Formal Verification in Hardware Design: A Survey

CHRISTOPH KERN and MARK R. GREENSTREET
Department of Computer Science, University of British Columbia

In recent years, formal methods have emerged as an alternative approach to ensuring the quality and correctness of hardware designs, overcoming some of the limitations of traditional validation techniques such as testing.

There are two main aspects to the application of formal methods in a design process: The formal framework used to specify desired properties of a design, and the verification techniques and tools used to reason about the relationship between a specification and a corresponding implementation. We survey a variety of frameworks and techniques which have been proposed in the literature and applied to actual designs. The specification frameworks we introduce include temporal logics, predicate logic, abstraction and refinement, as well as containment between ω-regular languages. The verification techniques presented include model checking, automata-theoretic techniques, automated theorem proving, and approaches that integrate the above methods.

In order to provide insight into the scope and limitations of currently available techniques, we present a selection of case studies where formal methods have been applied to industrial-scale designs, such as microprocessors, floating-point hardware, protocols, memory subsystems, and communications hardware.

Categories and Subject Descriptors: A.1 [**Introduction and Survey**]; B.7.2 [**Integrated Circuits**]: Design Aids—*Verification*

General Terms: Verification

Additional Key Words and Phrases: formal methods, formal verification, hardware verification, model checking, language containment, theorem proving, case studies, survey

1. INTRODUCTION

The correct design of complex hardware poses serious challenges. Economic pressures in rapidly evolving markets demand short design cycles while increasing complexity of designs makes simulation coverage less and less complete. Bugs in a design which are not uncovered in early design stages can be expensive, and bugs which remain undetected until after the shipping products can be *extremely* expensive. In safety-critical applications, such as nuclear power control or aviation, correct operation of a device is imperative.

This work was supported in part by NSERC research grant OGP-0138501, a UBC graduate fellowship, and a BC Advanced Systems Institute faculty fellowship.
Authors' address: 2366 Main Mall, Vancouver, B.C., Canada V6T 1Z4;
email: {ckern,mrg}@cs.ubc.ca; phone: +1 (604) 822-{5707,3065}; fax: +1 (604) 822-5485.

Formal methods, i.e. the application of mathematical methodologies to the specification and validation of systems, have emerged as a possible aid in tackling these challenges. This survey attempts to provide an understanding of what can be specified, how these properties can be verified, and presents examples of how formal methods have been applied to realistic designs. Earlier surveys include [Gupta 1992; Seger 1992; McFarland 1993; Shankar 1993]. The Gupta survey, in particular, provides a more comprehensive coverage of the theoretical aspects of formal hardware verification.

1.1 Formal Verification in Hardware Design

Hardware design typically starts with a high-level specification, usually given using block diagrams, tables, and informal text conveying the desired functionality. A combination of top-down and bottom-up design techniques are applied until a final design is obtained. *Validation* of the design involves checking that the fabricated physical design indeed meets its specification. In a traditional design flow, this is usually accomplished through simulation and testing. However, exhaustive testing for non-trivial devices is generally infeasible; thus, testing provides only a probabilistic assurance.

Formal verification, in contrast to testing, uses rigorous mathematical reasoning to show that a design meets all or parts of its specification. A prerequisite for the applicability of formal verification is the existence of formal descriptions for both the specification and the implementation. Such a description is given in a notation with a formal semantics which unambiguously associates a mathematical object with the description, permitting these objects to be reasoned about in a mathematical framework. Section 2 describes many such notations including predicate logic, temporal logic and a variety of hardware description languages (HDLs).

The time required for formal verification must be considered when applying these techniques to a real project. There are relatively automatic formal verification methods that are comparable to traditional simulation in time required and ease of use, but they tend to be limited to "small" designs. On the other end of the scale, formal verification frameworks exist which are in principle powerful enough to verify large, state-of-the-art designs [Brock et al. 1996]. Applying such a framework requires large amounts of time of highly skilled experts. Although formal verification methods have been used in the designs of several state-of-the-art microprocessors and other complicated chips, we are aware of no complete top-to-bottom verification for such a design. The cost of such verification still appears to be prohibitive.

Time is a decisive factor in the integration of formal methods in the design process. To achieve maximum leverage, formal methods must be applied in a way which ensures that they can keep up with the design flow [McMillan 1994]. If this is the case, formal methods can benefit the design process significantly, as they allow conceptual errors in the design to be uncovered much earlier through formal verification of the high-level design against the requirements.

