Reasoning about Abstract Open Systems with Generalized Module Checking

Patrice Godefroid

Bell Laboratories, Lucent Technologies, god@bell-labs.com

Abstract. We present a framework for reasoning about abstract open systems. Open systems, also called "reactive systems" or "modules", are systems that interact with their environment and whose behaviors depend on these interactions. Embedded software is a typical example of open system. Module checking [KV96] is a verification technique for checking whether an open system satisfies a temporal property no matter what its environment does. Module checking makes it possible to check adversarial properties of the "game" played by the open system with its environment (such as "is there a winning strategy for a malicious agent trying to intrude a secure system?"). We study how module checking can be extended to reason about 3-valued abstractions of open systems in such a way that both proofs and counter-examples obtained by verifying arbitrary properties on such abstractions are guaranteed to be sound, i.e., to carry over to the concrete system. We also introduce a new verification technique, called *generalized module checking*, that can improve the precision of module checking. The modeling framework and verification techniques developed in this paper can be used to represent and reason about abstractions automatically generated from a static analysis of an open program using abstraction techniques such as predicate abstraction. This application is illustrated with an example of open program and property that cannot be verified by current abstraction-based verification tools.

1 Introduction

Software verification via automatic abstraction and model checking is currently an active area of research (e.g., [BR01,CDH$^+$00,DD01,HJMS02,VHBP00]). This approach consists of automatically extracting a model out of a program by statically analyzing its code, and then of analyzing this model using model-checking techniques. If the model-checking results are inconclusive due to too much information being lost in the current abstraction, the model can then be automatically refined into a more detailed one provided the abstraction process can be parameterized and adjusted dynamically guided by the verification needs, as is the case with predicate abstraction [GS97] for instance. Current frameworks and tools that follow the above paradigm typically use traditional formalisms (such as Kripke structures or Labeled Transition Systems) for representing models, while the soundness of their analysis is based on using a simulation relation for relating the abstract model to the concrete program being analyzed. Two well-known drawbacks of these design choices are that the scope of verification is then limited to universal properties, and that counter-examples are generally unsound since abstraction usually introduces unrealistic behaviors that may yield spurious errors being reported when analyzing the model. In practice, the second limitation is perhaps more severe since the relative popularity of model checking (especially in industry) is due to its ability to detect errors that would be very hard to find otherwise, and not so much to its ability to prove "correctness".

Recently [GJ02,GHJ01,HJS01,BG00,BG99], it was shown how automatic abstraction can be performed to verify arbitrary formulas of the propositional μ-calculus [Koz83] in such a way that both correctness proofs and counter-examples are guaranteed to be sound. The key to make this possible is to represent abstract systems using richer models that distinguish properties that are true, false and unknown of the concrete system. Reasoning about such systems thus requires 3-valued temporal logics [BG99], i.e., temporal logics whose formulas may evaluate to *true*, *false* or $\perp$ ("unknown") on a given model. Then, by using an automatic abstraction process that generates by construction an abstract model which is less complete than the concrete system with respect to a completeness preorder logically characterized by 3-valued temporal logic, every temporal property ϕ that evaluates to *true* (resp. *false*) on the abstract model automatically holds (resp. does not hold) of the concrete system, hence guaranteeing soundness of both proofs and counter-examples. In case ϕ evaluates to $\perp$ on the model, a more precise verification technique called *generalized model checking* [BG00,GJ02] can be used to check whether there exist concretizations of the abstract model that satisfy ϕ or violate ϕ; if a negative answer is obtained in either one of these two tests, ϕ does not hold (resp. holds) of the concrete system. Otherwise, the analysis is still inconclusive and a more complete (i.e., less abstract) model is then necessary to provide a definite answer concerning this property of the concrete system. This approach can be used to both prove and refute arbitrary formulas of the propositional μ-calculus.

In this paper, we investigate how the scope of program verification via automatic abstraction can be extended to deal with programs implementing *open systems*. An *open system*, also called "reactive system" or "module", is a system that interacts with its environment and whose behavior depends on this interaction. Embedded software is a typical example of open system. It has been argued [KV96] that temporal properties of an open system should be verified with respect to all possible environments for that system. This problem is known as the *module checking* problem [KV96]. Module checking makes it possible to check adversarial properties of the "game" played by the open system and its environment (such as "is there a winning strategy for a malicious agent trying to intrude a secure system?").

We study how module checking can be extended to reason about 3-valued abstractions of open systems in such a way that both proofs and counter-examples obtained by verifying arbitrary properties of such abstractions are guaranteed to be sound, i.e., to carry over to the concrete system. We also introduce a new verification technique, called *generalized module checking*, that can improve the precision of module checking. The practical motivation of this paper is thus to develop a framework for representing and reasoning about abstractions automatically generated from a static analysis of an open program using abstraction techniques such as predicate abstraction. Our framework is illustrated by an example of application in Section 6. Note that existing software verification-by-abstraction frameworks and tools (e.g., [BR01,CDH$^+$00,DD01,HJMS02,VHBP00]) do not currently support verification techniques for open programs.

2 Background

2.1 3-Valued Models and Generalized Model Checking

In this section, we recall the main ideas and key notions behind the framework of [BG99,BG00,GHJ01,HJS01] for reasoning about partially defined systems. Examples of modeling formalisms for representing such systems are *partial Kripke structures* (PKS) [BG99], *Modal Transition Systems* (MTS) [LT88,GHJ01] or *Kripke Modal Transition Systems* (KMTS) [HJS01].

Definition 1. – *A* KMTS M *is a tuple* $(S, P, \xrightarrow{must}, \xrightarrow{may}, L)$*, where* S *is a nonempty finite set of states,* P *is a finite set of atomic propositions,* $\xrightarrow{may} \subseteq S \times S$ *and* $\xrightarrow{must} \subseteq S \times S$ *are transition relations such that* $\xrightarrow{must} \subseteq \xrightarrow{may}$*, and* $L : S \times P \rightarrow \{true, \bot, false\}$ *is an* interpretation *that associates a truth value in* $\{true, \bot, false\}$ *with each atomic proposition in* P *for each state in* S*.*
- *An MTS is a KMTS where* $P = \emptyset$*.*
- *A PKS is a KMTS where* $\xrightarrow{must} = \xrightarrow{may}$*.*
- *A Kripke structure (KS) is a PKS where* $\forall s \in S : \forall p \in P : L(s,p) \neq \bot$*.*

The third value $\bot$ (read "unknown") and *may*-transitions unmatched by *must*-transitions are used to model explicitly a loss of information due to abstraction concerning, respectively, state or transition properties of the concrete system being modeled. A standard, *complete* Kripke structure is a special case of KMTS where $\xrightarrow{must} = \xrightarrow{may}$ and $L : S \times P \rightarrow \{true, false\}$, i.e., no proposition takes value $\bot$ in any state. It is worth noting that PKSs, MTSs and KMTSs are all equally expressive (i.e., one can translate any formalism into any other) [GJ03].

