Methodology and System for Practical Formal Verification of Reactive Hardware

Ilan Beer, Shoham Ben-David, Daniel Geist, Raanan Gewirtzman and Michael Yoeli

IBM Science & Technology, Haifa, Israel

Abstract. Making formal verification a practicality in industrial environments is still difficult. The capacity of most verification tools is too small, their integration in a design process is difficult and the methodology that should guide their usage is unclear.

This paper describes a step-by-step methodology which was developed for the practical application of formal verification. The methodology was successfully realized in a production environment of hardware design. The realization involved the development of a system consisting of several tools, while using the SMV [McM93] verification tool as the system core.

This system was used in the verification of eight designs. We specifically elaborate on the verification of a bus-bridge design, which was particularly successful in uncovering and eliminating many hardware design errors.

1 Introduction

Most commercial formal verification packages are used for either verification of combinational logic or state machine comparison. Other packages employ theorem-proving methods to reason about systems, but their practical usage requires extensive user intervention. An important type of verification, which has attracted much attention in the academy, but is still under a limited use in the industry, is proving that a design satisfies certain properties which describe its desired *ongoing* behavior. This kind of verification is particularly applicable to *reactive systems* (cf. [Pnu86, MP91]), i.e., systems which continuously interact with their environment, such as process controllers.

Recently we have developed an industry-oriented formal verification methodology and a system which are especially suitable for the verification of reactive hardware designs. The system, which is based on advanced academic research tools, employs symbolic model-checking to verify behavioral properties of designs. It is interoperable with several industrial design environments and supports various hardware description languages. It also incorporates user-friendly means for error diagnostics. So far the system has been successfully applied in the verification of several hardware designs developed at IBM. It has detected many design errors which have been otherwise difficult to find. As a result, formal verification has become an integral part of the VLSI chip design methodology in the Haifa Design Group. We believe that the cooperation with several design teams has matured the system by focusing on industry-oriented considerations rather then research aspects.

This paper provides an outline of the formal verification activity at the IBM Haifa Research Laboratory. The rest of this section briefly reviews related work and background. Section 2 describes the verification methodology and Section 3 presents the main features of the system developed. Section 4 illustrates the application of the methodology and system to a real-life example. Section 5 concludes with more results, problems, and plans.

1.1 Related Work

In this section we survey two works which are related to our work in two main aspects: both are suitable for formal verification of reactive systems and both have been exercised in industrial environments.

Much of the theory relevant to this kind of verification, as well as its application, has been developed at Carnegie-Mellon University. These works are mainly based on temporal logic and model-checking (e.g. [CES83]). Our work was influenced by that of K. McMillan [McM93], who took a major part in the development of the theory of symbolic model checking and, based on it, implemented the SMV model checker. He used SMV to verify a protocol of an industrial computer, but did it within an academic framework.

A relevant methodology of considerable interest is presented in [Kur87]. An implementation of this methodology is provided by the verification tool called COSPAN (COordination-SPecification ANalyzer), which forms a part of the industrial verification system used at AT&T Bell Labs. COSPAN facilitates the development and formal verification of reactive systems, such as communication protocols. Within this methodology, system development starts with a high-level model and proceeds by successive refinements, until the final implementation is reached. Each refinement is accompanied by a formal proof. The COSPAN methodology is applicable to verification during development, while we are concerned with a posteriori verification.

1.2 Background

Temporal logic is a branch of formal logic, applicable to the verification of time-dependent systems. A survey of temporal logics can be found in [Eme89]. In this work we refer to **CTL** [CE81], a branching-time temporal logic. CTL is interpreted over infinite computation trees, representing all feasible execution sequences of a finite state machine. It allows formulation of a rich class of temporal properties, including safety and liveness. Examples of CTL formulas can be found in Section 4. **ACTL** [SG90, GL91] is a subset of CTL. It reasons about properties which are valid only if they hold on every path of the computation tree. As will be seen later, ACTL is useful in the context of abstraction.

