Combinations of Model Checking and Theorem Proving *

Tomás E. Uribe

Computer Science Department,
Stanford University, Stanford, CA. 94305-9045
uribe@cs.stanford.edu

Abstract. The two main approaches to the formal verification of reactive systems are based, respectively, on model checking (algorithmic verification) and theorem proving (deductive verification). These two approaches have complementary strengths and weaknesses, and their combination promises to enhance the capabilities of each. This paper surveys a number of methods for doing so. As is often the case, the combinations can be classified according to how tightly the different components are integrated, their range of application, and their degree of automation.

1 Introduction

Formal verification is the task of proving mathematical properties of mathematical models of systems. *Reactive systems* are a general model for systems that have an ongoing interaction with their environment. Such systems do not necessarily terminate, so their computations are modeled as infinite sequences of states and their properties specified using *temporal logic* [48]. The verification problem is that of determining if a given reactive system satisfies a given temporal property.

Reactive systems cover a wide range of hardware and software artifacts, including concurrent and distributed systems, which can be particularly difficult to design and debug. (Sequential programs and programs that terminate are a special case of this general model.) Reactive systems can be classified according to their number of possible states. For *finite-state systems*, the states are given by a finite number of finite-state variables. This includes, in particular, hardware systems with a fixed number of components. *Infinite-state systems* feature variables with unbounded domains, typically found in software systems, such as integers, lists, trees, and other datatypes. (Note that in both cases the computations are infinite sequences of states.)

The verification of temporal properties for finite-state systems is decidable: *model checking* algorithms can automatically decide if a temporal property holds

* This research was supported in part by the National Science Foundation under grant CCR-98-04100, by the Defense Advanced Research Projects Agency under contract NAG2-892, by the Army under grants DAAH04-96-1-0122 and DAAG55-98-1-0471, and by the Army under contract DABT63-96-C-0096 (DARPA).

for a finite-state system. Furthermore, they can produce a *counterexample computation* when the property does not hold, which can be very valuable in determining the corresponding error in the system being verified or in its specification. However, model checking suffers from the *state explosion problem*, where the number of states to be explored grows exponentially in the size of the system description, particularly as the number of concurrent processes grows.

The verification problem for general infinite-state systems is undecidable, and finite-state model checking techniques are not directly applicable. (We will see, however, that particular decidable classes of infinite-state systems can be model checked using specialized tools.) First-order logic is a convenient language for expressing relationships over the unbounded data structures that make systems infinite-state. Therefore, it is natural to use theorem proving tools to reason formally about such data and relationships. *Deductive verification*, based on general-purpose theorem proving, applies to a wide class of finite- and infinite-state reactive systems. It provides *relatively complete* proof systems, which can prove any temporal property that indeed holds over the given system, provided the theorem proving tools used are expressive and powerful enough [42]. Unfortunately, if the property fails to hold, deductive methods normally do not give much useful feedback, and the user must try to determine whether the fault lies with the system and property being verified or with the failed proof.

	Algorithmic Methods	Deductive Methods	Combination
Automatic / decidable?	yes	no	sometimes
Generates counterexamples?	yes	no	sometimes
Handles general infinite-state systems?	no	yes	yes

Table 1. Deductive vs. algorithmic verification

The strengths and weaknesses of model checking and deductive verification, as discussed above, are summarized in Table 1. Given their complementary nature, we would like to combine the two methods in such a way that the desirable features of each are retained, while minimizing their shortcomings. Thus, combinations of model checking and theorem proving usually aim to achieve one or more of the following goals:

- More automatic (or, less interactive) verification of infinite-state systems for which model checking cannot be directly applied.
- Verifying finite-state systems that are larger than what stand-alone model checkers can handle.
- Conversely, verifying infinite-state systems with control structures that are too large or complex to check with purely deductive means.

– Generating counterexamples (that is, falsifying properties) for infinite-state systems for which classical deductive methods can only produce proofs.
– Formalizing, and sometimes automating, verification steps that were previously done in an informal or manual way.

Outline: Section 2 presents background material, including the basic model for reactive systems. We then describe the main components of verification systems based on model checking (Section 3) and theorem proving (Section 4). In Section 5 we describe abstraction and invariant generation, which are important links between the two. This defines the basic components whose combination we discuss in the remaining sections.

In Section 6, we describe combination methods that use the components as "black boxes," that is, which do not require the modification of their internal workings. In Section 7, we present combination approaches that require a tighter integration, resulting in new "hybrid" formalisms.

References to combination methods and related work appear throughout the paper, which tries to present a general overview of the subject. However, since much of the work on formal verification during the last decade is related to the subject of this paper, the bibliography can only describe a subset of the work in the field. Furthermore, this paper is based on [58], which presents the author's own view on the subject, and thus carries all the biases which that particular proposal may have. For this, we apologize in advance.

2 Preliminaries

2.1 Kripke Structures and Fair Transition Systems

A *reactive system* $\mathcal{S} : \langle \Sigma, \Theta, R \rangle$ is given by a set of *states* Σ, a set of *initial states* $\Theta \subseteq \Sigma$, and a *transition relation* $R \subseteq \Sigma \times \Sigma$. If $\langle s_1, s_2 \rangle \in R$, the system can move from s_1 to s_2. A system $\mathcal{S}$ can be identified with the corresponding *Kripke structure*, or *state-space*. This is the directed graph whose vertices are the elements of Σ and whose edges connect each state to its successor states.

If Σ is finite, $\mathcal{S}$ is said to be *finite-state*. Describing large or infinite state-spaces explicitly is not feasible. Therefore, the state-space is *implicitly* represented in some other form: a hardware description, a program, or an ω-automaton.

Fair transition systems are a convenient formalism for specifying both finite- and infinite-state reactive systems [44]. They are "low-level" in the sense that many other formalisms can be translated or compiled into them.

The state-space of the system is determined by a set of *system variables* $\mathcal{V}$, where each variable has a given domain (e.g., booleans, integers, recursive datatypes, or reals). The representation relies on an *assertion language*, usually based on first-order logic, to represent sets of states.

Definition 1 (Assertion). *A first-order formula whose free variables are a subset of $\mathcal{V}$ is an* assertion*, and represents the set of states that satisfy it. For an assertion ϕ, we say that $s \in \Sigma$ is a* ϕ-state *if $s \models \phi$, that is, ϕ holds given the values of $\mathcal{V}$ at the state s.*

In practice, an assertion language other than first-order logic can be used. The basic requirements are the ability to represent predicates and relations, and automated support for validity and satisfiability checking (which need not be complete). Examples of other suitable assertion languages include *ordered binary decision diagrams* (OBDD's) [12] and their variants, for finite-state systems (see Section 3), and the abstract domains used in invariant generation (see Section 5.1).