It is important to note that the use of formal methods can be advantageous even if a complete top-to-bottom verification is not carried out: The exercise of formalizing the requirements or a high-level specification can be useful in itself because it tends to clarify many aspects, such as boundary conditions, which are easily overlooked in informal specifications [Miller and Srivas 1995]. Verifying properties of a high-level abstraction can catch many errors at an early stage of the design, avoiding costly corrections later. High-level descriptions can often be made concise enough to be tractable by automatic verification methods [McMillan 1994]. Finally, the cost of verification might be considered worthwhile for certain, particularly difficult to design subsystems, while other, "straightforward" modules can be treated with traditional methods.

1.2 The Meaning of Verification

Formal methods by themselves are not a panacea. Even verifying that the netlist model of a design satisfies a formal specification does not provide complete assurance that the physical device when manufactured will always work as intended. The reasons for this are fundamental [Cohn 1989b]: there is no formalization of the intentions the person had in mind who formalized the specification; therefore one cannot formally verify that the top-level requirements specification faithfully captures this intention. Similarly, the formal interpretation of the lowest-level description is still an abstract model of the physical device, and it cannot be formally verified that this model is accurate. Furthermore, verification generally requires that some assumptions are made on the behaviour of the environment in which a device is intended to operate. If the actual operating environment violates these assumptions, the device may fail despite successful verification. However, the application of formal methods offers the benefit of at least making these assumptions explicit.

The literature contains many claims of the form "the XYZ device has been formally verified". To be meaningful, such claims must be accompanied by a description of what properties were verified as well as a description of the form of the design description (e.g. RTL, abstract function blocks, behavioral models, gate, or switch level) that was used [Cohn 1989b]. For example, Brock and Hunt [1990] examined claims made about the application of formal methods to a commercial microprocessor noting many aspects of the design that were not verified.

1.3 Organization of the Paper

Section 2 introduces frameworks which are suitable for the formalization of specifications and implementation descriptions of hardware designs. Section 3 surveys methodologies and tools for the formal verification of assertions expressed in the introduced formalisms. Section 4 presents case studies of applications of formal verification to hardware design. We emphasize real, industrial designs in order to provide a realistic impression of what verification problems are tractable with current methods. We close the paper with a short summary and conclusions.

2. SPECIFICATIONS AND VERIFICATION CONDITIONS

There are two main approaches to the specification and corresponding verification of hardware (and also software) systems. The first approach is concerned with specifying desired *properties* which one wants to verify that the design satisfies. Formal verification is generally concerned with properties of of a temporal nature, i.e. they do not concern static attributes of the system but rather characteristics of the system's behaviours or executions. Temporal logics are a unifying framework for expressing such temporal properties. Verification amounts to showing that all of the system's possible behaviours satisfy the temporal properties comprising the system's specification.

The second approach is based on specification in terms of a high-level model of the system. Here, the "good" behaviours of a system are not given by a set of properties which they are required to obey, but rather by the set of all behaviours of the high-level model. Thus, verification requires showing that each possible behaviour of the system's implementation is consistent with some behaviour of its high-level specification.

The two approaches are often used in conjunction: First, a high-level model of the design is shown to satisfy a set of desired temporal properties. Then, a series of more and more detailed specifications are developed, each of which is an implementation of the specification at the next higher level. In an appropriate technical framework, the temporal properties of the highest level model are preserved by the refinement steps and thus carry through to the lowest, most detailed level. In this context, the first type of verification is also referred to as *design* or *property verification* while the second form is known as *implementation verification*. Note that the distinction between the two forms is only a conceptual one. Both types of verification are instances of the same problem: the specification defines some constraint on the allowed behaviours of a system, and verification requires showing that the implementation meets this constraint.

In this section, we will introduce the formalisms behind both approaches. Section 2.1 is concerned with the specification of temporal properties, while section 2.2 describes specification in terms of high-level models.

2.1 Specifications in Temporal Logic

In general, a temporal logic is a logic (usually propositional or first order logic) augmented with temporal modal operators which allow reasoning about how the truth values of assertions change over time [Emerson 1990]. Temporal logic can express conditions such as "property p holds at all times", or "if p holds at some instant in time, q must eventually hold at some later time". Propositions of this sort can be employed to specify desired properties of systems, such as "this bus controller will always grant at most one request for the bus at a time," and "every request will eventually be granted".

