Softw Syst Model (2012) 11:541–555
DOI 10.1007/s10270-012-0281-9

EXPERT'S VOICE

The hidden models of model checking

Willem Visser · Matthew B. Dwyer · Michael Whalen

Received: 5 December 2011 / Revised: 23 June 2012 / Accepted: 9 July 2012 / Published online: 24 August 2012

Abstract In the past, applying formal analysis, such as model checking, to industrial problems required a team of formal methods experts and a great deal of effort. Model checking has become popular, because model checkers have evolved to allow domain-experts, who lack model checking expertise, to analyze their systems. What made this shift possible and what roles did models play in this? That is the main question we consider here. We survey approaches that transform domain-specific input models into alternative forms that are invisible to the user and which are amenable to model checking using existing techniques—we refer to these as *hidden* models. We observe that keeping these models hidden from the user is in fact paramount to the success of the domain-specific model checker. We illustrate the value of hidden models by surveying successful examples of their use in different areas of model checking (hardware and software) and how a lack of suitable models hamper a new area (biological systems).

Keywords Model checking · Models temporal logic · Biological systems

Communicated by Prof. Jon Whittle and Gregor Engels.

W. Visser (✉) · M. B. Dwyer
Computer Science, Stellenbosch University,
Stellenbosch, South Africa
e-mail: visserw@sun.ac.za

M. B. Dwyer
Department of Computer Science and Engineering,
University of Nebraska-Lincoln, Lincoln, NE, USA
e-mail: dwyer@cse.unl.edu

M. Whalen
Software Engineering Center, University of Minnesota,
Minneapolis, MN, USA
e-mail: whalen@cs.umn.edu

1 Introduction

Model checking, since its introduction in the late 1970s, has become hugely popular, so much so that Clarke, Emerson and Sifakis received the 2007 Turing award for their pioneering work in the field. Although much of the early work on model checking focused on manual creation of models in the notation of the model checker's input, as the field matured there was a shift to model checkers that directly take domain-specific notations as input. We believe this shift is what made model checking such a successful technique, since it not only allowed domain experts that do not know how a model checker works to use one, but also allowed the model checkers to exploit domain knowledge to become more efficient. We argue that the key to making domain-specific model checking possible is the use of *hidden* models that are used during the transformation of domain notations to model checking input to support efficient analysis and to allow domain-specific output to reach the user.

The contribution of this work is to take a fresh look at model checking by surveying the roles that models play in different domains. Mostly, one considers just the input models when thinking of model checking, but here we consider the models you do not see, i.e. the hidden models, and highlight the significant role they play. We contend that it is when hidden models are exposed to users that model checking fails to be useful in practice. An example of a hidden model being exposed is when the translation to the model checker's input is not completely automatic. In such a scenario, a domain expert not only has to understand how to model the system they are interested in studying and the properties of that system that they would like to check, but they must then become expert in the input languages that a particular model checking tool accepts. Moreover, in all likelihood, they will need to understand how the model checker operates to produce

an encoding of their system for which the model checker is efficient.

The main message of this paper is that model checking becomes effective in a domain (a) when the natural notations in that domain are supported, and (b) when the models that are used to efficiently perform model checking are hidden from end users in the domain. It is not to advocate for specific domain-level notations, but rather to describe how these models can be transformed to effectively apply model checking techniques. As model checking is applied in new domains, these lessons and the techniques developed for model checking in existing domains should be leveraged to maximize the utility of model checking.

In the rest of the paper, we will give a brief introduction to model checking and the typical architecture of a domain-specific model checker (including the role of hidden models). We will then take two mature fields, namely model checking applied to hardware and software, to show how hidden models are used to allow efficient model checking. Model checking of biological processes in contrast is a new field, and we will discuss emerging approaches to model checking in this domain and the need for broader application of the lessons learned from software and hardware model checking.

2 Model checking

Model checking is an algorithmic verification technique that is particularly effective for finite-state systems. It was co-invented by Clarke and Emerson [22] and Queille and Sifakis [86] over three decades ago. Since that time, there have been dramatic advances in both model checking algorithms and the development of tools that implement those algorithms efficiently. Since the mid-1990s, such tools have been readily available and have been widely applied to reason about correctness properties of computer systems.

Model checking involves two inputs: a *state transition system* (S) that encodes a set of behaviors to be reasoned about and a *temporal logic formula* (ϕ) that encodes a set of desirable behaviors. The model checking problem asks whether S is a *logical model* [1] of ϕ, i.e., whether the behaviors encoded by S satisfy ϕ. Algorithms for model checking are able to answer this question and when the answer is negative they produce a *counterexample*, i.e., a behavior encoded by S that falsifies ϕ. Counterexamples provide valuable information for locating errors in the transition system model or in the temporal logic formula.

Often times one wishes to reason about a system that interacts with its environment, e.g., a software or hardware component. To apply model checking, the behavior of both the system and the environment must be incorporated into the transition system that is fed as input to the model checker. This is sometimes referred to as *closing* the system.

Model checking has been an active area of research for almost 30 years, and several different analysis approaches have been created. These approaches can be characterized in several dimensions, of which we consider three: (1) the language features of the models S, (2) the logics that can be used for describing properties over the models (ϕ), and (3) the algorithms used to perform the analysis. For language features, dimensions include whether the model is finite-state or infinite-state, describes system evolution using discrete time [24] or real time [53], and whether the behavior of the model is described using probabilities [63]. For logics for ϕ, some approaches allow only specification of *invariants*, which are formulae that must always be true in every state of the model. More complex logics such as linear temporal logic (LTL) [83] and computation tree logic (CTL) [24] allow specification of *liveness properties* that describe situations that must be eventually true in a model. Probabilistic temporal logics [63] allow one to reason about probabilistic models in terms of the likelihood that a formula is true. For algorithms, it is possible to broadly classify them into *explicit state* and *symbolic* algorithms [24]. With explicit state algorithms, the states of the system are generated on-the-fly as the algorithm proceeds, and tools are often distinguished by the brevity of the encoding. With symbolic algorithms, the model and set of explored states are represented symbolically as Boolean formulae. For symbolic approaches, there are two main "engines" that are used to solve the generated Boolean formulae: SAT/SMT solvers [38] and Binary-Decision Diagram solvers [12]. The choice of solver leads to different classes of algorithms for model checking.

There are many model checkers available, but two of the most popular are Spin[2] [54] and NuSMV[3] [20]. Figure 1 presents excerpts of inputs and outputs for the Spin model checker. On the left is the encoding of a transition system in Spin's PROMELA language. Since this particular system takes no input, except for the decisions about scheduling the order of execution of statements from the two `user` processes, there is no need for an encoding of the system's environment. Researchers have collected a variety of models for Spin [67] and NUSMV [66] to promote comparative

[1] When discussing applications of model checking the use of the word "model" has lead to some confusion. Typically the transition system that is provided to a model checker represents an abstract model that captures, e.g., a hardware design description or a software implementation. In this context it makes sense to "check" the "model" to detect errors and this is precisely what that application of model checking does. The word "model" in model checking refers, however, to the fact that model checking determines whether the system is a "logical model" of the property specification, i.e., whether the system satisfies the property.

[2] http://spinroot.com.

[3] http://nusmv.fbk.eu.

