Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Special section on automated verification of critical systems

Some current topics in model checking

Michael Huth

Department of Computing, Imperial College London, South Kensington campus, London, SW7 2AZ, United Kingdom

February 6, 2006

Abstract. Model checking is a particular approach to property verification of systems. One describes a system in a mathematical model, expresses the properties one wishes to verify for the system in a formal language, and then checks whether the model satisfies the formal property. Invented 25 years ago, this approach is fully automatic and has therefore gained wide acceptance and is increasingly being used in commercial research & development units. Impediments remain on the road to successful technology transfer. For one, the size of models often increases exponentially in the number of variables or sub-models, preventing scalable automation. Abstracting a model to reduce its size can be a cost-effective way of addressing this. For another, systems and models may be subject to change, e.g. in an incremental design process. One then seeks cost-effective means of ascertaining that property verifications remain to be valid as models evolve. This special section presents current research on such abstraction and change management of model checking.

Key words: model checking — temporal logic — slicing — game semantics — refinement checking — feature integration — incremental design

1 Introduction

Model checking [38,6] is a well-established methodology for formally verifying properties of systems. There is a wide range of systems and properties supported by model-checking frameworks. Qualitative systems such as those represented by programs in high-level programming languages [25], quantitative systems in which behavior incurs cost or risk such as in dynamic power management [34], probabilistic or stochastic systems such as those governing non-linear feedback systems as found in genetic networks within cells [4], timed systems such as controllers in electronic units [39] can all, in principle, be modeled and often checked algorithmically for compliance with certain properties. Systems may also blend aspects of time, probability etc and can still be checked automatically [20,28]. The nature of properties varies accordingly with the aspects present in systems. Quality of service demands, for example, may speak of completing certain tasks within specific time units with a given probability threshold.

But even properties of interest that focus on one aspect only can be grouped into different classes with different algorithmic costs for checking compliance. In qualitative systems it often suffices to consider *reachability* properties but other assurance needs may demand more complex properties. This requires model checking to be flexible and to offer a range of frameworks with varying degrees of expressiveness and computational cost.

In model checking, systems S are described as models M from a suitable class of mathematical models. Properties P are expressed as formulas ϕ from a suitable formal language. The satisfaction relation $S\,\mathsf{satisfies}\,P$ between systems and properties is then captured by means of a predicate $\models$ whose instances $M \models \phi$ define which models M satisfy what formulas ϕ.

Example 1. The system S could be a `C` program, the property of interest P could be "no out-of-bounds exceptions occur for any arrays in S", the model M of the system could be a Kripke structure synthesized automatically by a predicate abstraction [19] of the system S, the property P could be coded as ϕ specifying that such implicit or explicit exception program points are not reachable from legitimate initial program states, and $M \models \phi$ could be determined algorithmically through reachability analysis of labelled states in the Kripke structure M.

This lose definition of model checking is intentional as it allows needed flexibility. A system of components for web services would benefit from a notion of models that makes events explicit, e.g. as for the Business Process Execution Language (BPEL) (`www.bpelsource.com`) or a variant of the process algebra Pi-Calculus [33]. On the other hand, models of `C` programs for drivers need observables attached to states. Likewise, properties may be expressed in various guises, e.g. as a formula of temporal logic, a regular expression, an automata or simply another model. The meaning of the formal satisfaction relation $M \models \phi$ therefore depends on what we mean by model and property. For example if ϕ is expressed as yet another model, then $M \models \phi$ amounts to a kind of consistency check, e.g. one could check that all execution sequences of M are also execution sequences of ϕ. This approach is taken by the paper *Compositional Software Verification Based On Game Semantics and CSP*. On the other hand, if ϕ is a formal expression in a temporal logic, $M \models \phi$ would typically be defined in a syntax-directed manner over the structure of ϕ. Both approaches will be formalized below.

Model-checking framework Despite this generality and flexibility, a model-checking framework may be seen to consist of a class of models $\mathcal{M}$, a class of properties Φ, a predicate $\models\ \subseteq \mathcal{M} \times \Phi$ that associates models to properties they enjoy, and an algorithm for computing this predicate.

The undeniable success of model checking stems from the fact that it is a fully automated property verification technique for finite-state models: describe the finite-state model M, specify the property ϕ, and then push the button that triggers the fully automated computation of $M \models \phi$. The output is either "yes", in which case ϕ is verified to be true for M, or "no", in which case ϕ fails to hold for M and some diagnostic information may be presented for this failure by a model checker [7] if possible and indeed desired.

The transfer of this technology into industrial settings has reached many research & development divisions of companies in the sectors of communication technologies (e.g. Lucent Technologies), off-the-shelf software and operating systems (e.g. Microsoft), hardware design (e.g. Intel), and other sectors that produce or rely on critical software or hardware components. Such transfer took almost 20 years to reach blue-chip companies such as Microsoft. But model-checking technology is already part of commercial products (e.g. as offered by Cadence Berkeley Labs), companies have dedicated inhouse teams that build special purpose model checkers (e.g. SLAM [2] and Zing [1] at Microsoft), and mature free-ware model checkers (e.g. the NuSMV model checker at `http://nusmv.irst.itc.it/`) are available. All this looks like a definite success story for people working in the wider areas of "formal methods", corroborated by the aforementioned technology and tool transfer into industrial sectors — notably those pertaining to critical systems.

A more detailed look, however, tells a more sober and fine-grained story that reveals transfer-enabling research challenges for model checking. We begin with a trivial yet often neglected observation:

Representation Issue: We ultimately want to assure $S\,\mathsf{satisfies}\,P$, that systems satisfy properties, so what assurance can we possibly get from verifying $M \models \phi$ if M represents S and ϕ represents P?

The transition from the system S to the model typically involves some sort of abstraction. Modelers will not model all possible aspects of system S, but only those that matter for the properties one means to check. But how do modelers know which aspects of the system are relevant for which properties? An infinite regress threatens to appear. The paper *Formal Verification of the NASA Runway Safety Monitor*, e.g., verifies a protocol by safely over-approximating physical motion but limits the scope of the model to a small number of aircrafts, the implication being that a good model of physical motion is more important than having as many objects as a real airport may have. There is also the risk that the formalization of P as ϕ may be flawed or a mere approximation. Additionally, modelers can't be sure that the set of properties to be checked is fixed. Properties may change, be deleted or new properties with new atomic observables may become relevant. This may require a change of model M, after which previously verified instances of $M \models \phi$ may no longer hold. In summary, there is a certain risk that the verification of $M \models \phi$ may fool ourselves into believing that system S enjoys property P.

