# A Platform for Combining Deductive with Algorithmic Verification*

Amir Pnueli† Elad Shahar†

**Abstract.** We describe a computer-aided verification system which combines deductive with algorithmic (model-checking) verification methods. The system, called TLV (for temporal verification system), is constructed as an additional layer superimposed on top of the CMU SMV system, and can verify finite-state systems relative to *linear temporal logic* (LTL) as well as CTL specifications. The systems to be verified can be either hardware circuits written in the SMV design language or finite-state reactive programs written in a simple programming language (SPL).

The paper presents a common computational model which can support these two types of applications and a high-level interactive language TLV-BASIC, in which temporal verification rules, proofs, and complex assertions can be written. We illustrate the efficiency and generality gained by combining deductive with algorithmic techniques on several examples, culminating in verification of fragments of the Futurebus+ system. In the analysis of the Futurebus+ system, we even managed to detect a bug that was not discovered in a previous model-checking analysis of this system.

## 1 Introduction

As part of the general program for combining deductive with algorithmic methods for the verification of reactive systems (see [Man94] for a declaration of this manifest, and [RSS95] for an important contribution in this direction), we constructed a computer-aided verification system, called TLV (a *Temporal Logic Verifier*), for experimenting with some of these ideas.

Compared to algorithmic verification (model checking), deductive verification is handicapped by the requirement of user interaction, which necessitates a good understanding of the program and a certain degree of creative ability and high skills. Therefore, any proposal for replacing or even combining algorithmic methods with deductive methods must be accompanied by analysis of the expected gains from such a combination.

The main conceived advantages of combining deduction with model checking are:

**1. Generality** : In the finite-state world (which is the main concern of the work reported here), deductive verification can provide a *uniform* proof which establishes the correctness of a system of $N$ processes for any $N > 0$ in a single

* This research was supported in part by a basic research grant from the Israeli Academy of Sciences, and by the European Community ESPRIT Basic Research Action Project 6021 (REACT).

† Department of Computer Science, Weizmann Institute, Rehovot, Israel, e-mail: amir@wisdom.weizmann.ac.il

proof. In comparison, model checking can only examine the systems for particular values of $N$.

**2. Efficiency of Deduction**: Most of the model-checking algorithms are based on computation of the closure of the transition relation, which is applied either to the initial state or to some target states. This is an iterative process that may take a large number of steps to converge. In comparison, in the deductive verification of the same property, we only have to check the two implications

$$\Theta \rightarrow p \qquad \text{and} \qquad \rho \wedge p \rightarrow p',$$

where $\Theta$ is an assertion characterizing the initial condition and $\rho$ is the transition relation. It stands to reason that checking these implications takes less time and requires smaller BDDs than the iterative computation of the closure.

**3. Constrained model checking** : A possible way of combining deduction with model checking is to use deduction to establish the invariance of an assertion $\varphi$. Then, we can carry out regular model checking but use $\varphi$ to restrict the range of considered states. This amounts to model checking with the transition relation $\varphi \wedge \rho$ instead of the original $\rho$.

The (TLV) system described here has been constructed on top of the CMU SMV system, which supports verification of CTL specifications of finite-state systems ([BCM+92], [McM93]). TLV uses the BDD library and the SMV input language parser from SMV. The model checking algorithms were replaced by a layer which consists of a high-level interactive language, to which we refer as TLV-BASIC. The main data structure of TLV-BASIC is a quantifier-free assertion, obeying the SMV syntax for state-formulas, and represented internally by a BDD.

The TLV-BASIC language is used for three purposes:

- Temporal verification rules, such as the basic invariance rule BINV and the single-step response rule RESP, as well as algorithms for model-checking invariance and response properties, are written as TLV-BASIC procedures.
- For each particular system to be verified, the user usually prepares a *proof script* file which contains definitions of the assertions used in the property to be verified.
- The interactive dialog with the user is carried out in a restricted subset of TLV-BASIC.