In interpreting propositional operators on KMTSs, we use Kleene's strong 3-valued propositional logic [Kle87]. Conjunction $\wedge$ in this logic is defined as the function that returns *true* if both of its arguments are *true*, *false* if either argument is *false*, and $\bot$ otherwise. We define negation $\neg$ using the function '*comp*' that maps *true* to *false*, *false* to *true*, and $\bot$ to $\bot$. Disjunction $\vee$ is defined as usual using De Morgan's laws: $p \vee q = \neg(\neg p \wedge \neg q)$. Note that these functions give the usual meaning of the propositional operators when applied to values *true* and *false*.

Propositional modal logic (PML) is propositional logic extended with the modal operator AX (which is read "for all immediate successors"). Formulas of PML have the following abstract syntax: $\phi ::= p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid AX\phi$, where

p ranges over P. The following 3-valued semantics generalizes the traditional 2-valued semantics for PML.

Definition 2. *The value of a formula ϕ of 3-valued PML in a state s of a KMTS $M = (S, P, \xrightarrow{must}, \xrightarrow{may}, L)$, written $[(M,s) \models \phi]$, is defined inductively as follows:*

$$\begin{aligned}
[(M,s) \models p] &= L(s,p) \\
[(M,s) \models \neg\phi] &= comp([(M,s) \models \phi]) \\
[(M,s) \models \phi_1 \wedge \phi_2] &= [(M,s) \models \phi_1] \wedge [(M,s) \models \phi_2] \\
[(M,s) \models AX\phi] &= \begin{cases} true \text{ if } \forall s' : s \xrightarrow{may} s' \Rightarrow [(M,s') \models \phi] = true \\ false \text{ if } \exists s' : s \xrightarrow{must} s' \wedge [(M,s') \models \phi] = false \\ \perp \quad otherwise \end{cases}
\end{aligned}$$

This 3-valued logic can be used to define a preorder on KMTSs that reflects their degree of completeness. Let $\leq$ be the *information ordering* on truth values, in which $\perp \leq true$, $\perp \leq false$, $x \leq x$ (for all $x \in \{true, \perp, false\}$), and $x \not\leq y$ otherwise.

Definition 3 ($\preceq$). *Let $M_A = (S_A, P, \xrightarrow{must}_A, \xrightarrow{may}_A, L_A)$ and $M_C = (S_C, P, \xrightarrow{must}_C, \xrightarrow{may}_C, L_C)$ be KMTSs. The* completeness preorder *$\preceq$ is the greatest relation $\mathcal{B} \subseteq S_A \times S_C$ such that $(s_a, s_c) \in \mathcal{B}$ implies the following:*

- $\forall p \in P : L_A(s_a, p) \leq L_C(s_c, p)$,
- *if $s_a \xrightarrow{must}_A s'_a$, there is some $s'_c \in S_C$ such that $s_c \xrightarrow{must}_C s'_c$ and $(s'_a, s'_c) \in \mathcal{B}$,*
- *if $s_c \xrightarrow{may}_C s'_c$, there is some $s'_a \in S_A$ such that $s_a \xrightarrow{may}_A s'_a$ and $(s'_a, s'_c) \in \mathcal{B}$.*

This definition allows to abstract M_C by M_A by letting truth values of propositions become $\perp$ and by letting *must*-transitions become *may*-transitions, but all *may*-transitions of M_C must be preserved in M_A. We then say that M_A is *more abstract*, or *less complete*, than M_C. The inverse of the completeness preorder is called *refinement preorder* in [LT88,HJS01,GHJ01]. Note that relation $\mathcal{B}$ reduces to a simulation relation when applied to MTSs with *may*-transitions only.

It can be shown that 3-valued PML logically characterizes the completeness preorder [BG99,HJS01,GHJ01].

Theorem 1. *Let $M_A = (S_A, P, \xrightarrow{must}_A, \xrightarrow{may}_A, L_A)$ and $M_C = (S_C, P, \xrightarrow{must}_C, \xrightarrow{may}_C, L_C)$ be KMTSs such that $s_a \in S_A$ and $s_c \in S_C$, and let Φ be the set of all formulas of 3-valued PML. Then,*

$$s_a \preceq s_c \text{ iff } (\forall \phi \in \Phi : [(M_A, s_a) \models \phi] \leq [(M_C, s_c) \models \phi]).$$

In other words, KMTSs that are "more complete" with respect to $\preceq$ have more definite properties with respect to $\leq$, i.e., have more properties that are either *true* or *false*. Moreover, any formula ϕ of 3-valued PML that evaluates to *true* or *false* on a KMTS has the same truth value when evaluated on any more complete structure. This result also holds for PML extended with fixpoint operators, i.e., the propositional μ-calculus [BG99,BG00].

In [GHJ01], we showed how to adapt the abstraction mappings of [Dam96] to construct abstractions that are less complete than a given concrete program with respect to the completeness preorder.

Definition 4. *Let $M_C = (S_C, P, \xrightarrow{must}_C, \xrightarrow{may}_C, L_C)$ be a (concrete) KMTS. Given a set S_A of abstract states and a total*[1] *abstraction relation on states $\rho \subseteq S_C \times S_A$, we define the (abstract) KMTS $M_A = (S_A, P, \xrightarrow{must}_A, \xrightarrow{may}_A, L_A)$ as follows:*

- $s_a \xrightarrow{must}_A s'_a$ *if* $\forall s_c \in S_C : s_c \rho s_a \Rightarrow (\exists s'_c \in S_C : s'_c \rho s'_a \wedge s_c \xrightarrow{must}_C s'_c)$;
- $s_a \xrightarrow{may}_A s'_a$ *if* $\exists s_c, s'_c \in S_C : s_c \rho s_a \wedge s'_c \rho s'_a \wedge s_c \xrightarrow{may}_C s'_c$;
- $L_A(s_a, p) = \begin{cases} \mathit{true} \text{ if } \forall s_c : s_c \rho s_a \Rightarrow L_C(s_c, p) = \mathit{true} \\ \mathit{false} \text{ if } \forall s_c : s_c \rho s_a \Rightarrow L_C(s_c, p) = \mathit{false} \\ \bot \quad \mathit{otherwise} \end{cases}$

The previous definition can be used to build abstract KMTSs.