PROMELA

```
byte n, x, y, z;
active [2] proctype user(){
   byte me = _pid+1;
L: x = me;
   if
   :: (y != 0 && y != me)
      -> goto L
   :: (y == 0 || y == me)
   fi;
   ...
   if
   :: (z != me) -> goto L
   :: (z == me)
   fi;
   n = n+1;
   assert(n == 1);
   n = n - 1;
   goto L
}
```

LTL Formula

```
ltl { [](init -> [](open -> <>close)) }
```

Counterexample

```
1:  proc 1 f:5 [x = me]
2:  proc 1 f:7 [(y==0)||(y==me)]
3:  proc 1 f:10 [z = me]
4:  proc 1 f:12 [x==me]
5:  proc 0 f:5 [x = me]
...
51:  proc 1 f:15 [y = me]
52:  proc 1 f:17 [z==me]
53:  proc 1 f:21 [n = n+1]
assertion violated (at depth 54)
spin: trail ends after 54 steps
```

Fig. 1 SPIN PROMELA model, Counterexample, and LTL formula

evaluation—these also illustrate the diversity of systems that have been targetted with model checking.

On the upper right in Fig. 1 is a formula written in Spin's syntax for LTL. This specification assumes that boolean variables are present in the system indicating when the system is *init*ialized, when a resource is *open* and when it has been *close*d. The LTL formula states that after initialization a call to open will eventually be followed by a call to close. The `open`, `close`, and `init` must be defined in terms of the transition system model; they could capture the calls to a file open method, a file close method, and the creation of a file instance for a file API. Researchers have collected a variety of models from the literature [30,31] that illustrate the diversity of specification languages and properties targetted with model checking.

On the lower right in Fig. 1 is an excerpt of a counterexample which is composed of a sequence of 54 statements, indicated by the line number in a file (`f:10`) and the bracketed statement (`[z = me]`), that end in a violation of a property specified as an assertion.

3 Domain-specific model checking

Model checkers have proven their worth in performing certain forms of automated reasoning. For example, Microsoft's Static Driver Verifier is a toolset that model checks device driver implementations for correct usage of Kernel APIs and is a part of the regular device driver development process [77], IBM has applied their RuleBase model checker to numerous systems including their 1.3Ghz Power4 microprocessor design [8], and Rockwell Collins has integrated multiple model checking algorithms into a framework that has been applied to components of avionics software [73].

While researchers continue to improve the scalability and generality of the core model checking algorithms, there is a significant body of work focused on applying model checking to reason about problems in a variety of domains. Effective application of model checking requires that the input (output) models that are consumed (produced) by a model checker be *hidden* from domain engineers.

Figure 2 shows the architecture of a domain-specific model checker. The important thing to notice is that although the model checker itself is central to the process, there are a number of intermediate models flowing into, out of, and within the actual model checker. These models within the inner box of Fig. 2 (Automation boundary) are the hidden models of the model checking process. In the following sections, we will argue that the ultimate success of a model checker is heavily dependent on the efficiency of the analysis of these hidden models. Here, we will assume all translations between models are correct, but it should be acknowledged that proving such translations correct are non-trivial.

Let us first consider the three *visible* models that form the input:

Domain model This is the artifact that requires analysis. This model is related to but distinguished from the Logical Model that the model checker ultimately analyzes.

Environment model As stated in Sect. 2, model checkers analyze a closed system, i.e. a system composed with its execution environment. The so-called Environment Model is often an abstraction (both under- or over-approximation) of the actual environment.

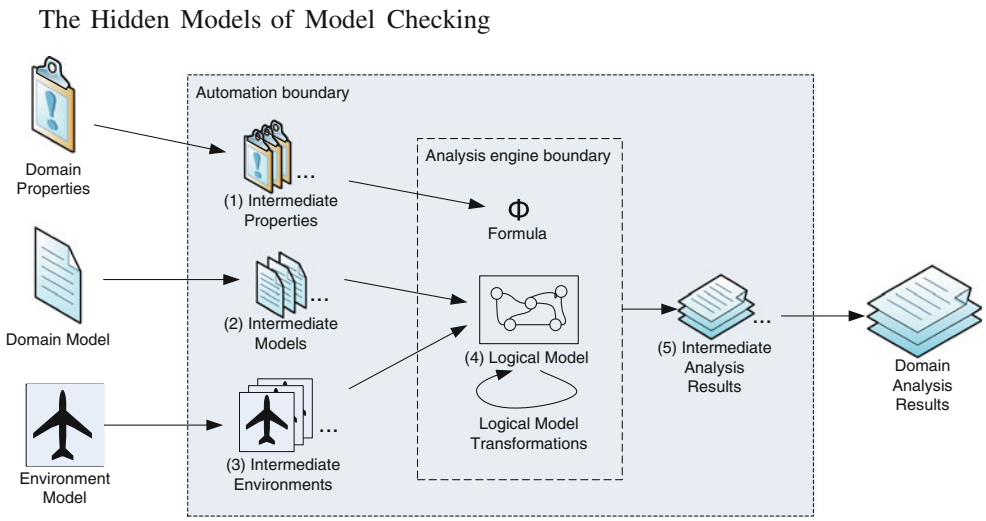

Fig. 2 The architecture of a domain-specific model checker

Domain properties The properties that the model checker must check of the composition of the Domain and Environment model. These can be given explicitly as logical properties, for example in temporal logic, or as implicit properties, e.g., the system should never deadlock, or the code never generates uncaught runtime exceptions.

The output of a model checker is typically an artifact that can be interpreted with respect to the input Domain Model. In the simple case if the property being checked holds, this might just state "Correct". The more common scenario, however, is for the model checker to produce a counterexample that illustrates why a property is being violated. This counterexample is another *visible* model of the model checking process. The remainder of this section discusses these different classes of models.

3.1 Domain model

When using an analysis technique such as model checking there is a *system* whose behaviors are the subject of the analysis. While a system might be expressed in any number of notations, typically, when working in a particular problem domain there are notations that are both familiar and convenient. For example, tele-communication protocols are often specified in SDL [60], software is expressed in its source language, such as Java, or C, and hardware components might be expressed as VHDL [56]. These notations have primitives that are designed to make descriptions concise and familiar to the engineers creating and consuming them.

The inputs and outputs of a mature domain-specific model checking system are typically quite different than the inputs and outputs of a specific model checking tool. In Fig. 2, this *semantic gap* is addressed by "(2) Intermediate Models" that transform the domain model to the input of the model checker. A successful domain specific model checker, bridges this semantic gap in a fully automated fashion. Requiring user assistance to construct this model means that only model checking specialists can use the system. Equally important, this effort must be repeated every time the system changes and is to be re-analyzed, significantly increasing the cost of analysis.

3.2 Hiding semantic model mismatches

Historically, model checkers have been designed with problems in a specific domain of application in mind. For example, Spin was developed to reason about network protocols and, consequently, PROMELA is well-suited for expressing such problems. When applying model checkers beyond their originally intended domain of application, one often encounters significant semantic mismatches between domain-specific notations and model checker notations.

One common approach to bridging the semantics from domain languages to model checker input notations is to develop appropriate intermediate representations (IR)—these are analogs of the IRs used in compilers. A number of succesful model checking projects have used this approach, e.g., SLAM's boolean programs [5], Bandera's BIR [59], and the TOPCASED project's FIACRE language [10].

Even with a well-designed IR numerous challenges must be addressed in achieving an efficient and effective translation. Initial efforts to model check Java programs using Spin [26,51] had to address the lack of heap allocated data, references, garbage collection, inheritance, and exception handling. It is widely accepted that automated techniques were needed to bridge the gap between domain models, e.g., Java programs, and model checker inputs, e.g., PROMELA, since

relying on human users to express their problems in model checker notations is costly and error prone. Moreover, when counterexamples are produced they describe property violations in terms of the model checker's input not the original domain-specific model. Thus, a kind of reverse translation must be performed and not just once, but for every counterexample generated.

Different model checking algorithms have different strengths. For example, for sequential software with recursion push-down model checking algorithms [34] have proven effective, whereas with multi-threaded software reduced explicit-state model checking [54,95] is more effective. Since programming languages, such as Java, can be used to express different types of software systems it makes sense to support the translation to multiple model checking algorithms [26,87]. Support for multiple model checking techniques only exacerbates the problem of mapping domain models to model checker inputs.

