In Manfred Broy and Bernhard Rumpe, editors, *Requirements Targeting Software and Systems Engineering,* volume 1526 of *Lecture Notes in Computer Science,* pages 273–292. Springer-Verlag, 1998.

Abstraction and Modular Verification of Infinite-State Reactive Systems ⋆

Zohar Manna,
Michael A. Colón, Bernd Finkbeiner,
Henny B. Sipma and Tomás E. Uribe

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

Abstract. We review a number of temporal verification techniques for reactive systems using modularity and abstraction. Their use allows the verification of larger systems, and the incremental verification of systems as they are developed and refined. In particular, we show how deductive verification tools, and the combination of finite-state model checking and abstraction, allow the verification of infinite-state systems featuring data types commonly used in software specifications, including real-time and hybrid systems.

1 Introduction

Reactive systems have an ongoing interaction with their environment. Many systems can be seen as reactive systems, including computer hardware, concurrent programs, network protocols, and concurrent software. *Temporal logic* is a convenient language for expressing properties of reactive systems [Pnu77]. A *temporal verification methodology* provides methods for proving that a given reactive system satisfies its temporal specification [MP95].

Computations of reactive systems are modeled as infinite sequences of states. For *finite-state* systems, the possible system states are determined by a fixed number of variables with finite domain, so there are finitely many such states. Algorithmic verification (model checking) can automatically decide the validity of temporal properties over such finite-state systems [CE81,QS82], and has been particularly successful for hardware [McM93].

Software systems, on the other hand, are usually *infinite-state*, since they contain system variables over unbounded domains, such as integers, lists, trees, and other data types. Most finite-state verification methods cannot be applied directly to such systems. The application of temporal verification techniques to

⋆ This research was supported in part by the National Science Foundation under grant CCR-95-27927, the Defense Advanced Research Projects Agency under NASA grant NAG2-892, ARO under grant DAAH04-95-1-0317, ARO under MURI grant DAAH04-96-1-0341, and by Army contract DABT63-96-C-0096 (DARPA).

software systems is further limited by the size and complexity of the systems analyzed. Such limitations already appear in the verification of large finite-state systems, e.g., complex hardware, where the state-explosion problem, and the limitations of symbolic methods, restrict the number of finite-state variables that can be considered by automatic methods.

Deductive verification, which relies on general theorem-proving and user interaction, provides complete proof systems that can, in principle, prove the correctness of any property over an infinite-state system, provided the property is indeed valid for that system. However, these methods are also limited by the size and complexity of the system being analyzed, becoming much more laborious as the system complexity grows.

To overcome these limitations, verification methods analogous to those used to manage complexity in software design are being investigated. *Modular verification* follows the classic divide-and-conquer paradigm, where portions of a complex system are analyzed independently of each other. It holds the promise of proof reuse and the creation of libraries of verified components. *Abstraction* is based on ignoring details as much as possible, often simplifying the domain of computation of the original system. This may allow, for instance, abstracting infinite-state systems to finite-state ones that can be more easily model checked.

This paper presents an overview of a number of abstraction and modular verification methods that we have recently investigated, geared to the verification of general infinite-state reactive systems. These methods are being implemented as part of the STeP (Stanford Temporal Prover) verification system (see Section 2.3). We show how they help design and debug complex systems, modularly described.

Outline: Section 2 presents the basic preliminary notions, and the STeP verification system. In Section 3 we briefly present abstraction, describing a number of simple (infinite-state) examples and their abstractions, generated and verified using STeP. Section 4 presents modular verification, including a larger example of a hybrid system that is modularly specified and verified, again using STeP. Section 5 presents our conclusions and briefly discusses related work.

2 Preliminaries

2.1 System and Property Specification

Transition Systems: We use *transition systems* [MP95] to model finite- and infinite-state reactive systems. An *assertion language*, usually based on first-order logic, is used to represent sets of system states, described in terms of a set of *system variables* $\mathcal{V}$. A reactive system is given by $\mathcal{V}$, a set of *initial states* θ, and a set of *transitions* $\mathcal{T}$. The *initial condition* Θ is described by an assertion over $\mathcal{V}$, and transitions are described by *transition relations*, assertions over the set of system variables $\mathcal{V}$ and a set of primed system variables $\mathcal{V}'$, giving the value of the system variables at the next state. A *run* of the system is an infinite

sequence of states $s_0, s_1, \ldots$ where s_0 satisfies θ, and for each $i \geq 0$, there is some transition $\tau \in \mathcal{T}$ such that (s_i, s_{i+1}) satisfies ρ_τ, the transition relation of τ.

Transitions can be *just* (weakly fair) or *compassionate* (strongly fair), indicating that they cannot be enabled infinitely often (continuously, in the case of justice) but never taken. A *computation* is a run that satisfies these fairness requirements. See [MP95] for details.

Temporal Logic: To express properties of reactive systems, we use *linear-time temporal logic* (LTL) [MP95], where we allow first-order quantification at the state-level.

Real-Time and Hybrid Systems: Real-time and hybrid systems can be modeled using *clocked* and *phase transition systems* [MP96], which use the basic transition system representation. Real-valued *clock variables* are updated by a *tick* transition that advances time, constrained by a global progress condition. In the case of hybrid systems, other continuous variables evolve over time, as given by a set of differential equations. This allows the reuse of the standard deductive verification techniques [MS98,KMP96].

Timed automata and hybrid automata can be easily translated into these formalisms. Furthermore, by adopting the general transition system representation, clocked and phase transition systems can model systems with an infinite-state control component. This includes, for instance, software with real-time constraints and software-controlled hybrid systems. No extension of temporal logic is required, since clock variables (including the global clock) can be directly referred to in specifications. For real-time and hybrid systems, fairness constraints are usually replaced by upper bounds on how long transitions can be enabled without being taken. Only runs that are *non-zeno*, where time grows beyond any bound, are considered to be computations.

2.2 Deductive and Algorithmic Verification

As mentioned in Section 1, the two main approaches to the verification of temporal properties of reactive systems are deductive verification (*theorem-proving*) and algorithmic verification (*model checking*). In deductive verification, the validity of a temporal property over a given system is reduced to the general validity of first-order verification conditions. In algorithmic verification, a temporal property is established by an exhaustive search of the system's state space, usually searching for a counterexample computation.