Theorem 2. *Given a KMTS M_C, any KMTS M_A obtained by applying Definition 4 is such that $M_A \preceq M_C$.*

Given a KMTS M_C, any abstraction M_A less complete than M_C with respect to the completeness preorder $\preceq$ can be constructed using Definition 4 by choosing the inverse of ρ as $\mathcal{B}$ [GHJ01]. When applied to MTSs with *may*-transitions only, the above definition coincides with traditional "conservative" abstraction. Building a 3-valued abstraction can be done using existing abstraction techniques at the same computational cost as building a conservative abstraction [GHJ01].

Since by construction $M_A \preceq M_C$, any temporal-logic formula ϕ that evaluates to *true* (resp. *false*) on M_A automatically holds (resp. does not hold) on M_C. It is shown in [BG00] that computing $[(M_A, s) \models \phi]$ can be reduced to two traditional (2-valued) model-checking problems on regular fully-defined systems (such as Kripke structures or Labeled Transition Systems), and hence that 3-valued model-checking for any temporal logic L has the same time and space complexity as 2-valued model checking for the logic L.

However, as argued in [BG00], the semantics of $[(M, s) \models \phi]$ returns $\bot$ more often than it should. Consider a KMTS M consisting of a single state s such that the value of proposition p at s is $\bot$ and the value of q at s is *true*. The formulas $p \vee \neg p$ and $q \wedge (p \vee \neg p)$ are $\bot$ at s, although in all complete Kripke structures more complete than (M, s) both formulas evaluate to *true*. This problem is not confined to formulas containing subformulas that are tautological or unsatisfiable. Consider a KMTS M' with two states s_0 and s_1 such that $p = q = \mathit{true}$ in s_0 and $p = q = \mathit{false}$ in s_1, and with a *may*-transition from s_0 to s_1. The formula $AXp \wedge \neg AXq$ (which is neither a tautology nor unsatisfiable) is $\bot$ at s_0, yet in all complete structures more complete than (M', s_0) the formula is *false*. This observation is used in [BG00] to define an alternative 3-valued semantics for temporal logics called the *thorough* semantics since it does more than the other semantics to discover whether enough information is present in a KMTS to give a definite answer. Let the *completions* $\mathcal{C}(M, s)$ of a state s of a KMTS M be the set of all states s' of complete Kripke structures M' such that $s \preceq s'$.

[1] That is, $(\forall s_c \in S_C : \exists s_a \in S_A : s_c \rho s_a)$ and $(\forall s_a \in S_A : \exists s_c \in S_C : s_c \rho s_a)$.

Logic	MC	SAT	GMC
Propositional Logic	Linear	NP-complete	NP-complete
PML	Linear	PSPACE-complete	PSPACE-complete
CTL	Linear	EXPTIME-complete	EXPTIME-complete
μ-calculus	NP$\cap$co-NP	EXPTIME-complete	EXPTIME-complete
LTL	PSPACE-complete	PSPACE-complete	EXPTIME-complete

Fig. 1. Known results on the complexity in the size of the formula for (2-valued and 3-valued) model checking (MC), satisfiability (SAT) and generalized model checking (GMC).

Definition 5. *Let ϕ be a formula of any two-valued logic for which a satisfaction relation $\models$ is defined on complete Kripke structures. The truth value of ϕ in a state s of a KMTS M under the* thorough *interpretation, written $[(M,s) \models \phi]_t$, is defined as follows:*

$$[(M,s) \models \phi]_t = \begin{cases} \mathit{true} \text{ if } (M',s') \models \phi \text{ for all } (M',s') \text{ in } \mathcal{C}(M,s) \\ \mathit{false} \text{ if } (M',s') \not\models \phi \text{ for all } (M',s') \text{ in } \mathcal{C}(M,s) \\ \perp \quad \mathit{otherwise} \end{cases}$$

It is easy to see that, by definition, we always have $[(M,s) \models \phi] \leq [(M,s) \models \phi]_t$. In general, interpreting a formula according to the thorough three-valued semantics is equivalent to solving two instances of the generalized model-checking problem [BG00].

Definition 6 (Generalized Model-Checking Problem). *Given a state s of a KMTS M and a formula ϕ of a temporal logic L, does there exist a state s' of a complete Kripke structure M' such that $s \preceq s'$ and $(M',s') \models \phi$?*

This problem is called *generalized model-checking* since it generalizes both model checking and satisfiability checking. At one extreme, where $M = (\{s_0\}, P, \xrightarrow{must} = \xrightarrow{may} = \{(s_0,s_0)\}, L)$ with $L(s_0,p) = \perp$ for all $p \in P$, all complete Kripke structures are more complete than M and the problem reduces to the satisfiability problem. At the other extreme, where M is complete, only a single structure needs to be checked and the problem reduces to model checking.

Algorithms and complexity bounds for the generalized model-checking problem for various temporal logics were presented in [BG00]. In the case of branching-time temporal logics, generalized model checking has the same complexity in the size of the formula as satisfiability. In the case of linear-time temporal logic, generalized model checking is EXPTIME-complete in the size of the formula, i.e., harder than both satisfiability and model checking, which are both PSPACE-complete in the size of the formula for LTL. Figure 1 summarizes the complexity results of [BG00]. These results show that the complexity in the size of the formula of computing $[(M,s) \models \phi]_t$ (GMC) is always higher than that of computing $[(M,s) \models \phi]$ (MC).

Regarding the complexity in the size of the model $|M|$, it is shown in [GJ02] that generalized model checking can in general require quadratic running time

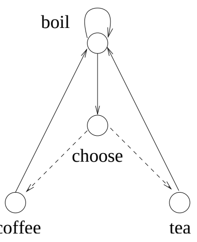

Fig. 2. Example of open system

in $|M|$, but that the problem can be solved in time linear in $|M|$ in the case of (LTL or BTL) *persistence* properties, i.e., properties recognizable by (word or tree) automata with a co-Büchi acceptance condition. Persistence properties include several important classes of properties of practical interest, such as all safety properties.

2.2 Module Checking

The framework described in the previous section assumes that the abstract system M is *closed*, i.e., that its behavior is completely determined by the state of the system. But what if M is an abstraction of an *open* system?