The main running example and one of the motivating drives for our system is the Futurebus+ system considered in [CGH+93]. That paper presented an SMV model for the Futurebus+ system and established several properties of the model, using the model-checking techniques of SMV. We considered it an interesting challenge to see whether the same properties can be verified using deductive techniques, and compare the efficiency and effectiveness of the two methods.

At its current state of implementation, the TLV system cannot yet consider variable-size systems where the system size is not fixed at analysis time. Therefore, we cannot yet demonstrate uniform proofs of such parameterized systems, and all the examples presented in this paper relate to specific values of the size parameter. To compensate for this temporary deficiency, we developed methods

by which the deductive proof of a parametric system can be parameterized itself, so that running a deduction for different values of the size parameter $n$ only requires modifying a line in the proof script file from "$n = 20$" to, say, "$n = 40$." In particular, we developed a special format by which one can specify an arbitrary configuration of Futurebus+ and generate automatically the proof appropriate for this configuration. Details about these instantiation mechanisms are given in [PS96].

Many approaches to the deductive verification of reactive systems and hardware circuits were proposed over the years, accompanied by systems supporting their automation. Examples of applications for hardware verification are the methods described in [Gor86] and [ORSS94]. An effective system for the deductive verification of linear temporal logic properties of reactive programs is reported in [MAB+94].

There have been also several approaches which combine deductive and algorithmic verification methods. The work in [JS93] combines the HOL theorem prover with the Voss system. Another combination of methodologies is reported in [KL93], where TLP, the proof checker for TLA, the temporal logic of actions, is combined with the COSPAN verifier. Perhaps closest to our work is [RSS95] which embeds symbolic model-checking into the PVS high-order prover.

The unique feature of our approach is that it is built as the minimal extension of an existing symbolic model checking system (SMV) needed in order to handle parametric systems. The specification language and associated deductive verification approach are based on linear temporal logic [MP95]. At present, the only deductive machinery we employ is provided by the BDD capabilities of the underlying SMV system.

The rest of the paper is organized as follows. In Section 2 we present the underlying computational model and its relation to the FTS model of [MP95]. In Section 3, we describe the languages that can serve as inputs to the TLV system. These include the TLV-BASIC language in which verification rules, model-checking procedures, and proof scripts are written; the SMV input languages used to specify systems; and the SPL language used to describe simple reactive programs [MP95]. In Section 4, we present some of the verification rules supported by the system. Section 5 presents several simple examples of deductive and combined verification, comparing their efficiency with standard model-checking verification of the same properties. In Section 6, we present our main case study, the Futurebus+ verification, and identify the bug that has escaped previous model-checking analysis.

## 2 The Computational Model

As an underlying computational model, we adopt the notion of an *always-enabled fair transition system* (ETS). The ETS model can be viewed as a variant of the *fair transition system* (FTS) model, introduced in [MP91] for the specification and verification of reactive systems. An ETS consists of the following components:

- $\mathcal{V}$ — A finite set of *state variables*. We define a state to be an interpretation of $\mathcal{V}$. The set of all states is denoted by $\Sigma$.

- $\Theta$ — Initial condition.
- $\mathcal{T}$ — A finite set of *transitions*. Each transition $\tau \in \mathcal{T}$ is a function
  $\tau : \Sigma \mapsto 2^{\Sigma} - \emptyset$,
  mapping a state $s$ to $\tau(s) \subseteq \Sigma$, a non-empty set of $\tau$-*successors* of $s$.
- $\mathcal{J} \subseteq \mathcal{T}$ — A *justice* (weak fairness) set of transitions.
- $\mathcal{C} = \{(\tau_1, \varphi_1), \ldots, (\tau_k, \varphi_k)\}$ — A *compassion* (strong fairness) set of pairs $(\tau_i, \varphi_i)$, $i = 1, \ldots, k$, each consisting of a transition $\tau_i$ and an assertion $\varphi_i$.

The requirement that every state has a non-empty set of successors implies that every transition is enabled on every state.