But there is plenty of good news, too. For example if the system S and property P have faithful representations within the class of models $\mathcal{M}$ and properties Φ as M and ϕ (respectively), then we know that $M \models \phi$ implies $S\,\mathsf{satisfies}\,P$. If that M is infinite-state, we can often *abstract* M into a finite-state model M' such that the automatic verification of $M' \models \phi$ implies $M \models \phi$ [8] (and therefore $M' \models \phi$ also implies $S\,\mathsf{satisfies}\,P$). The construction of such a sound abstraction M' depends on the kind of property. Safety properties [37] (nothing bad ever happens) require abstractions that are "safe simulations", liveness properties [37] (something good eventually happens) need "live simulations", and properties that mix safety and liveness aspects in non-trivial ways require abstractions that are 3-valued [13,14,22].

For the model-checking methodology there are also cost issues.

Cost Issue: What are the costs of producing a sound abstract model M from S, of formalizing a property specification ϕ from P, and of computing $M \models \phi$ and its diagnostics (if applicable)?

Getting "good" abstract models M from a system S may be time-consuming and may involve a feed-back loop from model checks. Say that M_0 is our initial model and $M_0 \models \phi$ fails to hold. Then the supplied diagnostics may suggest that this is a genuine flaw of S, or it may suggest that M_0 is not a "good" model of S, triggering the description of a finer model M_1 etc. For example, the over-approximation of physical motion and the specification of the number of aircrafts in the paper *Formal Verification of the NASA Runway Safety Monitor* may have benefited from incremental model builds to find a good tradeoff between precision and cost of the resulting model. These concerns may also apply to properties. We may struggle to express P within a particular formalism Φ and one may have to be an expert on Φ to understand whether P is indeed expressible in Φ. For example it is not trivial to realize that "atomic property p is true at every other state" is not expressible in the powerful temporal logic CTL*.

In that context, the Specification Patterns Library [16] can be seen as a laudable effort of mapping popular property patterns P into various back-end formalisms Φ. Last, but definitely not least, the cost of computing $M \models \phi$ may be prohibitively large. The application context will determine what costs are deemed to be acceptable. For example if $M \models \phi$ verifies an important hardware or software standard, we may be willing to wait a few weeks' time for this model check to provide assurance. If $M \models \phi$ verifies the correctness of feature integration of `C` code for switching boards, and if this information needs to be fed back to `C` programmers in real development time, only a cost of 1-2 hours of real time or less may be acceptable [21]. The paper *Model-checking the Preservation of Temporal Properties upon Feature Integration* offers algorithms for checking that correct features still operate as specified when they are integrated with new features. Models could either be mere specifications of features or abstractions of software that implements such features.

The bad news is that even if the cost of computing $M \models \phi$ is linear in the sizes of M and ϕ, the size of the model M is typically exponential in the number of data variables or the number of communicating sub-models. This is referred to as the notorious *state explosion problem*; it is also known as the *curse of dimensionality* in optimization and control theory. Much of the research in model checking tries to tackle this, and a lot of progress has been made in that regard in recent years — the second volume of STTT being a representative example thereof [10].

The model-checking framework as we presented it here is unable to capture the dynamic processes within which these activities may be embedded. We already hinted at a possible feed-back between model creation and model checking. But even this assumes that the system S is fixed and not subject to change, which may be untenable. Consider the potential use of model checking in software development where a particular agile and iterative development method, say a version of the Unified Process (UP), is being used. Work products and practices of that method may benefit from the use of model checking captured requirements (e.g. for consistency) or executable source code (for behavioral correctness checks). Let us just focus on requirements for sake of illustration. This instance of UP will have some change-management process in place that authorizes, denies and tracks changes, deletions or additions of functional or non-functional requirements.

This raises issues for model checking in the context of change management.

> **Change-Management Issue:** What are effective and transparent means of tracking any requirements change as to, which existing models it impacts, how those impacted models may have to be modified, which existing verifications can still be trusted, and which existing refutations may no longer hold?

Some of these desired abilities are similar to what is being offered by `make` files used in program development, or modern control version systems such as `cvs` used within integrated development environments or as stand-alone tools.

A more simplified view of such a development process is (linear) incremental design, where a final model M is being built by a successive sequence of models $M_0, M_1, \ldots, M_n = M$ such that M_{i+1} somehow incorporates M_i and enriches it with additional state or behavior. This restricted change policy offers opportunities of incremental model checks:

- for certain kinds of properties ϕ there may be a cost-effective way of establishing that $M_i \models \phi$ implies $M_{i+1} \models \phi$; in that case, such verifications remain to be verifications during incremental design
- dually, for certain ϕ we may have a cost-effective means of establishing that $M_i \not\models \phi$ implies $M_{i+1} \not\models \phi$; so refutations would remain refutations during incremental design.

Cost-effective here means "not more expensive than computing $M_{i+1} \models \phi$". At the extreme, it could mean "at no cost", for example, if the construction of M_{i+1} is such that one can prove the preservation of verifications $M_i \models \phi$ for certain ϕ as a meta-theorem. Such a theorem is given in the paper *CTL-Property transformations along an incremental design process* where the temporal-logic property has to be re-formulated when specified for the incremented model M_{i+1} to reflect how M_i is extended by M_{i+1}.

But as soon as we may *add and remove* states or behavior in the increment from M_i to M_{i+1}, we expect it to be much more costly to assure that verifications of ϕ in M_i are also verifications of ϕ in M_{i+1}. This is the setting of the paper *Model-checking the Preservation of Temporal Properties upon Feature Integration* which can provide such assurance with the instrumented use of SAT solvers. We observe a clear tradeoff between the degree of freedom in changing a model in an incremental design step and the cost of assuring that verifications remain verifications in moving to the incremental model.

Such incremental design can be achieved by means of a *stepwise refinement* process within a formal specification framework, e.g. the B method (`www.b-core.com`). Begin with an initial model M_0 that omits many details, and then add more details in each refinement step $M_i \mapsto M_{i+1}$ in a top-down fashion. For the models found in most model checkers, Kripke structures and labelled transition systems, the aforementioned tradeoff occurs: spelling out new details means that the new model will change its properties. This can be avoided if all M_i have to be bisimilar [35] but then it is difficult at best to add genuine details to models. At the other extreme, if models can be changed at will there are no guarantees at all on the preservation of any properties.

Modal and mixed transition systems [30], and their 3-valued variants [18], seem to offer an attractive compromise in that regard. Such systems consist of two specification parts: MAY information (may be implemented) and MUST information (has to be implemented). Any refinement process has to preserve all MUST information, but any piece of MAY information may be either removed (will not be implemented) or promoted to MUST information (and therefore will be implemented). This guarantees that verifications (now being established by a stronger 3-valued model check) are being preserved by refinement, therefore leading to potentially cost-effective change management for model checking.