An *open system* is a system that interacts with its environment and whose behavior depends on this interaction. Such systems are also often called *reactive*. In [KV96], Kupferman and Vardi argued that verifying temporal properties (especially those specified in branching temporal logics) of open systems must be handled differently than traditional model checking. Their argument is best illustrated with a simple example. Consider a vending machine M which repeatedly boils water, asks the environment to choose between coffee and tea, and deterministically serves a drink according to the external choice. This machine is depicted in Figure 2, where dotted transitions represent transitions taken by the environment. The machine has four states: *boil, choose, coffee* and *tea*. When M is in state *boil*, we know exactly what its possible next states are (namely, *choose* and *boil*). In contrast, when M is in state *choose*, the set of possible next states is unknown since it depends on the environment: it could be any subset of the set $\{$*coffee, tea*$\}$. To see the difference this makes semantically, consider the property "is it always possible for M to eventually serve tea?", which can be represented as the CTL formula[2] $AGEF tea$. If we evaluate this formula on M viewed as a traditional Kripke structure, thus ignoring whether transitions are taken by the system or its environment, we obtain $[M \models AGEF tea] = true$, since, for every state of M, there exists a path from that state to state *tea*. In contrast, if we take into account transitions taken by the environment, the answer should be *false* since, if the environment decides to always choose *coffee*, the machine M will never serve tea. This observation prompted the introduction of a variant of the model-checking problem for reasoning about open systems, called the *module checking* problem:

[2] See [Eme90] for a general introduction to the temporal logics used in this paper.

Given a module (M, s) and a temporal logic formula ϕ, does (M, s) satisfy ϕ in all possible environments?

Formally, a *module* is defined in [KV96] as a Kripke structure whose set of states is partitioned into two sets: a set of system states, representing the states where the system can make transitions (such as the states *boil, coffee* and *tea* in the example above), and a second set of environment states, where the environment can make transitions (such as the state *choose* in the previous example). Here, we will use an alternative formalization (already suggested in [KV96]) and represent modules as Modal Transition Systems instead: *must*-transitions will model system transitions while *may*-transitions will model environment transitions. Note that this second formalization is more general since it allows states from which both system and environment transitions exist. To simplify notations, we also use a set of atomic propositions and associate with each state a labeling from every propositions to the set $\{true, false\}$.

Definition 7 (Module). *A* module *is a tuple* $(S, P, \stackrel{must}{\longrightarrow}, \stackrel{may}{\longrightarrow}, L)$*, where* S *is a nonempty finite set of states,* P *is a finite set of atomic propositions,* $\stackrel{may}{\longrightarrow} \subseteq S \times S$ *and* $\stackrel{must}{\longrightarrow} \subseteq S \times S$ *are transition relations such that* $\stackrel{must}{\longrightarrow} \subseteq \stackrel{may}{\longrightarrow}$*, and* $L : S \times P \to \{true, false\}$ *is an* interpretation *that associates a truth value in* $\{true, false\}$ *with each atomic proposition in* P *for each state in* S*.*

A module is thus modeled as a KMTS where no proposition takes the value $\perp$. In this context, module checking can then be formally defined as follows. Recall that $\mathcal{C}(M, s)$ denotes the set of completions of a KMTS/module M (see previous section): $\mathcal{C}(M, s) = \{(M', s') | s \preceq s' \text{ and } M' \text{ is a KS}\}$.

Definition 8 (Module Checking). *Given a module* (M, s) *and a temporal logic formula* ϕ*, we say that the module* (M, s) *satisfies* ϕ*, denoted* $(M, s) \models_r \phi$*, if* $\forall (M', s') \in \mathcal{C}(M, s) : (M', s') \models \phi$*. Checking whether* $(M, s) \models_r \phi$ *is called the* module-checking problem*.*

Formalized this way[3], it is clear that module checking (ModC) is related to generalized model checking (GMC). In the next section, we study this relationship.

3 Comparing Module Checking and GMC

We first consider the case of properties specified in linear temporal logic (LTL) or in universal branching temporal logics such as $\forall CTL$ and $\forall CTL^*$. In this case, [KV96] shows that module checking and model checking coincide. Indeed, by definition, an LTL or universal property holds of a model if and only if all paths in that model satisfy the given (path) property. In the case of a module, module

[3] The definition of module checking in [KV96] actually requires the transition relation of any (M', s') in $\mathcal{C}(M, s)$ to be total, which has the effect of preventing the environment from blocking the system by refusing to execute any transition; this assumption is eliminated here to simplify the presentation.

checking can be reduced to checking whether the property holds of the module when placed in the maximal environment consisting of all possible environment transitions, which in turns is equivalent to model checking. This implies that LTL module checking has the same complexity as LTL model checking: it is PSPACE-complete in the size of the formula and can be done in linear time in the size of the model (it is also known to be NLOGSPACE-complete in $|M|$). In contrast, it is shown in [BG00,GJ02] that GMC for LTL can be more precise but also more expensive than LTL model checking: GMC for LTL is EXPTIME-complete in the size of the formula [BG00] and can be done in quadratic time in the size of the model [GJ02].

In the case of properties specified in branching temporal logics (BTL), thus including existential quantification, [KV96] shows that the complexity of module checking is higher than that of model checking, and is in fact as hard as satisfiability.

The following theorem states that, in the BTL case, GMC and module checking are interreducible.

Theorem 3. *For any branching temporal logic L, the GMC problem and the module checking problem for L are interreducible in linear time and logarithmic space.*

Proof. Consider a formula ϕ of the logic L and a module (M, s) represented by a KMTS as defined in Section 2.2. We have $(M, s) \models_r \phi$ iff $\forall (M', s') \in \mathcal{C}(M, s) : (M', s') \models \phi$ (by Definition 8) iff $\neg\exists(M', s') \in \mathcal{C}(M, s) : (M', s') \not\models \phi$. For any branching temporal logic, the latter is equivalent to $\neg GMC((M, s), \neg\phi)$ (by Definition 6). Thus, we have $(M, s) \models_r \phi$ iff $GMC((M, s), \neg\phi)$ does not hold. Conversely, $GMC((M, s), \phi)$ holds iff $(M, s) \not\models_r \neg\phi$.

In the BTL setting, generalized model checking and module checking are thus very closely related. One could even say that they are the "*dual*" of each other, in the same sense as the quantifier $\forall$ is the dual of $\exists$ since $\forall = \neg\exists\neg$. The previous theorem also explains why both problems have the same complexity in the case of BTL. For instance, [KV96] pointed out that the complexity in the size of the formula of module checking for a BTL L is the same as that of satisfiability for L, while a similar result was proved independently for GMC in [BG00].

This close correspondence also makes it possible to transfer unmatched results obtained for one problem to the other. For instance, [KV96] only shows that module checking for CTL can be done in PTIME in the size of M. Using the results of [GJ02] concerning the complexity of GMC in the size of M, we immediately obtain that module checking for CTL can require quadratic running time in $|M|$, but that it can be solved in time linear in $|M|$ in the case of CTL *persistence* properties, i.e., properties recognizable by tree automata with a co-Büchi acceptance condition. Conversely, [GJ02] does not provide a lower bound on the complexity of GMC in the size of M for BTL persistence properties. Using the proof of Theorem 2 in [KV96] which states that the program complexity of CTL and CTL* module checking is PTIME-complete, we obtain that GMC for BTL persistence properties (and hence CTL and CTL* in general) is PTIME-hard. (Moreover, it is easy to show that this proof carries over to GMC for LTL, which is thus also PTIME-hard.)