A *model* is an infinite sequence of states. Given an ETS $\Phi$, we define a *computation of* $\Phi$ to be a model

$$\sigma : s_0, s_1, s_2, \ldots,$$

satisfying the following requirements:

- *Initiation*: $s_0$ is an initial state (i.e it satisfies $\Theta$).
- *Consecution*: For each pair of consecutive states $s_i, s_{i+1}$ in $\sigma$, there exists a transition $\tau$ in $\mathcal{T}$ such that $s_{i+1} \in \tau(s_i)$. That is, $s_{i+1}$ is a $\tau$-successor of $s_i$.
- *Justice*: Every transition $\tau \in \mathcal{J}$ is taken infinitely many times.
- *Compassion*: For every $(\tau_i, \varphi_i) \in \mathcal{C}$, if $\varphi_i$ holds at infinitely many positions in $\sigma$ then $\tau_i$ is taken at $\varphi_i$-positions infinitely many times.

The main differences between the FTS and ETS models are the ETS requirement that transitions be always enabled, and the implications this requirement has on the requirements of justice and compassion.

The reason for this difference is that the natural SMV representation of transition relations, in particular those which result from SPL programs, is such that the transition can always be taken. Under the circumstances in which the corresponding FTS transition would be disabled, the ETS transition is still enabled but has no effect on the system variables, i.e., it changes the value of no system variable.

An FTS $\Phi$ is called a *leisurely fair transition system* (LFTS), if the idling transition $\tau_I$ is contained in the justice set of $\Phi$. Thus, every computation of an LFTS contains infinitely many idling steps, i.e. steps which preserve the values of all system variables. Obviously, every FTS $\Phi$ has a corresponding LFTS $\Psi$, such that $\Phi$ and $\Psi$ are equivalent up to stuttering.

The following claim shows that no expressive power is lost in moving from the FTS model into the ETS model.

**Claim 1** *A set of models $S$ is the set of computations of an* ETS $\Phi$ *iff it is the set of computations of some* LFTS $\Psi$.

In [PS96], we provide a proof of this claim.

# 3 The Languages of TLV

## 3.1 The SMV Input Language

Systems to be verified by TLV are described using the SMV input language [McM93], which has been slightly extended to allow for the richer set of fairness

requirements associated with the ETS model. In Fig. 1, we present file **sem.smv**, which contains the SMV description of a mutual exclusion algorithm MUX-SEM, which implements mutual exclusion by semaphores. Note that, standardly in our

```
MODULE main
VAR
  y : boolean; -- the semaphore variable. It is assigned by both processes.
  proc[1] : process user(y); -- The two processes have interleaved execution.
  proc[2] : process user(y);
ASSIGN
  init(y) := 1;
MODULE user(y)
VAR
  loc : {0,1,2,3,4};
ASSIGN
  init(loc) := 0;
  next(loc) :=
    case
      loc in {0,3}     : loc+1;
      loc = 1          : {1,2};
      loc = 2 & y = 1 : 3;
      loc = 4          : 0;
      1 : loc;
    esac;
  next(y) :=                          -- changes to the semaphore variable.
    case
      loc = 2 & next(loc) = 3 : 0; -- turned off when moving from l_2 to l_3
      loc = 4 & next(loc) = 0 : 1; -- turned on when moving from l_4 to l_0
      1 : y;
    esac;
JUSTICE
  proc[1], proc[2];
COMPASSION
  (proc[1],proc[1].loc = 2 & y > 0), (proc[2],proc[2].loc = 2 & y > 0)
```

**Fig. 1.** File **mux-sem.smv**: an SMV description of Algorithm MUX-SEM for $n = 2$ processes.

applications, we do not use the **FAIR** or **SPEC** declarations but introduce instead **JUSTICE** or **COMPASSION** declarations, wherever necessary.

Such an SMV specification is input into the TLV system which creates internally the ETS corresponding to the specification. In general, there will be one ETS transition for each process. Thus, in the **mux-sem.smv** example, the system will generate an ETS with two transitions, one corresponding to each process. The justice requirement requests that each of the two processes will be activated infinitely many times in every computation of this ETS.

