ACM Transactions on Software Engineering and Methodology, 2006, 15(1): 1–38
153. Sistla A P, Gyuris V, Emerson E A. SMC: a symmetry-based model checker for verification of safety and liveness properties. ACM Transactions on Software Engineering and Methodology, 2000, 9(2): 133–166
154. Chechik M, Devereux B, Easterbrook S M, Gurfinkel A. Multi-valued symbolic model-checking. ACM Transactions on Software Engineering and Methodology, 2003, 12(4): 371–408
155. Siegel S F, Mironova A, Avrunin G S, Clarke L A. Using model checking with symbolic execution to verify parallel numerical programs. In: Proceedings of ACM International Symposium on Software Testing and Analysis (ISSTA). 2006, 157–168
156. Gui L, Sun J, Liu Y, Si Y J, Dong J S, Wang X. Combining model checking and testing with an application to reliability prediction and distribution. In: Proceedings of ACM International Symposium on Software Testing and Analysis (ISSTA). 2013, 101–111
157. Fu X, Bultan T, Su J. Model checking XML manipulating software. In: Proceedings Proceedings of ACM International Symposium on Software Testing and Analysis (ISSTA). 2004, 252–262
158. Leesatapornwongsa T, Gunawi H S. SAMC: a fast model checker for finding heisenbugs in distributed systems (demo). In: Proceedings of ACM International Symposium on Software Testing and Analysis (ISSTA). 2015, 423–427
159. Chan W, Anderson R J, Beame P, Notkin D. Improving efficiency of symbolic model checking for state-based system requirements. In: Proceedings of ACM International Symposium on Software Testing and Analysis (ISSTA). 1998, 102–112
160. Groce A, Visser W. Model checking Java programs using structural heuristics. In: Proceedings of ACM International Symposium on Software Testing and Analysis (ISSTA). 2002, 12–21
161. Ramsay S J, Neatjerway R P, Ong C L. A type-directed abstraction refinement approach to higher-order model checking. In: Proceedings of the 41st POPL. 2014, 61–72
162. Flanagan C, Godefroid P. Dynamic partial-order reduction for model checking software. In: Proceedings of the 32nd POPL. 2005, 110–121
163. Chaki S, Rajamani S K, Rehof J. Types as models: model checking message-passing programs. In: Proceedings of the 29th POPL. 2002, 45–57
164. Schmidt D A, Data flow analysis is model checking of abstract interpretations. In: Proceedings of the 25th POPL. 1998, 38–48
165. Godefroid P. Model checking for programming languages using Verisoft. In: Proceedings of the 24th POPL. 1997, 174–186
166. Clarke E M, Grumberg O, Long D E. Model checking and abstraction. In: Proceedings of the 19th POPL. 1992, 342–354
167. Hsiung P-A, Chen Y-R, Lin Y-H. Model checking safety-critical systems using safe charts. IEEE Transactions on Computers, 2007, 56(5): 692–705
168. He F, Song X, Hung W N N, Gu M, Sun J. Integrating evolutionary computation with abstraction refinement for model checking. IEEE Transactions on Computers, 2010, 59(1): 116–126
169. Zheng H, Yao H, Yoneda T. Modular model checking of large asynchronous designs with efficient abstraction refinement. IEEE Transactions on Computers, 2010, 59(4): 561–573
170. Chen Y-R, Yeh J-J, Hsiung P-A, Chen S-J. Accelerating coverage estimation through partial model checking. IEEE Transactions on Computers, 2014, 63(7): 1613–1625
171. Zheng H, Zhang Z, Myers C J, Rodriguez E, Zhang Y. Compositional model checking of concurrent systems. IEEE Transactions on Computers, 2015, 64(6): 1607–1621
172. Roberson M, Boyapati C. Efficient modular glass box software model checking. In: Proceedings of OOPSLA. 2010, 4–21
173. Demsky B, Lam P. SATCheck: SAT-directed stateless model checking for SC and TSO. In: Proceedings of OOPSLA. 2015, 20–36
174. Jensen C S, Moller A, Raychev V, Dimitrov D, Vechev M T. Stateless model checking of event-driven applications. In: Proceedings of OOPSLA. 2015, 57–73
175. Darga P T, Boyapati C. Efficient software model checking of data structure properties. In: Proceedings of OOPSLA. 2006, 363–382
176. Roberson M, Harries M, Darga P T, Boyapati C. Efficient software model checking of soundness of type systems. In: Proceedings of OOPSLA. 2008, 493–504
177. Burckhardt S, Alur R, Martin M M K. CheckFence: checking consistency of concurrent data types on relaxed memory models. In: Proceedings of PLDI. 2007, 12–21
178. Huang J. Stateless model checking concurrent programs with maximal causality reduction. In: Proceedings of PLDI. 2015, 165–174
179. Kobayashi N, Sato R, Unno H. Predicate abstraction and CEGAR for higher-order model checking. In: Proceedings of PLDI. 2011, 222–233
180. Musuvathi M, Qadeer S. Fair stateless model checking. In: Proceedings of PLDI. 2008, 362–371
181. Choi Y. Safety analysis of trampoline OS using model checking: an experience report. In: Proceedings of the 22nd IEEE International Symposium on Software Reliability Engineering (ISSRE). 2011, 200–209
182. Liu Y, Sun J, Dong J S. PAT3: an extensible architecture for building multi-domain model checker. In: Proceedings of the 22nd IEEE International Symposium on Software Reliability Engineering (ISSRE). 2011, 190–199
183. Kim Y J, Kim M. Hybrid statistical model checking technique for reliable safety critical systems. In: Proceedings of the 23rd IEEE International Symposium on Software Reliability Engineering (ISSRE). 2012, 51–60
184. Letarte D. Conversion of fast inter-procedural static analysis to model