Model checking is the process of verifying that a temporal logic formula holds w.r.t. a suitably represented finite state machine. CTL model checking complexity is linear in the formula length and exponential in the number of state variables. It is efficient

compared to other relevant temporal logics such as LTL and CTL*, whose complexity is exponential in both formula length and number of state variables. Early CTL model checkers required an explicit representation of the complete state transition graph, which stressed the well-known state explosion problem. **Symbolic model checking** [McM93] provides efficient model checking facilities, without the need for explicit state enumeration. An improvement of the CTL model-checking algorithm provides for facilities to take into account only suitably selected paths by applying **fairness constraints.**

Reduction and Abstraction
Many real-life hardware designs are too large to be verified by straightforward model checking; even symbolic model checking may fail, due to the size of the design to be verified. Reduction and abstraction are used to alleviate this difficulty. Assume we are given a description of a hardware design M, and that we wish to verify that M satisfies a set F of formulas. **Reduction** is a construction of a simplified version of M, say M', such that $M' \models f \Leftrightarrow M \models f$ for every f in F. M' is called an exact reduction of M w.r.t. F; we also say that this reduction "strongly" preserves the properties of F. **Abstraction** is yet another way of simplifying a design. Simplification takes place by hiding internal details which are irrelevant to the verified property. Any ACTL formula valid in the abstracted design is also valid in the original one [GL91]. (Note that this property does not hold for CTL formulas in general.) Abstraction is a powerful tool because applying it may yield much smaller designs than when applying reduction. However, we are faced with a difficulty when the property of interest is not valid in the abstracted design.

2 Methodology

The methodology employed by our verification system is described in this section. It is based on concepts and methods described in [Bee92] and depicted in Figure 1. First, an overview is presented, and then we discuss some aspects which we find interesting.

2.1 Overview

The methodology suggests to follow these steps:

1. **Study of Expected Behavior**
   Unfortunately, it is not yet a common practice to supply formal specifications for hardware designs. Specification of the design, as well as the expected behavior of its environment, is usually provided by means of an informal description. Specifications can be obtained from several possible sources, such as published communication protocols, design documents and personal communication with designers.
2. **Partitioning**
   Formal verification systems have a limited capacity. The state explosion problem prevents verification of designs as a whole. Our solution is to partition the design and verify one part at a time. Usually designers develop their designs in a modular fashion and their partitioning is used for verification. When verifying one part,

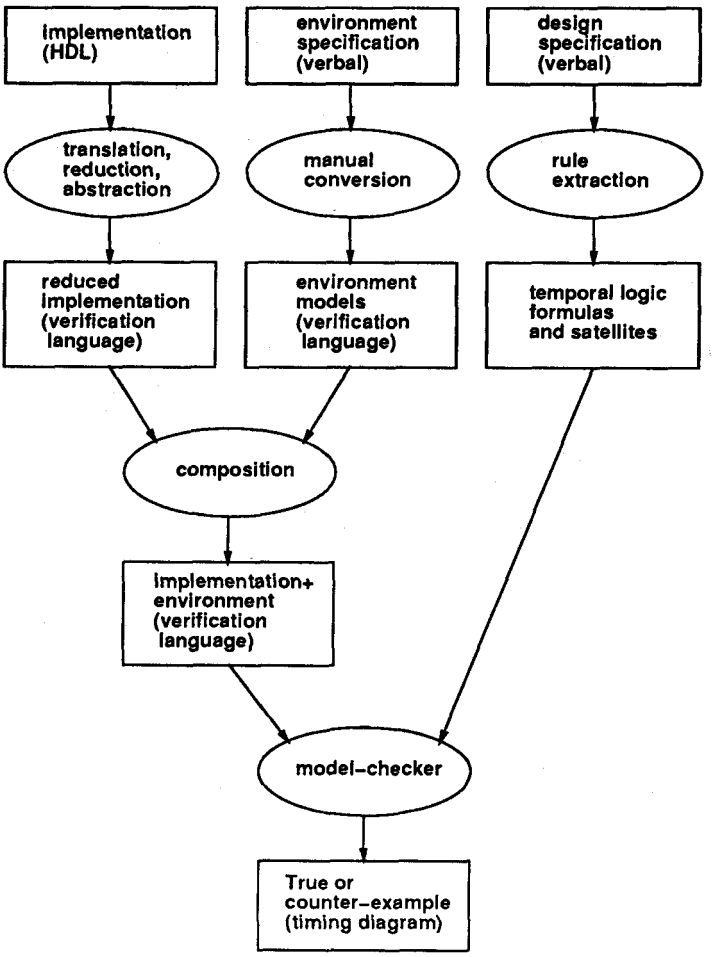

Fig. 1. Verification flow chart

abstract models of the others are treated as an environment. A specification of the interface between parts is obtained from the designers. In the sequel, when referring to a design, we mean a design partition. For more about partitioning see Section 2.3.