### 3.2 The SPL Input Language

While direct coding of hardware circuits in the SMV input language is a practice to which experienced users of the SMV system have resigned themselves, we can offer a higher description level for applications to reactive programming. To

represent reactive programs, we adopted the *simple programming language* (SPL) introduced in [MP91]. We refer the reader to [MP91] or [MP95] for details of this language. In Fig. 2, we present an SPL program for the MUX-SEM algorithm.

Here, we consider the instance $n = 2$ of this generic program. On reading the SPL file with the additional definition $n := 2$, the system translates it first into the SMV representation, presented in Fig. 1.

$$\begin{array}{l} \textbf{in} \quad n : \textbf{integer where } n > 0 \\ \textbf{local } y : \textbf{integer where } y = 1 \end{array}$$

$$\Big\|_{i=1}^{n} C[i] :: \left[\begin{array}{l} \ell_0 : \textbf{loop forever do} \\ \left[\begin{array}{l} \ell_1 : \textbf{noncritical} \\ \ell_2 : \textbf{request } y \\ \ell_3 : \textbf{critical} \\ \ell_4 : \textbf{release } y \end{array}\right] \end{array}\right]$$

**Fig. 2.** Program MUX-SEM (mutual exclusion by semaphores - general case).

### 3.3 TLV-BASIC

The TLV-BASIC language is easy to learn and simple to program with. It is used to program rules, model-checking algorithms, and compute assertions. The main (and only) data structure is a function with boolean arguments and integer range. As such, it can represent integers, booleans (a function with range $\{0, 1\}$, and assertions, which are represented as boolean functions. The underlying implementation is a BDD, which is manipulated using the SMV BDD library. Expressions in the language are constructed out of integer constants and variables to which we apply integer operations, integer comparisons, and all the boolean and quantifying operators available in the SMV language.

There are no variable declarations. Like BASIC, variables are created dynamically, whenever they are assigned values, or mentioned as parameters of a procedure. In addition, all the variables defined in an SMV input file which is loaded into the system can be referenced within TLV-BASIC expressions.

Following are some of the statements available in TLV-BASIC:

- `Let` *var* `:=` *exp* — Assign the value of expression *exp* to variable *var*.
- `Proc` *proc-name* $(par_1, \ldots, par_n)$; $S$ `End` — Define a procedure *proc-name* with parameters $par_1, \ldots, par_n$ and body $S$.
- `While (`*exp*`)` $S$ `End` — Repeatedly execute statement $S$ until *exp* is 0.
- `If (`*exp*`)` $S_1$ [ `else` $S_2$ ] `End` — If *exp* evaluates to a non-zero value, execute statement $S_1$. Otherwise, execute statement $S_2$.
- `Load "`*file-name*`"` — Load file *file-name* into the system. The loaded file can be a rules file or a proof script file.
- `Run` *proc-name* $par_1, \ldots, par_n$ — Invoke procedure *proc-name* with the given actual parameters.

The last two statements are the main commands that are used in interactive mode.

In Fig. 3 we present a TLV-BASIC proof script which computes the assertion

$$\texttt{mux:}\quad \bigwedge_{i=1}^{n}\bigwedge_{j=1}^{i-1}\neg(\texttt{proc[i].loc} = 3\ \&\ \texttt{proc[j].loc} = 3).$$

for $n = 10$. This assertion specifies mutual exclusion for program MUX-SEM. When we consider the same program for a different number of processes, say 11,

```
Let n := 10;
Proc prepare;
  Let mux := TRUE;
  Let i := n;
  While (i)
    Let j := i - 1;
    While (j)
      Let mux := mux & !(proc[i].loc = 3 & proc[j].loc = 3);
      Let j := j - 1;
    End -- end loop on j
    Let i := i - 1;
  End -- end loop on i
End -- end procedure

Run prepare
```

**Fig. 3.** File `mux-sem.pf`: Proof script for program MUX-SEM for $n = 10$.

it is only necessary to change the first statement in this file to `Let n := 11`.