The initial condition is now expressed as an assertion, characterizing the set of possible system states at the start of a computation. The transition relation R is described as a set of *transitions* $\mathcal{T}$. Each transition $\tau \in \mathcal{T}$ is described by its *transition relation* $\tau(\mathcal{V}, \mathcal{V}')$, a first-order formula over the set of system variables $\mathcal{V}$ and a *primed set* $\mathcal{V}'$, indicating their values at the next state.

Definition 2 (Transition system). *A* transition system $\mathcal{S} : \langle \mathcal{V}, \Theta, \mathcal{T} \rangle$ *is given by a set of* system variables $\mathcal{V}$*, an* initial condition Θ*, expressed as an assertion over* $\mathcal{V}$*, and a set of* transitions $\mathcal{T}$*, each an assertion over* $(\mathcal{V}, \mathcal{V}')$*.*

In the associated Kripke structure, each state in the state-space Σ *is a possible valuation of* $\mathcal{V}$*. We write* $s \models \varphi$ *if assertion* φ *holds at state* s*, and say that* s *is a* φ-state*. A state* s *is initial if* $s \models \Theta$*. There is an edge from* s_1 *to* s_2 *if* $\langle s_1, s_2 \rangle$ *satisfy* τ *for some* $\tau \in \mathcal{T}$*.*

The Kripke structure is also called the *state transition graph* for $\mathcal{S}$. Note that if the domain of a system variable is infinite, the state-space is infinite as well, even though the *reachable state-space*, the set of states that can be reached from Θ, may be finite. We can thus distinguish between *syntactically* finite-state systems and *semantically* finite-state ones. Establishing whether a system is semantically finite-state may not be immediately obvious (and is in fact undecidable), so we prefer the syntactic characterization.

The global transition relation is the disjunction of the individual transition relations: $R(s_1, s_2)$ iff $\tau(s_1, s_2)$ holds for some $\tau \in \mathcal{T}$. For assertions ϕ and ψ and transition τ, we write

$$\{\phi\}\,\tau\,\{\psi\} \stackrel{\text{def}}{=} (\phi(V) \wedge \tau(\mathcal{V}, \mathcal{V}')) \rightarrow \psi(\mathcal{V}') \ .$$

This is the *verification condition* that states that every τ-successor of a ϕ-state must be a ψ-state.

A *run* of $\mathcal{S}$ is an infinite path through the Kripke structure that starts at an initial state, i.e., a sequence of states $(s_0, s_1, \ldots)$ where $s_0 \in \Theta$ and $R(s_i, s_{i+1})$ for all $i \geq 0$. If $\tau(s_i, s_{i+1})$ holds, then we say that transition τ is *taken* at s_i. A transition is *enabled* if it can be taken at a given state.

Fairness and Computations: Our computational model represents concurrency by *interleaving*: at each step of a computation, a single action or transition is executed [44]. The transitions from different processes are combined in all possible ways to form the set of computations of the system. *Fairness* expresses the constraint that certain actions cannot be forever prevented from occurring—that is, that they do have a fair chance of being taken. Describing R as a set of transition relations is convenient for modeling fairness:

Definition 3 (Fair transition system). *A* fair transition system (FTS) *is one where each transition is marked as* just *or* compassionate*. A just (or* weakly fair*) transition cannot be continually enabled without ever being taken; a compassionate (or* strongly fair*) transition cannot be enabled infinitely often but taken only finitely many times. A* computation *is a run that satisfies these fairness requirements (if any exist).*

To ensure that R is total on Σ, so that sequences of states can always be extended to infinite sequences, we assume an *idling transition*, with transition relation $\mathcal{V} = \mathcal{V}'$. The set of all computations of a system $\mathcal{S}$ is written $\mathcal{L}(\mathcal{S})$, a language of infinite strings whose alphabet is the set of states of $\mathcal{S}$.

2.2 Temporal Logic

We use temporal logic to specify properties of reactive systems [48, 44]. *Linear-time temporal logic* (LTL) describes sets of sequences of states, and can thus capture *universal properties* of systems, which are meant to hold for all computations. However, LTL ignores the branching structure of the system's state-space, and thus cannot express *existential properties*, which assert the existence of particular kinds of computations. The logic CTL* includes both the branching-time *computation tree logic* (CTL) and LTL, and is strictly more expressive than both. Due to space limitations, we refer the reader to [44, 16] for the corresponding definitions.

A temporal formula is $\mathcal{S}$*-valid* if it holds at all the initial states of the kripke structures of $\mathcal{S}$, considering only the fair runs of the system. For LTL, a convenient definition of $\mathcal{S}$-validity can be formulated in terms of the set $\mathcal{L}(\varphi)$ of *models* of φ, which is the set of all infinite sequences that satisfy φ:

Proposition 1 (LTL system validity). *$\mathcal{S} \models \varphi$ for an LTL formula φ if and only if all the computations of $\mathcal{S}$ are models of φ, that is, $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\varphi)$.*

3 Finite-State Model Checking

Given a reactive system $\mathcal{S}$ and a temporal property φ, the verification problem is to establish whether $\mathcal{S} \models \varphi$. For finite-state systems, *model checking* [14, 50] answers this question by a systematic exploration of the state-space of $\mathcal{S}$, based on the observation that checking that a formula is true in a particular model is generally easier than checking that it is true in all models; the Kripke structure of $\mathcal{S}$ is the particular model in question, and φ is the formula being checked.

Model checkers, such as SPIN [32], SMV [45], Murφ [24], and those in STeP [7], take as input what is essentially a finite-state fair transition system and a temporal formula in some subset of CTL*, and automatically check that the system satisfies the property. (See [16] for a recent and comprehensive introduction to model checking.)

The complexity of model checking depends on the size of the formula being checked (linear for CTL and exponential for LTL and CTL*) and the size of the

system state-space (linear for all three logics). While the temporal formulas of interest are usually small, the size of the state-space can grow exponentially in the size of its description, e.g., as a circuit, program, or fair transition system. This is known as the *state explosion problem*, which limits the practical application of model checking tools.

Symbolic Model Checking: Model checking techniques that construct and explore states of the system, one at a time, are called *explicit-state*. In contrast, *symbolic model checking* combats the state-explosion problem by using specialized formalisms to represent sets of states. Ordered Binary Decision Diagrams (OBDD's) [12] are an efficient data structure for representing boolean functions and relations. They can be used to represent the transition relation of finite-state systems, as well as subsets of the systems' state-space. The efficient algorithms for manipulating OBDD's can be used to compute predicate transformations, such as pre- and post-condition operations, over the transition relation and large, symbolically represented sets of states [45].