Example 2. We oversimplify this idea in order to explain it in elementary terms. Let models M be total functions from a set of atomic propositions AP to the set $\{0, 1/2, 1\}$ where 0 represents "false", 1 represents "true", and 1/2 represents "may be true or may be false" [26]. The MUST information of such an M are those $p \in \mathsf{AP}$ for which $M(p) \neq 1/2$. MAY information of M are all those $p \in \mathsf{AP}$ with $M(p) = 1/2$. Stepwise refinement needs to preserve all MUST information and can deal with MAY information any way it pleases. That is if $M(p) \neq 1/2$, then $M'(p)$ has to equal $M(p)$ for all refinements M' of M. However, if $M(p) = 1/2$, then refinements M' of M have absolutely no constraints on the value of $M'(p)$.

To get the preservation of verifications under refinement, we evaluate formulas generated by the grammar $\phi ::= p \mid \neg p \mid \phi \wedge \phi \mid \phi \vee \phi$, with $p \in \mathsf{AP}$, conservatively over models:

- $M \models p$ iff $M(p) = 1$
- $M \models \neg p$ iff $M(p) = 0$
- $M \models \phi_1 \wedge \phi_2$ iff $M \models \phi_1$ and $M \models \phi_2$
- $M \models \phi_1 \vee \phi_2$ iff $M \models \phi_1$ or $M \models \phi_2$.

One can then prove that for all M and ϕ, and all refinements M' of M one has that $M \models \phi$ implies $M' \models \phi$. For example, $M \models p \vee \neg q$ holds if $M(p) = 1/2$ and $M(q) = 0$. If we refine M by setting $M'(p) = 0$, then $M' \models p \vee \neg q$ still holds. On the other hand, $M'' \not\models p \vee \neg q$ where $M''(p) = M''(q) = 1$ but M'' is also not a refinement of M as it does not preserve the value of M at q.

By the same token, one may think of MAY information as something provided by an external environment at run-time, and of the MUST information as a specification of the local system. Then stepwise refinement may be seen as an incremental concretization of an under-specified environment. It is therefore not surprising that such 3-valued models and their model checks have applications in open module checking [27], such as the interaction of cross-cutting features [31,32], e.g. the combination of a call-forward and a call-blocking feature in telecommunication systems. This suggests that the techniques proposed in the papers *Model-checking the Preservation of Temporal Properties upon Feature Integration* and *CTL-Property transformations along an incremental design process* may be applicable or may benefit from such 3-valued settings.

2 Two model-checking frameworks

For sake of illustration, we briefly sketch two model-checking frameworks, particular choices of $\models \; \subseteq \mathcal{M} \times \Phi$ that are used by most popular model checkers. In doing so, we won't discuss algorithmic issues for computing $\models$ but include references to salient algorithms.

2.1 State-based models

We begin with models for which information is residing within states and where there is only one kind of transition between states.

Formally, let AP be a set of atomic propositions. Typically, such properties are derived from systems in a straightforward way. For example, $p \in \mathsf{AP}$ could represent `x >= y + 2`, and that expression may be the boolean guard of an if-statement or while-statement in a `C` program. In most practical settings, AP may be chosen to be finite as only finitely many properties are being checked, each of which containing only finitely many atomic propositions. For such a set AP, we can define a Kripke structure M as a tuple (S, i, R, L), where

- S is a (possibly infinite) set of states
- $i \in S$ is the designated initial state
- $R \subseteq S \times S$ is the state transition relation, and
- $L\colon \mathsf{AP} \to \mathbb{P}(S)$ is the labelling function.

Variants may consider a set I of initial states. Sometimes, L may have type $S \to \mathbb{P}(\mathsf{AP})$ but we can transform such an L into an equivalent L' of type $\mathsf{AP} \to \mathbb{P}(S)$

by setting $L'(p) = \{s \in S \mid p \in L(s)\}$. State set S comprises the computational states the model could possibly be in, and system execution begins in the initial state i. The relation R similarly specifies what state transitions are possible within the model M: the notation $(s, s') \in R$, also written sRs', means that state s could evolve into state s' in model M within one "execution step". These models are generally non-deterministic as for most s there may be more than one s' with sRs'. Algorithms that compute $\models$ need to cope with this non-determinism and the combinatorial blow-up it may cause.

Finally, the labelling function tells us which atomic properties are true at what states: for each $p \in \mathsf{AP}$, the set $L(p)$ is understood to contain exactly those states of M that satisfy p. In particular no state in $S \setminus L(p)$ satisfies p. Similarly, if $(s, s') \notin R$, then within model M there simply cannot be a transition from state s to s' (in which case s may still reach s' through other transitions). We write $\mathcal{K}$ subsequently for the class of all Kripke structures.

If we are ultimately interested in verifying that the initial state i satisfies relevant properties, then the set of states that are reachable from i via R is of considerable interest. Formally, given a subset X of S, we write $X.R$ for the set of states in S that can be reached from X by exactly one transition step: $X.R = \{s' \in S \mid \exists s \in X\colon sRs'\}$. Then we can define the set of states reachable from i recursively by the following algorithm:

```
Cache = {};
Reach = {i};
while (Reach != Cache) {
  Cache = Reach;
  Reach = Reach + Reach.R;
}
return Reach;
```

where `{}` denotes the empty set and `+` denotes the union of sets. Note that this algorithm terminates if S is finite.

Intuitively, states that are not reachable from the initial state i via R should not impact whether i satisfies properties of interest. So one could first compute that set and restrict S to said set before model checking i. This is sometimes, but not always, more cost-effective.

An important restriction that many model checkers and this Section 2.1 make on such models is that R is *total*: for all $s \in S$ there is some s' with sRs'. This can easily be forced by adding a new state s_* to S and extending R with s_*Rs_*, and sRs_* whenever there is no s' in S with sRs'.