Model checking procedures are automatic, while deductive verification often relies on user interaction to identify suitable lemmas and auxiliary assertions. However, model checking is usually applicable only to systems with a finite, fixed number of states, while the deductive approach can verify infinite-state systems and *parameterized* systems, where an unbounded number of components with similar structure are composed.

2.3 The STeP System

The Stanford Temporal Prover (STeP) supports the computer-aided formal verification of reactive, real time and hybrid systems based on their temporal specifications, expressed in linear-time temporal logic. STeP integrates algorithmic and deductive methods to allow the verification of a broad class of systems, including parameterized (N-component) circuit designs, parameterized (N-process) programs, and programs with infinite data domains.

STeP is described in [BBC+95,BBC+96]. The latest release of STeP, version 2.0, is described in [MBB+98].

3 Abstraction

Abstraction reduces the complexity of a system being verified by considering a simpler *abstract system*, where some of the details of the original *concrete system* are hidden. There is much work on the theoretical foundations of reactive system abstraction [CGL94,DGG94,LGS+95,Dam96], usually based on the ideas of *abstract interpretation* [CC77].

Most abstractions *weakly preserve* temporal properties: if a property holds for the abstract system, then a corresponding property will hold for the concrete one. However, the converse will not be true: not all properties satisfied by the concrete system will hold at the abstract level. Thus, only positive results transfer from the abstract to the concrete level. This means, in particular, that abstract counterexamples will not always correspond to concrete ones.

Abstractions that remove too much information from the concrete system and are thus too *coarse* will fail to prove the property of interest. They then can be *refined*, by adding more detail, until the property can be proved or a concrete counterexample is found.

3.1 From Infinite- to Finite-State

The intuition that motivates the use of abstraction in the verification of software systems is the often limited interaction between control and data. The particular values taken on by data variables are often unimportant. Rather, it is the relationship between these variables which is relevant to verifying the system. For example, to decide which branch of a conditional statement will be taken, it is sufficient to know whether its guard is true or false, and this information can often be gleaned by an analysis of how each system transition affects the truth value of that guard.

Example: Bakery: Consider the version of Lamport's Bakery algorithm for mutual exclusion shown in Figure 1, as specified in the Simple Programming Language (SPL) of [MP95]. (STeP automatically translates such a program into the corresponding fair transition system.) This system contains two infinite-domain variables, y_1 and y_2, ranging over the non-negative integers. There is no

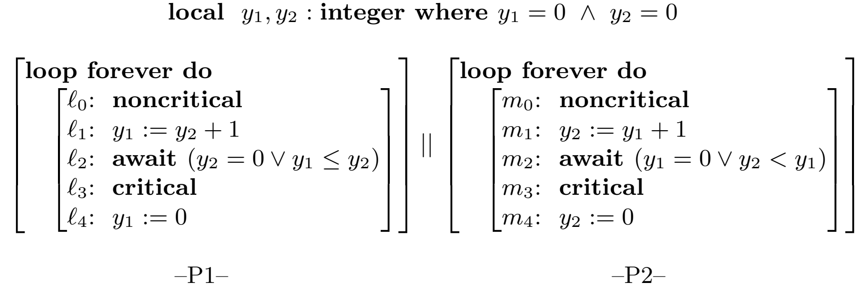

Fig. 1. Program BAKERY

upper bound on the values that these variables can take in a computation of the system. Thus, the system is infinite-state, and cannot be directly model checked.

However, knowing only the truth value of the assertions

$$\begin{aligned} b_1 &: y_1 = 0, \\ b_2 &: y_2 = 0 \\ b_3 &: y_1 \leq y_2 \end{aligned}$$

is sufficient to determine which branches of the conditional statements are feasible. Using these assertions to replace the original integer variables, and maintaining the finite-domain control variables, we can construct a finite-state abstraction of the Bakery algorithm, shown in Figure 2. This abstract program can be given to a model checker to verify the basic safety properties of the original system, including *mutual exclusion*,

$$\Box \neg (at_\ell_3 \wedge at_m_3) \ ,$$

stating that the two processes can never be both in their critical section at the same time, and *one-bounded overtaking*:

$$\Box (at_\ell_2 \ \rightarrow \ \neg at_m_3 \ \mathcal{W} \ (at_m_3 \ \mathcal{W} \ (\neg at_m_3 \ \mathcal{W} \ at_\ell_3))) \ ,$$

which states that if process P1 is waiting to enter its critical section, then process P2 can only enter its critical section at most once before P1 does.

All transitions in the concrete BAKERY program are just, except for the **noncritical** statements at ℓ_0 and m_0. Under certain conditions, the abstract transitions can inherit the fairness properties of the original ones [KMP94,CU98]. This is the case here, so we can also prove *accessibility*,

$$\Box (at_\ell_1 \rightarrow \Diamond at_\ell_3) \ ,$$

by model checking the abstract system with the inherited fairness requirements.

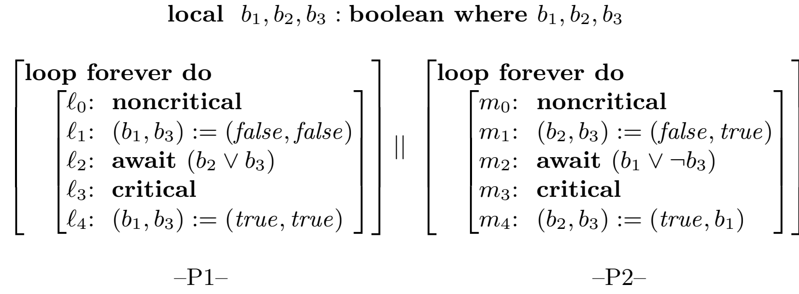

Fig. 2. Abstraction of Program BAKERY

3.2 Generating Abstractions

Constructing abstract systems manually can be time-consuming, and requires that the correctness of the abstraction be checked during a separate phase. If this is not done formally, a new potential source of error is introduced.

The Bakery example in the preceding section is an instance of *assertion-based abstraction*, where a finite number of assertions $\{b_1, \dots, b_n\}$ are used as boolean variables in the abstract system, replacing the concrete variables they refer to. An algorithm that generates an abstract system automatically, given such a set of assertions (called the *basis* of the abstraction), is presented in [CU98]. The algorithm uses a validity checker to establish relationships between the basis elements, compositionally abstracting the transition relations of the system to directly produce abstract transition relations. This algorithm has been implemented as part of STeP, using the STeP validity checker, and automatically generated the abstractions in this section.