checking. In: Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM). 2010, 1–2

185. Yang G, Dwyer M B, Rothermel G. Regression model checking. In: Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM). 2009, 115–124
186. Sabetzadeh M, Nejati S, Liaskos S, Easterbrook S M, Chechik M.Consistency checking of conceptual models via model merging. In: Proceedings of the 15th IEEE International Requirements Engineering Conference (RE). 2007, 221–230
187. Fuxman A, Mylopoulos J, Pistore M, Traverso P. Model checking early requirements specifications in tropos. In: Proceedings of the 9th IEEE International Requirements Engineering Conference (RE). 2001, 174–181
188. Biere A, Cimatti A, Clarke E M, Zhu Y. Symbolic model checking without BDDs. In: Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). 1999, 193–207
189. McMillan K L. Symbolic Model Checking. Boston: Kluwer Academic Publishers, 1993
190. Holzmann G J, Smith M H. A practical method for the verifying event-driven software. In: Proceedings of the 21st International Conference on Software Engineering (ICSE). 1999, 597–607
191. Holzmann G J, Smith M H. Automating software feature verification. Bell Labs Technical Journal, 2000, 5(2): 72–87
192. Holzmann G J. Logic verification of ANSI-C code with SPIN. In: Proceedings of the 7th SPIN Workshop. 2000, 131–147
193. Holzmann G J. From code to models. In: Proceedings of the 2nd International Conference on Application of Concurrency to System Design. 2001, 3–10
194. Holzmann G J, Smith M H. An automated verification method for distributed systems software based on model extraction. IEEE Transactions on Software Engineering, 2002, 28(4): 364–377
195. Holzmann G J, Joshi R. Model driven software verification. In: Proceedings of the 11th SPIN Workshop. 2004, 76–91
196. Wooldridge M, Huget M P, Fisher M, Parsons S. Model checking for multi agent systems: the Mable language and its applications. International Journal on Artificial Intelligence Tools, 2006, 15(2): 195–226
197. Vortler T, Rulke S, Hofstedt P. Bounded model checking of Contiki applications. In: Proceedings of the 15th IEEE International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS). 2012, 258–261
198. Eisler S, Scheidler C, Josko B, Sandmann G, Stroop J. Preliminary results of a case study: model checking for advanced automotive applications. In: Proceedings of International Symposium on Formal Methods (FM). 2005, 533–536
199. Espada A R, Gallardo M, Salmeron A, Merino P. Using model checking to generate test cases for android applications. In: Proceedings of the 10th MBT. 2015, 7–21
200. Aceto L, Morichetta A, Tiezzi F. Decision support for mobile cloud computing applications via model checking. In: Proceedings of the 3rd Mobile Cloud. 2015, 199–204
201. Huang Y W, Yu F, Hang C, Tsai C H, Lee D T, Kuo S Y. Verifying Web applications using bounded model checking. In: Proceedings of IEEE International Conference on Dependable Systems and Networks. 2004, 199–208
202. Armando A, Carbone R, Compagna L, Li K, Pellegrino G. Model checking driven security testing of web-based applications. In: Proceedings of the 3rd IEEE International Conference on Software Testing, Verification, and Validation Workshops. 2010, 361–370
203. Li L, Miao H, Chen S. Test generation for web applications usingmodel-checking. In: Proceedings on 11th ACIS International Conference on Software Engineering Artificial Intelligence Networking and Parallel/Distributed Computing. 2010, 237–242
204. Choi E H, Watanabe H. Model checking class specifications for Web applications. In: Proceedings of the 12th Asia-Pacific Software Engineering Conference. 2005, 67–78
205. Sciascio E D, Donini F M, Mongiello M, Totaro R, Castelluccia D. Design verification of Web applications using symbolic model checking. In: Proceedings of the 5th International Conference on Web Engineering. 2005, 69–74
206. Nakajima S. Model-checking behavioral specification of BPEL applications. Electronic Notes in Theoretical Computer Science, 2006, 151(2): 19–105
207. Ghezzi C, Pezze M, Tamburrelli G. Adaptive REST applications viamodel inference and probabilistic model checking. In: Proceedings of IFIP/IEEE International Symposium on Integrated Network Management. 2013, 1376–1382
208. Gligoric M, Majumdar R. Model checking database applications. In: Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. 2013, 549–564
209. Cofer D D, Engstrom E, Goldman R P, Musliner D J, Vestal S. Applications of model checking at Honey well Laboratories. In: Proceedings of the 8th SPIN Workshop. 2001, 296–303
210. Cimatti A. Industrial applications of model checking. In: Proceedings of the 4th MOVEP. 2000, 153–168
211. Hoque K A, Mohamed O A, Savaria Y, Thibeault C. Early analysis of soft error effects for aerospace applications using probabilistic model checking. In: Proceedings of the 2nd FTSCS Workshop. 2013, 54–70
212. Hoque K A, Mohamed O A, Savaria Y. Towards an accurate reliability, availability and maintainability analysis approach for satellite systems based on probabilistic model checking. In: Proceedings the Design, Automation & Test in Europe Conference & Exhibition. 2015, 1635–1640
213. Armando A, Carbone R, Compagna L. SATMC: a SAT-based model checker for security-critical systems. In: Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. 2014: 31–45
214. Loding H. Model-based scenario testing and model checking with applications in the railway domain. Dissertation for the Doctoral Degree, 2014
215. Azimi S, Gratie C, Ivanov S, Manzoni L, Petre I, Porreca A E. Complexity of model checking for reaction systems. Theoretical Computer Science, 2016, 623: 103–113
216. Zuliani P. Statistical model checking for biological applications. International Journal on Software Tools for Technology Transfer, 2015, 17(4): 527–536
217. Clarke E M, Faeder J R, Langmead C J, Harris L A, Jha S K, Legay A. Statistical model checking in biolab: applications to the automated analysis of T-cell receptor signaling pathway. In: Proceedings of International Conference on Computational Methods in Systems Biol-