Finally, model checking tools are built to support the semantics of their input language. When expressing a domain model in such a language the semantics of the domain model are lost. For instance, when translating Java to PROMELA the semantics of the JVM memory model cannot be encoded in a form that can be exploited by Spin's partial order reduction algorithms [41]. This leads to suboptimal performance since memory model semantics, e.g., thread local heap data, can be exploited for significant state space reduction [32]. In our view, the hidden models should carry relevant semantics to the model checking algorithms, so that they can be exploited for good performance, but this only serves to further complicate those models, thus making them more difficult for domain experts to create and understand.

In summary, model checker notations should be hidden from domain engineers to relieve them of the burden of translating to and from model checker specific notations.

3.3 Environment model

No useful system stands alone—they interact with other systems and/or with the physical world. A system has an *interface* which defines how it receives and transmits values to entities outside the system, which we refer to as the system's *environment*. The environment performs complementary actions with respect to the system, i.e., when the system receives a value the environment has to have previously transmitted it, and one models the behavior of the environment at the system's interface as an *environment model*.

Unlike with domain modeling, standards for defining environment models have not been widely adopted. For software environments often one uses a mixture of source code with special primitives that allow for any value from a set of possibilities to be chosen—these simulate the lack of detailed knowledge about what the environment will do next.

Creating the environment model is one of the most difficult parts of model checking [92], and we argue this is because there are no hidden models readily available to simplify the process for a domain expert. For example, a domain expert might create a model that allows any interaction accepted by the domain model, this is referred to as the universal environment model, but this model can cause the model checker to fail due to a state-explosion. Conversely, the expert might provide too restricted an environment that allows a tool to "prove" a property that may not hold in the system's actual environment. The former is an example of an over-approximation and the latter an under-approximation abstraction of the environment; however, producing just the correct level of abstraction is very hard and the essence of the assume-guarantee model checking field [39,79].

In areas where there have been significant industrial success with software model checking systems, it is often the case that the domain has been limited to the extent that environment models can be created and packaged with the model checking system. An example of this is the SLAM Driver Verifier which comes with its own model of the Windows Kernel functions [4] and thus allows device driver developers to run the tool on their new drivers. A similar approach is taken in JPF[4] [95] and Pex & Moles[5] [28] where mechanisms were added to override any existing method call with a model of the call; this allows one to replace complex Java and .NET library calls with simplified versions. For hardware analysis, it is often the case that the systems must perform correctly in a universal or nearly universal environment; while these environments can be expensive to analyze, they are simple to construct.

3.4 Property model

Unlike domain or environment modeling, the process of describing a desirable system *property* is much more focused. Rather than describing all desirable system behaviors in one property, typically one describes how a certain part of the system should behave in a specific circumstance. For example, one might specify that when a component is given a positive value, then it should compute the square of that value— such a description leaves out the behavior of the component for non-positive inputs.

In certain domains, notations for property modeling are widely accepted. For example, after several decades during which competing temporal logics were used for specifying hardware properties in 2005 IEEE standardized the property specification language (PSL) [33]. For software, the use of *contracts*, e.g., [65], has gained considerable momentum as

[4] http://babelfish.arc.nasa.gov/trac/jpf/.

[5] http://research.microsoft.com/en-us/projects/pex/.

a means of specifying the input/output behavior of software modules.

3.5 Analysis results model

When a model checker detects a violation of a property it generates a representation of the ways in which the system's behavior violates the specified property—a counterexample. The nature of the counterexample depends on the type of specification. For example, the violation of a property stating that variable x is always positive would be described as a sequence of steps in the system execution, starting at the initial state and interacting with the environment model as appropriate that ends in a state in which $x \leq 0$.

Most existing software model checking tools that we are aware of provide support for interpreting the counterexample as an execution of the system model, i.e., the program source code [4,26,95]. In hardware, counterexamples are generally provided as input/output traces for the circuit to be analyzed that can be replayed as test cases through simulation tools [57,68].

Note that although most emphasis in model checking is placed on the result where a counterexample is produced there are two other options: the property holds or the model checker ran out of resources before producing a result. When a property is true of a domain model and its environment, there is the option that the wrong property was specified (or the property was specified incorrectly). In this case, the ideal results model should contain a justification of why the property holds, which would in turn allow the domain expert to determine if the property was specified correctly. Vacuity checking of the property [9] and coverage data of the domain model [47] are additional sources of information used to validate whether the property checked was the *right* property. When resource limits are exceeded, coverage data can also give an indication of the extent to which the model checking was able to explore the set of system behaviors. More research is needed to explore how best to leverage the results of such *partial* model checks in future model checking runs.

3.6 Hiding transient models

Model checking can be costly—it can consume time and space that is exponential in size of the input transition system. Consequently, most of the research on model checking algorithms over the past several decades has focused on developing state space reduction techniques to improve time and space performance. During the application of these reductions, models are created that preserve relevant behavior of the original model, but simplify the model checking. Since those models only exist during the model checking run they are referred to as *transient*.

Many reduction techniques can be viewed as constructing a transition system model that preserves the ability to answer the model checking question (or at least to answer it in the affirmative). For example, property directed model slicing [50] eliminates portions of the transition system that are provably unrelated to the temporal logic formula under analysis. Partial order reductions (POR), e.g., [41] and symmetry reductions [58] effectively ignore behaviors of the input transition system that are provably equivalent relative to the temporal logic formula under analysis. Essentially, they construct a reduced model on-the-fly during analysis. Predicate abstraction [44] techniques transform the transition system model by selectively replacing non-deterministic choice with abstracted branch predicates to sharpen the precision of analysis results while minimizing the cost of analysis.

In all of these cases, the preservation properties of the reduction techniques mean that the answer to the model checking question is the same as it would have been on the original transition system, it is just computed more quickly—in some cases orders of magnitude more quickly [32].

In addition to the translation performed from domain models into the source notation of a model checking tool, there is often a significant amount of translation from the source notation of the model checking tool to its underlying representation. Model checkers operate by creating transient models of (portions of) the state space being analyzed. For example, in automata-theoretic model checking, both the model and the formula to be analyzed are translated on-the-fly into Büchi Automata [90], and these representations are intersected to create a *product automata* [94].

Similarly, there are several different transient models created during symbolic model checking depending on the algorithm used. For *transition relation* based approaches, e.g., [20], a Boolean formula is created that describes a single computational step (transition) of the system. This encoding requires flattening the structure of the model and encoding the computations as operations over Boolean formulae. The encodings are usually very large and not easily readable or traceable (by humans) to the source language.

Symbolic approaches work by manipulating formulae that describe the computation of the model. In the case of fixpoint approaches such as interpolation [72] and McMillan's initial symbolic algorithm [71], the transition relation is conjoined with a formula that represent the set of states that have been explored. Other approaches (such as bounded model checking [25] and k-induction [49]) create models through successive unrollings of the transition relation to describe the behavior of the system over several computational steps.

In addition, as the formulae are being solved, it is often the case that the solver changes the formula encoding to make it more efficient to solve. For example, a Binary Decision Diagram (BDD) is a directed acyclic graph representation of a Boolean formula that is ordered by the Boolean

variables in the formula. Changing the order of variables, called *dynamic variable reordering* [12], often significantly shrinks the size of the representation and commensurately speeds up the analysis.

In summary, state space reduction techniques produce new transition systems for mitigating the cost of model checking, but since they guarantee property preservation the details of those reduced systems can be hidden from users.

4 Domains for model checking

In the following sections, we discuss two domains in which the model checking architecture of Fig. 2 has been instantiated with significant success: for hardware designs and for software implementations. We discuss the domain models and the hidden models that have allowed those instantiations to scale to real-world systems.