Example: Fischer: As a second example, consider Fischer's mutual exclusion algorithm, as shown in Figure 3. This is a real-time system, with lower and upper bounds L and U on the amount of time that each process can wait at any control location. Provided that $2L > U$, the program satisfies mutual exclusion [MP96].

The program can be modeled as a clocked transition system, with clock variables c_1 and c_2 measuring the time each process has been waiting at each control location. Because of these clock variables, the system is infinite-state and cannot be model checked directly.[1]

Examining the system suggests that the truth value of the assertions

$$b_1 : c_1 \geq L$$
$$b_2 : c_2 \geq L$$

together with the value of the finite-domain variables, is sufficient to determine when transitions can be taken. However, these two assertions are not inductive,

[1] Specialized model checkers for real-time and hybrid systems, such as HyTech [HH95] and Kronos [DOTY96], can automatically prove properties of such systems, but are restricted to linear hybrid systems with finite control.

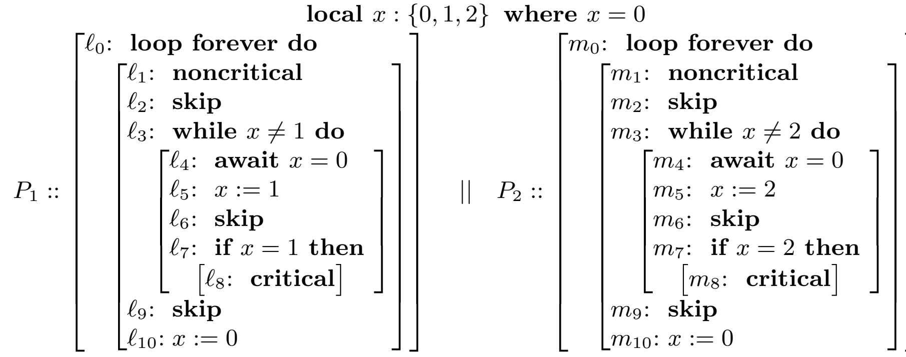

Fig. 3. Fischer's mutual exclusion algorithm.

i.e., the system contains transitions for which knowing the truth values of these assertions is not sufficient to determine them after the transitions are taken. To remedy this situation, we also consider the assertions:

$$\begin{aligned} b_3 &: c_1 \geq c_2 \\ b_4 &: c_2 \geq c_1 \\ b_5 &: c_1 \geq c_2 + L \\ b_6 &: c_2 \geq c_1 + L \end{aligned}$$

These additional assertions yield sufficient information about the relationships between the clock variables to generate a finite-state abstraction fine enough to establish mutual exclusion: $\Box\neg(at_\ell_4 \wedge at_m_4)$.

Example: BRP: Finally, we turn to the bounded retransmission protocol (see, e.g. [HS96,GS97,DKRT97]). The protocol consists of two processes, a sender and a receiver, communicating over two lossy channels. The sender sends a list of items (of some unspecified type) one by one, by repeatedly transmitting a frame containing the current item until the frame is acknowledged. The receiver repeatedly waits for a frame to arrive, acknowledges it, and appends the corresponding item to the end of its own list.

To detect the arrival of duplicate frames and acknowledgements, each process maintains a bit that is compared against a bit included in the frames and acknowledgements. Which each outgoing frame, the sender attaches its bit, which the receiver later copies into the acknowledgement it sends for that frame (if the frame is not lost in transit). The sender ignores any acknowledgements that arrive with the wrong bit, and flips its bit upon the arrival of a correct acknowledgement. The receiver acknowledges every frame it receives, but only appends the carried item to its list if the frame's bit agrees with its own, and flips its own bit in this case. If the number of retransmissions of any frame exceeds a fixed, predetermined bound, the sender aborts transmission. The sender and receiver each report the status of the transmission when they terminate. Both sender and receiver report `OK` (resp. `NOT_OK`) when they detect that the transmis-

sion has succeeded (resp. failed). In addition, the sender may report DONT_KNOW should it happen to abort while transmitting the last item of the list.

One property we would like to establish is that the status reports are consistent: either both processes report OK, both report NOT_OK, or the sender reports DONT_KNOW and the receiver OK or NOT_OK. This is specified as the invariance of:

$$\begin{array}{l} \texttt{sendDone} \wedge \texttt{recvDone} \rightarrow \\ \left[\begin{array}{c} (\texttt{sendStatus} = \texttt{OK} \wedge \texttt{recvStatus} = \texttt{OK}) \vee \\ (\texttt{sendStatus} = \texttt{NOT_OK} \wedge \texttt{recvStatus} = \texttt{NOT_OK}) \vee \\ \texttt{sendStatus} = \texttt{DONT_KNOW} \ \wedge \ (\texttt{recvStatus} = \texttt{OK} \vee \texttt{recvStatus} = \texttt{NOT_OK}) \end{array}\right] \end{array}$$

The system cannot be directly model checked: not only are the sender and receiver lists unbounded, but the retransmission count sendCount is unbounded as well, since the retransmission bound is unspecified. However, a finite-state abstraction of the system can be generated. The assertion

$$b_1 : \texttt{sendList} = \texttt{nil}$$

determines if the sender has successfully transmitted and received acknowledgements for all the items to be sent. In this case, the sender cannot abort transmission. The assertions

$$\begin{array}{l} b_2 : \texttt{sendList} = \texttt{cons}(\texttt{head}(\texttt{sendList}), \texttt{nil}) \\ b_3 : \texttt{sendCount} = 0 \end{array}$$

ensure that the generated abstraction accurately models the sender's behavior when it chooses to abort with one item remaining to be sent. In that case, the sender can report NOT_OK only if it has yet to transmit the last item. Otherwise, the sender must report DONT_KNOW, since it is unclear whether the frame or its acknowledgement was lost in transmission.

Using this basis ($\{b_1, b_2, b_3\}$) to abstract the unbounded variables, and preserving the finite-domain variables (which includes all the variables in the invariant to be proved), STeP automatically generates a finite-state abstraction for which the above invariance is then model checked, automatically as well.