## 4 Verification Rules

The TLV system comes equipped with a set of deductive verification rules as well as various model-checking algorithms. As previously explained, these rules are implemented using the TLV-BASIC language. This means that a sophisticated user can easily modify any of the existing rules, as well as write new ones.

In Fig. 4, we present the two verification rules that have been used for verifying the examples presented in this paper.

$$\frac{\begin{array}{ll}\text{B1} : \Theta \rightarrow p & \\ \text{B2} : \rho_\tau \wedge p \rightarrow p' & \forall \tau \in \mathcal{T}\end{array}}{\Box p}$$

Rule BINV

$$\frac{\begin{array}{l}\text{A1} : \varphi \wedge \delta = 0 \rightarrow q \\ \text{A2} : (\varphi \wedge \neg q) \rightarrow \exists \tau \in \mathcal{T}\, \exists V' (\rho_\tau \rightarrow \delta \succ \delta')\end{array}}{\mathbf{AG\ EF}\ q}$$

Rule AGEF

**Fig. 4.** Verification rules.

## 5 Simple Verification Examples

In this section we illustrate the use of the TLV system for the verification of several simple examples taken from [MP95].

**Program** MUX-SEM

In Fig. 2, we presented the general MUX-SEM program parameterized by $n$. Fig. 1 illustrated its SMV translation for the case $n = 2$. The main safety property of this program can be specified by the invariance of assertion **mux** presented above.

Direct application of rule BINV failed (and produced a counter-example). According to the terminology of [MP95], this means that assertion **mux** is invariant but not *inductive*, i.e., it does not carry sufficient information to rule out inaccessible states. The standard remedy is to *strengthen* assertion **mux** by additional invariants, which will exclude such states.

Indeed, our next step in the verification process, was to formulate the auxiliary invariant assertion

$$\texttt{phi:}\quad \texttt{y} \quad <-> \quad \bigwedge_{i=1}^{n} \neg(\texttt{proc[i].loc in \{3,4\}})$$

Application of rule BINV to the conjunction **mux & phi** succeeded which established the invariance of both **mux** and **phi** over program MUX-SEM.

This experimentation was carried out for the low value of $n = 2$. However, once the strategy was established we prepared a proof script for computing the conjunction **mux & phi** and can now run the verification for various values of $n$, changing only the value of the parameter between successive runs.

To compare the time and space complexity of conventional model checking and the deductive approach, we plot in Fig. 5 the time and space complexity of verifying the invariance of assertion **mux** by the two approaches for increasing number of processes in program MUX-SEM. The line labeled SMV represents the conventional model-checking approach, while the line labeled TLV represents the deductive approach.

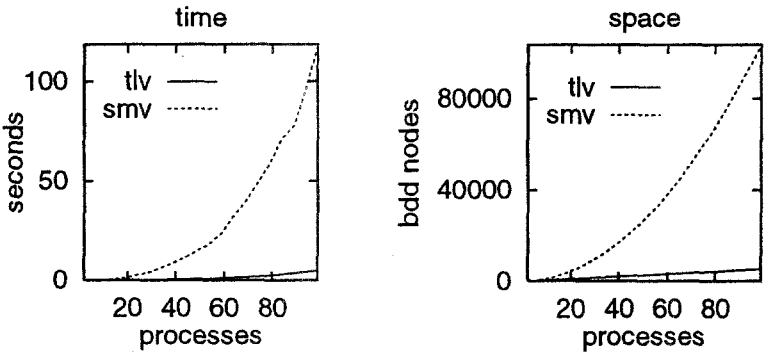


**Fig. 5.** Comparison of SMV and TLV for MUX-SEM

**Program** RES-SV

As the next example, we considered program RES-SV, presented in Fig. 6. Program RES-SV consists of an alloator process $A$ and customer processes $C[i]$, $i = 1, \ldots, n$. The allocator provides a centralized control which is exptected to guarantee mutual exclusion between the customers. We refer readers to [MP95] for