We also discuss an emerging domain, the analysis of biological systems, that has yet to see widespread use of effective domain-specific model checking techniques. We identify succesful efforts in applying model checking to biological systems that have been developed and that can be built on to target other biological systems.

4.1 Software implementations

Model checking has been applied at every level of the software development process, from requirements, design to implementation. Early approaches mostly focused on hand-translations of software models into model checkers' input notations and were one time activities. The modern trend is to use the architecture from Fig. 2 and automate the creation of the hidden models to allow domain experts to do the analysis. In this section, we will focus on model checking applied directly to source code to show an instantiation of the architecture from Sect. 3; however, numerous examples exist of model checking applied to requirements and design models, e.g., [17–19].

4.1.1 Domain model

Although there were attempts to apply model checkers for other domains to the problem of source code analysis [26,51], it became quickly apparent that the semantic gap could not be easily bridged and custom model checkers were developed instead [4,40,87,95]. These custom checkers took the original source code as input and were able to exploit the domain knowledge captured within the models to improve the model checking.

An interesting case is the SPIN model checker [54] that in its early form only allowed PROMELA models, i.e. custom notation for modeling protocols, but now supports also the analysis of C code. The C code must be analyzed within the context of an environment that is specified in PROMELA, since it allows for example non-determinism that is very useful for succinct environment specifications [55].

4.1.2 Hiding semantic model mismatches

The Verisoft model checker [40] was one of the first to target source code and used an underlying logical model based on interleavings of concurrent statements. The C library code for threading was instrumented to allow all possible enabled actions to be executed. Furthermore, it was the so-called state-less model checking since no states were stored and executions were replayed from the start rather than use a backtracking based search. The logical model was thus a tree of concurrent transitions (or code segments) rather than the more common graph of states. Verisoft used, therefore, a very simple hidden model to allow the model checking to proceed and could be used on every C program for which the threading library can be instrumented.

JPF [95] and BOGOR[6] [87] also support concurrent programs, but in this case Java programs. The logical model used in both cases is a graph of reachable states that allow one to check both safety and liveness properties. Also, unlike the Verisoft case, state-space searches can be pruned, since previously visited states are stored. However, the hidden models used here are more complex and based on executing Java bytecode. Since not all bytecode-based program features can be handled (since they can call native methods, use reflection, dynamic class loading etc.), the user sometimes is confronted with a hidden model restriction; these lead to much frustration as can be seen on, e.g., the JPF mailing list.[7]

SLAM [4] and CBMC[8] [21] only focus on sequential programs, but use very different logical models: SLAM uses boolean programs derived from abstractions of C code and CBMC encodes the reachability problem for C/C++ programs as a satisfiability problem and uses a technique called bounded model checking to do the analysis.

A new trend in software model checking is to analyze the code symbolically [14,15,29,42,43,62,85,91]. There are two approaches commonly followed: classic symbolic execution [29,85] and dynamic symbolic execution [14,42,43,91]. In classic symbolic execution, the interpretation of the execution is changed to operate symbolically and involves checking feasibility of paths, by invoking a decision procedure. In dynamic symbolic execution, the code is executed in a standard fashion, but via instrumentation, a symbolic constraint on the input for that path is constructed on the side; this symbolic constraint is then used to guide the execution of a next

[6] http://sireum.org.

[7] http://groups.google.com/group/java-pathfinder?pli=1.

[8] http://www.cprover.org/cbmc/.

concrete path. The important thing to notice is that dynamic symbolic execution has a much simpler hidden model and is, therefore, commonly considered the more applicable to industrial software.

4.1.3 Environment model

There are two types of environment models required to do software model checking: models to represent underlying aspects of the execution environment of the domain and models that represent the environment of the system under analysis. The former is a set of environment models that are distributed with the model checking framework and represent, e.g., libraries that are abstracted to allow analysis. A classic example of this might be input–output libraries that read and write to a file. Both JPF and PEX (via Moles) allow one to override the behavior of any method with a custom version and as such makes it easy to develop these kinds of environment models.

The second type of environment model is much more complicated to construct and, as pointed out in Sect. 3.3, is often times not hidden at all. In fact environment generation is mostly manual. An example where automated environment generation has been quite successful is in the analysis of graphical user interface code [92]: GUI code cannot be analyzed directly since it requires human input which is outside the closed system required for model checking. Another field where automated environment generation is used is in compositional (or assume-guarantee) reasoning where the environment is "learned" that will ensure a property is true of a composition of the system and its environment [80].

Note that all software model checkers augment the classic programming languages they analyze with the capability of expressing non-deterministic choice. This capability allows one to specify (among other things) the range of possible inputs to be used when analyzing the program. Without a non-deterministic choice operation, useful environments cannot be generated and in some sense model checking degenerates to testing (in the sequential case at least; for concurrent programs the scheduler is still non-deterministic in general). For example, JPF has a highly user-customizable *ChoiceGenerator* framework that allows all forms of non-determinism, including custom data choices, scheduling choices, etc.

The symbolic execution-based model checkers use a slightly different notion of an environment, since part of the benefit of using these techniques is that they produce the input values for which a certain behavior will be exhibited, i.e. they generate part of the environment by themselves. However, which method will be called in which order is typically still part of the environment that must be provided and only the input parameters are left symbolic.

4.1.4 Property model

Although model checking is historically focused on using temporal logic properties for analysis, software model checking tends to use only safety properties and not liveness properties. State-less model checkers cannot handle liveness by definition since they cannot detect cycles, but even model checkers like JPF and BOGOR do not support liveness; SPIN does support liveness but when using C code in the models, one must specify which part of the state must be tracked during model checking (exposing the hidden model).

The most common types of properties are simply local safety properties in the form of assertion violations (including contracts) or uncaught runtime exceptions (for example null-pointer dereferences). Another class of property is finite-state machine properties, such as every *open* should be followed by a *close* operation. These can be used to check type-state properties (i.e. API contracts) and are used in SLAM [4] and KLEE[9] [14].

4.1.5 Analysis results model

In software model checking, the results model is quite standard and involves an ability to review/replay the counterexample if a property is violated. There has, however, been work on trying to minimize the counterexample, by trying to explain why the error occurred [45,46]. In addition, coverage data are calculated during the model checking to allow both the system to proceed to behavior it has not analyzed, and reported at the end of a run to allow a user to check the adequacy of the model checking run if no error was found [47,97].

4.1.6 Hiding transient models

The internal optimizations used in software model checking are numerous, but essentially two techniques stand out: partial-order reductions [41] and predicate abstraction [44].

Partial-order reductions reduce the number of interleavings that need to be analyzed during model checking of concurrent programs, by not considering interleavings of independent transitions. The dependency information was traditionally determined by a static analysis before model checking, but due to the dynamic nature of software, dynamic partial-order reduction [37] is more popular in software model checking. During dynamic partial-order reduction, the dependencies between transitions are calculated during the execution by considering whether variables can be accessed by other threads. The more precise dynamic analysis typically leads to massive state-space reductions [32]. The calculation of the hidden dependency model during partial-order

[9] http://klee.llvm.org/.

reductions are completely hidden from the user. Another approach for reducing the number of interleavings to analyze is used by the CHESS [76] model checker that limits the number of context switches between threads. Where partial-order reduction is a precise abstraction of the original program (i.e. no behaviors are added or removed) the bounded context switching approach is an under-approximation that could miss errors. However, empirical evidence suggests that many concurrency errors are revealed with very few context switches (often times only 3). All of these reduced systems are calculated during model checking and are thus transient models.

Predicate abstraction allows one to reason about an over-approximation of the system behaviors and is often used within the so-called Counter-Example Guided Abstraction Refinement (CEGAR) framework [23]. CEGAR refers to the process where one starts with a gross over-approximation of the system behaviors (typically the control flow of the program) and with each counterexample this abstraction is refined (i.e. refinement guided by the counterexample) and the process is repeated until either a concrete counterexample is found (one with no abstractions that can thus happen in the real un-abstracted program) or the property is proved. Each of these abstracted models is hidden and the user only sees the final output.