Example 3. Let S be the infinite set of natural numbers $n \geq 0$ and 0 be the designated initial state. The state transition relation R consists of four mutually disjoint subsets: $R_1 = \{(0, n) \mid n \geq 1\}$, $R_2 = \{(n, n/2) \mid n \text{ is even}, n > 1\}$, $R_3 = \{(n, 3n+1) \mid n \text{ is odd}, n > 1\}$, and $R_4 = \{(1, 1)\}$. The set of atomic propositions is $\{\mathbf{1}, o, e\}$ with $L(\mathbf{1}) = \{1\}$, $L(e) = \{2, 4, 6, \dots\}$, and $L(o) = \{1, 3, 5, \dots\}$. Thus we have three atomic propositions: $\mathbf{1}$ meaning "being equal to 1", o meaning "being odd and positive", and e meaning "being even and positive". Ignoring the labelling function L, we may think of this model as a representation of the program

```
int Collatz(c : int) {
  int n = max(1,c);
  while (n != 1) {
    if (n % 2 == 0) { n = n % 2;
    } else { n = 3 * n + 1; }
  }
  return n;
}
```

Note how 0 models any non-positive input value of `c`. The state-transition component R_1 models the statement `int n = max(1,c);` soundly and completely, as well as the un-bounded non-determinism present in the header `(c : int)` of that method. The actual value of `c` is only known when method calls take place at run-time. Finally, R_2 models all possible executions of the if-branch whereas R_3 models all possible executions of the else-branch. The component R_4 has no match in the program but ensures that R is total. The components R_2, R_3, and R_4 are deterministic, only R_1 captures the non-determinism at the abstract input state 0. The meaning of the while-statement is implicit in this Kripke structure: the determinacy of R_2, R_3, and R_4 force the control flow of that while-statement as observed at run-time.

We now turn to formal property languages. The grammar

$$\phi ::= p \mid \neg\phi \mid \phi \wedge \phi \mid \mathsf{X}\phi \mid \phi \mathsf{U} \phi$$

defines linear-time temporal logic [36] (LTL). These formulas are evaluated over computation paths of models M, which are infinite sequences $s_0 s_1 s_2 \dots$ of states in S such that $s_i R s_{i+1}$ for all $i \geq 0$. In that case, we also write $s_0 R s_1 R s_2 R \dots$.

The relation $\models\ \subseteq \mathcal{K} \times \text{LTL}$ is defined in two stages. Given a computation path $\pi = s_0 R s_1 R \dots$ in any Kripke structure M with total R, we define $\pi \models \phi$ by structural induction on π and ϕ:

- $\pi \models p$ iff $s_0 \in L(p)$
- $\pi \models \neg\phi$ iff $\pi \not\models \phi$
- $\phi \models \phi_1 \wedge \phi_2$ iff $\pi \models \phi_1$ and $\pi \models \phi_2$
- $\phi \models \mathsf{X}\phi$ iff $s_1 R s_2 R \cdots \models \phi$
- $\phi \models \phi_1 \mathsf{U} \phi_2$ iff there is some $j \geq 0$ such that
  - for all $0 \leq i < j$, $s_i R s_{i+1} R \cdots \models \phi_1$
  - $s_j R s_{j+1} R \cdots \models \phi_2$

Note that this predicate $\models$ does not have type $\mathcal{K} \times$ LTL as it applies to computation paths, not models. To get the right type, we overload the symbol $\models$ and set $M \models \phi$ to be true iff $\pi \models \phi$ holds for all computation paths π whose first state is the initial state of M.

The operators X and U are called neXT and Until operator (respectively). Formula $p\mathsf{U}q$ closely corresponds to the regular expression p^*q since $\pi \models p\mathsf{U}q$ checks whether π begins with a finite "word" of p-states, immediately followed by a q-state. Apart from the usual logical operators $\vee$ and $\rightarrow$, additional temporal operators can be defined from the basic LTL grammar:

– ϕ holds at some point in the Future: $\mathsf{F}\phi = (p \vee \neg p)\mathsf{U}\phi$, literally: "true" is true until ϕ is true (and ϕ needs to become true)
– ϕ holds Globally: $\mathsf{G}\phi = \neg\mathsf{F}\neg\phi$, literally: ϕ is and remains to be true forever (as it is not true that $\neg\phi$ holds at some point in the future).

Example 4. Figure 1 shows a Kripke structure with four states, initial state 0, and three atomic observables $\mathsf{AP} = \{\mathbf{1}, o, e\}$. States 0 and 1 stand for their respective natural numbers. State odd$_+$ abstracts all integers that are odd and greater than 1, the set $\{3, 5, \dots\}$, into a single state. Similarly, state even$_+$ abstracts even and positive numbers, the set $\{2, 4, 6, \dots\}$, into a single state. The transitions are such that the model can move from 0 to any state other than 0, and 1 can only move to itself. The remaining transitions are abstract interpretations [12] of the program rules that map integers $n > 1$ to $n/2$ if n is even; and map n to $3n + 1$ if n is odd. Such transitions are present iff there is such a possible transition in the represented concrete state sets. For example, there is a transition odd$_+ \rightarrow$ even$_+$ in Figure 1 since 5 is greater than 1 and odd, and $3 \cdot 5 + 1$ is even and positive. On the other hand, there is no transition from odd$_+$ to 1 in Figure 1 since there is no odd $n > 1$ such that $3n + 1$ equals 1. Note that the transitions out of 0 and 1 obey that same abstract interpretation.

We illustrate the LTL semantics with some checks on that model: we have

– $0\,\mathrm{even}_+\,\mathrm{odd}_+ \cdots \models \mathsf{X}e$ since $\mathrm{even}_+\,\mathrm{odd}_+ \cdots \models e$, the latter since $\mathrm{even}_+ \in L(e)$
– $0\,\mathrm{even}_+\,\mathrm{odd}_+\,\mathrm{even}_+\,1\,1 \cdots \models (e \vee \neg e)\mathsf{U}(\mathbf{1} \wedge o)$ since that path reaches state 1 which satisfies $\mathbf{1} \wedge o$
– $\mathrm{odd}_+ \models (\mathsf{G}\,\mathsf{F}\,e) \vee (\mathsf{G}\,\mathsf{F}\,o)$ since every path beginning in odd$_+$ satisfies that e (or o) is true infinitely often, all cycles contain some state labelled with e or o
– $0 \not\models \mathsf{G}\,(e \rightarrow \mathsf{F}\,o)$ since $0\,\mathrm{even}_+\,\mathrm{even}_+ \cdots \not\models \mathsf{G}\,(e \rightarrow \mathsf{F}\,o)$, the latter since we have
  – $0\,R\,\mathrm{even}_+$
  – $\mathrm{even}_+ \models e$
  – $\mathrm{even}_+\,\mathrm{even}_+ \cdots \not\models \mathsf{F}\,o$.