Symbolic model checking extends the size of finite-state systems that can be analyzed, and is particularly successful for hardware systems. However, it is still restricted to finite-state systems of fixed size. The size of the OBDD representation can grow exponentially in the number of boolean variables, leading to what can be called the *OBDD explosion problem*, where the model checker runs out of memory before the user runs out of time. In these cases, efficient explicit-state model checkers such as Murφ [24] are preferred.

We arrive now at our first general combination scheme, which we can informally call "SMC(X);" here, symbolic model checking is parameterized by the constraint language used to describe and manipulate sets of states. For instance, extensions of OBDD's are used to move from bit-level to word-level representations [15]; more expressive assertions are used in [36]. Similarly, finite-state *bounded model checking* [5] abandons BDD's and relies instead on the "black-box" use of a propositional validity checker. This method can find counterexamples in cases for which BDD-based symbolic model checking fails.

4 Deductive Verification

For assertions φ and p,

I1. $\Theta \rightarrow \varphi$

I2. $\{\varphi\}\ \tau\ \{\varphi\}$ for each $\tau \in \mathcal{T}$

I3. $\varphi \rightarrow p$

$\mathcal{S} \models \Box p$

Fig. 1. General invariance rule G-INV

Figure 1 presents the *general invariance rule*, G-INV, which proves the $\mathcal{S}$-validity of formulas of the form $\Box p$ for an assertion p [44]. The premises of the rule are first-order verification conditions. If they are valid, the temporal conclusion must hold for the system $\mathcal{S}$.

An assertion is *inductive* if it is preserved by all the system transitions and holds at all initial states. The invariance rule relies on finding an inductive auxiliary assertion φ that *strengthens* p, that is, φ implies p.

The soundness of the invariance rule is clear: if ϕ holds initially and is preserved by all transitions, it will hold for every reachable state of $\mathcal{S}$. If p is implied by φ, then p will also hold for all reachable states. Rule G-INV is also *relatively complete*: if p is an invariant of $\mathcal{S}$, then the strengthened assertion φ always exists [44]. Assuming that we have a complete system for proving valid assertions, then we can prove the $\mathcal{S}$-validity of any $\mathcal{S}$-valid temporal property. Note, however, that proving invariants is undecidable for general infinite-state systems, and finding a suitable φ can be non-trivial.

Other verification rules can be used to verify different classes of temporal formulas, ranging from safety to progress properties. Together, these rules are also relatively complete and yield a direct proof of any $\mathcal{S}$-valid temporal property [42]. However, they may require substantial user guidance to succeed, and do not produce counterexample computations when the property fails.

When we require that a premise of a rule or a verification condition be valid, we mean for it to be $\mathcal{S}$-valid; therefore, verification conditions can be established with respect to invariants of the system, which can be previously proved or generated automatically. In general, axioms and lemmas about the system or the domain of computation can also be used. As we will see, this simple observation is an important basis for combined verification techniques.

4.1 Decision Procedures (and their Combination)

Most verification conditions refer to particular theories that describe the domain of computation, such as linear arithmetic, lists, arrays and other data types. *Decision procedures* provide specialized and efficient validity checking for particular theories. For instance, equality need not be axiomatized, but its consequences can be efficiently derived by congruence closure. Similarly, specialized methods can efficiently reason about integers, lists, bit vectors and other datatypes frequently used in system descriptions.

Thus, using appropriate decision procedures can be understood as specializing the assertion language to the particular domains of computation used by the system being verified. However, this domain of computation is often a *mixed* one, resulting in verification conditions that do not fall in the decidable range of any single decision procedure. Thus, combination problems in automated deduction are highly relevant to the verification task. The challenge is to integrate existing decision procedures for the different decidable fragments and their combination into theorem-proving methods that can be effectively used in verification. See Bjørner [6] for examples of such combinations.

4.2 Validity Checking and First-Order Reasoning

Decision procedures usually operate only at the *ground* level, where no quantification is allowed. This is sufficient in many cases; for instance, the Stanford Validity Checker (SVC) [2] is an efficient checker specialized to handle large ground formulas, including uninterpreted function symbols, that occur in hardware verification.

However, program features such as parameterization and the *tick* transition in real-time systems introduce quantifiers in verification conditions. Fortunately, the required quantifier instantiations are often "obvious," and use instances that can be provided by the decision procedures themselves. Both the Extended Static Checking system (ESC) [23] and the Stanford Temporal Prover (STeP) [6, 7, 9] feature integrations of first-order reasoning and decision procedures that can automatically prove many verification conditions that would otherwise require the use of an interactive prover.

Finally, we note that automatic proof methods for first-order logic and certain specialized theories are necessarily incomplete, not guaranteed to terminate, or both. In practice, automatic verification tools abandon completeness and focus instead on quickly deciding particular classes of problems that are both tractable and likely to appear when verifying realistic systems.

To achieve completeness, interactive theorem proving must be used, and the above techniques must be integrated into interactive theorem proving frameworks, as done, for instance, in STeP and PVS [47]. Combining decision procedures, validity checkers and theorem provers, and extending them to ever-more expressive assertion languages, are ongoing challenges.

5 Abstraction and Invariant Generation

Abstraction is a fundamental and widely-used verification technique. Together with modularity, it is the basis of most combinations of model checking and deductive verification [37], as we will see in Sections 6 and 7.

Abstraction reduces the verification of a property φ over a *concrete system* $\mathcal{S}$, to checking a related property $\varphi^{\mathcal{A}}$ over a simpler *abstract system* $\mathcal{A}$. It allows the verification of infinite-state systems by constructing abstract systems that can be model checked. It can also mitigate the state explosion problem in the finite-state case, by constructing abstract systems with a more manageable state-space.

Abstract interpretation [18] provides a general framework and a methodology for automatically producing abstract systems given a choice of the abstract domain $\Sigma_{\mathcal{A}}$. The goal is to construct abstractions whose state-space can be represented, manipulated and approximated in ways that could not be directly applied to the original system. Originally designed for deriving safety properties in static program analysis, this framework has recently been extended to include reactive systems and general temporal logic, e.g., [39, 20].

One simple but useful instance of this framework is based on *Galois connections*. Two functions, $\alpha : 2^{\Sigma_C} \mapsto \Sigma_{\mathcal{A}}$ and $\gamma : \Sigma_{\mathcal{A}} \mapsto 2^{\Sigma_C}$, connect the lattice

of sets of concrete states and an abstract domain $\Sigma_{\mathcal{A}}$, which we assume to be a complete boolean lattice. The *abstraction function* α maps each set of concrete states to an abstract state that represents it. The *concretization function* $\gamma : \Sigma_{\mathcal{A}} \to 2^{\Sigma_C}$ maps each abstract state to the set of concrete states that it represents.