ogy. 2008, 231–250

218. Blackham B, Heiser G. Sequoll: a framework for model checking binaries. In: Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium. 2013, 97–106
219. Christodorescu M, Jha S. Static analysis of executables to detect malicious patterns. In: Proceedings on 12th USENEX Security Symposium. 2003
220. Merz S. Model checking: a tutorial overview. In: Cassez F, Jard C, Rozoy B, et al, eds. Modelling and Verification of Parallel Processes. Berlin: Springer-Verlag, 2001, 3–38
221. Chen J, Zhou H, Bruda S D. Combining model checking and testing for software analysis. In: Proceeding of International Conference on Computer Science and Software Engineering. 2008, 206–209
222. Steffen B. Data flow analysis as model checking. In: Proceedings of International Symposium on Theoretical Aspects of Computer Software. 1991, 346–365
223. Bryant R E. Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers, 1986, 100(8), 677–691
224. Valmari A. A stubborn attack on state explosion. In: Proceedings of the 2nd International Conference on Computer Aided Verification (CAV). 1990, 156–165
225. Peled D A. All from one, one for all: model checking using representatives. In: Proceedings of the 5th International Conference on Computer Aided Verification (CAV). 1993, 409–423
226. Godefroid P. Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem. Dissertation for the Doctoral Degree, 1996
227. Yang Y, Chen X, Gopalakrishnan G, Kirby R M. Distributed dynamic partial order reduction based verification of threaded software. In: Proceedings of the 14th SPIN Workshop. 2007, 58–75
228. Kahlon V, Wang C, Gupta A. Monotonic partial order reduction: an optimal symbolic partial order reduction technique. In: Proceedings of the 21st International Conference on Computer Aided Verification (CAV). 2009, 398–413
229. Tasharofi S, Karmani R K, Lauterburg S, Legay A, Marinov D, Agha G. Trans DPOR: a novel dynamic partial-order reduction technique for testing actor programs. In: Proceedings of Formal Techniques for Distributed Systems. 2012, 219–234
230. Abdulla P A, Aronis S, Jonsson B, Sagonas K F. Optimal dynamic partial order reduction. In: Proceedings of the 41st POPL. 2014, 373–384
231. Sheeran M, Singh S, Stalmarck G. Checking safety properties using induction and a SAT-solver. In: Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design. 2000, 108–125
232. McMillan K L. Applying SAT methods in unbounded symbolic model checking. In: Proceedings of the 14th International Conference on Computer Aided Verification (CAV). 2002, 250–264
233. McMillan K L. Interpolation and SAT-based model checking. In: Proceedings of the 15th International Conference on Computer Aided Verification (CAV). 2003, 1–13
234. Ganai M K, Gupta A, Ashar P. Efficient SAT-based unbounded symbolic model checking using circuit cofactoring. In: Proceedings of IEEE/ACM International Conference on Computer-aided Design. 2004, 510–517