Another property we would like to establish is that the list is correctly transmitted if the sender does not abort. That is, we would like to prove

$$\varphi : \ \Box(\texttt{sendDone} \wedge \texttt{sendList} = \texttt{nil} \ \rightarrow \ \texttt{recvList} = \texttt{LIST})$$

where LIST is the complete list being transmitted. Again, we use abstraction. The assertions

$$\begin{array}{l} b_1 : \texttt{sendList} = \texttt{nil} \\ b_2 : \texttt{recvList} = \texttt{LIST} \end{array}$$

let us track the formula φ to be proven over the abstract system. The assertion

$$b_3 : \texttt{frameItem} = \texttt{head}(\texttt{sendList})$$

tracks the current item as it moves from sender to receiver. Finally,

$$b_4 : \texttt{recvList}{+}{+}\texttt{sendList} = \texttt{LIST}$$
$$b_5 : \texttt{recvList}{+}{+}\texttt{tail}(\texttt{sendList}) = \texttt{LIST}$$

where ++ is the list concatenation operator, capture an inductive relationship between the sender's and the receiver's lists at any point in time, and the list being transmitted; $b_4 \vee b_5$ is an invariant of the system. Given this basis, $\{b_1, \ldots, b_5\}$, STeP automatically generates and model checks a finite-state abstraction.

4 Modular Specification and Verification

The advantages of modular description and development of complex systems are well-known. From the formal verification point of view, decomposing systems into modules allows verification that more closely follows the design structure of the system. For instance, general properties of a parameterized module can be proved once and then reused when the module is instantiated.

[FMS98] presents *modular transition systems*, a system specification formalism that allows systems to be built from transition modules. Modules consist of an *interface*, which describes the interaction with the environment, including a list of shared variables and the name of exported transitions, and a *body*, which describes the module's actions, as transitions that can be synchronized or interleaved with those of other modules.

Complex modules are constructed from simpler ones by module expressions. The description language includes recursive module definitions, module composition and instantiation, variable hiding and renaming, and augmenting and restricting module interfaces. Composition can be synchronous or asynchronous; transitions with the same label are composed synchronously, while the rest are interleaved. This modular system specification language is being added to STeP, together with the corresponding modular proof rules.

When designing a system modularly, one would like to prove simple properties of individual modules, and combinations of modules, before the entire system is specified. *Assumption-guarantee reasoning* is often used to prove properties of a module that depend on its environment, before that environment is fully specified. Abstraction can facilitate this process: Modular properties can be model checked for abstractions, relative to assumptions on the environment. Furthermore, for real-time and hybrid systems, part or all of the complex real-time behavior can be abstracted away when debugging individual modules. More expensive verification methods should only be used after the design components and some of their combinations pass these simple (and fast) checks.

4.1 Example: Steam Boiler Case Study

The *steam boiler* case study [ABL96] is a benchmark for specification and verification methods for hybrid controlled systems. At the time of its appearance we

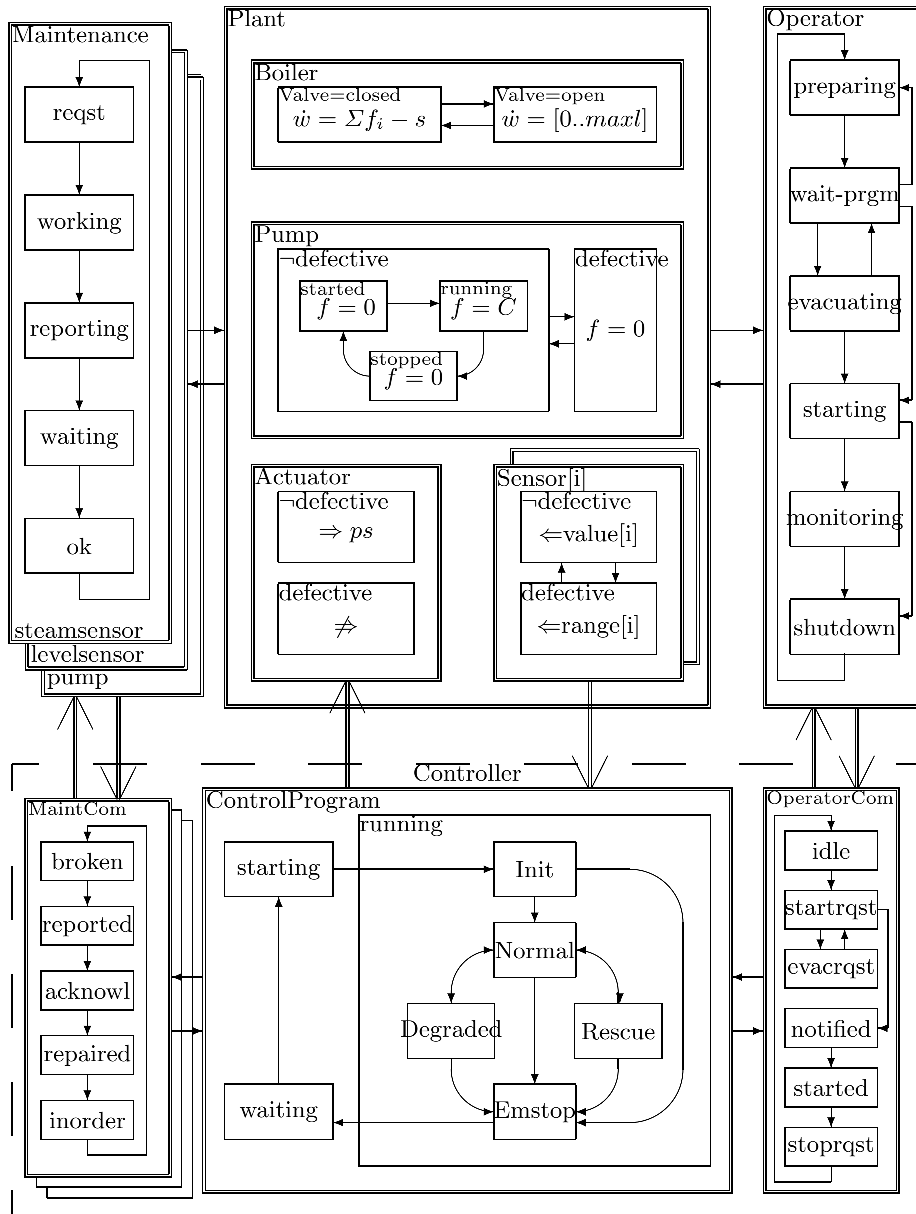

Fig. 4. Schematic overview of the Steam Boiler system

developed a STeP implementation of the system, including both the plant and the controller, consisting of some 1000 lines of STeP SPL code.