In general, the abstract and concrete systems are described using different assertion languages, specialized to the respective domains of computation (see Section 4.1). We say that an assertion or temporal formula is *abstract* or *concrete* depending on which language it belongs to.

For an abstract sequence of states $\pi^{\mathcal{A}} : a_0, a_1, \ldots$, its concretization $\gamma(\pi^{\mathcal{A}})$ is the set of sequences $\{s_0, s_1, \ldots \mid s_i \in \gamma(a_i) \text{ for all } i \geq 0\}$. The *abstraction* $\alpha(S)$ of a set of concrete states S is

$$\alpha(S) \stackrel{\text{def}}{=} \bigwedge^{\mathcal{A}} \{a \in \Sigma_{\mathcal{A}} \mid S \subseteq \gamma(a)\} \quad .$$

This is the smallest point in the abstract domain that represents all the elements of S. In practice, it is enough to soundly over-approximate such sets [17].

Definition 4 (Abstraction and concretization of CTL* properties). *For a concrete CTL* temporal property φ, its* abstraction $\alpha^t(\varphi)$ *is obtained by replacing each assertion f in φ by an abstract assertion $\alpha^-(f)$ that characterizes the set of abstract states $\alpha^-(f) : \bigvee^{\mathcal{A}} \{a \in \Sigma_{\mathcal{A}} \mid \gamma(a) \subseteq f\}$. Conversely, given an abstract temporal property $\varphi^{\mathcal{A}}$, its* concretization $\gamma(\varphi^{\mathcal{A}})$ *is obtained by replacing each atom a in $\varphi^{\mathcal{A}}$ by an assertion that characterizes $\gamma(a)$.*

We can now formally define *weak property preservation*, where properties of the abstract system can be transferred over to the concrete one:

Definition 5 (Weak preservation). *$\mathcal{A}$ is a* weakly preserving abstraction *of $\mathcal{S}$ relative to a class of concrete temporal properties $\mathcal{P}$ if for any property $\varphi \in \mathcal{P}$,*

1. *If $\mathcal{A} \models \alpha^t(\varphi)$ then $\mathcal{S} \models \varphi$. Or, equivalently:*
2. *For any abstract temporal property $\varphi^{\mathcal{A}}$ where $\gamma(\varphi^{\mathcal{A}}) \in \mathcal{P}$, if $\mathcal{A} \models \varphi^{\mathcal{A}}$ then $\mathcal{S} \models \gamma(\varphi^{\mathcal{A}})$.*

Note that the failure of $\varphi^{\mathcal{A}}$ for $\mathcal{A}$ does *not* imply the failure of ϕ for $\mathcal{S}$. *Strong* property preservation ensures the transfer of properties from $\mathcal{S}$ to $\mathcal{A}$ as well; however, it severely limits the degree of abstraction that can be performed, so weak preservation is more often used.

There are two general applications of this framework: In the first, we can transfer any property of a correct abstraction over to the concrete system, independently of any particular concrete property to be proved. Thus, this is called the *bottom-up* approach. In the second, given a concrete property to be proved, we try to find an abstract system that satisfies the corresponding abstract property. The abstraction is now tailored to a specific property, so this is called the *top-down* approach. We return to this classification in Section 7.

The question now is how to determine that a given abstract system is indeed a sound abstraction of $\mathcal{S}$. The following theorem expresses sufficient conditions for this, establishing a *simulation relation* between the two systems:

Proposition 2 (Weakly preserving ∀CTL* abstraction). *Consider systems* $\mathcal{S} : \langle \Sigma_{\mathcal{C}}, \Theta_{\mathcal{C}}, R_{\mathcal{C}} \rangle$, *and* $\mathcal{A} : \langle \Sigma_{\mathcal{A}}, \Theta_{\mathcal{A}}, R_{\mathcal{A}} \rangle$ *such that for a concretization function* $\gamma : \Sigma_{\mathcal{A}} \to 2^{\Sigma_{\mathcal{C}}}$ *the following hold:*

1. INITIALITY*:* $\Theta_{\mathcal{C}} \subseteq \gamma(\Theta_{\mathcal{A}})$.
2. CONSECUTION*: If* $R_{\mathcal{C}}(s_1, s_2)$ *for some* $s_1 \in \gamma(a_1)$ *and* $s_2 \in \gamma(a_2)$*, then* $R_{\mathcal{A}}(a_1, a_2)$.

Then $\mathcal{A}$ *is a weakly preserving abstraction of* $\mathcal{S}$ *for* $\forall CTL^*$.

Informally, the conditions ensure that $\mathcal{A}$ can do everything that $\mathcal{S}$ does, and perhaps some more. Note that this proposition is limited to universal properties and does not consider fairness. The framework can, however, be extended to include existential properties and take fairness into account—see [20, 58].

5.1 Invariant Generation

Once established, invariants can be very useful in all forms of deductive and algorithmic verification, as we will see in Section 6. Given a sound ∀CTL$*$-preserving abstraction $\mathcal{A}$ of $\mathcal{S}$, if $\Box\varphi^{\mathcal{A}}$ is an invariant of $\mathcal{A}$, then $\Box\gamma(\varphi^{\mathcal{A}})$ is an invariant of $\mathcal{S}$. Thus, in particular, the concretization of the reachable state-space of the abstract system is an invariant of the concrete one; furthermore, any *over-approximation* of this state-space is also an invariant. This is the basis for most automatic invariant generation methods based on abstract interpretation, which perform the following steps:

1. Construct an abstract system $\mathcal{A}$ over some suitable domain of computation;
2. Compute an over-approximation of the state-space of $\mathcal{A}$, expressed as an abstract assertion or a set of constraints;
3. Concretize this abstract assertion, to produce an invariant over the concrete domain.

Widening, a classic abstract interpretation technique, can be used to speed up or ensure convergence of the abstract fixpoint operations, by performing safe over-approximations. This is the approach taken in [8] to automatically generate invariants for general infinite-state systems. The abstract domains used include set constraints, linear arithmetic, and polyhedra. These methods are implemented as part of STeP [7].

6 Loosely Coupled Combinations

The preceding sections have presented model checking, deductive verification, and abstraction, including the special case of invariant generation. Numerous stand-alone tools that perform these tasks are available. This section surveys combination methods that can use these separate components as "black boxes," while Section 7 describes methods that require a closer integration.

6.1 Modularity and Abstraction

Given the complementary nature of model checking and theorem proving, as discussed in Section 1, a natural combination is to decompose the verification problem into sub-problems that fall within the range of application of each of the methods. Since theorem proving is more expressive and model checking is more automatic, the most reasonable approach is to use deductive tools to reduce the main verification problem to subgoals that can be model checked.