In summary, generalized model checking and module checking are different, yet related, problems. The former is a framework for reasoning about partially-specified, i.e., abstract, systems, while the latter is a framework for reasoning about open systems. It is then natural to ask whether these two techniques could be combined into a framework for reasoning about abstract open systems. The rest of this paper investigates this idea.

4 Modeling and Reasoning about Abstract Open Systems

We now discuss how to model abstract open systems. Our goal is to define a modeling formalism for representing abstractions of programs implementing open systems. Such abstractions could then be automatically generated from source code by static analysis tools using abstraction techniques like predicate abstraction. We thus focus here on a semantic model to represent abstract open systems, not on a modeling language (with a specific syntax).

Combining the ideas of Section 2, an abstract open system can simply be represented as a KMTS with two distinct types of "unknowns": the third truth value $\perp$ can model loss of information due to abstraction, while *may*-transitions unmatched by *must*-transitions can model uncertainty due to environment transitions. Formally, an *abstract module*, or *3-valued module*, can be defined as follows.

Definition 9 (Abstract Module). *An* abstract module *is a KMTS, i.e., a tuple* $(S, P, \overset{must}{\longrightarrow}, \overset{may}{\longrightarrow}, L)$ *where* S *is a nonempty finite set of states,* P *is a finite set of atomic propositions,* $\overset{may}{\longrightarrow} \subseteq S \times S$ *and* $\overset{must}{\longrightarrow} \subseteq S \times S$ *are transition relations such that* $\overset{must}{\longrightarrow} \subseteq \overset{may}{\longrightarrow}$*, and* $L : S \times P \to \{true, \perp, false\}$ *is an* interpretation *that associates a truth value in* $\{true, \perp, false\}$ *with each atomic proposition in* P *for each state in* S*. A module is an abstract module such that* $\forall s \in S : \forall p \in P : L(s,p) \neq \perp$*. An abstract model is an abstract module such that* $\overset{must}{\longrightarrow} = \overset{may}{\longrightarrow}$*.*

It is worth emphasizing that modeling abstract open systems this way does not restrict expressiveness of either kind of "unknowns" since KMTSs have the same expressiveness as PKSs and MTSs [GJ03]. In other words, any modeling of incomplete information using $\perp$ can also be done using *may*-transitions instead, and vice versa. This implies that it does not matter which of $\perp$ or *may*-transitions are used to model abstraction or the environment. What matters is that the two sources of incomplete information are modeled differently, so that they can be distinguished in the model. Indeed, these two types of partial information are treated differently, as we will see later.

We now turn to the definition of *abstract module checking*, or *3-valued module checking*. For doing so, we first need to define the set of possible environments in which an abstract module can be executed. We formally represent this set as the set of completions of the abstract module with respect to a preorder $\preceq_{MTS}$, which we define as follows.

Definition 10 **($\preceq_{MTS}$).** *Let* $M_A = (S_A, P, \xrightarrow{must}_A, \xrightarrow{may}_A, L_A)$ *and* $M_C = (S_C, P, \xrightarrow{must}_C, \xrightarrow{may}_C, L_C)$ *be KMTSs. The preorder* $\preceq_{MTS}$ *is the greatest relation* $\mathcal{B} \subseteq S_A \times S_C$ *such that* $(s_a, s_c) \in \mathcal{B}$ *implies the following:*

- $\forall p \in P : L_A(s_a, p) = L_C(s_c, p)$,
- *if* $s_a \xrightarrow{must}_A s'_a$, *there is some* $s'_c \in S_C$ *such that* $s_c \xrightarrow{must}_C s'_c$ *and* $(s'_a, s'_c) \in \mathcal{B}$,
- *if* $s_c \xrightarrow{may}_C s'_c$, *there is some* $s'_a \in S_A$ *such that* $s_a \xrightarrow{may}_A s'_a$ *and* $(s'_a, s'_c) \in \mathcal{B}$.

The preorder $\preceq_{MTS}$ is similar to the completeness preorder on MTSs [LT88], but is defined on KMTSs and leaves truth values of atomic propositions unchanged. Definition 10 is also similar to Definition 3 defining $\preceq$ on KMTSs except for the first condition which prevents the refinement of unknown truth values. Let $\mathcal{C}_{MTS}((M, s)) = \{(M', s') | s \preceq_{MTS} s' \text{ and } M' \text{ is a PKS}\}$ denote the set of completions of a module (M, s) with respect to $\preceq_{MTS}$. We can now define abstract module checking as follows.

Definition 11 **(Abstract Module Checking).** *Given an abstract module* (M, s) *and a temporal logic formula* ϕ, *computing the value*

$$[(M, s) \models_r \phi] = \begin{cases} \textit{true} \ \ \textit{if} \ \forall (M', s') \in \mathcal{C}_{MTS}(M, s) : [(M', s') \models \phi] = \textit{true} \\ \textit{false} \ \textit{if} \ \exists (M', s') \in \mathcal{C}_{MTS}(M, s) : [(M', s') \models \phi] = \textit{false} \\ \perp \qquad \textit{otherwise} \end{cases}$$

is defined as the abstract module checking problem.

The previous definition generalizes the definition of module checking: when (M, s) is a concrete module (i.e., a module where no atomic proposition has the value $\perp$ in any state), Definition 11 coincides with Definition 8 defining module checking.

Abstract module checking defines a new 3-valued logic for reasoning about abstract modules: its syntax is as usual and its semantics is defined by $[(M, s) \models_r \phi]$. The following preorder $\preceq_{PKS}$ on KMTSs measures the degree of completeness of abstract modules with respect to the new semantics derived from abstract module checking.