3. **Environment Modeling**

A hardware design should usually be verified in conjunction with its environment, which imposes restrictions on the feasible behavior of the design. Verifying the design on its own, may lead to finding errors in execution sequences which are not possible in reality. Out of the informal information provided about the environment, a formal model is generated. This model represents all possible interactions between the environment and the design, along their common interface. All other irrelevant activities of the environment are hidden by using abstraction techniques. We found

it beneficial to model the environment in a verification language which provides abstraction mechanisms, rather than a general hardware description language. To increase the confidence in the environment models, we check them w.r.t. temporal-logic formulas describing their expected behavior.

4. **Formal Specification**
   The informal specification of the design to be verified is replaced by a finite set of CTL formulas (hereafter: rules), possibly with fairness constraints and satellites (see Section 2.2). Implementation-oriented rules, which provide additional requirements not mentioned in the written specifications, are obtained from designers and transformed into CTL formulas.
5. **Translation**
   The design is implemented in a hardware description language, such as VHDL. This description is translated into a language acceptable by the verification system. The translation process depends on the actual development environment. The system described in Section 3 is an example of such a process.
6. **Reduction and Abstraction**
   Often, the design size after partitioning is still too large to fit into the verification system. Usually, only a fragment of it, which has a reasonable size, participates in the verification of a specific formula. The rest is removed by per-formula reduction and abstraction methods (see Section 1.2). Much of the reduction can be done automatically. If it is insufficient, manual abstraction is required.
7. **Actual Verification**
   The translated and reduced implementation is composed with the environment models and verified w.r.t. the specification formulas, using model checking facilities.
8. **Handling Failures**
   In case of a failure, the system produces, whenever feasible, an execution trace illustrating the problem. The cause is not necessarily a design error. The failure may be due to a mistaken rule formulation, an unsuitable abstraction of the environment, or the application of an inadmissible reduction step. Only after the failure has been pinpointed to an actual design error, it is presented to the design engineer in a suitably familiar form, such as a timing diagram.

Verification Modes

Two verification modes are defined: development and maintenance. During development, models are established, most of the rules are formulated, and the main reduction steps are performed. Model checking is applied until the design is successfully verified w.r.t. most rules. At this stage the design is relatively stable and maintenance mode begins. New versions of the implementation, obtained from designers, are translated, reduced and verified automatically. New rules are added if necessary. Development mode is the responsibility of the verification persons, while the maintenance mode can often be operated by designers.

2.2 Satellites

CTL formulas are not powerful enough to express all state machine properties which one would wish to verify. Furthermore, some properties expressible in CTL require formulas

which are cumbersome to construct and difficult to understand. In many cases linear-time formulas are more adequate than branching-time ones. Although more expressive logics have been established (e.g. CTL* [Eme89]), relevant model checkers are inherently less efficient, and practical realizations are not available. To overcome some of these CTL limitations, the concept of a **satellite**, an extension of history variables [Cli73], was introduced in [Bee92]. A satellite is a small finite state machine connected externally to the design to be verified. It can read the design output at any moment but, to prevent the satellite from affecting the design behavior, the design does not read the satellite's state. A satellite records particular events of interest, to be retrieved later by CTL formulas. A concept similar to satellites has been introduced in [Lon93] as *observer processes*. Satellites resemble PLTL (linear time logic of the past [Eme89]), but their expressive powers are incomparable: satellites cannot capture the fact that an event occurred infinitely often (PLTL can), while they can easily capture that a property holds in time t *modulo* k (PLTL cannot). Using CTL formulas combined with satellites, one can reason about branching-time future and linear-time past. Satellites do not come for free. Their main disadvantage is the increase in the number of state variables, causing an earlier explosion of the state-space.

2.3 Partitioning and Modular Verification

Partitioning can be a problem if the verification of a property requires the presence of more than one partition. Modular Verification [GL91] is a powerful method dealing with such situations. Suppose that the design is a composition of the form: $M = M_1||M_2||...||M_n$. If we construct abstract models $A_1, A_2, ..., A_n$, such that A_i is an abstraction of partition M_i, then $A = A_1||A_2||...||A_n$ is an abstraction of M. If A is relatively small we can check it w.r.t ACTL formulas. If a formula is valid in A it is also valid in M (the opposite is not necessarily true).