The truth of a formula in temporal logic is understood relative to a system M which is perceived to be in one of a (possibly infinite) set of states S at any point in time, and which performs transitions between states as time progresses. It is assumed that there are *atomic propositions* associated with each state in S; these form the basic building blocks for temporal formulas. In verification practice, the state space often consists of Boolean n-tuples; in this case a suitable form of the atomic propositions is "component i of the state tuple has value *true* (*false*)". The temporal operators permit the expression of relationships between the truth values of atomic propositions at different times; this corresponds to a statement about the truth values of atomic propositions as the system proceeds through a sequence of states. Pnueli [1977] first proposed using temporal logic to reason about system behaviours.

There are many different forms of temporal logics; in particular there are a number of options with regard to the underlying model of time, such as branching versus linear time, discrete versus continuous time or the use of temporal operators on points versus intervals in time. In this section, we will first introduce Computation Tree Logic (CTL) as a representative example of temporal logics and demonstrate how it can be used for specification purposes. Later, we will examine other variants of propositional temporal logics more briefly. For a more complete survey of temporal logics, see [Pnueli 1986; Emerson 1990].

The choice of an appropriate logic for a given specification task is an important problem. There are three main considerations to take into account: Firstly, temporal logics differ in their expressiveness; there are properties which can be expressed in some logics, but not in other, less expressive ones. Secondly, the complexity of the verification task depends on the choice of logic. Generally, it is more difficult or computationally expensive to show that a system satisfies a temporal formula in a more expressive logical framework. Finally, temporal logic specifications are often the top-level specification and thus need to be validated with respect to the informal intention the specifier has in mind. This is often easier in a simpler (and therefore most often less expressive) framework. Thus, it is usually advantageous to choose the least expressive logic which is expressive enough to capture the properties one has in mind.

2.1.1 *Computation Tree Logic.* Computation Tree Logic (CTL) [Clarke et al. 1986] is a propositional logic of branching time; i.e. it is based on propositional logic and uses a discrete model of time where, at each instant, time may split into more than one possible future. We will first introduce the syntax and formal semantics of CTL, and then use a simple example to demonstrate how CTL can be used to specify desired behaviours of systems.

2.1.1.1 *CTL Formulae and their Truth Semantics.* The truth semantics of a CTL formula is defined with respect to a *temporal structure* $M = (S, R, L)$, where S is a set of states, $R \subseteq S \times S$ is a total binary relation (i.e. $\forall_{s \in S}\ \exists_{t \in S}\ (s,t) \in R$), and $L : S \rightarrow 2^{\mathcal{P}}$ is a labeling of states with the atomic propositions in $\mathcal{P}$ which are true in a given state. R is the next-state relation of the structure, i.e. if the system is in state s at a given time instant, it will be in any of the states in the set $\{t \in S \mid (s,t) \in R\}$ at the following time instant. The totality requirement for R is included because CTL formulae have no sensible interpretation for states without successors. A *path* is defined as an infinite sequence of states $s_0, s_1, \ldots$ such that $\forall_{i \geq 0}\ (s_i, s_{i+1}) \in R$).

Let $\mathcal{P}$ be a set of *atomic propositions*. Formulae of CTL are defined recursively as:

(1) Every atomic proposition $p \in \mathcal{P}$ is a CTL formula.
(2) If f_1 and f_2 are CTL formulae, then so are $\neg f_1$, $f_1 \wedge f_2$, $\mathbf{AX}\, f_1$, $\mathbf{EX}\, f_1$, $\mathbf{A}[f_1\, \mathbf{U}\, f_2]$, and $\mathbf{E}[f_1\, \mathbf{U}\, f_2]$.

Intuitively, **AX** means "all successors": the formula $\mathbf{AX}\, f_1$ holds in a system state s_0 *iff* f_1 holds in every successor state of s_0. Likewise, **EX** means "there exists a successor". The formula $\mathbf{A}[f_1\, \mathbf{U}\, f_2]$ means "always-until," i.e. along all possible paths f_1 holds until f_2 is satisfied. The formula $\mathbf{A}[f_1\, \mathbf{U}\, f_2]$ holds in state s_0 *iff* for every path starting from s_0, $s = s_0, s_1, \ldots$, there is some $i \geq 0$ such that f_2 holds in state s_i and f_1 holds in all states from s_0 to s_{i-1}. Note that $\mathbf{A}[f_1\, \mathbf{U}\, f_2]$ holds in any state where f_2 holds. In the same fashion eventually, $\mathbf{E}[f_1 \, \mathbf{U} \, f_2]$ means "exists-untils," i.e. there exists a path such that f_1 holds until f_2 is satisfied. Additional temporal operators are defined in terms of the ones above:

(1) $\mathbf{AF}\, f \equiv \mathbf{A}[\mathit{true}\, \mathbf{U}\, f]$ (f must hold *eventually*)
(2) $\mathbf{EF}\, f \equiv \mathbf{E}[\mathit{true}\, \mathbf{U}\, f]$ (there is a reachable state in which f holds)
(3) $\mathbf{EG}\, f \equiv \neg\mathbf{AF}\,\neg f$ (there is some path on which f always holds).
(4) $\mathbf{AG}\, f \equiv \neg\mathbf{EF}\,\neg f$ (f must always hold on all possible paths).

Consider a CTL formula f, a model structure $M = (S, R, L)$, and a state $s_0 \in S$. We denote the statement "f holds in M at state s_0" with $M, s_0 \models f$. We write $s_0 \models f$ if the underlying model structure is understood, $M, S_0 \models f$ to abbreviate $\forall_{s \in S_0 \subseteq S}\ M, s \models f$, and $M \models f$ to abbreviate $M, S \models f$. The relation $\models$ defines the formal truth semantics for CTL and is defined recursively as follows:

(1) $M, s_0 \models p$ *iff* $p \in L(s_0)$.
(2) $M, s_0 \models \neg f$ *iff* not $(M, s_0 \models f)$.
(3) $M, s_0 \models f_1 \wedge f_2$ *iff* $(M, s_0 \models f_1)$ and $(M, s_0 \models f_2)$
(4) $M, s_0 \models \mathbf{AX}\, f$ *iff* $\forall_{t \in S}\ (s_0, t) \in R \Rightarrow (M, t \models f)$.
(5) $M, s_0 \models \mathbf{EX}\, f$ *iff* $\exists_{t \in S}\ (s_0, t) \in R \wedge (M, t \models f)$.
(6) $M, s_0 \models \mathbf{A}[f_1\, \mathbf{U}\, f_2]$ *iff* for all paths $s_0, s_1, \ldots,$
$\exists_{i \geq 0}\ (M, s_i \models f_2) \wedge \forall_{j=0 \ldots i-1}\ M, s_j \models f_1$.
(7) $M, s_0 \models \mathbf{E}[f_1\, \mathbf{U}\, f_2]$ *iff* for some path $s_0, s_1, \ldots,$
$\exists_{i \geq 0}\ (M, s_i \models f_2) \wedge \forall_{j=0 \ldots i-1}\ M, s_j \models f_1$.

2.1.1.2 *Specifications in CTL.* We will now introduce a simple system which will be used as an example throughout this paper. Consider the controller of a (very abstract) traffic-light at a four-way intersection where the lights in opposing directions always show the same colour. Let $\mathcal{C} = \{r, y, g\}$ be the set of traffic-light colours. The state space of the controller can be modeled as $\mathcal{S} = \mathcal{C} \times \mathcal{C}$, where the first (second) component denotes the colour of the lights in the north-south (east-west) direction. Figure 1 shows the state-transition graph of the traffic-light controller. The corresponding state-transition relation R is defined such that $(s, t) \in R$ *iff* there exists an arc from state s to state t in the graph. The states on the right are states which we expect a correct and properly initialized implementation of a traffic-light not to reach. We have added self-loops to each of them to ensure that R is indeed a total relation. A suitable set of atomic propositions is given by

$$\begin{aligned} L(s) \;=\; & \{[ns = c] \mid c \in \mathcal{C} \wedge \exists_{d \in \mathcal{C}}\ s = (c, d)\} \cup \\ & \{[ew = c] \mid c \in \mathcal{C} \wedge \exists_{d \in \mathcal{C}}\ s = (d, c)\}. \end{aligned}$$

For example, the atomic proposition $[ns = g]$ holds in states $(g, r), (g, y)$ and (g, g). Let $M = (\mathcal{S}, R, L)$ be the temporal structure corresponding to our traffic-light controller.

There are two important classes of properties which one usually wants to verify for a given system [Owicki and Lamport 1982]:

Safety Properties. These are properties which intuitively assert that "bad things never happen".