One can even define past-tense (i.e. "backwards") versions of X and U within LTL and mix them with their forward versions within specifications. To achieve this, we need to enrich the judgment $s_0Rs_1R \cdots \models \phi$ with a placeholder for positions, $s_0Rs_1R \dots, i \models \phi$, with the interpretation that ϕ holds at the computation path

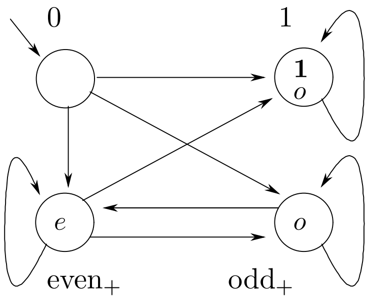

Fig. 1. A Kripke structure with state set $\{0, 1, \mathrm{even}_+, \mathrm{odd}_+\}$, initial state 0 (the incoming transition without any source), $\mathsf{AP} = \{\mathbf{1}, o, e\}$, and labelling function L satisfying $L(\mathbf{1}) = \{1\}$, $L(o) = \{1, \mathrm{odd}_+\}$, and $L(e) = \{\mathrm{even}_+\}$. Transitions are indicated as directed arrows: sRs' iff there is an arrow $s \rightarrow s'$

$s_iRs_{i+1}R \dots$, the suffix of $s_0Rs_1R \dots$ from position i onwards. Our semantics for LTL above is easily re-written in this style. The meaning of X^-, e.g., is then given by

$$\pi, i \models \mathsf{X}^-\phi \text{ iff } i > 0 \text{ and } \pi, i-1 \models \phi\,.$$

In particular, $\pi, 0 \models X^-\phi$ is false for all π and ϕ as the entire path π cannot be shifted one unit into the past.

Even without past-tense operators, we may need to modify the definition of $\models$ if the state-transition relation R is not total or if computation paths are limited to a specified length $n > 0$, as used in bounded model checking [3] where only states reachable within n steps are being explored. Bounded model checking is the verification technique used in the paper *Improved Verification of Hardware Designs through Antecedent Conditioned Slicing.*

We already saw that basic LTL model checks are conducted over computation paths and then promoted to states through a universal quantification. Temporal logics can benefit from making such quantification explicit and from also allowing it in existential mode. The resulting grammar

$$\phi ::= p \mid \neg\phi \mid \phi \wedge \phi \mid \mathsf{EX}\phi \mid \mathsf{AF}\phi \mid \mathsf{E}[\phi\mathsf{U}\phi]$$

where $p \in \mathsf{AP}$, defines Computation Tree Logic [6] (CTL), whose formulas apply to *states.*

Intuitively, $\mathsf{EX}\phi$ is an existentially quantified version of $\mathsf{X}\phi$: "there is a next state satisfying ϕ"; similarly, $\mathsf{AF}\phi$ is universally quantified $\mathsf{F}\phi$: "on all paths, at some future ϕ"; whereas $\mathsf{E}[\phi_1\mathsf{U}\phi_2]$ is existentially quantified $\phi_1\mathsf{U}\phi_2$: "there is a path on which ϕ_1 holds until ϕ_2 holds. Since in CTL each temporal operator is immediately captured by a path quantifier, we obtain for a model $M = (S, i, R, L)$ a predicate $\models$ of type $S \times \mathrm{CTL}$ as follows:

– $s \models p$ iff $s \in L(p)$
– $s \models \neg\phi$ iff $s \not\models \phi$
– $s \models \phi_1 \wedge \phi_2$ iff $s \models \phi_1$ and $s \models \phi_2$
– $s \models \mathsf{EX}\phi_1$ iff there is sRs' such that $s' \models \phi$

– $s \models \mathsf{AF}\phi$ iff for all computation paths $s_0 R s_1 R \ldots$ with $s = s_0$ there is some $j \geq 0$ such that $s_j \models \phi$
– $s \models \mathsf{E}[\phi_1 \mathsf{U} \phi_2]$ iff there is some $s_0 R s_1 R \ldots$ with $s = s_0$ and some $j \geq 0$ such that
  – for all $0 \leq i < j$, $s_i \models \phi_1$
  – $s_j \models \phi_2$

We overload $\models$ to type $\mathcal{K} \times \mathrm{CTL}$ by setting $M \models \phi$ iff $i \models \phi$, where i is the designated initial state of M. Again, we can derive other temporal operators:

$$\begin{aligned} \mathsf{AX}\phi &= \neg\mathsf{EX}\neg\phi \\ \mathsf{EF}\phi &= \mathsf{E}[(p \vee \neg p)\mathsf{U}\phi] \\ \mathsf{EG}\phi &= \neg\mathsf{AF}\neg\phi \\ \mathsf{AG}\phi &= \neg\mathsf{EF}\neg\phi \end{aligned}$$

and last, but not least,

$$\mathsf{A}[\phi_1 \mathsf{U} \phi_2] = \neg(\mathsf{EG}\neg\phi_2 \vee \mathsf{E}[\neg\phi_2 \mathsf{U} \neg\phi_1 \wedge \neg\phi_2]) \,.$$

The temporal operator $\mathsf{A}[\phi_1 \mathsf{U} \phi_2]$ specifies that, on all computation paths ϕ_1 holds until ϕ_2 holds (and ϕ_2 will hold). It is of interest to note that EU cannot be defined within CTL by any other combination of temporal operators [29].

Example 5. We do some CTL model checks for the model in Figure 1:

– $0 \models \mathsf{EX}(e \wedge \neg o)$ holds since there is a transition $0\, R\, \mathrm{even}_+$ such that $\mathrm{even}_+ \models e \wedge \neg o$
– $0 \models \mathsf{AF}(\mathsf{EG}e \vee \mathsf{EG}o)$ holds since all paths beginning in 0 reach some state from which there is either a path where e holds throughout, or a path where o holds throughout (this property is not expressible in LTL)
– $0 \models \mathsf{AF}(\mathsf{AG}e \vee \mathsf{AG}o)$ does not hold, for there is a path $0\, \mathrm{even}_+\, \mathrm{odd}_+\, \mathrm{even}_+\, \mathrm{odd}_+ \ldots$ which does not reach a state whose reachable states in the model all satisfy e, or all satisfy o.

LTL and CTL are arguably the most frequently used temporal logics in model checkers. The complexity of deciding $M \models \phi$ is linear in the size of M in each case, linear in the size of ϕ for CTL, and exponential in the size of ϕ for LTL [9]. The latter is not that big of an issue considering that the LTL formulas used in practice are typically quite short. For both logics, the linear dependency on the size of M means trouble as the state explosion problem suggests that this very size may be exponential to begin with.

The choice between LTL or CTL may also be based on other reasons [42]. For example, past tense operators are definable in LTL whereas they are not definable in CTL due to the branching-time nature of its quantifiers. On the other hand, CTL formulas like $\mathsf{AG}\,\mathsf{EF}\,\mathtt{reboot}$ are of interest in the context of self-stabilizing systems but this formula is not expressible in LTL.