However, at that time STeP did not provide any modularity or abstraction techniques. Although we managed to prove some simple properties over that program and discovered numerous bugs in our model of the system, we quickly decided that full verification was not feasible with the tool at hand.

With modularity and abstraction techniques in place in STeP, the case study was revived. The system was rewritten as a modular transition system consisting of ten modules with a total of 80 transitions, 18 real-valued variables, and 28 finite-domain variables. In the following we briefly describe the system, and then present some of the techniques we have used to analyze it.

System Description: Our specification of the system is shown schematically in Figure 4. The system consists of a *physical plant*, a *controller*, a *maintenance department* and an *operator desk*.

The plant, at the top of Figure 4, contains the *boiler* itself, a *pump* that supplies water to the boiler, *sensors* that measure the water level in the boiler and the steam flow out of the boiler, and an *actuator* that can start and stop the pump.

The controller, at the bottom of Figure 4, consists of three sub-modules: the *control program* processes the sensor values and determines the output to the actuator, and generally monitors the plant. It is responsible to detect unsafe conditions and faulty equipment and, if necessary, generate an emergency stop. The *maintenance communication* sub-module, `MaintCom`, keeps track of equipment status, based on input from the maintenance department and the central control program. The *operator communication* sub-module, `OperatorCom`, processes the input from the operator desk during the start-up phase.

Space considerations prohibit showing the entire system, but to illustrate our modular description language, Figure 5 shows the top-level composition of the various modules. There are eight basic modules: `Maintenance`, `MaintCom`, `Operator`, `OperatorCom`, `ControlProgram`, `Boiler`, `Pump` and `Environment`. The sensor and actuator are incorporated into the boiler and the pump, respectively.

The `MaintFun` module is the parallel composition of instances of the `MaintCom` and `Maintenance` modules. Multiple instances of the same module may be used in a composition. For example, the full system, module `BoilerSystem`, contains three instances of the `MaintFun` module, one for the steam flow sensor, one for the level sensor, and one for the pump. In each case the generic variables `equipmentDefective` and `equipmentState` are instantiated as the variable specific to the corresponding piece of equipment. The `Environment` module specifies how variables may be modified by the physical environment of the plant. Finally, we *close* the system by hiding all shared variables, indicating that their behavior is determined completely by the modules included in the system.

Modules can communicate with each other through shared variables or by synchronization of transitions. For example, the `Maintenance`, `Boiler`, `Pump` and `Operator` modules communicate via shared variables, as do the three parts of the `ControlProgram`, whereas the communication between the `Controller` and

```
Module MaintFun: (M: Maintenance) || (MC: MaintCom)

Module OperatorFun: (O: Operator) || (MO: OperatorCom)

Module BoilerSystem:
          Hide(  levelEqState, steamEqState, pumpEqState,
                 levelDefective, steamDefective, pumpDefective in
                 ( (C:ControlProgram) || (B:Boiler) || (P:Pump)
                    || Rename((SteamMaint:MaintFun)
                          equipmentDefective = steamDefective;
                          equipmentState     = steamEqState)
                    || Rename((LevelMaint:MaintFun)
                          equipmentDefective = levelDefective;
                          equipmentState     = levelEqState)
                    || Rename((PumpMaint:MaintFun)
                          equipmentDefective = pumpDefective;
                          equipmentState     = pumpEqState)
                    || (Ops: OperatorFun) || (E: environment)))
```

Fig. 5. Top-level modular specification of the Steam Boiler system

the other modules is solely via synchronized transitions. This reflects the fact that the controller only has access to the current plant data at the time that sensors are sampled.

4.2 System Analysis: Modularity and Inheritance

The modular structure of the system allows us to prove properties of the system at various levels. For example, we may want to prove the consistency between the internal states of the `Maintenance` and the `MaintCom` modules for all the pieces of equipment. It is attractive to prove this property over the `MaintFun` module and then let all of its instances in the full system inherit it, rather than proving it directly over the entire system; furthermore, since the `MaintFun` module is finite-state, we can use a model checker for the modular proof, whereas the full system contains real-valued variables, ruling out the use of a model checker within STeP.

The `MaintFun` module contains two shared variables, `equipmentDefective` and `equipmentState`. To be able to inherit properties, we must assume that these variables can be arbitrarily modified by the module's environment. However, to prove the consistency property we need a stronger assumption on the environment.

We provide two ways to state such assumptions: the most general way is to specify the property as an *assume-guarantee* property, of the form $\mathcal{A} \rightarrow \mathcal{G}$, where the assumption $\mathcal{A}$ is discharged upon parallel composition of the module with its environment. However, in some cases this leads to rather large, unintuitive temporal formulas. A second, weaker way to specify an assumption is to include

an explicit *environment restriction* in the module. This restriction becomes part of the environment transition when the modular property is proven. When a module with an environment restriction is composed, the transition relations of the composing modules are required to satisfy the restriction. We now show an example of each approach:

Environment Restriction: Specifying the consistency property as an assume-guarantee property is possible, but rather awkward, mainly because within a temporal formula it is hard to separate the actions of the module from those of its environment. On the other hand, we can include the assertion

$$(\texttt{equipmentState}' = \texttt{equipmentState}) \;\vee\; (\texttt{equipmentState}' = \texttt{broken})$$

as an environment restriction for the `MaintFun` module, stating that the environment can set `equipmentState` to `broken`, but cannot modify it in any other way. We can then prove the property directly over the `MaintFun` module. Subsequently, we ensure that the modules composed with `MaintFun` satisfy this property when building the full system.

Assume-Guarantee Reasoning: An example of a temporal property for which an assume-guarantee proof is well-suited is

$$\mathcal{G} : \Box(\texttt{equipmentDefective} \rightarrow \Diamond\, \texttt{equipmentState} = \texttt{inOrder}) \;,$$

stating that if a piece of equipment is defective then it will eventually be in working order again.

The sequence of events in an equipment failure are as follows: At an arbitrary time, the environment can set any equipment to be defective, resulting in either a faulty sensor reading, or the pump failing to start when requested. The `Controller` detects that the equipment is broken, and sets the corresponding equipment status to `broken`. The maintenance function of the `Controller` then informs the maintenance department, which acknowledges the report and repairs the equipment, eventually setting the equipment to `inOrder`. All of the steps involved in achieving the `inOrder` condition are controlled by the `MaintFun` module, except for the detection of the failure, which is done by the `ControlProgram` module. Thus we can specify the property $\mathcal{G}$ under assumption $\mathcal{A}$, as $\mathcal{A} \rightarrow \mathcal{G}$:

$$\begin{array}{c}\Box(\texttt{equipmentDefective} \rightarrow \Diamond\, \texttt{equipmentState} = \texttt{broken}) \\ \rightarrow \\ \Box(\texttt{equipmentDefective} \rightarrow \Diamond\, \texttt{equipmentState} = \texttt{inOrder})\end{array}$$

This implication can be model checked over the `MaintFun` module.