The underlying technology required for predicate abstraction (as well as symbolic execution) is decision procedures for satisfiability that can reason over the domain of the program instructions, e.g. linear integer arithmetic, strings, arrays, bitvectors, etc. However, not all language features in programs can be represented using current decision procedures (e.g., nonlinear arithmetic). This is again an undesirable case where the hidden model is exposed to the user. Dynamic symbolic execution tries to alleviate this problem by using concrete values to solve constraints outside of their decision procedure domains. This is another reason why they are considered more robust for industrial use. Recent advances in classic symbolic execution also address this problem by solving the part of the constraint that the decision procedure can handle and using that as concrete values to solve the rest [81].

4.2 Hardware designs

In the 1990s, symbolic model checking matured to the point where it became industrially viable and applied at scale. Commercial interest increased significantly after the Pentium floating point division bug [84], which cost Intel almost half a billion dollars. It is now a mature technology that is a standard part of commercial design flows, and is applied regularly and repeatably by IBM, Intel, AMD, and Motorola. A good deal of the success of of model checking in this domain is due to the seamless integration of this technology into the tools that are used by hardware designers and to the close relationship between verification and hardware synthesis, which uses many of the same technologies.

4.2.1 Domain model

Hardware designers use hardware description languages (HDLs) to describe the design of electronic circuits. The two most popular languages are Verilog [89] (and its superset SystemVerilog [88]) and VHDL [56], though there are a wide range of notations that are used. HDLs differ from software languages in their treatment of time and concurrency. Since hardware is inherently concurrent, the languages are structured to allow straightforward expression of parallel evaluation. In addition, time is represented explicitly to support the analysis of time durations necessary for signals to stabilize within the circuit.

4.2.2 Hiding semantic model mismatches

The process of translating from HDLs to analysis models is primarily one of throwing information away; usually, model checking is performed only considering the Boolean logic portion of the model (netlist) and throwing away the timing information. The advent of symbolic methods in the early 1990s [71] led to a watershed in the application of model checking to HDLs, in large part because the netlist description is quite similar to the symbolic transition relation that is implemented by the Logical Model. That said, aspects of the translation of Verilog and VHDL can be difficult due to the semantics of binary signals: they can not only take on the values true and false, but also other indeterminate values. For example, in Verilog, there are two additional values X and Z, standing for 'unknown' and 'tri-stated' respectively. X in particular can be difficult, because it is interpreted differently by different tools: it can mean 'don't-care' or 'wildcard' depending on context [93]. VHDL has nine such additional values for signals [56].

4.2.3 Environment model

Environmental models in hardware describe expectations on the inputs to the circuit being constructed. Often, hardware is expected to work in relatively unconstrained environments, leading to simple environmental models. These assumptions are by definition system dependent and must be manually constructed. On the other hand, it is often the case that assume-guarantee reasoning is necessary to analyze large systems; in this case, it is necessary to manage the assumptions that different subsystems impose upon each other to ensure that the analysis results are sound. Several techniques have been developed to support this kind of circular reasoning between components [3,69], and many of the tool

suites ensure that mutual assumptions between components are kept consistent [57,70].

4.2.4 Property model

Unlike software, much of the work in hardware is focused on equivalence checking, where a reference model of a circuit is compared against an optimized model. This can be done at the level of *combinatorial equivalence checking* which does not consider the state of the two designs and *sequential equivalence checking*, which does consider state.

Verification of reference models is often performed using temporal formulae. Unlike software, most of the property languages support both safety and liveness formulae, and it is not uncommon to see liveness specifications for hardware models. Properties are often expressed through temporal logic such as LTL and CTL, extended regular expressions [88], and combinations of the two formats, as in PSL [33]. Recent work in hardware verification has focused on creating *verification units* that are stored along with the hardware designs being created. Notations such as PSL and SystemVerilog allow a designer to specify complex contracts within a hardware design. These contracts can contain local definitions in the host language, assumptions about the external environment, and assertions about expected behavior. The goal is to support both simulation-based verification and formal analysis using a single set of specifications.

4.2.5 Analysis results model

Counterexamples have been traditionally expressed as graphical waveforms, showing the behavior of output signals with respect to a set of inputs over time. However, with the introduction of richer datatypes, it can be more difficult to visualize the waveforms, and textual results are displayed. Many of the commercial design automation tools (e.g., [1,2]) support sophisticated step simulators of the hardware that allow forward and backward stepping through counterexamples, which helps in the diagnosis and correction of errors.

4.2.6 Hiding transient models

There are a number of transient models that are created to support hardware model checking and in fact the analysis process often translates between several representations to analyze large systems effectively. An interesting aspect is that synthesis and analysis are often intertwined. The synthesis process is concerned with minimizing circuits with respect to the number of gates involved (*minarea retiming* [96]) and minimizing timing (*mintime retiming*). Minarea retiming can also significantly improve analysis time, as it usually simplifies the description of the circuit. The synthesis process can involve equivalence checks on the model, so it may involve a level of symbolic analysis.

To perform minimization, models are often translated into And-Inverter-Graphs (AIGs) [27] which support techniques such as *structural hashing* [27], a graph algorithm to simplify the boolean representation of the circuit by identifying syntactically identical subtrees. The process iterates between removing syntactically identical subtrees using AIGs and structural hashing and performing model checking over heuristically-chosen subtrees thought to be equal but not structurally identical (*fraiging*) [74]. Alternate representations, such as reduced Boolean circuits [11], have also been shown to be very effective at reducing models.

For equivalence checking, the two circuits to be checked for equivalence are turned into a hidden model of a single circuit called a *miter* [13] derived by combining pairs of inputs that have the same names and feeding pairs of outputs into EXOR gates which are then fed into an OR gate. If the output of the OR gate is a constant 0, then the two circuits are equivalent.

For functional verification, the same techniques used for software analysis, such as POR and predicate abstraction refinement loops that abstract portions of the model, are widely used. In addition, some of the analysis techniques rely on adding *additional* information to the model. Inductive techniques rely on addition of additional level of lemma generation [61,82] to be practically useful. These lemmas define additional information about the model that is not exposed to the user.

4.3 Biological systems

Languages for describing aspects of biological systems have enabled dramatic advances in biology. For example, the encoding of DNA as a string over symbols A, T, C, G effectively abstracts from biochemical properties and permits the application of algorithmic techniques for identifying and manipulating DNA. Inspired by such advances, in recent years researchers have begun to explore how algorithmic verification techniques, such as model checking, can be applied to supplement existing theoretical and experimental study of a broad range of biological systems.

Like hardware and software systems, biological systems are enormously complex and span multiple levels of abstraction. Biological systems range over an extremely diverse set of abstraction levels from low-level biochemical properties, up through genetic, cellular, tissue, organ, organ system, organism, community, population, and even ecosystem levels. This breadth of system description has not yet been targeted by model checking, but there has been promising work on two classes of biological systems, cellular signalling pathways and genetic regulatory networks, which we describe below.

Unlike with hardware and software systems, biological models should be viewed as hypotheses that are subject to corroboration by experimental data obtained from the actual biological system [36]. As such, these models are imperfect descriptions of biological structures and mechanisms. In these systems, the lack of knowledge about the exact behavior of a biological system is often abstracted by incorporating stochastic behavior.

The use of stochastic modeling requires appropriate model checking algorithms. Model checking frameworks that provide such support, e.g., PRISM[10] [64], in addition to more traditional symbolic model checking frameworks, e.g., NuSMV [20], have been put to use in reasoning about biological systems.