Definition 12 **($\preceq_{PKS}$).** *Let* $M_A = (S_A, P, \xrightarrow{must}_A, \xrightarrow{may}_A, L_A)$ *and* $M_C = (S_C, P, \xrightarrow{must}_C, \xrightarrow{may}_C, L_C)$ *be KMTSs. The preorder* $\preceq_{PKS}$ *is the greatest relation* $\mathcal{B} \subseteq S_A \times S_C$ *such that* $(s_a, s_c) \in \mathcal{B}$ *implies the following:*

1. $\forall p \in P : L_A(s_a, p) \leq L_C(s_c, p)$,
2. *if* $s_a \xrightarrow{may}_A s'_a$, *there is some* $s'_c \in S_C$ *such that* $s_c \xrightarrow{may}_C s'_c$ *and* $(s'_a, s'_c) \in \mathcal{B}$,
3. *if* $s_a \xrightarrow{must}_A s'_a$, *there is some* $s'_c \in S_C$ *such that* $s_c \xrightarrow{must}_C s'_c$ *and* $(s'_a, s'_c) \in \mathcal{B}$,
4. *if* $s_c \xrightarrow{may}_C s'_c$, *there is some* $s'_a \in S_A$ *such that* $s_a \xrightarrow{may}_A s'_a$ *and* $(s'_a, s'_c) \in \mathcal{B}$,
5. *if* $s_c \xrightarrow{must}_C s'_c$, *there is some* $s'_a \in S_A$ *such that* $s_a \xrightarrow{must}_A s'_a$ *and* $(s'_a, s'_c) \in \mathcal{B}$.

The preorder $\preceq_{PKS}$ is similar to the completeness preorder on PKSs [BG99], but is defined on KMTSs and requires a bisimulation-like relation on both *may* and *must* transitions. The above definition extends Definition 3 defining $\preceq$ on KMTSs by also requiring conditions (2) and (5). An important property of $\preceq_{PKS}$ is the following.

Lemma 1. *Let* $M_A = (S_A, P, \xrightarrow{must}_A, \xrightarrow{may}_A, L_A)$ *and* $M_C = (S_C, P, \xrightarrow{must}_C, \xrightarrow{may}_C, L_C)$ *be KMTSs. Given any two states* $s_a \in S_A$ *and* $s_c \in S_C$*,* $s_a \preceq_{PKS} s_c$ *implies the two following properties:*

- $\forall (M'_A, s'_a) \in \mathcal{C}_{MTS}(M_A, s_a) : \exists (M'_C, s'_c) \in \mathcal{C}_{MTS}(M_C, s_c) : s'_a \preceq_{PKS} s'_c$*, and*
- $\forall (M'_C, s'_c) \in \mathcal{C}_{MTS}(M_C, s_c) : \exists (M'_A, s'_a) \in \mathcal{C}_{MTS}(M_A, s_a) : s'_a \preceq_{PKS} s'_c$.

Intuitively, the previous lemma states that, if $s_a \preceq_{PKS} s_c$, then the set of "possible environments" (i.e., $\mathcal{C}_{MTS}$) for s_a and s_c are equivalent: any environment of s_a is a possible environment of s_c and vice versa. This lemma is useful to prove our next theorem.

Theorem 4. *Let* $M_A = (S_A, P, \xrightarrow{must}_A, \xrightarrow{may}_A, L_A)$ *and* $M_C = (S_C, P, \xrightarrow{must}_C, \xrightarrow{may}_C, L_C)$ *be two abstract modules (KMTSs) such that* $s_a \in S_A$ *and* $s_c \in S_C$*, and let* Φ *be the set of all formulas of 3-valued PML. Then,*

$$s_a \preceq_{PKS} s_c \text{ iff } (\forall \phi \in \Phi : [(M_A, s_a) \models_r \phi] \leq [(M_C, s_c) \models_r \phi]).$$

Proof. Omitted due to space limitations.

The previous theorem thus states that 3-valued PML defined with the semantics $[(M, s) \models_r \phi]$ logically characterizes the preorder $\preceq_{PKS}$.[4] This implies that abstract module checking cannot distinguish abstract modules that are equivalent with respect to the preorder $\preceq_{PKS}$. Another important corollary of this theorem is that, by generating an abstraction (M_A, s_a) from (a static analysis of) a concrete module (M_C, s_c) such that $s_a \preceq_{PKS} s_c$, we can then *both prove and disprove arbitrary properties* of s_c by doing module checking of its abstraction s_a. How to automatically generate abstractions preserving $\preceq_{PKS}$ is discussed with an example later in Section 6.

In the case of LTL, abstract (3-valued) module checking, i.e., computing $[(M, s) \models_r \phi]$, reduces to abstract (3-valued) model checking, i.e., computing $[(M, s) \models \phi]$ with M viewed as a PKS where all the *may*-transitions are also *must*-transitions. In the BTL case, 3-valued module checking can be *approximated* by 3-valued model checking using the 3-valued semantics on KMTS defined in Definition 2. Indeed, $[(M, s) \models \phi] = true$ (respectively *false*) implies that $\forall (M', s') \in \mathcal{C}(M, s) : (M', s') \models \phi$ (resp., $(M', s') \not\models \phi$), which in turn implies that $[(M, s) \models_r \phi] = true$ (resp., *false*). We thus always have $[(M, s) \models \phi] \leq [(M, s) \models_r \phi]$. Since 3-valued model checking can be done at the same cost as traditional 2-valued model checking [BG00], computing $[(M, s) \models \phi]$ is less computationally expensive than computing $[(M, s) \models_r \phi]$ in the BTL case (see Theorem 3), and can thus be used as a cheaper but less precise partial algorithm for testing whether $[(M, s) \models_r \phi]$ is *true* or *false* in that case.

5 Generalized Module Checking

As in the case of 3-valued model checking, precision of 3-valued module checking can be improved by defining a 3-valued *thorough semantics*, denoted $[(M, s) \models_r$

[4] As in the 2-valued case, this result also holds for the propositional μ-calculus.

$\phi]_t$. Let $\mathcal{C}_{PKS}((M,s)) = \{(M',s') | s \preceq_{PKS} s'$ and M' is a module $\}$ denote the set of possible (concrete) modules for an abstract module (M,s).

Definition 13 (Thorough Abstract Module Checking). *Given an abstract module (M,s) and a temporal logic formula ϕ, computing the value*

$$[(M,s) \models_r \phi]_t = \begin{cases} \textit{true if } (M',s') \models_r \phi \textit{ for all } (M',s') \textit{ in } \mathcal{C}_{PKS}(M,s) \\ \textit{false if } (M',s') \not\models_r \phi \textit{ for all } (M',s') \textit{ in } \mathcal{C}_{PKS}(M,s) \\ \perp \quad \textit{otherwise} \end{cases}$$

is defined as the thorough abstract module checking problem.

The next theorem states that thorough abstract module checking is consistent with abstract module checking.