Abstraction is one of the two main methods for doing this: abstractions are deductively justified and algorithmically model checked. The other is *modular verification*. Here, the system being verified is split into its constituent components, which are analyzed independently, provided assumptions on their environment. The properties of the individual components are then combined, usually following some form of *assumption-guarantee reasoning*, to derive properties of the complete system. This avoids the state-space explosion and allows the reuse of properties and components. Different specialized tools can be applied to different modules; finite-state modules can be model checked, and infinite-state modules can be verified deductively.

Abstraction and modularity are orthogonal to each other: modules can be abstracted, and abstractions can be decomposed. Together, they form a powerful basis for scaling up formal verification [37, 52].

6.2 General Deductive Environments

A general-purpose theorem prover can formalize modular decomposition and assume-guarantee reasoning, formalize and verify the correctness of abstractions, apply verification rules, and put the results together. It can also handle *parameterization*, where an arbitrary number of similar modules are composed, and provide decision procedures and validity checkers for specialized domains (see Section 4.1). This often includes OBDD's for finite-state constraint solving and model checking.

Provers such as HOL, the Boyer-Moore prover, and PVS, have been equipped with BDD's and used in this way. From the theorem-proving point of view, this is a tight integration: model checking becomes a proof rule or a tactic. However, it is usually only used at the leaves of the proof tree, so the general verification method remains loosely coupled.

The above theorem provers do not commit to any particular system representation, so the semantics of reactive systems and temporal properties must be formalized within their logic. The STeP system [7] builds in the notion of fair transition systems and LTL, and provides general model checking and theorem proving as well (and more specialized tools, which we will see below). The Symbolic Model Prover (SyMP) [4] is an experimental deductive environment oriented to model checking, featuring a modular system specification language and interfaces to SMV and decision procedures.

Verification environments such as the above, which include model checking and theorem proving under a common deductive environment, support the following tasks:

Debugging using Model Checking: A simple but important application of model checking in a deductive environment is to check (small) finite instances of an infinite-state or parameterized system specification. In this way, many errors can quickly be found before the full deductive verification effort proceeds.

Incremental Verification: As mentioned in Section 4, verification conditions need not be valid in general, but only valid with respect to the invariants of the system. Therefore, deductive verification usually proceeds by proving a series of invariants of increasing strength, where each is used as a lemma to prove subsequent ones [44]. The database of system properties can also help justify new reductions or abstractions, which in turn can be used to prove new properties.

Automatically generated invariants can also be used to establish verification conditions. Different abstract domains generate different classes of invariants; in some cases, expressing the invariants so that they can be effectively used by the theorem proving machinery is a non-trivial task, and presents another combination challenge. So is devising invariant generation methods that take advantage of system properties already proven.

Invariants can also be used to constrain the set of states explored in symbolic model checking [49], for extra efficiency. Here, too, it may be necessary to translate the invariant into a form useful to the model checker, e.g. if an individual component or an abstracted finite-state version is being checked.

Formal Decomposition: System abstraction and modular verification are often performed manually and then proved correct. If done within a general theorem proving environment, these steps can be formally checked. For instance, Müller and Nipkow [46] use the Isabelle theorem prover to prove the soundness of an I/O-automata abstraction, which is then model checked. Hungar [33] constructs model-checkable abstractions based on data independence and modularity. Abstraction is also used by Rajan *et. al.* [51] to obtain subgoals that can be model checked, where the correctness of the abstraction is proved deductively. Kurshan and Lamport [38] use deductive modular decomposition to reduce the correctness of a large hardware system to that of smaller components that can be model checked.

6.3 Abstraction Generation Using Theorem Proving

Many of the works cited above use theorem proving to prove that an abstraction given *a priori* is correct. This abstraction is usually constructed manually, which can be a time-consuming and error-prone task. An attractive alternative is to use theorem proving to *construct* the abstraction itself.

The domain most often used for this purpose is that of *assertion-based abstractions*, also known as *predicate* or *boolean abstractions*. Here, the abstract state-space is the complete boolean algebra over a finite set of assertions $B : \{b_1, \ldots, b_n\}$, which we call the *basis* of the abstraction. For a point p in the boolean algebra, its *concretization* $\gamma(p)$ is the set of concrete states that satisfy p (see Section 5). For an abstract assertion $f^{\mathcal{A}}$, a concrete assertion that characterizes $\gamma(f^{\mathcal{A}})$ is obtained simply by replacing each boolean variable in $f^{\mathcal{A}}$ by the

corresponding basis element. Similarly, for an abstract temporal formula $\varphi^{\mathcal{A}}$, its concretization $\gamma(\varphi^{\mathcal{A}})$ is obtained by replacing each assertion in $\varphi^{\mathcal{A}}$ by its concretization. Since the abstract system is described in terms of logical formulas, off-the-shelf theorem proving can be used to construct and manipulate it.

Graf and Saidi [28] presented the first automatic procedure for generating such abstractions, using theorem proving to explicitly generate the abstract state-space. Given an abstract state s, an approximation of its successors is computed by deciding which assertions are implied by the postcondition of $\gamma(s)$. This is done using a tactic of the PVS theorem prover [47]. If the proof fails, the next-state does not include any information about the corresponding assertions, thus safely *coarsening* the abstraction: the abstract transition relation over-approximates the concrete one.

An alternative algorithm is presented by Colón and this author in [17], using the decision procedures in STeP. Rather than performing an exhaustive search of the reachable abstract states while constructing $\mathcal{A}$, this algorithm transforms $\mathcal{S}$ to $\mathcal{A}$ directly, leaving the exploration of the abstract state-space to an off-the-shelf model checker. Thus, this procedure is applicable to systems whose abstract state-space is too large to enumerate explicitly, but can still be handled by a symbolic model checker. The price paid by this approach, compared to [28], is that a coarser abstraction may be obtained.

Bensalem *et. al.* [3] present a similar framework for generating abstractions. Here, the invariant to be proved is assumed when generating the abstraction, yielding a better abstract system.

From the combination point of view, all of these approaches have the advantage of using a validity checker purely as a black box, operating only under the assumption of soundness; more powerful checkers will yield better abstractions. For instance, SVC [2] (see Section 4.1) has been used to generate predicate abstractions as well [21]. As with invariant generation, these methods can be parameterized by the abstract assertion language and validity checker used. Note that some user interaction is still necessary, in the choice of assertions for the abstraction basis; but the correctness of the resulting abstraction is guaranteed. A related method for generating abstract systems is presented in [40], intended for the practical analysis and debugging of complex software systems.