$$\begin{array}{l}\textbf{in} \quad n \quad : \textbf{integer where } n > 0\\ \textbf{local } g, r : \textbf{array } [1..n] \textbf{ of boolean where } g = \mathbf{F},\ r = \mathbf{F}\end{array}$$

$$A :: \left[\begin{array}{l}\textbf{local } t : \textbf{integer where } t = 1\\ m_0 : \textbf{loop forever do}\\ \left[\begin{array}{l} m_1 : \textbf{if } r[t] \textbf{ then}\\ \left[\begin{array}{l} m_2 : g[t] := \mathbf{T}\\ m_3 : \textbf{await } \neg r[t]\\ m_4 : g[t] := \mathbf{F}\end{array}\right]\\ m_5 : t := t \oplus_n 1\end{array}\right]\end{array}\right] \quad \| \quad \|_{i=1}^{n} C[i] :: \left[\begin{array}{l}\ell_0 : \textbf{loop forever do}\\ \left[\begin{array}{l}\ell_1 : \textbf{noncritical}\\ \ell_2 : r[i] := \mathbf{T}\\ \ell_3 : \textbf{await } g[i]\\ \ell_4 : \textbf{critical}\\ \ell_5 : r[i] := \mathbf{F}\\ \ell_6 : \textbf{await } \neg g[i]\end{array}\right]\end{array}\right]$$

**Fig. 6.** Program RES-SV (resource allocator).

details of this algorithm and its verification. Here we set to ourselves the more modest goal of verifying mutual exclusion between customers $C[1]$ and $C[2]$ in a system of $n \geq 2$ customers.

This property can be specified as the invariance of the assertion

$$\texttt{mux}: \quad \neg(at\_\ell_4[1] \wedge at\_\ell_4[2]),$$

where, for any $i$ and $j$, $at\_\ell_i[j]$ stands for `C[j].loc = i`.

As in the previous case, assertion `mux` is an invariant of the program but is not inductive. To complete the proof, we used six strengthening assertions for $i \in \{1, 2\}$. The first two assertions of this set are:

$$\begin{array}{ll}\varphi_1[i] : & at\_m_{3,4} \wedge t = i \leftrightarrow g[i]\\ \varphi_2[i] : & at\_\ell_{3..5} \quad \leftrightarrow r[i]\end{array}$$

Using these strengthening invariants, assertion `mux` has been proven an invariant of program RES-SV.

In Fig. 7, we plot the time and space complexity of verifying the invariance of assertion `mux` over program RES-SV as a function of the number of processes. Again, the conventional model checking and deductive approaches are compared.

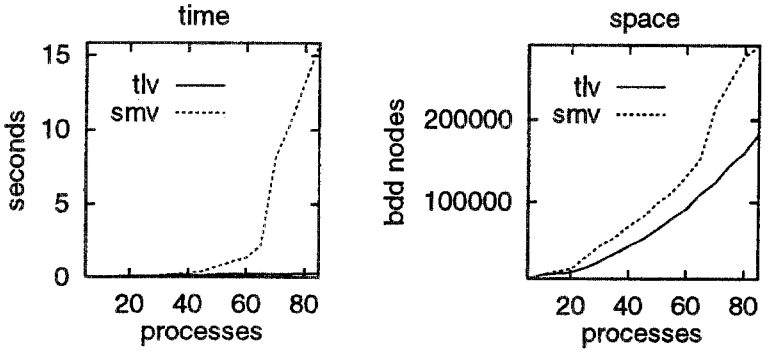


**Fig. 7.** Comparison of SMV and TLV for RES-SV

## Constrained Model Checking

In addition to the purely deductive approach, we also implemented and tested a mixed (or combined) approach, in which we use deductively derived invariants to

restrict the range of the transition function in computing the backwards closure, usually employed in model checking for invariance properties.