Example 6. Considering the Kripke structure M_{Col} defined in Example 3, we may wonder whether the CTL model check $M_{\mathrm{Col}} \models \mathsf{AF}\,\mathbf{1}$ holds. Since $\mathsf{AF}\,\mathbf{1}$ is expressible in LTL as $\mathsf{F}\,\mathbf{1}$ we could alternatively conduct the LTL check $M_{\mathrm{Col}} \models \mathsf{F}\,\mathbf{1}$. Two things are worth noticing: the model M_{Col} has infinitely many states; and the answer to any of these two (equivalent) model checks is a well known open problem. This illustrates that finding abstract finite-state models that soundly verify (or refute) a property of the concrete model they abstract may be extremely difficult.

The example also illustrates that the patterns $\mathsf{F}p$ and $\mathsf{AF}p$ can be used to check whether all computations in a model terminate: identify all states in M that represent termination (if possible), add a new state s_* and transitions from all termination states to s_*, add a transition $s_* R s_*$, let $p \notin \mathsf{AP}$ and extend L to sort $\mathsf{AP} \cup \{p\}$ with $L(p) = \{s_*\}$. In particular, infinite-state LTL and CTL model checking reduce to the Halting Problem and are therefore undecidable. But recently, sophisticated approaches to termination checking, based on the automatic synthesis of predicate abstractions and well-founded orders, and counter-example guided abstraction refinements (CEGAR) have been proposed [11].

Having model checking at our disposal, we can identify models that satisfy the same set of properties. Given a subset Ψ of properties Φ (e.g. a subset of LTL), we define $M \equiv_\Psi M'$ iff for all $\psi \in \Psi$, we have $M \models \psi$ iff $M' \models \psi$. So if Ψ is the set of properties we are interested in, there is no semantic difference between model checking M or M' whenever $M \equiv_\Psi M'$. Of course, there could be cost differences. For example, one of the models could be finite-state, the other one infinite-state.

Also, Ψ may mention less atomic propositions than M so model M could be "sliced" to a model M' that contains atomic propositions present in Ψ only. For programs, this can be done by specifying a slicing criterion, doing a dependency analysis to determine which program statements need to be retained to preserve the semantics for the slice, and by then compressing transitions through the removal of redundant statements [43]. This is the context of the work reported in the paper *Improved Verification of Hardware Designs through Antecedent Conditioned Slicing.* Again, there is a tradeoff between the cost of computing such a smaller model and the cost savings obtained from model checking the compressed model.

2.2 Action-based models

In some systems observable information does not reside within states but in the state transitions themselves. In testing software as a black box, for example, we don't know any internal state of the software but we trigger events by entering data in GUIs or moving the mouse etc. The software may then reply with other events such as opening another GUI or redirecting the user to a data entry point. We are then interested in seeing which events can occur in which contexts or orders, as well as in the infeasibility of certain events or their ordered execution. We use the terms "event" and "action" interchangeably here.

This view is particularly fruitful for systems that are component-based and where components offer encapsulation of data and so observations are limited to the information inherent in the events and their spatio-temporal unfolding. We mention web services and biological systems as examples for which action-based models render good representations. It is apt to point out that both of these examples will benefit from quantitative or stochastic aspects in models and model checks but our paper limits itself to qualitative model checking only.

Given a set Act of actions, a basic form of action-based model M is a tuple (S, i, R) with

– a (possibly infinite) set of states S
– a designated initial state $i \in S$, and
– a state transition relation $R \subseteq S \times \mathsf{Act} \times S$.

The roles of S and i are as for Kripke structures. But now we have no labelling function L and no atomic propositions AP. This is compensated for by the type of the state transition relation. Transitions are now triples $(s, \alpha, s') \in R$ expressing that *if* event α occurs in state s, then the next state of the model may be s'. Alternatively, we may think of R as a *set* of transition relations of type $S \times S$, one such relation for each event $\alpha \in \mathsf{Act}$.

Process algebras, such as CSP and its model checker FDR [40], are formalisms for describing action-based models. Consider the grammar

$$p ::= \mathbf{0} \ \mid \ \mathbf{1} \ \mid \ \alpha.p \ \mid p + p \qquad (1)$$

with $\alpha \in \mathsf{Act}$. Let $\mathbf{0}$ denote a process that cannot engage in any event of Act. Dually, let $\mathbf{1}$ be the process that can engage in all events of Act, whereupon it reaches its own state again. The expression $\alpha.p$ describes a process that can only engage in event α, and if done so will turn into process p. Finally, $p+q$ denotes a process that can engage in any event that p or q can engage in, but in doing so it will turn into a continuation of p or q, respectively.

We can formalize these intuitions by mapping each process term of (1) to the model it represents:

– the model for $\mathbf{0}$ is $(\{\mathbf{0}\}, \{\}, \{\mathbf{0}\})$: there is only one state which is also the initial state, and there are no transitions
– the model for $\mathbf{1}$ is $(\{\mathbf{1}\}, \{(\mathbf{1}, \alpha, \mathbf{1}) \mid \alpha \in \mathsf{Act}\}, \{\mathbf{1}\})$: there is only one state which is also the initial state, and there is one transition for each action α
– the model for $\alpha.p$ is defined recursively; let (S, i, R) be the model already computed for p; then the model for $\alpha.p$ has the disjoint union $S \cup \{\alpha.p\}$ as state space, $\alpha.p$ as initial state, and the disjoint union $R \cup \{(\alpha.p, \alpha, p)\}$ as state transition relation

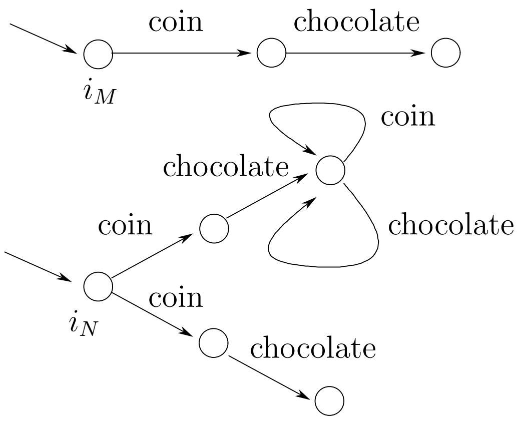

Fig. 2. Two action-based models M (top) and N (bottom) with respective initial states i_M and i_N. Model M is the meaning of the process term coin.chocolate.$\mathbf{0}$. Model N is the meaning of the process term coin.chocolate.$\mathbf{0}$ + coin.chocolate.$\mathbf{1}$

– similarly, the model for $p + q$ makes use of already computed models (S_p, i_p, R_p) and (S_q, i_q, R_q) for p and q (respectively); the new state space is the disjoint union $S_p \setminus \{i_p\} \cup S_q \setminus \{i_q\} \cup \{p + q\}$, the initial state is $p + q$, and the new state transition relation is the disjoint union of R_p and R_q where in transitions all occurrences of p and q are renamed to $p + q$.