4.3.1 Domain model

For biological systems, the models are not like the hardware and software models discussed in earlier sections. Rather these models are theories put forth by researchers that seek to explain the biological mechanisms that give rise to experimental observations.

One domain where probabilistic model checking has been applied is the analysis of cellular signalling pathways. Pathways are made up of a complex set of biochemical and molecular processes, typically involving the interaction of proteins, that are specific to the function of the pathway.

In [78], the signaling of T Cell's in immune system responses is studied in an attempt to understand discrepancies between existing theoretical models and observations of immune responses. This study is carried out by identifying a set of mappings from chemical reactions to fragments of code in PRISM's input language, then applying those mappings, by hand, to translate an existing model of T Cell pathway signaling. A similar pattern-based translation has been used in other applications of model checking to pathway signaling [16,52]. In principle, this type of pattern-based translation resembles the earliest version of the JPF software model checker [51] and could be automated using similar methods.

The RoVerGene project[11] [6] focuses on genetic regulatory networks which are small systems comprised of genes, proteins and small molecules whose joint behavior is key to many cellular processes. In RoVerGene, these networks are modeled as a system of piecewise-affine equations which are translated to a form that is amenable to model checking.

4.3.2 Hiding semantic model mismatches

The approach of RoVerGene adapts a well-established strategy from hardware and software model checking to bridge the gap between domain models and model checker inputs. It calculates an abstract transition system that captures the networks behavior in much the same way as predicate abstraction-based software model checking tools such as SLAM [4]—although the nature of the abstraction is much different. The resulting abstracted system is then checked using the NuSMV [20] symbolic model checker.

Another approach is to enrich the model checker inputs to better match the domain primitives. For example, the primitives for coordinating between components that are present in existing model checker input languages are unable to express the fact that components of biological systems, e.g., pairs of cells involved in signaling, operate asynchronously, but do not drift too far apart in performing local actions. Researchers have observed this in experiments and have captured this as the *bounded asynchrony* coordination mechanism [35]. This can be exploited to achieve state space reductions akin to partial order reductions when model checking biological systems. This strategy is an instance of the more general insight behind modern software model checkers [87,95] which provide the ability to customize the primitives of the modeling language, e.g., in the case of JPF to handle JVM bytecodes directly.

4.3.3 Property model

Rather than check a specific property, in the work on T Cell receptors PRISM was used to explore the responsiveness, selectivity, and speed of the response of cells under varying environmental factors. In this respect, the model checker is used more for "experimentation" with the model rather than property assurance.

In other work on pathway signaling [52], specifications are written in Continuous Stochastic Logic (CSL) that express, e.g., the number of times two proteins bind another during a time interval. RoVerGene also uses a temporal logic for property specification.

Using such logics directly is a challenge even for trained experts. To address this, researchers working on model checking cellular interactions have identified a number of recurring patterns that reflect common queries that domain scientists seek to answer [75]. This work is a domain-specific version of the more general property specification patterns [30] that have been used in many software model checking efforts.

4.3.4 Environment model

In the work on T Cell receptors, the process of constructing the PRISM input model produced a closed system, i.e., there were no unconstrained inputs. When models are constructed manually it is possible to blur the line between the system and the environment and to construct such a closed system,

[10] http://www.prismmodelchecker.org/.

[11] http://iasi.bu.edu/~batt/rovergene/rovergene.htm.

but this is more difficult to achieve when using automated techniques for generating models suitable for model checking.

As mentioned, the RoVerGene project [6,7] models the network as a system of piecewise-affine equations, and formulates properties using temporal logic. It expresses the environment as a set of input parameters that are varied across a range of analyses. For certain problems, RoVerGene is asked to calculate the set of input parameters for which the system is guaranteed to satisfy the property. This is the strategy that has recently been explored in work on learning minimal environment specifications in software model checking [39].

4.3.5 Analysis results model

The way that model checking has been most commonly applied in reasoning about biological systems avoids many of the challenges faced in mapping counterexamples back to software and hardware models. Instead, the results generally are presented in the form of sets of parameter values, described symbolically [7], which makes them easy to understand by domain specialists.

4.3.6 Hiding transient models

Manual attempts to leverage model checking tools to analyze biological systems have the disadvantage of having to produce transient intermediate models that may be unfamiliar to domain experts. In the case of the T Cell work, an additional encoding of the system in the stochastic π-calculus was produced—though it was exploited for simulation purposes. Generally, it is better to hide such models and RoVerGene does exactly that.

In fact, the generality of the piecewise-affine system model used in RoVerGene permits translations from other biological models. Recently, researchers have developed a mapping from cardiac cell models to RoVerGene which enables the exploration of questions related to tachycardia and other cardiac disorders [48]. In this work, the genetic regulatory networks are themselves a transient representation of the original domain models, not to mention the abstracted transition system models that are ultimately submitted to NuSMV. As RoVerGene-like tools begin to spread to other areas of biological systems one expects to see similar layered solutions, where a model in one domain is hidden when reasoning in another.

While there remains much more work to be done to realize a broad range of usable domain-specific model checking approaches for biological systems, we believe that this recent work represents an important step. Further improvement will come by continuing to build on the techniques and lessons learned in providing model checking support in other domains and developing new techniques as needed.

5 Conclusions

In many fields, modeling is an inherently human activity. Models are built by humans to capture their understanding or intent. Models are built for humans to abstract complex information sources and facilitate the communication of key details. In model checking, these human-oriented models play a crucial role. Users must capture the behavior of the system they wish to reason about and pose questions about that behavior in the form of property specifications. They must also be able to interpret counterexamples that demonstrate why a system violates a property specification.

Not all models are fit for human consumption, however. In modern optimizing compilers, a variety of different internal representations that model the structure and computation of the source program are created. These models are for intra-compiler communication. They serve to bridge the semantic gap from source language semantics to architecture specific object code and to expose key factors that can be exploited by optimization phases. Users of compilers gain no value from viewing such models.

When model checking is effectively tailored to an application domain, it operates much like a compiler. It hides internal transient models that exist to bridge the gap from domain-specific models to model checker notations and to make for more efficient processing. Developers of hardware and software systems benefit from the abstraction afforded by domain-specific model checking. As other domains begin to embrace the power of model checking as an analytic engine, they should follow the same pattern by hiding models whenever appropriate.

References

1. Cadence Inc. Incisive Enterprise Simulator product web site. http://www.cadence.com/products/sd/enterprise_simulator
2. Mentor Graphics Inc. ModelSim product web site. http://www.mentor.com/products/fv/modelsim
3. Amla, N., Emerson, A.E., Namjoshi, K.S., Trefler, R.J.: Abstract patterns of compositional reasoning. In: CONCUR, pp. 423–438 (2003)
4. Ball, T., Rajamani, S.: The SLAM toolkit. In: Berry, G., Comon, H., Finkel, A. (eds.) Proceedings of CAV, Paris, France. LNCS, vol. 2102, pp. 260–264. Springer, Berlin (2001)
5. Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for boolean programs. In: Proceedings of the 7th International SPIN Workshop on Model Checking and Software Verification, pp. 113–130 (2000)
6. Batt, G., Belta, C., Weiss, R.: Model checking liveness properties of genetic regulatory networks. In: Tools and Algorithms for the Construction and Analysis of Systems, vol. 4424, pp. 323–338. Springer, Berlin (2007)
7. Batt, G., Page, M., Cantone, I., Goessler, G., Monteiro, P.T., de Jong, H.: Efficient parameter search for qualitative models of regulatory networks using symbolic model checking. Bioinformatics **26**(18) (2010)