2.4 More about Environment Abstraction

The informal description of the environment is frequently either partial or parametric, and it presents a variety of actions from which an actual environment is free to choose. It is essential that the formal model precisely reflects this situation: i.e., the model includes all feasible actions, but no others. Otherwise not all possible execution sequences of the design will be verified or, alternatively, non-feasible ones will be examined. To illustrate this point, assume that the environment informal description contains the statement *"State A always implies the eventual occurrence of state B"*. In this case the environment is free to arbitrarily choose any finite delay. The relationship between states A and B can be represented by the finite state machine M shown in Figure 2, where the transition from A to B is non-deterministic. To prevent an infinite self loop around state A, which means a (forbidden) infinite delay, we impose on M a fairness constraint "$GF(\neg A)$", which means that the environment is infinitely often not in state A.

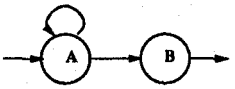

Fig. 2. Environment model example

3 Verification System

A system supporting the methodology of Section 2 is described below. The system is composed of existing tools and of new programs developed by us. Only its main features are described.

3.1 The SMV Model Checker

The core of the verification system is SMV - a symbolic model-checker developed by K.McMillan at CMU [McM93]. SMV is used for checking finite-state systems against specifications written in the temporal logic CTL. Its input language is very similar to other hardware description languages but, in addition to describing design implementations, it also allows to specify abstract, nondeterministic models. SMV employs symbolic methods: it represents logic functions, relations and sets of states as Binary Decision Diagrams (BDD [Bry86]).

To increase the capacity and speed of SMV, we augmented it with an algorithm for ordering the transition-relation partitions [GB94]. We are also using several heuristics to order BDD variables. These improvements allow us to verify larger designs than with the original system.

3.2 Translation

Our system deals with real design implementations, which are given in a Hardware Description Language (HDL). The system can read structural VHDL and IBM-internal HDLs. Existing compilers are used to translate the HDL files into gate level, which is then loaded into an IBM design-automation database for further manipulations, such as reduction. Finally this internal representation is translated into the SMV language for actual verification. The translation process is fully automated.

3.3 Reduction and Abstraction

Usually, the implementation is too big to fit into SMV. We therefore have to reduce it and extract only the parts relevant for verification. Different rules may refer to different parts of the design, thus the reduction made is based on the rule to be verified. The reduction process is semi-automatic: the user assigns attributes to interface signals (sometimes to internal signals also) and the system uses this information to reduce the design automatically. There are several possible attributes of signals. Some important attributes: *essential*, *constant* and *cutting* are described here.

The *essential* attribute is assigned to signals which either participate in a rule or must be observed by an environment model. The basic reduction algorithm leaves only cones of influence of essential signals, removing design components which are not necessary for the evaluation of these signals.
Signals whose exact contents are not supposed to affect the correctness of the checked properties can be assigned constant values. This is often the situation with some bits of wide input buses. A constant-elimination algorithm propagates the constants into the design, thus simplifying its structure by reducing the number of inputs and memory elements. This operation sometimes over-reduces the design and must be used with care, to avoid the elimination of feasible execution paths.
The reduction described so far is sometimes not sufficient and the reduced design is still too large. In these cases, an internal "surgery" is needed. This is done by assigning the *cutting* attribute to internal signals, thus making them design inputs. This technique can be used for the replacement of sub-blocks, whose internal operation is irrelevant to the verification of a specific property, by simpler abstract blocks with an equivalent external behavior. The cutting signals interface between the abstract blocks and the rest of the design. This operation is an abstraction, thus only ACTL formulas can be used, and a failure might occur even when the property holds in the original design.
Another reduction algorithm is used for detection and merging of memory elements which may become equivalent as a result of the previous reduction and abstraction steps.

3.4 Counter Examples

When the design fails to obey a rule, SMV presents a counter-example which can be viewed as a short test that exhibits the wrong behavior. The system translates the counter-example into a timing diagram which can be browsed interactively using an X-Windows based GUI. While browsing, the designer can inspect values of interface signals as well as internal signals. The timing diagram looks exactly like standard simulation results, thus giving the designer a familiar framework for debugging. A timing diagram example is shown in Figure 3.
The counter-example is also used to produce a simulation test that can be given as an input for the simulator. Designers use this test to reconstruct the error in their environment and to verify that they have corrected it. The test can also be used while checking the integrated design, when formal methods can no longer be applied because of size limits.