Example 7. Let $\mathsf{Act} = \{\text{coin}, \text{chocolate}\}$. The model M for the process term coin.chocolate.$\mathbf{0}$ is depicted in Figure 2(top). The model N for the process term

$$\text{coin.chocolate.}\mathbf{0} + \text{coin.chocolate.}\mathbf{1} \qquad (2)$$

is seen in Figure 2(bottom). Note that the state transition relation R may not be total.

Both model M and the property of interest, ϕ, may be specified by process terms p and q (respectively). The model check $M \models \phi$ may then ask whether all maximal sequences of events that p can generate from its initial state are also maximal sequences of events specified in q. Formally, the traces of a model $M = (S, i, R)$ are defined as

$$\mathsf{Trace}(M) = \{\alpha_1\alpha_2 \cdots \mid \exists_{\max}(i, \alpha_1, s_1)R(s_1, \alpha_2, s_2)R \dots\}$$

where each sequence $(i, \alpha_1, s_1)R(s_1, \alpha_2, s_2)R\dots$ is maximal: it is either infinite, or it ends in (s_n, α, s_{n+1}) such that there is no $s' \in S$ and no $\alpha \in \mathsf{Act}$ with $(s_{n+1}, \alpha, s') \in R$. Each such maximal sequence of transitions determines a sequence of events, the ones that occurred in that order. The set $\mathsf{Trace}(M)$ records all such event sequences generated from the initial state of M.

Example 8. For the model M in Figure 2 we have that $\mathsf{Trace}(M) = \{\text{coin}\,\text{chocolate}\}$. Note that $\mathsf{Trace}(M) \subseteq \mathsf{Trace}(N)$ since coin chocolate is also a trace of N, realized by choosing the left summand of (2). Therefore

$M \models N$ holds if $\models$ is understood as checking trace containment. In the literature this is also called *refinement checking*. Conversely, we have $N \not\models M$ as $\mathsf{Trace}(N)$ is not a subset of $\mathsf{Trace}(M)$: for example, the infinite sequence coin chocolate chocolate chocolate . . . is in $\mathsf{Trace}(N)$ but not in $\mathsf{Trace}(M)$. Indeed, the set $\mathsf{Trace}(N)$ is uncountable whereas $\mathsf{Trace}(M)$ is a singleton.

One can vary the notion of trace, e.g., by adding to $\mathsf{Trace}(M)$, as defined in this paper, all its finite prefixes. This notion of trace is close to that used with refinement checking in the paper *Compositional Software Verification Based On Game Semantics and CSP*. Action-based models can easily be enriched with labelling functions L of the same type as for Kripke structures. It is also possible to encode Kripke structures as action-based models, and vice versa. Thus it is not surprising that action-based models and their trace inclusion checks can be applied to checking safety properties of computer programs. For example, we may see a fragment of code as an open system that reacts to and initiates certain events. Asking questions about the input, and answering questions in the output can then both be seen as events. This gives rise to the interpretation of open program fragments as dialogue games [41,23], which are the semantic formalism for the verification method proposed in the paper *Compositional Software Verification Based On Game Semantics and CSP*.

3 The special section

All papers in this special issue demonstrate advances in the automated verification of critical systems, specifically in the area of model checking. We discuss briefly the contributions of each of these papers:

The paper *Formal Verification of the NASA Runway Safety Monitor* is a substantial case study of applying model checking to real world designs. It uses a tool for model checking Petri nets to model and analyze a component of NASA's Runway Incursion Prevention System. Model checks corroborated the overall robustness of this protocol but also pointed out potential problems to designers, who then took corrective action. The proposed method of discretizing continuous variables should be of interest for the verification of hybrid systems in which the discrete portion is quite substantial and complex. This case study judiciously over-approximates physically possible behavior but also under-approximates as it limits the scope of the model, e.g. by putting a bound on the number of aircrafts involved. This implicitly appeals to Daniel Jackson's *Small Scope Hypothesis* [24] saying that the vast majority of flaws in a system can be found by exhaustively testing all input parameters within some small scope of values. This work also exploits data structures and algorithms for efficient reachability analysis over Petri nets or similar *globally asynchronous, locally synchronous* systems [5].

The paper *Improved Verification of Hardware Designs through Antecedent Conditioned Slicing* advances upon existing slicing techniques and validates the proposed improvement on a case study, a Verilog Register Transfer Level implementation of the USB 2.0 functional core. Given a property to be checked, its observables determine a slicing criterion which drives a static analysis that transforms source code by ignoring statements that are revealed not to change the meaning of the check one wants to perform. The novel contribution is to exploit the logical structure occurring in specification patterns, here the antecedents of logical implications — which often capture pre-conditions or filters [17] — in that analysis. This exploitation of logical structure means that the model simplifications go beyond the cone of influence reduction technique that may already be implicitly enforced in model checkers. Antecedent conditioned slicing can result in considerably smaller models and faster checks. It is hoped that slicing techniques will increasingly exploit the logical structure of properties they mean to verify.

The paper *Compositional Software Verification Based On Game Semantics and CSP* applies game semantics in the area of software verification. It considers a non-trivial programming language, call-by-name second-order recursion-free Idealized Algol, and models open program fragments in a compositional manner as processes in the process algebra CSP [40]. These models encode the semantics of program fragments in their interpretation as dialogue games [41,23]. Although this approach appears similar to representing and checking program fragments as regular expressions, it enables the use of existing tool suites and optimization techniques for CSP processes; we mention compositional state-reduction techniques, automated refinement checks, and interactive debugging. The suggested approach is illustrated on a number of small examples and the paper offers additional results on parameterized model checking of polymorphic programs and their properties. This game-semantics approach to software verification has now reached a proof-of-concept stage and supports counter-example-guided abstraction refinement [15].

The paper *Model-checking the Preservation of Temporal Properties upon Feature Integration* is a foundational study concerned with feature integration and the preservation of properties. Adding features to existing systems corresponds to removing some, and adding other, behavior. Therefore feature integration may invalidate properties that had been verified prior to such integration, incurring costs for re-checking. The authors propose an implementable method that can determine whether a feature integration will preserve already established safety properties. The criteria put forward in this method are sound and incomplete, but appear to be complete for safety properties as far as practical scenarios are concerned. The paper also shows that the method is valid for certain patterns of liveness properties. It would be of interest to see whether this work is extendable to 3-valued settings in which the environment is made explicit such that refinement can capture feature composition [32].