235. Gunther H, Weissenbacher G. Incremental bounded software model checking. In: Proceedings of SPIN Workshop, 2014, 40–47
236. Tinelli C. SMT-based model checking. In: Proceedings of the 4th NASA Formal Methods. 2012
237. Garcia M, Monteiro F R, Cordeiro L C, Filho E B L. ESBMC: a bounded model checking tool to verify Qt applications. In: Proceedings of the 23rd SPIN Workshop. 2016, 97–103
238. Hinton A, Kwiatkowska M Z, Norman G, Parker D. PRISM: a tool for automatic verification of probabilistic systems. In: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). 2006, 441–444
239. Hartmanns A, Timmer M. Sound statistical model checking for MDP using partial order and confluence reduction. International Journal on Software Tools for Technology Transfer, 2015, 17(4): 429–456
240. Jegourel C, Legay A, Sedwards S. Command-based importance sampling for statistical model checking. Theoretical Computer Science, 2016, 649: 1–24
241. Geldenhuys J. State caching reconsidered. In: Proceedings of the 11th SPIN Workshop. 2004, 23–38
242. Clarke E M. Counterexample-guided abstraction refinement. In: Proceedings of TIME-ICTL. 2003, 7–8
243. Rabbi F, Wang H, MacCaull W, Rutle A. A model slicing method for work flow verification. Electronic Notes in Theoretical Computer Science, 2013, 295: 79–93
244. Self J P, Mercer E G. On-the-Fly dynamic dead variable analysis. In: Proceedings of the 14th SPIN Workshop. 2007, 113–130
245. Evangelista S. Dynamic delayed duplicate detection for external memory model checking. In: Proceedings of the 15th SPIN Workshop. 2008, 77–94
246. Chen Z, Motet G. Never trace claims for model checking. In: Proceedings of the 17th SPIN Workshop. 2010, 162–179
247. Lugiez D, Niebert P, Zennou S. Dynamic bounds and transition merging for local first search. In: Proceedings of the 9th SPIN Workshop. 2002, 221–229
248. Holzmann G J. State compression in SPIN: recursive indexing and compression training runs. In: Proceedings of the 3rd SPIN Workshop. 1997, 1–10
249. Nguyen V Y, Ruys T C. Incremental hashing for Spin. In: Proceedings of the 15th SPIN Workshop. 2008, 232–249
250. Rangarajan M, Dajani-Brown S, Schloegel K, Cofer D D. Analysis of distributed SPIN applied to industrial-scale models. In: Proceedings of the 11th SPIN Workshop. 2004, 267–285
251. Melatti I, Palmer R, Sawaya G, Yang Y, Kirby R M, Gopalakrishnan G. Parallel and distributed model checking in Eddy. In: Proceedings of the 13th SPIN Workshop. 2006, 108–125
252. Laster K, Grumberg O. Modular model checking of software. In: Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). 1998, 20–35
253. Brim L, Crhova J, Yorav K. Using assumptions to distribute CTL model checking. Electronic Notes in Theoretical Computer Science, 2002, 68(4): 559–574
254. Holzmann G J, Bosnacki D. Multi-core model checking with SPIN. In: Proceedings of IEEE International Parallel and Distributed Processing Symposium. 2007, 1–8