8. Baumgartner, J., Heyman, T., Singhal, V., Aziz, A.: Model checking the IBM gigahertz processor: an abstraction algorithm for high-performance netlists. In: Proceedings of the 11th International Conference on Computer Aided Verification, pp. 72–83 (1999)
9. Beer, I., Ben-David, S., Eisner, C., Rodeh, Y.: Efficient detection of vacuity in temporal model checking. Formal Methods Syst. Des. **18**, 141–163 (2001)
10. Berthomieu, B., Garavel, H., Lang, F., Vernadat, F.: Verifying dynamic properties of industrial critical systems using TOPCASED/FIACRE. ERCIM News **2008**(75) (2008)
11. Bjesse, P., Boralv, A.: Dag-aware circuit compression for formal verification. In: Proceedings of ICCAD '04, pp. 42–49 (2004)
12. Brace, K.S., Rudell, R.L., Bryant, R.E.: Efficient implementation of a BDD package. In: Proceedings of the 27th Design Automation Conference, pp. 40–45 (1990)
13. Brand, D.: Verification of large synthesized designs. Proc. ICCAD **1993**, 534–537 (1993)
14. Cadar, C., Dunbar, D., Engler, D.: Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08, Berkeley, CA, USA, pp. 209–224. USENIX Association (2008)
15. Cadar, C., Godefroid, P., Khurshid, S., Păsăreanu, C.S., Sen, K., Tillmann, N., Visser, W.: Symbolic execution for software testing in practice: preliminary assessment. In: ICSE, pp. 1066–1071 (2011)
16. Calder, M., Gilmore, S., Hillston, J., Vyshemirsky, V.: Formal methods for biochemical signalling pathways. In: Formal Methods: State of the Art and New Directions, pp. 185–215. Springer, Berlin (2010)
17. Chan, W., Anderson, R.J., Beame, P., Jones, D.H., Notkin, D., Warner, W.E.: Optimizing symbolic model checking for statecharts. IEEE Trans. Softw. Eng. **27**(2), 170–190 (2001)
18. Chan, W., Anderson, R.J., Beame, P., Notkin, D.: Improving efficiency of symbolic model checking for state-based system requirements. In: ISSTA, pp. 102–112 (1998)
19. Choi, Y., Rayadurgam, S., Heimdahl, M.P.E.: Toward automation for model-checking requirements specifications with numeric constraints. Requir. Eng. **7**(4), 225–242 (2002)
20. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV Version 2: an open source tool for symbolic model checking. In: Proceedings of the International Conference on Computer-Aided Verification (CAV 2002). Springer, Berlin (2002)
21. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004). Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer, Berlin (2004)
22. Clarke, E.M., Emerson, A.E.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Workshop on Logics of Programs, pp. 52–71 (1981)
23. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV, pp. 154–169 (2000)
24. Clarke, E.M.: Grumberg. Model Checking. MIT Press, Orna (1999)
25. Clarke, E.M., Kroening, D., Ouaknine, J., Strichman, O.: Completeness and complexity of bounded model checking. In: Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation, pp. 85–96 (2004)
26. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Păsăreanu, C.S., Zheng, R., Zheng, H.: Bandera: extracting finite-state models from Java source code. In: International Conference on Software Engineering, pp. 439–448 (2000)
27. Darringer, A., Joyner, W.H. Jr., Berman, C.L., Trevillyan, L.: Logic synthesis through local transformations. IBM J. Res. Dev. **25**(4), 272–280 (1981)
28. de Halleux, J., Tillmann, N.: Moles: tool-assisted environment isolation with closures. In: Objects, Models, Components, Patterns, pp. 253–270. Springer, Berlin (2010)
29. Deng, X., Lee, J., Robby: Bogor/Kiasan: a k-bounded symbolic execution for checking strong heap properties of open systems. In: ASE (2006)
30. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the International Conference on Software Engineering, pp. 411–420 (1999)
31. Dwyer, M.B., Corbett, J.C., Avrunin, G.: Spec patterns. http://patterns.projects.cis.ksu.edu (1999)
32. Dwyer, M.B., Hatcliff, J., Robby, Ranganath, V.P.: Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. Formal Methods Syst. Des. **25**(2-3), 199–240 (2004)
33. Eisner, C., Fisman, D.: A Practical Introduction to PSL. Springer, Berlin (2006)
34. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Computer Aided Verification, pp. 232–247 (2000)
35. Fisher, J., Henzinger, T., Mateescu, M., Piterman, N.: Bounded asynchrony: concurrency for modeling cell–cell interactions. In: Formal Methods in Systems Biology, pp. 17–32. Springer, Berlin (2008)
36. Fisher, J., Henzinger, T.A.: Executable cell biology. Nat. Biotechnol. **25**(11), 1239–1249 (2007)
37. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '05, pp. 110–121. ACM, New York (2005)
38. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast decision procedures. In: Alur, R., Peled, D., (eds.) Proceedings of the 16th International Conference on Computer Aided Verification, CAV'04 (Boston, Massachusetts). Lecture Notes in Computer Science, vol. 3114, pp. 175–188. Springer, Berlin (2004)
39. Giannakopoulou, D., Păsăreanu, C.S., Cobleigh, J.M.: Assume-guarantee verification of source code with design-level assumptions. In: Proceedings of the 26th International Conference on Software Engineering, pp. 211–220 (2004)
40. Godefroid, P.: Model checking for programming languages using Verisoft. In: Proceedings of POPL, pp. 174–186. ACM, New York (1997)
41. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems—An Approach to the State-Explosion Problem. Lecture Notes in Computer Science, vol. 1032. Springer, Berlin (1996)
42. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. SIGPLAN Not. **40**(6), 213–223 (2005)
43. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: Network Distributed Security Symposium (NDSS). Internet Society (2008)
44. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) Computer Aided Verification. Lecture Notes in Computer Science, vol. 1254, pp. 72–83. Springer, Berlin (1997)
45. Groce, A., Kroening, D., Lerda, F.: Understanding counterexamples with Explain. In: CAV, pp. 453–456 (2004)
46. Groce, A., Visser, W.: What went wrong: explaining counterexamples. In: SPIN, pp. 121–135 (2003)
47. Groce, A., Visser, W.: Heuristics for model checking Java programs. STTT **6**(4), 260–276 (2004)
48. Grosu, R., Batt, G., Fenton, F.H., Glimm, J., Le Guernic, C., Smolka, S.A., Bartocci, E.: From cardiac cells to genetic regulatory networks. In: Computer Aided Verification—23rd