These finite-state abstractions can be used to generate invariants; as noted in Section 5.1, the concretization of the reachable state-space of $\mathcal{A}$ is an invariant of the original system (but may be an unwieldy formula); on the other hand, previously proven invariants of $\mathcal{S}$ can be used as lemmas in the abstraction generation process, yielding finer abstract systems.

7 Tight Combinations

We now describe and classify more tightly-coupled combination methods that are based on abstraction; its counterpart, modularity, is only briefly mentioned in the interest of satisfying space constraints.

Abstraction Refinement: Recall that weak property preservation (Section 5) only guarantees that $\mathcal{S} \models \phi$ whenever $\mathcal{A} \models \varphi^{\mathcal{A}}$; if the abstract property fails, it is possible that φ does hold for $\mathcal{S}$, but the abstraction was not *fine enough* to prove it. Thus, in general, abstractions must be *refined* in order to prove the desired property. In predicate abstraction, refinement occurs by performing new validity checks over the existing basis, or adding new assertions to the basis.

A generic abstraction-based verification procedure for proving $\mathcal{S} \models \phi$ proceeds as follows: First, an initial weakly-preserving abstraction $\mathcal{A}$ of $\mathcal{S}$ is given by the user or constructed automatically. If model checking $\mathcal{A} \models \varphi^{\mathcal{A}}$ succeeds, the proof is complete. Otherwise, $\mathcal{A}$ is used as the starting point for producing a finer abstraction $\mathcal{A}'$, which is still weakly-preserving for $\mathcal{S}$ but satisfies more properties. This process is repeated until φ is proved (if it indeed holds for $\mathcal{S}$).

At each step, the model checker produces an *abstract* counterexample, which does not necessarily correspond to any concrete computation. However, it can help choose the next refinement step, or serve as the basis for finding a concrete counterexample that indeed falsifies φ. Finding effective procedures to distinguish between these two cases and perform the refinement remains an intriguing research problem, which is addressed by some of the methods described below. (In the general case, we are still faced with an undecidable problem, so no method can guarantee that the process will terminate.)

Methods that combine deductive and algorithmic verification through abstraction can be classified according to the following criteria:

- whether the abstraction is constructed *a priori* or or dynamically refined (*static* vs. *dynamic*);
- whether the abstraction generation is tailored specifically for the property of interest (*bottom-up* vs. *top-down*);
- whether the process is automatic or interactive.

Focusing on a particular formula allows for coarser abstract systems, which satisfy fewer properties but can be easier to construct and model check.

The automatic abstraction generation algorithms of Section 6.3 are generally bottom-up, but can be focused towards a particular temporal property by including its atomic subformulas as part of the assertion basis. The tighter combinations described below are mostly top-down, using a mixture of model checking and theorem proving that is specialized to the particular temporal property being proved.

7.1 Diagram-based Formalisms

We begin by describing a number of *diagram-based* verification formalisms that have been developed as part of the STeP project [7]. They offer deductive-algorithmic proof methods that are applicable to arbitrary temporal properties.

Static Abstractions: GVD's: The *Generalized Verification Diagrams* (GVD's) of Browne *et. al.* [11] provide a graphical representation of the verification conditions needed to establish an arbitrary temporal formula, extending the specialized diagrams of Manna and Pnueli [43].

A GVD serves as a proof object, but is also a weakly-preserving assertion-based abstraction of the system, where the basis is the set of formulas used in the diagram [41]. Each node in the diagram is labeled with an assertion f, and corresponds to an abstract state representing the set of states that satisfy f. The diagram Φ is identified with a set of computations $\mathcal{L}(\Phi)$, and a set of verification conditions associated with the diagram show that $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\Phi)$. This proves, deductively, that Φ is a correct abstraction of $\mathcal{S}$. The proof is completed by showing that $\mathcal{L}(\Phi) \subseteq \mathcal{L}(\varphi)$. This corresponds to model checking φ over Φ, when Φ is seen as an abstract system, and is done algorithmically, viewing the diagram as an ω-automata.

Diagrams can be seen as a flexible generalization of the classic deductive rules. (As noted in, e.g., [3, 37], the success of a deductive rule such as G-INV of Section 4 implies the existence of an abstraction for which the proven property can be model checked.) Since the diagram is tailored to prove a particular property φ, this can be classified as a *top-down* approach. Since the formalism does not include refinement operations in the case that the proof attempt fails, it is *static*.

Dynamic Abstractions: Deductive Model Checking: *Deductive Model Checking* (DMC), presented by Sipma *et. al.* [57], is a method for the interactive model checking of possibly infinite-state systems. To prove a property φ, DMC searches for a counterexample computation by refining an abstraction of the $(\mathcal{S}, \neg\varphi)$ *product graph*.

The DMC procedure interleaves the theorem proving and model checking steps, refining the abstraction as the model checking proceeds. This focuses the theorem proving effort to those aspects of the system that are relevant to the property being proved. On the other hand, the expanded state-space is restricted by the theorem proving, so that space savings are possible even in the case of finite-state systems.

Sipma [56] describes DMC, GVD's, and their application to the verification of real-time and hybrid systems. The *fairness diagrams* of de Alfaro and Manna [22] present an alternate dynamic refinement method that combines the top-down and bottom-up approaches.

7.2 Model Checking for Infinite-State Systems

Abstraction is also the basis of model checking algorithms for decidable classes of infinite-state systems, such as certain classes of real-time and hybrid systems. In some cases, the exploration of a finite quotient of the state-space is sufficient. For others, the convergence of fixpoint operations is ensured by the right choice of abstract assertion language, such as polyhedra [30] or Presburger arithmetic [13]. The underlying principles are explored in [26, 31].

A number of "local model checking" procedures for general infinite-state systems, such as that of Bradfield and Stirling [10], are also hybrid combinations of deductive verification rules and model checking. Another top-down approach is presented by Damm *et. al.* [19] as a "truly symbolic" model checking procedure, analyzing the separation between data and control. A tableau-based procedure

for ∀CTL generates the required verification conditions in a top-down, local manner, similarly to DMC.

Model Checking and Static Analysis: The relationship between model checking and abstract interpretation continues to be the subject of much research. Static program analysis methods based on abstract interpretation have been recently re-formulated in terms of model checking. For instance, Schmidt and Steffen [55] show how many program analysis techniques can be understood as the model checking of particular kinds of abstractions.

ESC [23] and Nitpick [34] are tools that automatically detect errors in software systems by combining static analysis and automatic theorem proving methods. Another challenge is to further combine program analysis techniques with formal verification and theorem proving (see [52]).