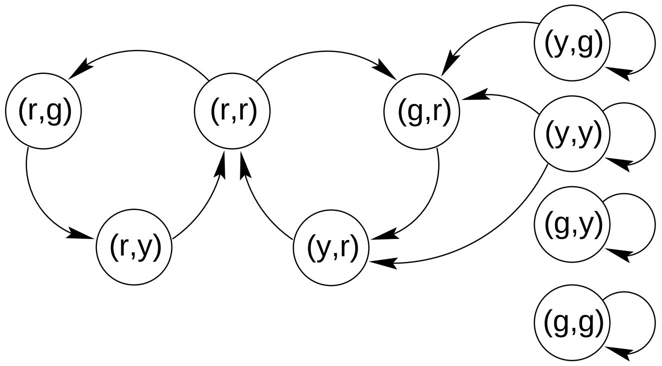

Fig. 1. State Transition Graph for the Traffic Light Controller

Liveness Properties. Properties in this class state that "good things happen eventually". They are also referred to as *eventuality* or *progress* properties.

For a more rigorous classification of liveness and safety properties, see [Pnueli 1986; Emerson 1990].

An important safety property of our traffic-light is that it shows a red light to at least one direction at all times. In CTL, this property can be expressed as $P \equiv \mathbf{AG}\,([ns = r] \vee [ew = r])$. Note that $M \not\models P$ since, for example, $M, (g, y) \not\models P$. However, one can easily convince oneself by exploring all possible state transitions that e.g. $M, (r, r) \models P$. An equivalent formulation is

$$M \models ([ns = r] \wedge [ew = r]) \Rightarrow \mathbf{AG}\,([ns = r] \vee [ew = r]).$$

This specification is in a commonly encountered generic form of safety properties, $M \models P_{init} \Rightarrow \mathbf{AG}\, P_s$, where P_{init} and P_s are *instantaneous formulae* not containing temporal operators. A safety property of this form has the intuitive interpretation that every computation of M which starts in a state satisfying P_{init} also satisfies P_s at all times.

A desirable liveness property of the traffic-light is that a particular direction will "always eventually" see green lights. This property can be expressed in CTL as

$$Q \;\equiv\; \mathbf{AG}\,(\mathbf{AF}\,[ns = g]) \wedge \mathbf{AG}\,(\mathbf{AF}\,[ew = g]).$$

However, $M, (r, r) \models Q$ does not hold, because the transition relation R specifies two successor states for (r, r). This means that R allows, for example, the path that repeatedly cycles through the three states $(r, r), (r, y), (r, g), \ldots$. Such a path is a counter-example for the first conjunct of Q. We can phrase a weaker property

$$Q' \;\equiv\; \mathbf{AG}\,(\mathbf{EF}\,[ns = g]) \wedge \mathbf{AG}\,(\mathbf{EF}\,[ew = g]),$$

such that indeed $M, (r, r) \models Q'$. Q' states that from all states on all paths, M *permits* a sequence of transitions to a state in which a particular direction sees green.

The stronger liveness property Q holds for M only under additional fairness assumptions stating that M does not consistently "discriminate" against state transitions. However, it can be shown that liveness under fairness assumptions cannot be expressed in CTL [Emerson and Halpern 1986; Emerson 1990]. In the following section, we consider more general branching time logics which permit such assertions.

2.1.2 *More Expressive Branching Time Logics.* Emerson and Halpern [1986] define the logic CTL* as a generalization of CTL where a path quantifier may be followed by arbitrary temporal formulae. Formally, CTL* formulae are defined recursively in terms of *path formulae* and *state formulae* as follows:

(1) Any atomic proposition is a state formula.
(2) If s_1, s_2 are state formulae and p is a path formula, then $s_1 \wedge s_2, \neg s_1$ and $\mathbf{E}p$ are state formulae.
(3) If s is a state formula and p_1, p_2 are path formulae, then $s, p_1 \,\mathbf{U}\, p_2, \neg p_1, p_1 \wedge p_2, \mathbf{X}p_1$ are path formulae.
(4) Any state formula is a CTL* formula.

In addition, the abbreviations $\mathbf{F}p \equiv \mathit{true}\,\mathbf{U}\,p$, $\mathbf{G}p \equiv \neg\mathbf{F}\neg p$ and $\mathbf{A}p \equiv \neg\mathbf{E}\neg f$ are introduced. Informally, the truth of a state formula is defined in the obvious way for the Boolean operators, and such that $\mathbf{E}p$ holds in a state r if there is a path starting with r which satisfies p. The truth of a path formula is defined with respect to a given path in a natural way, for example $p_1 \,\mathbf{U}\, p_2$ holds for a path q if p_1 holds on a finite and possibly empty prefix of q which is followed by a state in which p_2 holds. For a formal treatment of the semantics of CTL*, see [Emerson and Halpern 1986; Emerson 1990].