4 Project Example

Carmel is a communication bridge chip between two buses, which comply with different protocols, one of which is the PCI bus [PCI93]. The chip's function is to detect transactions in one bus which are addressed to devices on the other bus and to enable these transactions. Carmel was developed by the Haifa Design Group, IBM Israel. The implementation language was VHDL 1076.
Both formal verification and traditional testing by simulation were employed to verify Carmel. Formal verification was applied only after successful simulation of most

test cases, and was used to uncover relatively subtle design errors. We were mainly interested in verifying Carmel's control logic, leaving the relatively simple data-path for simulation. Trying to verify the data-path might have caused an immediate state explosion. The control logic as a whole was too large to fit into the verification system. The designers partitioned Carmel, considering modularity, into several modules, and we used this partitioning as a basis for the formal verification process.

Table 1 summarizes relevant details of two of the partitions we verified. To demonstrate the influence of reduction on the speed of verification, we include two versions of each partition, with different degrees of reduction. It can be seen that a small decrease in the number of state variables can increase the speed significantly. In the table, $Before$ is the number of state variables (memory elements and inputs) before reduction, $After$ is the number of state variables after reduction, $Rules$ is the number of rules verified, $Bugs$ is the number of design errors detected, $RunTime$ is the CPU-time of a typical rule (in seconds, on a Risc System 6000 model 550), and BDD is the number of BDD nodes allocated by the model-checker.

Table 1. Partition details

Function	Before	After	Rules	Bugs	RunTime	BDD
PCI slave[1]	663	74	55	30	1600	1820K
PCI slave[2]		70			550	660K
PCI master[1]	748	76	40	16	9500	1840K
PCI master[2]		63			190	243K

The errors we found were about 40% of all errors detected throughout the entire verification of the chip, including data-path errors. Some designers had initially been skeptical towards formal verification, but they became enthusiastic after receiving the first error reports. They also appreciated the debugging aids provided by our system.
Carmel has been fabricated and tested extensively. No bugs have been found yet.

4.1 Rules

Many rules concerning the bus interfaces have been extracted from published bus standards. These rules are implementation-independent and we used them again in other designs which interfaced these types of buses. More rules, mainly implementation-dependent ones, have been added by the designers. Some were refinements of existing rules, such as addition of timing constraints. Others verified the internal behavior of partitions by referring to internal signals. The following examples exhibit interesting aspects of rules:

Long Counter-Example
In some cases, counter-examples which demonstrated design errors were considerably long. Recalling that a counter-example is a near-shortest simulation test which can detect the error, it is clear that the probability that one will find these errors is low, if one is using traditional methods. The following example, involving the PCI protocol, illustrates such a case: a rule states that *"In a case of target-retry, the master must retry the transaction within 3 time cycles"*. (This is an implementation-specific requirement that refines a generic PCI rule.) *Target-retry* is characterized by *Devsel* and *Stop* low while *Trdy* is high. *Frame* transition from high to low denotes transaction beginning. The rule is represented by the formula: "$SPEC\ AG(\ (\neg Devsel \wedge \neg Stop \wedge Trdy) \rightarrow ABF\ 1..3\ Fall(Frame)\ )$". *Fall* is a transition-detecting satellite and *ABF* is a bounded-future operator. The resulting counter-example is displayed in Figure 3. It shows that a very special sequence of events must be generated in order to uncover the problem. (For those who are familiar with PCI: three back-to-back transactions and two target-aborts are required before the problem shows up.) The problem hiding behind this simple symptom was a serious one, and required major design modifications.

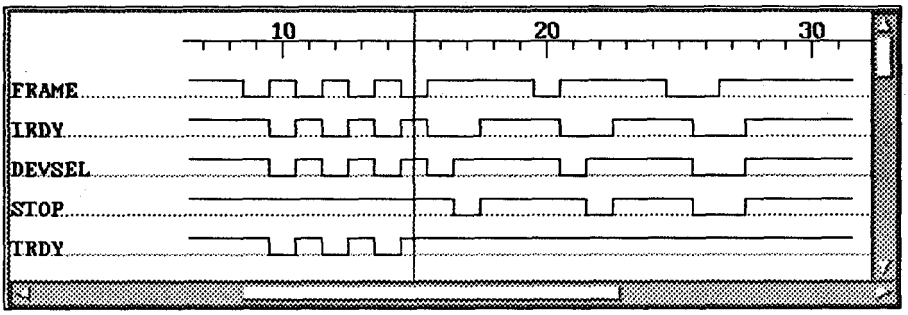

Fig. 3. Counter example timing diagram

Using a Satellite
One of the requirements was that at most K data units are stored in the bridge at any time. Another requirement stated that the bridge cannot write out more data than it reads in. To verify these requirements, we defined a satellite that counted the number of data-read operations minus the number of data-write operations, and then wrote a CTL formula asserting that the counter's value must neither exceed K nor be below zero.