When the `MaintFun` module is instantiated, the full system inherits three assume-guarantee properties of the form $\mathcal{A}' \rightarrow \mathcal{G}'$:

$$\begin{array}{c}\Box(\texttt{steamDefective} \rightarrow \Diamond\, \texttt{steamEqState} = \texttt{broken}) \\ \rightarrow \\ \Box(\texttt{steamDefective} \rightarrow \Diamond\, \texttt{steamEqState} = \texttt{inOrder})\end{array}$$

for the steam sensor,

$$\begin{array}{c}\Box(\texttt{levelDefective} \rightarrow \Diamond\, \texttt{levelEqState} = \texttt{broken}) \\ \rightarrow \\ \Box(\texttt{levelDefective} \rightarrow \Diamond\, \texttt{levelEqState} = \texttt{inOrder})\end{array}$$

for the level sensor, and

$$\begin{array}{c}\Box(\texttt{pumpDefective} \rightarrow \Diamond\, \texttt{pumpEqState} = \texttt{broken}) \\ \rightarrow \\ \Box(\texttt{pumpDefective} \rightarrow \Diamond\, \texttt{pumpEqState} = \texttt{inOrder})\end{array}$$

for the pump. We then need separate proofs for each of the three assumptions, specific to the particular failure detection method of that piece of equipment.

4.3 System Analysis: Abstraction

As we saw in Section 3, abstraction can reduce an infinite-state system to a finite-state one by capturing relationships between infinite-domain variables, in the form of assertions. As an example of the use of abstraction, consider the property

$$\varphi : \Box \left(\begin{array}{l} \quad \texttt{steamEqState} = \texttt{inOrder} \;\wedge\; \neg\texttt{steamDefective} \\ \rightarrow \;\; (\texttt{steamEqState} = \texttt{inOrder}\; \mathcal{W}\; \texttt{steamDefective}) \end{array} \right) ,$$

stating that as long as the steam flow sensor is not defective its status will be `inOrder`. That is, the controller will not detect a failure in nondefective equipment. This property is certainly desirable, since the failure status of a piece of equipment may cause a plant shutdown. We will prove it over the full system.

To check whether the steam flow sensor is operating correctly, at each cycle the controller predicts the range of possible sensor readings for the next reading, based on a minimum and maximum assumed gradient in the flow. If the reading is outside this range, it is considered defective, and the controller will set its status to `broken`.

Although the property involves only finite-domain variables, its validity is obviously dependent on real-valued variables such as the actual and predicted steam flow. However, the property does not depend on the particular values of these variables, but only on certain relationships between them. The following assertion basis is sufficient to prove the property:

$$\begin{array}{ll}
b_1 : & \texttt{steamflow} = \texttt{C.s} \\
b_2 : & \texttt{C.s} = \texttt{B.sf} \\
b_3 : & \texttt{C.s} \geq \texttt{C.sPredLow} \\
b_4 : & \texttt{C.s} \leq \texttt{C.sPredHigh} \\
b_5 : & \texttt{C.s} \leq \texttt{B.sf} - \texttt{mingrad} * \texttt{delta} \\
b_6 : & \texttt{C.s} \geq \texttt{B.sf} - \texttt{maxgrad} * \texttt{delta} \\
b_7 : & \texttt{C.s} = \texttt{C.sLow} \\
b_8 : & \texttt{C.s} = \texttt{C.sHigh} \\
b_9 : & \texttt{C.sPredLow} = \texttt{C.s} + \texttt{mingrad} * \texttt{delta} \\
b_{10} : & \texttt{C.sPredHigh} = \texttt{C.s} + \texttt{maxgrad} * \texttt{delta}
\end{array}$$

Here, `steamflow` is the sensor reading, `C.s` is the local value of the steam flow within the controller, and `B.sf` is the actual steam flow going out of the boiler. Parameters `mingrad` and `maxgrad` are the minimum and maximum gradients of the steam flow, and `delta` is the sampling interval. We assume that $\texttt{mingrad} < 0$, $\texttt{maxgrad} > 0$, and $\texttt{delta} > 0$.

The addition of these variables to the system allows us to remove all real-valued variables and construct a finite-state abstraction that contains sufficient information to prove the property.

Abstraction Test Points: When constructing abstract transition relations, the algorithm of [CU98] uses a set of *test points*, built from the abstraction basis $\{b_1, \ldots, b_n\}$, to determine the effect of a transition on the truth value of the basis elements under different circumstances. By default, the test points used are of the form $\{p_1 \rightarrow p_2'\}$, where p_1 and p_2 are basis elements or their negation. When generating the abstraction, a validity checker is used, in essence, to check the implication

$$p_1 \wedge \ \rho_\tau \rightarrow p_2'$$

for every transition (compositionally over the structure of the formula that describes the transition relation); if valid, the implication is added to the abstracted transition relation.

These default test points are enough to generate the abstractions described in Section 3. However, in some cases a more precise abstraction is required to prove the desired property, which could have been obtained if more complex relationships between basis elements had been explored.

Thus, our implementation allows additional test points to be specified explicitly for particular transitions, letting p_1 and p_2' above be general boolean combinations of basis elements. This has the effect of refining the abstracted system, producing an abstraction for which more properties can be proved.