International Conference, Proceedings, pp. 396–411. Springer, Berlin (2011)
49. Hagen, G., Tinelli, C.: Scaling up the formal verification of Lustre programs with SMT-based techniques. In: Proceedings of the Formal Methods in Computer-Aided Design, pp. 1–9 (2008)
50. Hatcliff, J., Dwyer, M.B., Zheng, H.: Slicing software for model construction. High. Order Symb. Comput. **13**(4), 315–353 (2000)
51. Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder (1998)
52. Heath, J., Kwiatkowska, M., Norman, G., Parker, D., Tymchyshyn, O.: Probabilistic model checking of complex biological pathways. Theor. Comput. Sci. **319**(3), 239–257 (2008)
53. Henzinger, T.A.: The theory of hybrid automata. Theor. Comput. Sci **138**, 3–34 (1995)
54. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, Boston (2003)
55. Holzmann, G.J., Joshi, R.: Model-driven software verification. In: Model Checking Software: 11th International SPIN, Workshop, pp. 76–91 (2004)
56. IEEE Standard 1076–2008: VHDL Language Reference Manual. Technical report, Institute of Electrical and Electronics Engineers (2009)
57. Cadence Incisive Formal Verifier data sheet. Cadence Design Systems, Inc. http://www.cadence.com/rl/Resources/datasheets/IncisiveFV_ds.pdf
58. Iosif, R.: Symmetry reductions for model checking of concurrent dynamic software. STTT **6**(4), 302–319 (2004)
59. Iosif, R., Dwyer, M.B., Hatcliff, J.: Translating Java for multiple model checkers: the Bandera back-end. Formal Methods Syst. Des. **26**(2), 137–180 (2005)
60. Specification and description language. Technical report, International Telecommunication Union, November 1988. ITU Recommendation Z.100
61. Kahsai, T., Ge, Y., Tinelli, C.: Instantiation-based invariant discovery. In: Bobaru, M., Havelundand, K., Holzmann, G., Joshi, R. (eds.) Proceedings of the 3rd NASA Formal Methods Symposium (Pasadena, CA, USA). Lecture Notes in Computer Science, vol. 6617, pp. 192–207. Springer, Berlin (2011)
62. Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Proceedings of TACAS, pp. 553–568 (2003)
63. Kwiatkowska, M.: Model checking for probability and time: from theory to practice. In: Proceedings 18th Annual IEEE Symposium on Logic in Computer Science (LICS'03), pages 351–360. IEEE Computer Society Press, New York (2003). Invited Paper
64. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11), pp. 585–591 (2011)
65. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. ACM SIGSOFT Softw. Eng. Notes **31**(3), 1–38 (2006)
66. Lluch, A.: NuSMV examples: the collection. http://nusmv.fbk.eu/examples/examples.html (1999)
67. Lluch, A.: Promela database. http://www.albertolluch.com/research/promelamodels (2012)
68. Mathworks Inc. Simulink Design Verifier product web site. http://www.mathworks.com/products/sldesignverifier
69. McMillan, K.L.: Circular compositional reasoning about liveness. Technical Report 1999–02, Cadence Berkeley Labs, Berkeley, CA 94704 (1999)
70. McMillan, K.L.: Symbolic Model Verifer (SMV)—Cadence Berkeley Laboratories Version. Cadence Design Systems, Inc. http://www-cad.eecs.berkeley.edu/~kenmcmil/smv
71. McMillan, K.L.: Symbolic Model Checking. Kluwer, Dordrecht (1993)
72. McMillan, K.L.: Applications of Craig interpolants in model checking. In: Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, Proceedings, pp. 1–12 (2005)
73. Miller, S.P., Whalen, M.W., Cofer, D.D.: Software model checking takes off. Commun. ACM **53**(2), 58–64 (2010)
74. Mishchenko, A., Chatterjee, S., Brayton, R., Een, N.: Improvements to combinational equivalence checking. In: Proceedings of ICCAD, vol. 2006, pp. 836–843 (2011)
75. Monteiro, P.T., Ropers, D., Mateescu, R., Freitas, A.T., de Jong, H.: Temporal logic patterns for querying dynamic models of cellular interaction networks. In: ECCB'08 Proceedings, Seventh European Conference on Computational Biology, pp. 227–233 (2008)
76. Musuvathi, M., Qadeer, S.: Chess: Systematic stress testing of concurrent software. In LOPSTR, pp. 15–16 (2006)
77. Orwick, P., Smith, G.: Developing Drivers with the Windows Driver Foundation. Microsoft Press, Redmond (2007)
78. Owens, N., Timmis, J., Greensted, A., Tyrrell, A.: Modelling the tunability of early T cell signalling events. In: 7th International Conference on Artificial Immune Systems (ICARIS'08), pp. 12–23 (2008)
79. Păsăreanu, C., Dwyer, M., Huth, M.: Assume-guarantee model checking of software: a comparative case study. In: Theoretical and Practical Aspects of SPIN Model Checking, pp. 168–183. Springer, Berlin (1999)
80. Păsăreanu, C.S., Giannakopoulou, D., Bobaru, M.G., Cobleigh, J.M., Barringer, H.: Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. Formal Methods Syst. Des. **32**(3), 175–205 (2008)
81. Păsăreanu, C.S., Rungta, N., Visser, W.: Symbolic execution with mixed concrete-symbolic solving. In: ISSTA, pp. 34–44 (2011)
82. Per Bjesse, K.C.: SAT-based verification without state space traversal. In: International Conference on Formal Methods in Computer Aided Design (2000)
83. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, pp. 46–57 (1977)
84. Pratt, V.R.: Anatomy of the Pentium bug. In: Proceedings of the 6th International Joint Conference on Theory and Practice of Software Development, pp. 97–107 (1995)
85. Păsăreanu, C.S., Mehlitz, P.C., Bushnell, D.H., Gundy-Burlet, K., Lowry, M., Person, S., Pape, M.: Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In: Proceedings of ISSTA (2008)
86. Queille, J.-P., Sifakis, J.: Specification and verification of concurrent systems in Cesar. In: International Symposium on Programming, 5th Colloquium, Proceedings, pp. 337–351 (1982)
87. Robby, M.B.D., Hatcliff, J.: Bogor: An extensible and highly-modular model checking framework. ACM SIGSOFT Softw. Eng. Notes **28**(5), 267–276 (2003)
88. Sutherland, S., Davidmann, S., Flake, P.: System Verilog for Design. Springer, Berlin (2006)
89. Thomas, D.E., Moorby, P.R.: The Verilog Hardware Description Language. Kluwer, Norwell (1998)
90. Thomas, W.: Automata on infinite objects. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B), pp. 133–192 (1990)
91. Tillmann, N., De Halleux, J.: Pex: white box test generation for .NET. In: TAP, pp. 134–153. Springer, Berlin (2008)
92. Tkachuk, O., Dwyer, M.B., Păsăreanu, C.S.: Automated environment generation for software model checking. In: ASE, pp. 116–129 (2003)
93. Turpin, M.: The dangers of living with an X. ARM Ltd. http://www.arm.com/files/pdf/Verilog_X_Bugs.pdf
94. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification (preliminary report). In: Proceedings

of the Symposium on Logic in Computer Science, pp. 332–344 (1986)
95. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. **10**(2), 203–232 (2003)
96. Wang, J., Zhou, H.: An efficient incremental algorithm for min-area retiming. In: Design Automation Conference (2008)
97. Xie, T., Tillmann, N., de Halleux, P., Schulte, W.: Fitness-guided path exploration in dynamic symbolic execution. In: Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009), pp. 359–368 (2009)

Author Biographies

Willem Visser is a professor in Computer Science at Stellenbosch University, South Africa. Before joining Stellenbosch in 2009, he spent 8 years at NASA Ames Research Center, where he was one of the research leads for the Java PathFinder project. His research interests include model checking, testing, and symbolic execution for test generation. He has been co-chair of ASE in 2008 and the ICSE Experience Report track in 2010, is currently on the steering committee for ASE and SPIN, on the executive committee of ACM SIGSOFT and is a member of the editorial board of TOSEM. More information can be found on his webpage at: http://www.cs.sun.ac.za/~wvisser/.

Matthew B. Dwyer is the Henson Professor of Software Engineering in the Department of Computer Science and Engineering at the University of Nebraska - Lincoln. Since receiving his PhD from the University of Massachusetts at Amherst in 1995 he has explored his research interests in software analysis, verification and testing. His research has received several awards including the ICSE "Most Influential Paper" and SIGSOFT "Impact Paper" awards in 2010. Dr. Dwyer has served as program chair for the ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE 2004), the International Conference on Software Engineering (ICSE 2008), and the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2012).

Michael Whalen is the Program Director at the University of Minnesota Software Engineering Center. Dr. Whalen is interested in formal analysis, language translation, testing, and requirements engineering. He has developed simulation, translation, testing, and formal analysis tools for Model-Based Development languages including Simulink, Stateflow, SCADE, and RSML-e, and has published more than 30 papers on these topics. He has led successful formal verification projects on large industrial avionics models, including displays (Rockwell-Collins ADGS-2100 Window Manager), redundancy management and control allocation (AFRL CerTA FCS program) and autoland (AFRL CerTA CPD program). His PhD dissertation involved using higher-order abstract syntax as a basis for a provably-correct code generation tool from the RSML-e specification language into a subset of C.