255. Laarman A, Pol J, Weber M. Parallel recursive state compression for free. In: Proceedings of the 18th SPIN Workshop. 2011, 38–56

256. Ditter A, Ceska M, Luttgen G. On parallel software verification using boolean equation systems. In: Proceedings of the 19th SPIN Workshop. 2012, 80–97

257. Holzmann G J. Parallelizing the SPIN model checker. In: Proceedings of the 19th SPIN Workshop. 2012, 155–171

258. Burns E, Zhou R. Parallel model checking using abstraction. In: Proceedings of the 19th SPIN Workshop. 2012, 172–190

259. Barnat J, Brim L, Simecek P. I/O efficient accepting cycle detection. In: Proceedings of the International Conference on Computer Aided Verification (CAV). 2007, 281–293

260. Barnat J, Brim L, Simecek P, Weber M. Revisiting resistance speed sup i/o-efficient LTL model checking. In: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). 2008, 48–62

261. Edelkamp S, Sanders P, Simecek P. Semi-external LTL model checking. In: Proceedings of the 20th International Conference on Computer Aided Verification (CAV). 2008, 530–542

262. Wu L, Huang H, Su K, Cai S, Zhang X. An i/o efficient model checking algorithm for large-scale systems. IEEE Transactions on Very Large Scale Integration Systems, 2015, 23(5): 905–915

263. Ganai M K, Wang C, Li W. Efficient state space exploration: interleaving stateless and state-based model checking. In: Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD). 2010, 786–793

264. Evangelista S, Kristensen L M. Combining the sweep-line method with the use of an external-memory priority queue. In: Proceedings of the 19th SPIN Workshop. 2012, 43–61

265. Havelund K. Java Path Finder, a translator from Java to Promela. In: Proceedings of the 6th SPIN Workshop. 1999

266. Cimatti A, Clarke E M, Giunchiglia F, Roveri M. NUSMV: a new symbolic model checker. International Journal on Software Tools for Technology Transfer, 2000, 2(4): 410–425

267. Sun J, Liu Y, Roychoudhury A, Liu S, Dong J S. Fair model checking with process counter abstraction. In: Proceedings of International Symposium on Formal Methods. 2009, 123–139

268. Klein J, Baier C, Chrszon P, Daum M, Dubslaff C, Kluppelholz S, Marcker S, Muller D. Advances in symbolic probabilistic model checking with PRISM. In: Proceedings of the 22nd nternational Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). 2016, 349–366

269. Godefroid P, Levin M Y, Molnar D A. SAGE: white box fuzzing for security testing. Communications of the ACM, 2012, 55(3): 40–44

270. Thomas A H, Ranjit J, Rupak M, Gregoire S. Software verification with BLAST. In: Proceedings of the 10th SPIN Workshop. 2003, 235–239