To illustrate the abstraction process, some examples of concrete and corresponding abstract transition relations are given below. The evolution of the physical system, modeled by the concrete relation

$$\rho^C_{Ev} : \texttt{B.sf}' \geq \texttt{B.sf} + \texttt{mingrad} * \texttt{delta} \ \wedge \ \texttt{B.sf}' \leq \texttt{B.sf} + \texttt{maxgrad} * \texttt{delta}$$

is abstracted to

$$\rho^A_{Ev} : (b_2 \rightarrow b_5' \wedge b_6') \wedge \mathit{preserve}(\{b_1, b_3, b_4, b_7, b_8, b_9, b_{10}\}) \ ,$$

where $\mathit{preserve}(S)$ stands for $\wedge_{x \in S}(x' = x)$.[2]

The two transitions involved in the prediction of the acceptable steam flow range are

$$\rho^C_{P1} : \begin{array}{l} \texttt{C.steamReliable} \ \wedge \ \texttt{C.sLow}' = \texttt{C.s} \ \wedge \ \texttt{C.sHigh}' = \texttt{C.s} \\ \vee \ (\neg \texttt{C.steamReliable} \wedge \ \ldots) \end{array}$$

[2] We only include the transition relation fragments relevant to the abstraction; in particular, finite-state control variables that are retained are not included. We plan to make the entire concrete and abstract systems available elsewhere.

and

$$\rho^C_{P2} : \begin{array}{l} \texttt{C.sPredLow}' = \texttt{C.sLow} + \texttt{mingrad} * \texttt{delta} \\ \wedge\ \texttt{C.sPredHigh}' = \texttt{C.sHigh} + \texttt{maxgrad} * \texttt{delta}\ . \end{array}$$

They are abstracted to

$$\rho^A_{P1} : \begin{array}{l} \texttt{C.steamReliable} \rightarrow\ b_7' \wedge b_8' \wedge \mathit{preserve}(\{b_1, \ldots, b_6, b_9, b_{10}\}) \\ \wedge\ \neg\texttt{C.steamReliable} \rightarrow \ldots \end{array}$$

and

$$\rho^A_{P2} : (b_7 \rightarrow\ b_3' \wedge b_9') \wedge (b_8 \rightarrow\ b_4' \wedge b_{10}' \wedge \mathit{preserve}(\{b_1, b_2, b_5, b_6, b_7, b_8\})\ .$$

For these transition relations, the default test points suffice. The transition that models the sensor sampling, `BoilerSensorsC`, is a synchronized transition between the `Boiler` and the `ControlProgram` modules, with transition relation

$$\rho^C_S : \begin{array}{l} \texttt{C.s}' = \texttt{steamflow}' \wedge \\ \texttt{steamflow}' = \texttt{if steamDefective then outofrange else B.sf}\ . \end{array}$$

This transition requires an extra test point in order to establish a sufficiently strong postcondition, namely:

$$\{\neg\texttt{steamDefective} \wedge b_5 \wedge b_6 \wedge b_9 \wedge b_{10}\}\, \texttt{BoilerSensorsC}\, \{b_3 \wedge b_4\}\ .$$

This results in the abstract transition relation

$$\rho^A_S : \begin{array}{l} b_1' \wedge (\neg\texttt{steamDefective} \rightarrow b_2' \wedge b_5' \wedge b_6') \wedge \\ (\neg\texttt{steamDefective} \wedge b_5 \wedge b_6 \wedge b_9 \wedge b_{10} \rightarrow b_1' \wedge b_2' \wedge b_3' \wedge b_4')\ . \end{array}$$

This abstraction allows us to prove the desired property φ above.

5 Conclusions and Related Work

We have shown how abstraction and modularity allow for more automatic and incremental verification of reactive systems. Deductive methods allow the verification and abstraction of infinite-state systems, including the unbounded data types used in software systems.

Clearly, much has to be done before these techniques are practical for large-scale software system design. In practice, a combination of formal and informal methods is required. However, we believe that abstraction, refinement and modularity will be useful in all of these settings.

Proving simple properties can help debug systems while they are being designed. The abstraction and verification of individual modules can be regarded as a "lightweight" formal method, in the sense of [JW96], which can be used before moving on to more "heavyweight" ones. Initial negative results can help debug the system; positive results will establish simple properties that will be useful in more complex, global proofs.

In the important special case of hybrid and real-time systems, untimed components can be isolated and debugged using model checking, and timed components can be abstracted to model checkable ones.

Related Work

Abstraction and Deductive Verification: The methods for *automatic invariant generation* presented in [BBM97] are a special case of abstraction, where abstract interpretation is carried out using pre-defined abstract domains for which fixpoints (or their approximations) can be efficiently computed. These methods are implemented in STeP, automatically generating local, linear, and polyhedral invariants, depending on the abstract domain used.

Verification diagrams [MP94,BMS95] provide a visual representation of the proof of the system validity of particular temporal properties. *Deductive model checking* [SUM98] interactively explores and refines an abstraction of the system state-space in search for a counterexample. Both of these verification methods can be seen as providing an appropriate assertion-based abstraction, when they succeed. Furthermore, they incorporate well-founded domains, for those cases where a finite-state abstraction does not exist.

Abstraction, Modularity and Model Checking: A procedure that explicitly generates an abstract state-space for an assertion-based abstraction, similar to our abstraction algorithm, is presented in [GS97]; another automatic abstraction procedure that uses validity checking is presented in [BLO98].

In [HLP98], a system specified in LISP code is abstracted, manually, to a model-checkable finite-state system, uncovering significant flaws in the original design. [DGH95] investigates the separation of control and data in infinite-state systems, combining model checking with the generation of verification conditions that are established deductively. [Lon93,CGL94] show how abstraction and modularity can be combined for finite-state systems that are synchronously composed and symbolically model checked.

Refinement: In general, refinement can be seen as the dual of abstraction, and used as a formal system design methodology [dBdRR90,KMP94]: first, a high-level version of the algorithm can be verified to meet the desired specifications. Then, implementation details can be added to the system, while ensuring that the desired properties still hold.

STeP: [BMSU97] presents the modular specification and verification of the well-known *generalized railroad crossing* real-time case study. Invariants are proved, or automatically generated, separately for each module. They are then used to prove properties for combinations of modules which, in turn, are used to prove the desired properties for the entire system.

Other STeP test cases are reported in [BLM97,MS98]. For more information, including abstraction and verification examples, see the STeP web pages at:

`http://www-step.stanford.edu/`

Acknowledgements: We thank Nikolaj Bjørner and Anca Browne for their comments.

References

[ABL96] J.R. Abrial, E. Boerger, and H. Langmaack, editors. *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, vol. 1165 of *LNCS*. Springer-Verlag, 1996.

[AHS96] R. Alur, T.A. Henzinger, and E.D. Sontag, editors. *Hybrid Systems III: Verification and Control*, vol. 1066 of *LNCS*. Springer-Verlag, 1996.