Theorem 5. *For any abstract module (M,s) and temporal logic formula ϕ, we have $[(M,s) \models_r \phi] \leq [(M,s) \models_r \phi]_t$.*

Proof. By definition, $[(M,s) \models_r \phi] = \textit{true}$ implies $[(M,s) \models_r \phi]_t = \textit{true}$. Moreover, $[(M,s) \models_r \phi] = \textit{false}$ implies $[(M,s) \models_r \phi]_t = \textit{false}$ since $\exists (M',s') \in \mathcal{C}_{MTS}(M,s) : \forall (M'',s'') \in \mathcal{C}_{PKS}(M',s') : (M'',s') \not\models \phi$ implies $\forall (M',s') \in \mathcal{C}_{PKS}(M,s) : \exists (M'',s'') \in \mathcal{C}_{MTS}(M',s') : (M'',s') \not\models \phi$.

Checking whether $[(M,s) \models_r \phi]_t = \textit{true}$ can be reduced to solving an instance of the generalized model checking problem for (M,s) and $\neg\phi$ with respect to the completeness preorder $\preceq$ on KMTSs, because of the two universal quantifications defining $[(M,s) \models_r \phi]_t = \textit{true}$. In contrast, checking whether $[(M,s) \models_r \phi]_t = \textit{false}$ involves an alternation between $\exists$ and $\forall$, and requires solving an instance of the following problem, which we call *generalized module checking*.

Definition 14 (Generalized Module-Checking Problem). *Given a state s of an abstract module M and a formula ϕ of a temporal logic L, does there exist a state s' of a module M' such that $s \preceq_{PKS} s'$ and $(M',s') \models_r \phi$?*

Clearly, generalized module checking (GModC) generalizes both module checking and generalized model checking for any (i.e., BTL or LTL) temporal logic since it includes both as particular sub-problems. Hence, generalized module checking is at least as hard as generalized model checking, which is itself harder than module checking in the LTL case and as hard as module checking in the BTL case (see Section 3). Is GModC harder than GMC? The next theorem shows that this is not the case by providing a reduction from GModC to GMC.

Theorem 6. *Given any (LTL or BTL) temporal logic L, any instance of the generalized module checking problem for L can be reduced in linear time and logarithmic space to an instance of the generalized model checking problem for L.*

Proof. (Sketch) Consider a formula ϕ of the logic L and an abstract module (M,s) represented by a KMTS $M = (S, P, \xrightarrow{must}, \xrightarrow{may}, L)$ as defined in Section 4. Without loss of generality, let us assume that ϕ is in positive form in which negation can apply only to atomic propositions. We first define a PKS $M' = (S', P', \rightarrow', L')$ that simulates exactly

```
program C() {
  x,y= 1,0;
  if (read(z) == 0) then {
    x,y = 2*f(x),f(y);
    x,y = 1,0;
  } else
    x,y = f(x),f(y);
}
```

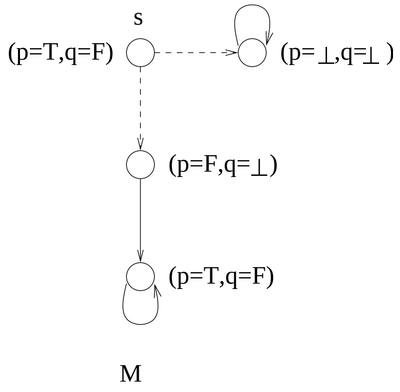

Fig. 3. Example of open program and abstract module

the *may* and *must* transitions of M with an additional proposition p_{must} as follows: $S' = S \times \{must, may\}$, $P' = P \cup \{p_{must}\}$, $\rightarrow' = \{((s,x),(s',x')) | (s,s') \in \xrightarrow{may}, x' = must$ if $(s,s') \in \xrightarrow{must}$ or $x' = may$ otherwise$\}$, $\forall (s,x) \in S' : \forall p \in P : L'((s,x),p) = L(s,p)$, and $L'((s,x),p_{must}) = true$ if $x = must$ or $L'((s,x),p_{must}) = false$ otherwise. Second, we translate the formula ϕ into the formula $T(\phi)$ defined with a set $P' = P \cup \{p_{must}\}$ of atomic propositions by applying recursively the following rewrite rules: for all $p \in P$, $T(p) = p$ and $T(\neg p) = \neg p$; $T(\phi_1 \wedge \phi_2) = T(\phi_1) \wedge T(\phi_2)$; $T(\phi_1 \vee \phi_2) = T(\phi_1) \vee T(\phi_2)$; $T(AX\phi) = AX(T(\phi))$; $T(EX\phi) = EX(p_{must} \wedge T(\phi))$. (Note that the above rules are for PML; in the case of the mu-calculus, fixpoint operators are left unchanged; a similar translation exists for LTL). Third, we show that $GModC((M,s),\phi) = true$ iff $GMC((M',(s,must)),T(\phi)) = true$.

To summarize, GModC has the same complexity as GMC. This implies the following, maybe surprising, corollary: even though computing $[(M,s) \models_r \phi]_t$ (thorough abstract module checking) can be more precise than computing $[(M,s) \models_r \phi]$ (abstract module checking), it has the same complexity in the BTL case.

6 Application

The practical motivation for this paper is to provide a framework for verifying properties of programs implementing open systems using automatic abstraction techniques such as predicate abstraction.

An example of program implementing an open system is shown on the left of Figure 3. In this program, x and y denote variables controlled by the program (system), f denotes some unknown function of the system, while z denotes a variable controlled by the environment. The notation "$x, y = 1, 0$" means that variables x and y are simultaneously assigned to values 1 and 0, respectively.

Imagine we are interested in checking the property "(eventually y is odd) and (at any time, x is odd or y is even)". From this property, we define two predicates p : "is x odd?" and q : "is y odd?". Given these two predicates, the property can be represented by the LTL formula $\phi = \Diamond q \wedge \Box(p \vee \neg q)$.

Logic	MC	ModC	GModC
Propositional Logic	Linear	Linear	NP-complete
PML	Linear	PSPACE-complete	PSPACE-complete
CTL	Linear	EXPTIME-complete	EXPTIME-complete
μ-calculus	NP∩co-NP	EXPTIME-complete	EXPTIME-complete
LTL	PSPACE-complete	PSPACE-complete	EXPTIME-complete

Fig. 4. Complexity in the size of the formula of model checking (MC), module checking (ModC), and generalized module checking (GModC).

Using predicate abstraction techniques and predicates p and q, one can automatically compute an abstract module for this program that satisfies the preorder $\preceq_{PKS}$ and thus Theorem 4, by construction. An example of such an abstract module (M, s) for this program is shown on the right of Figure 3. The truth values of atomic propositions p and q is defined in each state as indicated in the figure. Dotted transitions indicate *may*-transitions unmatched by *must*-transitions and represent transitions controlled by the environment. Note how the unknown function f is modeled using $\perp$, while uncertainty due to the environment is modeled using *may*-transitions.