Finally, we note that there is also much work on applying abstraction to *finite-state* systems, particularly large hardware systems. Abstracting from the finite-state to an *infinite-state* abstract domain has proved useful here, namely, using uninterpreted function symbols and symbolically executing the system using a decidable logic, as shown, e.g., by Jones *et. al.* [35].

7.3 Integrated Approaches

A number of verification environments and methodologies have been proposed that combine the above ingredients in a systematic way.

As mentioned above, STeP [7] includes deductive verification, generalized verification diagrams, symbolic and explicit-state model checking, abstraction generation, and automatic invariant generation, which share a common system and property specification language. To this, modularity and compositional verification are being added as well—see Finkbeiner *et. al.* [27].

Dingel and Filkorn [25] apply abstraction and error trace analysis to infinite-state systems. The abstract system is generated automatically given a *data abstraction* that maps concrete variables and functions to abstract ones. If an abstract counterexample is found that does not correspond to a concrete one, an *assumption* that excludes this counterexample is generated. This is a temporal formula that should hold for the concrete system $\mathcal{S}$. The model checker, which takes such assumptions into account, is then used again. The process is iterated until a concrete counterexample is found, or model checking succeeds under a given set of assumptions. In the latter case, the assumptions are deductively verified over the concrete system. If they hold, the proof is complete.

Rusu and Singerman [53] present a framework that combines abstraction, abstraction refinement and theorem proving specialized to the case of invariants, where the different components are treated as "black boxes." After an assertion-based abstraction is generated (using the method of Graf and Saidi [28]), abstract counterexamples are analyzed to refine the abstraction or produce a concrete counterexample. Conjectures generated during the refinement process are given to the theorem prover, and the process repeated.

A similar methodology is proposed by Saidi and Shankar [54]. The ongoing Symbolic Analysis Laboratory (SAL) project at SRI [52] proposes a collection of multiple different analysis tools, including theorem provers, model checkers, abstraction and invariant generators, that communicate through a common language in a blackboard architecture.

An Abstraction-based Proposal: Table 2 summarizes the classification of the abstraction-based methods discussed in this section. For general infinite-state systems, the automatic methods are incomplete, and the complete methods are interactive.

Method	refinement?	uses φ?	automatic?
Static Analysis and Invariant Generation (Abstract Interpretation)	static	bottom-up	automatic
(Generalized) Verification Diagrams (includes verification rules)	static	top-down	interactive
Deductive Model Checking	dynamic	top-down	interactive
Fairness Diagrams	dynamic	[both]	interactive
Infinite-state Model Checking	dynamic	top-down	[both]

Table 2. Classification of some combination methods based on abstraction

In [58], this author proposes combining these approaches by exploiting their common roots in abstraction. Each different proof attempt (including failed ones) and static analysis operation provides additional information about the system being verified. This information can be captured, incrementally, as an *extended finite-state abstraction*, which includes information about fairness constraints and well-founded orders over the original system. Once generated, these abstractions can be combined and re-used, given a (correspondingly extended) model checker to reason about them.

Thus, abstractions can serve as the repository for all the information about the system that is shared by the different components. An important challenge is to manage and combine abstractions from different domains, finding a common language to express them.

The tight integration found in DMC's and GVD's (Section 7.1) is an obstacle for their implementation using off-the-shelf tools. However, it allows for more detailed user input and feedback. Given tools whose input is expressive and flexible enough, such as a model checker for the extended abstractions, it will be possible to implement such tightly coupled methods in a more modular way.

The abstraction framework has the advantage of leaving the combinatorial problems to the automatic tools, as well as getting the most out of the automatic theorem-proving tools; the user can then focus on defining the right abstractions and guiding their refinement, until a proof or counterexample are found. We believe that such approaches will lead to more automated, lightweight and useful verification tools.

Acknowledgements: The author thanks: the FROCOS organizers, for their kind invitation; the STeP team, and in particular Nikolaj Bjørner, Anca Browne, Michael Colón, Bernd Finkbeiner, Zohar Manna, and Henny Sipma, for their inspiration and feedback—but even though much of the work described above is theirs, they are not to be held responsible for the views expressed herein; and the Leyva-Uribe family, for its hospitality and patience while the final version of this paper was prepared.

References

1. R. Alur and T. A. Henzinger, editors. *Proc. 8^{th} Intl. Conf. on Computer Aided Verification*, vol. 1102 of *LNCS*. Springer, July 1996.
2. C. Barrett, D. L. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *1st Intl. Conf. on Formal Methods in Computer-Aided Design*, vol. 1166 of *LNCS*, pp. 187–201, Nov. 1996.
3. S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Proc. 10^{th} Intl. Conf. on Computer Aided Verification*, vol. 1427 of *LNCS*, pp. 319–331. Springer, July 1998.
4. S. Berezin and A. Groce. *SyMP: The User's Guide*. Comp. Sci. Department, Carnegie-Mellon Univ., Jan. 2000.
5. A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Autom. Conf. (DAC'99)*, 1999.
6. N. S. Bjørner. *Integrating Decision Procedures for Temporal Verification*. PhD thesis, Comp. Sci. Department, Stanford Univ., Nov. 1998.
7. N. S. Bjørner, A. Browne, E. S. Chang, M. Colón, A. Kapur, Z. Manna, H. B. Sipma, and T. E. Uribe. STeP: Deductive-algorithmic verification of reactive and real-time systems. In [1], pp. 415–418.
8. N. S. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Comp. Sci.*, 173(1):49–87, Feb. 1997.
9. N. S. Bjørner, M. E. Stickel, and T. E. Uribe. A practical integration of first-order reasoning and decision procedures. In *Proc. of the 14^{th} Intl. Conf. on Automated Deduction*, vol. 1249 of *LNCS*, pp. 101–115. Springer, July 1997.
10. J. C. Bradfield and C. Stirling. Local model checking for infinite state spaces. *Theoretical Comp. Sci.*, 96(1):157–174, Apr. 1992.
11. A. Browne, Z. Manna, and H. B. Sipma. Generalized temporal verification diagrams. In *15th Conf. on the Foundations of Software Technology and Theoretical Comp. Sci.*, vol. 1026 of *LNCS*, pp. 484–498. Springer, 1995.
12. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.
13. T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using Presburger arithmetic. In Grumberg [29], pp. 400–411.
14. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs*, vol. 131 of *LNCS*, pp. 52–71. Springer, 1981.
15. E. M. Clarke, M. Fujita, and X. Zhao. Hybrid decision diagrams. Overcoming the limitations of MTBDDs and BMDs. In *IEEE/ACM Intl. Conf. on Computer-Aided Design*, pp. 159–163, Nov. 1995.
16. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Dec. 1999.