[BBC+95] N.S. Bjørner, A. Browne, E.S. Chang, M. Colón, A. Kapur, Z. Manna, H.B. Sipma, and T.E. Uribe. STeP: The Stanford Temporal Prover, User's Manual. Technical Report STAN-CS-TR-95-1562, Computer Science Department, Stanford University, November 1995.

[BBC+96] 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 R. Alur and T.A. Henzinger, editors, *Proc.* 8th *Intl. Conference on Computer Aided Verification*, vol. 1102 of *LNCS*, pages 415–418. Springer-Verlag, July 1996.

[BBM97] N.S. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, February 1997. Preliminary version appeared in 1st *Intl. Conf. on Principles and Practice of Constraint Programming*, vol. 976 of LNCS, pp. 589–623, Springer-Verlag, 1995.

[BLM97] N.S. Bjørner, U. Lerner, and Z. Manna. Deductive verification of parameterized fault-tolerant systems: A case study. In *Intl. Conf. on Temporal Logic*. Kluwer, 1997. To appear.

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In Hu and Vardi [HV98], pages 319–331.

[BMS95] A. Browne, Z. Manna, and H.B. Sipma. Generalized temporal verification diagrams. In *15th Conference on the Foundations of Software Technology and Theoretical Computer Science*, vol. 1026 of *LNCS*, pages 484–498. Springer-Verlag, 1995.

[BMSU97] N.S. Bjørner, Z. Manna, H.B. Sipma, and T.E. Uribe. Deductive verification of real-time systems using STeP. In *4th Intl. AMAST Workshop on Real-Time Systems*, vol. 1231 of *LNCS*, pages 22–43. Springer-Verlag, May 1997.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In 4th *ACM Symp. Princ. of Prog. Lang.*, pages 238–252. ACM Press, 1977.

[CE81] 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*, pages 52–71. Springer-Verlag, 1981.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Trans. on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CU98] M.A. Colón and T.E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In Hu and Vardi [HV98], pages 293–304.

[Dam96] D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, July 1996.

[dBdRR90] J.W. de Bakker, W.P. de Roever, and G. Rosenberg, editors. *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, vol. 430 of *LNCS*. Springer-Verlag, 1990.

[DGG94] D.R. Dams, O. Grümberg, and R. Gerth. Abstract interpretation of reactive systems: Abstractions preserving ∀CTL*, ∃ECTL*, CTL*. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET 94)*, pages 573–592, June 1994.

[DGH95] W. Damm, O. Grümberg, and H. Hungar. What if model checking must be truly symbolic. In *First Intl. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 95)*, vol. 1019 of *LNCS*, pages 230–244. Springer-Verlag, May 1995.

[DKRT97] P.R. D'Argenio, J.P. Katoen, T. Ruys, and G.T. Tretmans. The bounded retransmission protocol must be on time! In *3rd Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, vol. 1217 of *LNCS*, pages 416–432. Springer-Verlag, 1997.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In Alur et al. [AHS96], pages 208–219.

[FMS98] B. Finkbeiner, Z. Manna, and H.B. Sipma. Deductive verification of modular systems. In *International Symposium on Compositionality, COMPOS'97*, LNCS. Springer-Verlag, 1998. To appear.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th Intl. Conference on Computer Aided Verification*, vol. 1254 of *LNCS*, pages 72–83. Springer-Verlag, June 1997.

[HH95] T.A. Henzinger and P. Ho. HYTECH: The Cornell hybrid technology tool. In *Hybrid Systems II*, vol. 999 of *LNCS*, pages 265–293. Springer-Verlag, 1995.

[HLP98] K. Havelund, M. Lowry, and J. Penix. Formal verification of a space craft controller. Technical report, NASA Ames Research Center, 1998.

[HS96] K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe*, pages 662–681, March 1996.

[HV98] A.J. Hu and M.Y. Vardi, editors. *Proc. 10th Intl. Conference on Computer Aided Verification*, vol. 1427 of *LNCS*. Springer-Verlag, June 1998.

[JW96] D. Jackson and J. Wing. Lightweight formal methods. *IEEE Computer*, April 1996.

[KMP94] Y. Kesten, Z. Manna, and A. Pnueli. Temporal verification of simulation and refinement. In J.W. de Bakker, W.P. de Roever, and G. Rosenberg, editors, *A Decade of Concurrency*, vol. 803 of *LNCS*, pages 273–346. Springer-Verlag, 1994.

[KMP96] Y. Kesten, Z. Manna, and A. Pnueli. Verifying clocked transition systems. In Alur et al. [AHS96], pages 13–40.

[LGS$^+$95] 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.

[Lon93] D.E. Long. *Model Checking, Abstraction, and Compositional Verification*. PhD thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, July 1993.

[MBB$^+$98] Z. Manna, N.S. Bjørner, A. Browne, M. Colón, B. Finkbeiner, M. Pichora, H.B. Sipma, and T.E. Uribe. An update on STeP: Deductive-algorithmic verification of reactive systems. In *Tool Support for System Specification, Development and Verification*, pages 87–91. Christian-Albrechts-Universitat, Kiel, June 1998. Bericht Nr. 9803.

[McM93] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Pub., 1993.

[MP94] Z. Manna and A. Pnueli. Temporal verification diagrams. In M. Hagiya and J.C. Mitchell, editors, *Proc. International Symposium on Theoretical Aspects of Computer Software*, vol. 789 of *LNCS*, pages 726–765. Springer-Verlag, 1994.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[MP96] Z. Manna and A. Pnueli. Clocked transition systems. Technical Report STAN-CS-TR-96-1566, Computer Science Department, Stanford University, April 1996.

[MS98] Z. Manna and H.B. Sipma. Deductive verification of hybrid systems using STeP. In T. Henzinger and S. Sastry, editors, *Hybrid Systems: Computation and Control*, vol. 1386 of *LNCS*, pages 305–318. Springer-Verlag, 1998.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. Found. of Comp. Sci.*, pages 46–57. IEEE Computer Society Press, 1977.

[QS82] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Intl. Symposium on Programming*, vol. 137 of *LNCS*, pages 337–351. Springer-Verlag, 1982.

[SUM98] H.B. Sipma, T.E. Uribe, and Z. Manna. Deductive model checking. To appear in *Formal Methods in System Design*, 1998. Preliminary version appeared in *Proc.* 8th *Intl. Conference on Computer Aided Verification*, vol. 1102 of *LNCS*, Springer-Verlag, pp. 208–219, 1996.