In this example, we have $[(M, s) \models_r \phi] = \perp$, while $[(M, s) \models_r \phi]_t = \mathit{false}$. In other words, using the thorough interpretation and generalized module checking is needed to obtain a definite answer in this case. Indeed, this result is obtained by proving that, for all possible completions (M', s') such that $(M, s) \preceq_{PKS} (M', s')$, there exists an environment of (M', s') (namely the one which forces the system down the leftmost path) where ϕ is violated. We are not aware of any decision procedure more efficient than generalized module checking to prove automatically that the open program C of Figure 3 violates property ϕ.[5]

Finally note how GModC differ from GMC in the presence of *may*-transitions: while $GModC((M, s), \phi) = \mathit{false}$, we have $GMC((M, s), \phi) = \perp$.

7 Conclusions

We have presented a framework for representing and reasoning about abstract open systems. Our framework is designed to be used in conjunction with automatic abstraction tools for generating abstractions from static program analysis. We identified the preorder $\preceq_{PKS}$ as the one being logically characterized by the 3-valued semantics derived from the definition of module checking of [KV96]. Any abstraction that preserves $\preceq_{PKS}$ can then be used to both prove and disprove arbitrary temporal properties of the concrete program. We introduced new variants of the module checking problem suitable for reasoning about such abstractions, namely abstract, thorough abstract and generalized module checking. We also studied the complexity of these problems. The complexity of generalized

[5] Note that, since GMC for LTL can be reduced to SAT for CTL* (using Theorem 23 of [BG00]), the above verification results could be obtained using a SAT solver for CTL*, but at a much higher complexity both in $|M|$ and $|\phi|$.

module checking is summarized in the last column of Figure 4. The precision of generalized module checking was illustrated with an example of program and property that is beyond the scope of current abstraction-based verification tools.

Note that generating an abstraction (M_A, s_a) preserving $\preceq_{PKS}$ assumes that it is possible to determine which transitions of the concrete module (M_C, s_c) are controlled by the system and which transitions are controlled by the environment. Our framework does not currently support a way to safely approximate (M_C, s_c) in the case this information is unknown (i.e., cannot be determined exactly by a static analysis). Previous work on alternating refinement relations [AHKV98,AH01] provides a way to conservatively abstract a game (like the one played between a system and its environment) while preserving the existence of a winning strategy for one of the players. However, such game abstractions do not preserve the existence of a winning strategy for the other player: they are *conservative* in the same sense as a simulation relation defines a conservative abstraction of a system, which can be used for proving universal properties but not refuting these. An interesting topic for future work is therefore to study how to combine our framework with techniques for abstracting games with the goal of designing "3-valued game abstractions" that would preserve winning strategies for both players while allowing abstraction. Such a way, one could extend the framework developed in this paper to allow sound approximations of the partitioning between system and environment transitions while preserving the ability to prove and disprove any temporal property of the interaction of the system with its environment.

References

[AH01] L. de Alfaro and T. Henzinger. Interface Automata. In *Proceedings of the 9th ACM Symposium on the Foundations of Software Engineering (FSE'01)*, 2001.

[AHKV98] R. Alur, T. Henzinger, O. Kupferman, and M. Vardi. Alternating Refinement Relations. In *Proc. 10th Conference on Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 163–178. Springer-Verlag, 1998.

[BG99] G. Bruns and P. Godefroid. Model Checking Partial State Spaces with 3-Valued Temporal Logics. In *Proceedings of the 11th Conference on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 274–287, Trento, July 1999. Springer-Verlag.

[BG00] G. Bruns and P. Godefroid. Generalized Model Checking: Reasoning about Partial State Spaces. In *Proceedings of CONCUR'2000 (11th International Conference on Concurrency Theory)*, volume 1877 of *Lecture Notes in Computer Science*, pages 168–182, University Park, August 2000. Springer-Verlag.

[BR01] T. Ball and S. Rajamani. The SLAM Toolkit. In *Proceedings of CAV'2001 (13th Conference on Computer Aided Verification)*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264, Paris, July 2001. Springer-Verlag.

[CDH$^+$00] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting Finite-State Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, 2000.

[Dam96] D. Dams. *Abstract interpretation and partition refinement for model checking*. PhD thesis, Technische Universiteit Eindhoven, The Netherlands, 1996.

[DD01] S. Das and D. L. Dill. Successive Approximation of Abstract Transition Relations. In *Proceedings of LICS'2001 (16th IEEE Symposium on Logic in Computer Science)*, pages 51–58, Boston, June 2001.

[Eme90] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier/MIT Press, Amsterdam/Cambridge, 1990.

[GHJ01] P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based Model Checking using Modal Transition Systems. In *Proceedings of CONCUR'2001 (12th International Conference on Concurrency Theory)*, volume 2154 of *Lecture Notes in Computer Science*, pages 426–440, Aalborg, August 2001. Springer-Verlag.

[GJ02] P. Godefroid and R. Jagadeesan. Automatic Abstraction Using Generalized Model Checking. In *Proceedings of CAV'2002 (14th Conference on Computer Aided Verification)*, volume 2404 of *Lecture Notes in Computer Science*, pages 137–150, Copenhagen, July 2002. Springer-Verlag.

[GJ03] P. Godefroid and R. Jagadeesan. On the Expressiveness of 3-Valued Models. In *Proceedings of VMCAI'2003 (4th Conference on Verification, Model Checking and Abstract Interpretation)*, volume 2575 of *Lecture Notes in Computer Science*, pages 206–222, New York, January 2003. Springer-Verlag.

[GS97] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, June 1997. Springer-Verlag.

[HJMS02] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 58–70, Portland, January 2002.

[HJS01] M. Huth, R. Jagadeesan, and D. Schmidt. Modal Transition Systems: a Foundation for Three-Valued Program Analysis. In *Proceedings of the European Symposium on Programming (ESOP'2001)*, volume 2028 of *Lecture Notes in Computer Science*. Springer-Verlag, April 2001.

[Kle87] S. C. Kleene. *Introduction to Metamathematics*. North Holland, 1987.

[Koz83] D. Kozen. Results on the Propositional Mu-Calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[KV96] O. Kupferman and M. Vardi. Module Checking. In *Proc. 8th Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 75–86, New Brunswick, August 1996. Springer-Verlag.

[LT88] K. G. Larsen and B. Thomsen. A Modal Process Logic. In *Proceedings of Third Annual Symposium on Logic in Computer Science*, pages 203–210. IEEE Computer Society Press, 1988.

[VHBP00] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proceedings of ASE'2000 (15th International Conference on Automated Software Engineering)*, Grenoble, September 2000.