The path quantifiers $\mathbf{E}$ and $\mathbf{A}$ apply to a path formula and result in a state formula. These quantifiers transform a statement about computations starting from a state into an assertion about the state itself. In particular, $\mathbf{E}p$ holds for state s_0 if there is a path, $s_0, s_1, s_2, \ldots$ such that p holds for this path. Furthermore, if we constrain the syntax of CTL* such that path quantifiers must be followed immediately by one of the modalities $\mathbf{X}$, $\mathbf{U}$, $\mathbf{F}$ or $\mathbf{G}$, we obtain the logic CTL embedded in CTL*. The essential difference between CTL* and CTL is that all sub-formulae of a CTL formula must be state formulae; this precludes assertions about individual computations.

Liveness properties under fairness assumptions can be expressed in CTL* for example in the form

$$\mathbf{A}((\mathbf{GF}p_1 \wedge \cdots \mathbf{GF}p_n) \Rightarrow \mathbf{F}q). \tag{1}$$

Here, q is a property which must eventually hold on every path which "always eventually" satisfies each of the properties $p_1, \ldots p_n$. The key point is that the sub-formulae $\mathbf{GF}p_i$ and $\mathbf{F}q$ all refer to the *same* path which can be thought of as being "bound" by the path quantifier $\mathbf{A}$. As mentioned above, CTL cannot express assertions about paths; this gives an intuitive explanation why liveness properties such as (1) cannot be stated in CTL.

In the traffic-light example, we could require the fairness assumption that the controller "always eventually" takes each of the two transitions from (r, r). The formulae $[ns = r] \wedge [ew = r] \wedge \mathbf{X}[ew = g]$ and $[ns = r] \wedge [ew = r] \wedge \mathbf{X}[ns = g]$ hold for a path starting in (r, r) if this path's first transition is $(r, r) \rightarrow (r, g)$ or

$(r, r) \rightarrow (g, r)$, respectively. Thus, the fairness assumption can be stated as

$$\begin{aligned} T \;\equiv\; & \mathbf{GF}([ns = r] \wedge [ew = r] \wedge \mathbf{X}[ew = g]) \wedge \\ & \mathbf{GF}([ns = r] \wedge [ew = r] \wedge \mathbf{X}[ns = g]). \end{aligned}$$

The desired liveness property does indeed hold under this fairness assumption:

$$M, (r, r) \models \mathbf{A}(T \Rightarrow (\mathbf{F}[ns = g] \wedge \mathbf{F}[ew = g])).$$

It is not always necessary to resort to full CTL* to express certain notions of fairness. Clarke et al. [1986] propose the logic CTL^F which has the same syntax as CTL. However, *fairness constraints* for a model can be stated in terms of sets of states which have to be visited infinitely often on paths which are to be considered fair. The semantics of CTL^F is defined in exactly the same way as for CTL, except that all path quantifiers range over fair paths only.

The logics ∀CTL and ∀CTL* are defined as subsets of CTL and CTL*, respectively, which are free of existential path quantifiers [Clarke et al. 1994].

2.1.3 *Linear Time Logic.* In the branching time framework described in the previous sections, the non-determinism of a system is captured by allowing for more than one future at any time instant. An alternative view is to regard time as being linear and to consider only the one future or execution which "actually takes place" [Lamport 1980]. Thus, a linear time temporal formula is an assertion about one particular sequence of states; to reason about non-deterministic systems it is necessary to consider more than one execution.

Syntactically, formulae of linear-time logic (LTL) look like path-quantifier-free CTL* formulae. The semantics of an LTL formula is defined with respect to a *linear-time structure* $M = (S, x, L)$, where S is a set of states, $x = x_0, x_1, \ldots$ is an infinite sequence of states, and $L : S \rightarrow 2^{\mathcal{P}}$ is a labeling of states with atomic propositions in $\mathcal{P}$ [Emerson 1990]. Essentially, the truth of an LTL formula with respect to a sequence is defined in the same manner as truth of a CTL∗ path-formula is defined with respect to a path.