We considered again program RES-SV but used the deductive approach to verify only the two first invariants in the list: $\varphi_1[i]$ and $\varphi_2[i]$. These are very simple invariants, which can be discovered automatically by various heuristics (as explained in [MP95]). At this point we ceased using deductive methods, and invoked a special model-checking procedure CMCINV, written in TLV-BASIC, with a constraint parameter, which is the conjunction of $\varphi_1[i]$ and $\varphi_2[i]$. This procedure performs regular backwards closure computation, but eliminates all states which do not satisfy the given constraint.

In Fig. 8, we present plots of time and space complexity which compare regular model checking with constrained model checking for program RES-SV. The line representing constrained model checking is labeled by CMC, as compared to regular model checking which is labeled by MC. Both were performed by appropriate TLV-BASIC procedures.

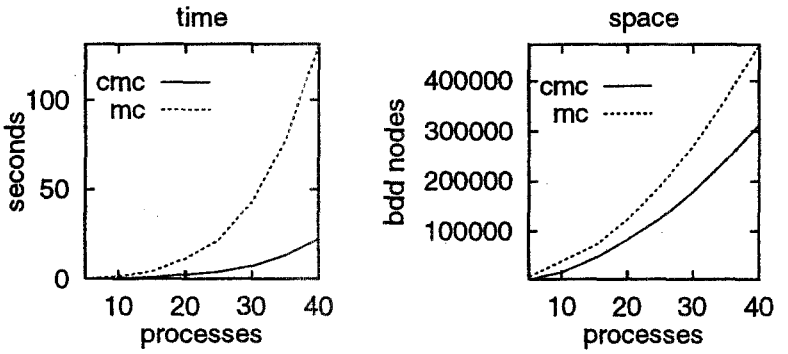


**Fig. 8.** Comparison of Model Checking and Constrained Model Checking for RES-SV

## 6 Verification of the Futurebus+

The IEEE Futurebus+ protocol specification is a technology-independent protocol for single-bus and multiple-bus multiprocessor systems. Part of this standard is the cache coherence protocol designed to work in a hierarchically structured multiple-bus system. Coherence is maintained by having the caches observe all bus transactions. Coherence across buses is maintained using bus bridges. A bus bridge is a memory agent/cache agent pair, each of them on a different bus, which can communicate. The memory agent represents the memory on its bus. The cache agent represents all the *remote caches*, caches on the bus of the corresponding memory agent, which may need to get access to the cache line via the bus bridge.

The protocol defines various transactions which let caches on a bus obtain readable and writable copies of *cache lines*. A cache line is a series of consecutive memory locations that is treated as a unit for coherence purposes.

We refer the reader to [CGH+93] for additional explanations and details about the SMV coding of the Futurebus+.

## 6.1 Specifying and verifying Cache Coherence

The following specifications are the ones which were proved in [CGH+93]. We repeated their verification, using deductive methods. There are four classes of safety properties and one for liveness.

The first class of safety properties is used to check that no device ever observes an illegal combination of bus signals or an unexpected transaction. Thus, we have the following formulas for every device d:

$$\mathbf{AG}\ \neg d.bus\text{-}error \qquad\qquad \mathbf{AG}\ \neg d.error$$

If these formulas are true then we say that the model is *error free.*

The *exclusive write* property states that if a cache has an exclusive modified copy of some cache line, then no other cache has a copy of that line. The specification includes the formula

$$\mathbf{AG}\ (p1.writable \rightarrow \neg p2.readable).$$

for each pair of caches $p1$ and $p2$. $p1.writable$ is true when $p1$ is in the exclusive-modified state. Similarly, $p2.readable$ is true when $p2$ is not in the invalid state.