17. M. A. Colón and T. E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Proc.* 10^{th} *Intl. Conf. on Computer Aided Verification*, vol. 1427 of *LNCS*, pp. 293–304. Springer, July 1998.
18. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In 4^{th} *ACM Symp. Princ. of Prog. Lang.*, pp. 238–252. ACM Press, 1977.
19. W. Damm, O. Grumberg, and H. Hungar. What if model checking must be truly symbolic. In *TACAS'95*, vol. 1019 of *LNCS*, pp. 230–244. Springer, May 1995.
20. D. R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking.* PhD thesis, Eindhoven Univ. of Technology, July 1996.
21. S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *Proc.* 11^{th} *Intl. Conf. on Computer Aided Verification*, vol. 1633 of *LNCS*. Springer, 1999.
22. L. de Alfaro and Z. Manna. Temporal verification by diagram transformations. In [1], pp. 287–299.
23. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Tech. Report 159, Compaq SRC, Dec. 1998.
24. D. L. Dill. The Murφ verification system. In [1], pp. 390–393.
25. J. Dingel and T. Filkorn. Model checking of infinite-state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In *Proc.* 7^{th} *Intl. Conf. on Computer Aided Verif.*, vol. 939 of *LNCS*, pp. 54–69, July 1995.
26. E. A. Emerson and K. S. Namjoshi. On model checking for non-deterministic infinite-state systems. In *Proc. 13th IEEE Symp. Logic in Comp. Sci.*, pp. 70–80. IEEE Press, 1998.
27. B. Finkbeiner, Z. Manna, and H. B. Sipma. Deductive verification of modular systems. In *COMPOS'97*, vol. 1536 of *LNCS*, pp. 239–275. Springer, Dec. 1998.
28. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In Grumberg [29], pp. 72–83.
29. O. Grumberg, editor. *Proc.* 9^{th} *Intl. Conf. on Computer Aided Verification*, vol. 1254 of *LNCS*. Springer, June 1997.
30. T. A. Henzinger and P. Ho. HyTech: The Cornell hybrid technology tool. In *Hybrid Systems II*, vol. 999 of *LNCS*, pp. 265–293. Springer, 1995.
31. T. A. Henzinger and R. Majumdar. A classification of symbolic transition systems. In *Proc. of the 17th Intl. Conf. on Theoretical Aspects of Comp. Sci. (STACS 2000)*, LNCS. Springer, 2000.
32. G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, Engelwood Cliffs, NJ, 1991.
33. H. Hungar. Combining model checking and theorem proving to verify parallel processes. In *Proc.* 5^{th} *Intl. Conf. on Computer Aided Verification*, vol. 697 of *LNCS*, pp. 154–165. Springer, 1993.
34. D. Jackson and C. A. Damon. Nitpick reference manual. Tech. report, Carnegie-Mellon Univ., 1996.
35. R. B. Jones, J. U. Skakkebæk, and D. L. Dill. Reducing manual abstraction in formal verification of out-of-order execution. In G. Gopalakrishnan and P. Windley, editors, *2nd Intl. Conf. on Formal Methods in Computer-Aided Design*, vol. 1522 of *LNCS*, pp. 2–17, Nov. 1998.
36. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In Grumberg [29], pp. 424–435.
37. Y. Kesten and A. Pnueli. Modularization and abstraction: The keys to practical formal verification. In *Mathematical Foundations of Comp. Sci.*, vol. 1450 of *LNCS*, pp. 54–71, Aug. 1998.

38. R. P. Kurshan and L. Lamport. Verification of a multiplier: 64 bits and beyond. In *Proc.* 5^{th} *Intl. Conf. on Computer Aided Verification*, vol. 697 of *LNCS*, pp. 166–179. Springer, 1993.
39. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:1–35, 1995.
40. M. Lowry and M. Subramaniam. Abstraction for analytic verification of concurrent software systems. In *Symp. on Abstraction, Reformulation, and Approx.*, May 1998.
41. Z. Manna, A. Browne, H. B. Sipma, and T. E. Uribe. Visual abstractions for temporal verification. In A. Haeberer, editor, *Algebraic Methodology and Software Technology (AMAST'98)*, vol. 1548 of *LNCS*, pp. 28–41. Springer, Dec. 1998.
42. Z. Manna and A. Pnueli. Completing the temporal picture. *Theoretical Comp. Sci.*, 83(1):97–130, 1991.
43. Z. Manna and A. Pnueli. Temporal verification diagrams. In M. Hagiya and J. C. Mitchell, editors, *Proc. Intl. Symp. on Theoretical Aspects of Computer Software*, vol. 789 of *LNCS*, pp. 726–765. Springer, 1994.
44. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer, New York, 1995.
45. K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Pub., 1993.
46. O. Müller and T. Nipkow. Combining model checking and deduction for I/O-automata. In *TACAS'95*, vol. 1019 of *LNCS*, pp. 1–12. Springer, May 1995.
47. S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In [1], pp. 411–414.
48. A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. Found. of Comp. Sci.*, pp. 46–57. IEEE Computer Society Press, 1977.
49. A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In [1], pp. 184–195.
50. J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Intl. Symp. on Programming*, vol. 137 of *LNCS*, pp. 337–351. Springer, 1982.
51. S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking with automated proof checking. In *Proc.* 7^{th} *Intl. Conf. on Computer Aided Verification*, vol. 939 of *LNCS*, pp. 84–97, July 1995.
52. J. Rushby. Integrated formal verification: Using model checking with automated abstraction, invariant generation, and theorem proving. In *Theoretical and Practical Aspects of SPIN Model Checking*, vol. 1680 of *LNCS*, pp. 1–11, July 1999.
53. V. Rusu and E. Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. In *TACAS'99*, vol. 1579 of *LNCS*. Springer, Mar. 1999.
54. H. Saidi and N. Shankar. Abstract and model check while you prove. In *Proc.* 11^{th} *Intl. Conf. on Computer Aided Verification*, vol. 1633 of *LNCS*. Springer, 1999.
55. D. A. Schmidt and B. Steffen. Program analysis *as* model checking of abstract interpretations. In *Proc. 5th Static Analysis Symp.*, LNCS. Springer, Sept. 1998.
56. H. B. Sipma. *Diagram-based Verification of Discrete, Real-time and Hybrid Systems.* PhD thesis, Comp. Sci. Department, Stanford Univ., Feb. 1999.
57. H. B. Sipma, T. E. Uribe, and Z. Manna. Deductive model checking. *Formal Methods in System Design*, 15(1):49–74, July 1999.
58. T. E. Uribe. *Abstraction-based Deductive-Algorithmic Verification of Reactive Systems.* PhD thesis, Comp. Sci. Department, Stanford Univ., Dec. 1998. Tech. Report STAN-CS-TR-99-1618.