Lamport [1980] uses linear-time structures of the form $M_X = (S, X, L)$, where S and L are defined as above, and X is a *set* of sequences of states corresponding to the set of all possible computations of a system. In this framework, a LTL formula p holds for a structure M_X, $M_X \models p$, *iff* p holds for all $x \in X$. We write $M_X, x \models p$ to denote that p holds for some $x \in X$. The universal quantification implicit in the notation $M_X \models p$ can be made explicit by considering the CTL* formula $\mathbf{A}p$ with respect to a temporal structure whose transition relation generates exactly the sequences in X.

The truth of an LTL formula can also be extended to temporal structures $M_R = (S, R, L)$, where R is the transition relation (see section 2.1.1). Let $M_{R,X} = (S, X_R, L)$, where $X_R = \{x \,|\, x \text{ is a path of } M_R\}$, denote the corresponding linear time structure. Then, $M_R \models \varphi$ *iff* $M_X \models \varphi$, i.e. $x \models \varphi$ for all paths x generated by M_R.

The main difference between LTL and BTL is the lack of existential quantification over paths in LTL. Emerson and Halpern [1986] have shown that CTL* is strictly more expressive than LTL, i.e. there are properties such as $\mathbf{EF}p$ which cannot be expressed in LTL. There is some argument on whether existential path quantification is essential in specifications or should not be used at all. Emerson and Halpern [1986] argue that existential quantification plays an important role in reasoning about concurrent systems. Others, however, point out that specifications should capture properties of all possible computation of a system, not only some, in the sense that one usually wants to specify what a system *must* do, and not only what it *might* do [Clarke et al. 1994].

Note that LTL and BTL are equivalent if the model structure is deterministic, in which case it gives rise to exactly one computation.

2.1.4 *μ-Calculus.* An alternative characterization of temporal modalities can be given in terms of fixpoints of monotonic predicate transformers. Let $M = (S, R, L)$ be a temporal structure as above. Consider the set of predicates on states, and let each predicate be represented by the subset of $P \subseteq S$ such that the predicate is true in and only in the states in P. Let $\mathbf{AX}\, P$ denote the set of states such that all of those state's successors states are in P, i.e.

$$\mathbf{AX}\, P \;=\; \{s \in S \mid \forall_{s' \in S}\ (s, s') \in R \Rightarrow s' \in P\}. \tag{2}$$

Consider the sequence of sets defined by

$$\begin{aligned} Q_0 &= S, \\ Q_{i+1} &= P \cap \mathbf{AX}\, Q_i. \end{aligned}$$

The set $Q_i, i > 0$ has the property that for every state $s \in Q_i$, P holds, and for all successors s' of s, P holds on a prefix of length $i-1$ of all paths starting in s'. In other words, P holds on any path prefix of length i starting from a state $s \in Q_i$. One can show that the greatest fixpoint $\nu Q.P \cap \mathbf{AX}\, Q$ exists and that

$$Q_\infty = lim_{i \to \infty} Q_i = \nu Q.P \cap \mathbf{AX}\, Q, \tag{3}$$

(see [Emerson 1990]). We can interpret Q_∞ as the set of states such that for any state $s \in Q_\infty$, P holds on all states of all paths starting in s. This justifies the definition $\mathbf{AG}\, P = \nu Q.P \cap \mathbf{AX}\, Q$, which provides a characterization of the temporal operator $\mathbf{EG}$ in terms of a fixpoint. Similar fixpoint characterizations can be derived for the other temporal operators.

Propositional μ-calculus [Kozen 1993; Bradfield 1992] is a temporal logic based on fixpoint characterizations. Formulae of the μ-calculus are composed of simple predicates, the next-state operators on sets $\mathbf{EX}$ and $\mathbf{AX}$, Boolean connectives, and the least and greatest fixpoint operators μ and ν. The semantics of a formula with respect to a structure M is given in terms of sets of states in a similar fashion as above. Thus, the denotation of a formula f is a subset $Q_f \subseteq S$ which is intuitively the set of states in which f holds. The possibility of nesting the fixpoint operators provides a considerable expressive power. In particular, μ-calculus subsumes CTL* and LTL [Emerson 1990].

The fixpoint characterizations for the temporal operators are of practical importance, even if one does not require the full expressive power of the μ-calculus. For a structure M whose set of states S is finite, the sequence $Q_i, i = 0, 1, \ldots$ in eq. (3) must converge in a finite number of steps. Thus, the iterative fixpoint computation provides the basis for an algorithm which computes the set of states satisfying a temporal formula in e.g. CTL or any other temporal logic which can be expressed in the μ-calculus. This approach is explored in greater detail in section 3.1.2.