5 Further Experience

In addition to the example of Carmel, discussed in Section 4, we have participated so far (January 94) in the verification of seven other designs, four of them have already been produced. We have found dozens of errors in most of the designs. Many of them, such as deadlocks and other kinds of hard-to-produce situations, were very difficult to

find by conventional simulation. After a short experience with the system, even the most skeptical designers gained faith in our methods and cooperated with enthusiasm. Formal verification is now an integral part of the design methodology in the Haifa Design Group.

Observations

- We found the system and methodology particularly useful in the verification of designs with limited influence of the data on the flow of control.
- Experiments with asynchronous designs, or with multiple unsynchronized clocks, resulted in an early state explosion.
- No correlation was found between the complexity of rules and the types of errors they have found: simple rules discovered symptoms of very subtle errors.
- Automated BDD variable ordering is a necessity in an industrial system, because of the frequent design modification and the dependency of the order on the per-formula reduction.

Problems and Plans

- Capacity limitation is a major problem in every formal verification system. Currently we can verify only one partition at a time. To increase our system's capacity, we are investigating several directions, such as improved compositional verification methods, reduction algorithms, and BDD variable ordering methods.
- Another serious problem is the difficulty to decide when a set of CTL formulas, each of them representing a partial specification, covers the whole informal design specification. We seek criteria of adequate coverage.
- CTL formulas are difficult to read and write. We are planning a user friendly specification language on top of CTL and the satellites.

6 Acknowledgements

We thank the designers of the Haifa Design Group, whose cooperation contributed to the maturity of both the methodology and the system. We also thank Israel Berger and Aharon Aharon for supporting this work, and to Yossi Lichtenstein and Orna Grumberg for reviewing this paper.

References

[Bee92] I. Beer, "Formal Verification of Hardware", M.Sc. Thesis, EE Department, Technion, 1992 (in Hebrew).

[Bry86] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", IEEE Trans. Computers, Vol. C-35, August 1986.

[CE81] E.M. Clarke and E.A. Emerson, "Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic", in: Proceedings of the Workshop on Logic of Programs, LNCS 131, 1981. Temporal Logic Specifications: A Practical Approach", Tenth ACM Symposium on Principles of Programming Languages, Austin, Texas, 1983.

[CES83] E.M. Clarke, E.A. Emerson and A.P. Sistla, "Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications: A Practical Approach", Tenth ACM Symposium on Principles of Programming Languages, Austin, Texas, 1983.

[Cli73] M. Clint, "Program Proving: Coroutines", Acta informatica, 2(1), 50-63, 1973.

[Eme89] E.A. Emerson, "Temporal and Modal Logic", in: Handbook of Theoretical Computer Science, J. van Leeuwen, ed., North-Holland, 1989.

[GB94] D. Geist and I. Beer, "Efficient Model Checking by Automated Ordering of Transition Relation Partitions", submitted for publication, 1994.

[GL91] O. Grumberg and D.E. Long, "Model Checking and Modular Verification", in: LNCS 527, 1991.

[Kur87] R.P. Kurshan, "Reducibility in Analysis of Coordination", in: LNCS 103, 1987.

[Lon93] D.E. Long, "Model Checking, Abstraction and Compositional Verification", Ph.D. Thesis, CMU, 1993.

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

[MP91] Z. Manna and A. Pnueli, "The Temporal Logic of Reactive and Concurrent Systems; Specification", Springer-Verlag, 1991.

[PCI93] "PCI Local Bus Specification, Revision 2.0", PCI Special Interest Group, 1993.

[Pnu86] A. Pnueli, "Applications of Temporal Logic to the Specification and Verification of Recative Systems: A Survey of Current Trends", in: Current Trends in Concurrency, J.W.de Bakker et al., eds., LNCS 224, 1986.

[SG90] G. Shurek and O. Grumberg, "The modular Framework of Computer-Aided Verification: Motivation, Solutions and Evaluation Criteria", CAV90.