The paper *CTL-Property transformations along an incremental design process* uses Moore machines to model flow-control oriented hardware devices. An incremental design process of such devices leads to the addition of behavior at each iteration of the design. If added behavior does not override existing behavior, then each iteration transforms CTL formulas without increasing the modal depth of formulas such that the transformed formula holds on the incremented design if, and only if, the original formula holds on the design as given prior to the increment. The utility of this approach is demonstrated on the non-regression analysis of bus protocol converters.

References

1. T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A Model Checker for Concurrent Software. In *Proc. of the 16th Intl. Conference on Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 484–487, Boston, Massachussets, 13-17 July 2004. Springer Verlag.
2. T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *Conference Record of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, Portland, Oregon, 16-18 January 2002. ACM Press.
3. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proc. of the Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer Verlag, 1999.
4. R. Blossey, L. Cardelli, and A. Phillips. Stochastic dynamics of gene networks. *Theoretical Computer Science*, 2005. To appear.
5. G. Ciardo. Reachability set generation from Petri nets: can brute force be smart? In *Proc. of the 25th Intl. Conference on Applications and Theory of Petri Nets*, volume 3099 of *Lecture Notes in Computer Science*, pages 17–34, Bologna, Italy, June 2004. Springer Verlag.
6. E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In D. Kozen, editor, *Logic of Programs Workshop*, volume 131 of *LNCS*, pages 244–263. Springer Verlag, 1981.
7. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of the 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169, Berlin, Germany, 2000. Springer Verlag.
8. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.
9. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, January 2000.
10. R. Cleaveland. Pragmatics of model checking: an STTT special section. *International Journal on Software Tools for Technology Transfer*, 2:208–218, 1999.
11. B. Cook, A. Podelski, and A. Rybalchenko. Abstraction Refinement for Termination. In *Proc. of the 12th International Symposium on Static Analysis*, volume 3672 of *Lecture Notes in Computer Science*, pages 87–101, London, England, 7-9 September 2005. Springer Verlag.
12. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs. In *Proc. 4th ACM Symp. on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.
13. D. Dams. *Abstract interpretation and partition refinement for model checking*. PhD thesis, Technische Universiteit Eindhoven, The Netherlands, 1996.
14. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM TOPLAS*, 19:253–291, 1997.
15. A. Dimovski, D. R. Ghica, and R. Lazić. Data-Abstraction Refinement: A Game Semantic Approach. In *Proc. of the 12th International Symposium on Static Analysis*, volume 3672 of *Lecture Notes in Computer Science*, pages 102–117, London, England, 7-9 September 2005. Springer Verlag.
16. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In *Proc. of the second workshop on Formal Methods in software practice*, pages 7–15, Clearwater Beach, Florida, 4-5 March 1998. ACM Press.
17. M. B. Dwyer and D. A. Schmidt. Limiting State Explosion with Filter-Based Refinement. In *Proc. of the ILPS'97 Workshop on Verification, Model Checking, and Abstraction*, 1997.
18. P. Godefroid and R. Jagadeesan. On The Expressiveness of 3-Valued Models. In L. D. Zuck, P. C. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *Proc. of 4th Conference on Verification, Model Checking and Abstract Interpretation*, volume 2575 of *LNCS*, pages 206–222, New York, January 2003. Springer Verlag.
19. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. of the 9th Intl. Conference on Computer Aided Verification*, volume 1254 of *LNCS*, pages 72–83, Haifa, Israel, June 1997. Springer Verlag.
20. H. A. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.
21. G. J. Holzmann and M. H. Smith. Automating Software Feature Verification. *Bell Labs Technical Journals*, 5(2):72–87, 2000.
22. M. Huth, R. Jagadeesan, and D. A. Schmidt. Modal transition systems: a foundation for three-valued program analysis. In D. Sands, editor, *Proc. of the European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 155–169. Springer Verlag, April 2001.
23. J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II and III. *Information and Computation*, 163:285–400, 2000.
24. D. Jackson and C. A. Damon. Elements of style: Analyzing a software design feature with a counterexample

detector. *IEEE Transactions on Software Engineering*, 22(7):484–495, 1996.

25. K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
26. S. C. Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.
27. O. Kupferman, M. Y. Vardi, and P. Wolper. Module Checking. *Information and Computation*, 164(2):322–344, January 2001.
28. M. Kwiatkowska. Model checking for probability and time: from theory to practice. In *18th Annual IEEE Symposium on Logic in Computer Science*, pages 351–360, Ottawa, Canada, 22-25 June 2003. IEEE Computer Society.
29. F. Laroussinie. About the expressive power of CTL combinators. *Information Processing Letters*, 54(6):343–345, 1995.
30. K. G. Larsen and B. Thomsen. A Modal Process Logic. In *Proc. of the 3rd Annual Symposium on Logic in Computer Science*, pages 203–210. IEEE Computer Society Press, 1988.
31. H. Li, S. Krishnamurthi, and K. Fisler. Verifying cross-cutting features as open systems. In *Proc. of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 89–98, Charleston, South Carolina, 18-22 November 2002. ACM Press.
32. H. Li, S. Krishnamurthi, and K. Fisler. Modular Verification of Open Features Through Three-Valued Model Checking. *Journal of Automated Software Engineering*, July 2003, Accepted for publication.
33. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. *Information and Computation*, 100(1):1–40, September 1992.
34. G. Norman, D. Parker, M. Kwiatkowska, S. Shukla, and R. Gupta. Using probabilistic model checking for dynamic power management. *Formal Aspects of Computing*, 17(2):160–176, August 2005.
35. D. M. R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *In Proc. of the 5th GI Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer Verlag, 1989.
36. A. Pnueli. The temporal logic of programs. In *Proc. of the 18th IEEE Symposium on the Foundations of Computer Science*, pages 46–57, 1977.
37. A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In J.W. de Bakker, editor, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, pages 510–584. Springer-Verlag, 1986.
38. J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. of the 5th International Symposium on Programming*, 1981.
39. J. Romberg, J. Jürjens, and G. Wimmel. AutoFOCUS and the MoDe Tool. In *Proc. of the Third International Conference on Application of Concurrency to System Design*, pages 249–250, Guimaraes, Portugal, 18-20 June 2003. IEEE Computer Society Press.
40. A. W. Roscoe. *Concurrent and Distributed Systems: The Theory and Practice of Concurrency*. Prentice Hall, 1997.
41. S.Abramksy, P. Malacaria, and R. Jagadeesan. Full abstraction for PCF. In *Proc. of the Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 1–15, Sendai, Japan, 19-22 April 1994.
42. M. Y. Vardi. Branching vs. Linear Time: Final Showdown. In *Proc. of the 7th Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22, Genova, Italy, 2-6 April 2001. Springer Verlag.
43. M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.