271. James C K. Symbolic execution and program testing. Communications of the ACM, 1976, 19(7): 385–394

272. Ciortea L, Zamfir C, Bucur S, Chipounov V, Candea G. Cloud9: a software testing service. ACM SIGOPS Operating Systems Review, 2010, 43(4): 5–10

273. Karna A K, Du J, Shen H, Zhong H, Gong J, Yu H, Ma X, Zhao J. Tuning parallel symbolic execution engine for better performance. Frontiers of Computer Science. 2018, 12(1): 86–100

274. Khurshid S, Pasareanu C S, Visser W. Generalized symbolic execution for model checking and testing. In: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). 2003, 553–568

275. Amjad H. Combining model checking and theorem proving. Dissertation for the Doctoral Degree, 2004

276. Uribe T E. Combinations of model checking and theorem proving. In: Proceedings of the 3rd International Workshop on Frontiers of Combining Systems (FroCoS). 2000, 151–170

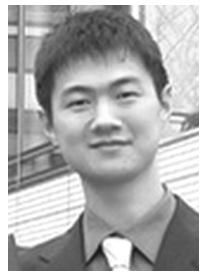
277. Hungar H. Combining model checking and theorem proving to verify parallel processes. In: Proceedings of the 5th International Conference on Computer Aided Verification (CAV). 1993, 154–165

278. Dybjer P, Haiyan Q, Takeyama M. Verifying Haskell programs by combining testing, model checking and interactive theorem proving. Information & Software Technology, 2004, 46(15): 1011–1025

279. Amjad H. Verification of AMBA using a combination of model checking and theorem proving. Electronic Notes in Theoretical Computer Science, 2006, 145: 45–61

280. Seidel P. A case for multi-level combination of theorem proving and model checking tools. In: Proceedings of the 15th MTV Workshop. 2014, 90–97

Anil Kumar Karna received his BE degree in computer science from Huazhong University of Science and Technology, China in 2004, and his MS degree in software engineering from Shanghai Jiao Tong University (SJTU), China in 2010. He worked as web-master, lead IT support engineer, and IT instructor in Nepal, India, and China, respectively from 2004 to 2007. He joined STAP lab as a PhD candidate in the Department of Computer Science and Engineering at SJTU. His current research interests include software reliability, system crashes, program analysis, testing & debugging, and verification & validation.



Yuting Chen received the BS and MS degrees in computer science from Nanjing University, China in 2000 and 2003, respectively. He received the PhD degree in computer science from Hosei University, Japan in 2007. Yuting Chen is now an associate professor at the Computer Science Department of Shanghai Jiao Tong University, China. Chen's research interests include program analysis, software testing, verification, and validation.

Haibo Yu received her BS degree in computer science from Tsinghua University, China in 1987. She received her MS and PhD degrees in computer science from Kyushu University, Japan in 1999 and 2009, respectively. She joined the School of Software at Shanghai Jiao Tong University, China in March, 2010. Her research interests include software engineering, information retrieval and web application systems.

Hao Zhong received his BS degree from Civil Aviation University of China, China in 2002; and both his MS and PhD degrees from Peking University, China in 2005 and 2009, respectively. After graduation, he worked as an assistant professor at Institute of Software, Chinese Academy of Sciences, China, and was promoted as an associate professor in 2012. From 2013 to 2014, he was a visiting scholar at University of California, Davis. Since 2014, he has become an associate professor at Shanghai Jiao Tong University, China. He is a member of the IEEE and ACM. His research interest is in the area of software engineering, with an emphasis on mining software repositories.

Jianjun Zhao received the BS degree in computer science from Tsinghua University, China in 1987, and the PhD degree in computer science from Kyushu University, Japan in 1997. He is now a professor in the Department of Advanced Information Technology, Kyushu University, Japan. Before joining the faculty in April 2016, he was a professor in the School of Software and also the Department of Computer Science and Engineering from November 2005 to March 2016, at Shanghai Jiao Tong University, China. His research interests include software engineering and programming language, especially program analysis, software testing, code searching and automatic programming.