The *consistency* property requires that if two caches have copies of a cache line, then they agree on the data in that line:

$$\mathbf{AG}\ (p1.readable \wedge p2.readable \rightarrow p1.data = p2.data)$$

The *memory consistency* property is similar to the consistency property. It specifies that any cache line that has a readable copy must agree with the memory device on the data.

$$\mathbf{AG}\ (p1.readable \wedge \neg m.memory\text{-}line\text{-}modified \rightarrow p1.data = m.data)$$

There is only one class of liveness specifications. It is used to check that it is always possible for a cache to get read and write accesses to a line. In a sense, it says that the model does not get stuck.

$$\mathbf{AG}\ \mathbf{EF}\ p.readable \qquad\qquad \mathbf{AG}\ \mathbf{EF}\ p.writable$$

All these properties were verified for small configurations, using deductive methods. We refer the reader to [PS96] for details of the inductive assertions that were used.

## 6.2 A Bug was Found

During our verification process, we came across a bug which seems to have escaped the attention of the previous verifiers of this design. In all probability, this is due to the fact that they have not considered the particular configuration in which this particular bug was lurking. We managed to prove the specifications for this configuration after fixing this bug.

The bug is manifested under the following circumstances. Consider a bus with a memory agent and three processors. We start from a reachable state where all processors have a shared copy of the cache line and the memory agent is in the `remote-shared-unmodified-invalid` state which indicates that the current bus has shared copies on it but the memory agent itself does not have a copy. Suppose that process p1 wants an exclusive copy of the cache line. It issues an invalidate transaction on the bus, which tells all other caches to release

their copies of the cache line. However, the other two processors, p2 and p3, choose to split the request so they continue to hold a shared copy but they each owe a response. Eventually, p2 responds and enters an invalid state. The memory agent observes this and enters the **remote-exclusive-modified** state. This means that the memory agent thinks that p1 already has an exclusive-modified copy but, in fact, p1 and p3 still hold shared copies. When p3 issues a response the memory agent sets on the error flag since, if only one process has a copy of the cache line, no other process should owe a response indicating a release of its hold on a shared copy.

## References


[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[CGH+93] E.M Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the Futurebus+ cache coherence protocol. In L. Claesen, editor, *Proc. of the 11th Int. Symp. on Computer Hardware Description Languages and their Applications*. North-Holland, April 1993.

[Gor86] M. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In *Formal Aspects of VLSI Design*, pages 153–177. Elsevier Science Publishers (North Holland), 1986.

[JS93] J.J Joyce and C.-J.H. Seger. Linking BDD-based symbolic evaluation to interactive theorem proving. In *Proc. of the 30th Design Automation Conf.*. ACM, 1993.

[KL93] R. Kurshan and L. Lamport. Verification of a multiplier: 64 bits and beyond. In C. Courcoubetis, editor, *Proc. of 5th CAV*, volume 697 of *Lect. Notes in Comp. Sci.*, pages 166–179. Springer-Verlag, 1993.

[MAB+94] Z. Manna, A. Anuchitanukul, N. Bjørner, A. Browne, E. Chang, M. Colón, L. De Alfaro, H. Devarajan, H. Sipma, and T. Uribe. STeP: The Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Dept. of Comp. Sci., Stanford University, Stanford, California, 1994.

[Man94] Z. Manna. Beyond model checking. In D. L. Dill, editor, *Proc. of 6th CAV*, volume 818 of *Lect. Notes in Comp. Sci.*, pages 220–221. Springer-Verlag, 1994. Invited talk.

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

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.

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

[ORSS94] S. Owre, J.M. Rushby, N. Shankar, and M.K. Srivas. A tutorial on using PVS for hardware verification. In R. Kumar and T. Kropf, editors, *Proc. of the 2nd Conf. on Theorem Provers in Circuit Design*, pages 167–188. FZI Publication, Universität Karlsruhe, 1994. Preiminary Version.

[PS96] A. Pnueli and E. Shahar. The TLV system and its applications. Technical report, The Weizmann Institute, 1996.

[RSS95] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model checking with automated proof checking. In P. Wolper, editor, *Proc. of 7th CAV*, volume 939 of *LNCS*, pages 84–97. Springer-Verlag, 1995.