2.1.5 *Invariants and Safety Properties.* In the previous subsections, we have introduced some variants of temporal logic with rather comprehensive expressiveness. In many applications however, it is sufficient, and often more intuitive, to specify a system in terms of rather simple assertions.

One of the simplest forms of temporal specification is in terms of invariants and safety properties. An invariant I of a structure M is a property such that for all states of M which satisfy I, their successor states also satisfy I. This can be expressed in CTL as $M \models (I \Rightarrow \mathbf{AX}\, I)$. One can easily verify that e.g. $I_1 \equiv [ns = r] \vee [ew = r]$ and $I_2 \equiv [ns = g] \wedge \neg [ew = r]$ are invariants of the traffic light controller. In section 2.1.1.2, we have already encountered the traffic light's safety property $P_{init} \Rightarrow \mathbf{AG}\, P_s$, with $P_{init} \equiv [ns = r] \wedge [ew = r]$ and $P_s \equiv [ns = r] \vee [ew = r]$.

There is an important connection between invariants and safety properties: Given

(1) $M \models P_{init} \Rightarrow I$

(2) $M \models I \Rightarrow \mathbf{AX}\, I$

(3) $M \models I \Rightarrow P_s$,

we can conclude $M \models P_{init} \Rightarrow \mathbf{AG}\, P_s$ using a simple induction argument over reachable states. This allows safety properties to be established inductively, where the inductive step corresponds to showing that I is an invariant [Lamport and Schneider 1984]. In the traffic-light example, once I_1 has been shown to be an invariant, simple Boolean algebra suffices to show $P_{init} \Rightarrow I_1$ and $I_1 \Rightarrow P_s$, which together establishes the desired safety property.

?] present a framework where properties of modular designs are expressed in terms of invariants of individual components as well as the entire system. Verification proceeds by proving invariants of individual modules and combining the results to prove invariants of their composition.

2.1.6 *Trajectory Formulas.* Trajectory Formulas (TF) [Seger and Bryant 1995] form a restricted temporal logic which offers only the next-time operator and does not allow negation or disjunction. Trajectory formulas can be efficiently verified using techniques which are described in section 3.1.4. Trajectory formulas are constructed from assertions about node values and expressions of symbolic Boolean variables. For example, if `in` is a node of the circuit being modeled, then the formula `in` is 0 asserts that the node `in` always has the value `0` (i.e. *false*).

A *trajectory assertion* has the form $[A \Rightarrow C]$, where A and C are trajectory formulas. Informally, a trajectory assertion $[A \Rightarrow C]$ holds for a model structure M, written $M \models [A \Rightarrow C]$, *iff* each sequence of states of M which satisfies the *antecedent* A also satisfies the *consequent* C. Typically, A specifies constraints on how the inputs of a circuit are driven, while C asserts the expected results on the output nodes. For example, the formula

$$[\mathtt{in}\,\mathbf{is}\,0 \;\Rightarrow\; \mathbf{N}(\mathtt{out}\,\mathbf{is}\,1)]$$

asserts that from any state in which the node `in` is `0`, the node `out` must be `1` in the next state ($\mathbf{N}$ is the next state operator).

Rather than writing a separate assertion for every possible combination of inputs, symbolic variables may be used to write much more succinct specifications. For example, an inverter with a one time-unit delay can be specified using the symbolic variable a and the symbolic trajectory assertion:

$$[\texttt{in}\,\mathbf{is}\,a \;\Rightarrow\; \mathbf{N}(\texttt{out}\,\mathbf{is}\,\neg a)]$$

A trajectory assertion is implicitly universally quantified over any symbolic variables that appear in it. The complete set of constructions for trajectory formulas are:

- Simple predicates: if `nd` is a node in the circuit model, then `nd is 0` and `nd is 1` are trajectory formulas.
- Conjunction: If F_1, F_2 are trajectory formulas, then so is $(F_1 \wedge F_2)$.
- Next time: If F is a trajectory formula, then so is $\mathbf{N}F$.
- Domain restriction: If F is a trajectory formula and E is an expression over symbolic variables, then $F\,\mathbf{when}\,E$ is a trajectory formula.

Many useful abbreviations can be defined using these operators. For example, we write $\texttt{nd}\,\mathbf{is}\,b$ as an abbreviation for

$$((\texttt{nd}\,\mathbf{is}\,0)\,\mathbf{when}\,\neg b) \wedge ((\texttt{nd}\,\mathbf{is}\,1)\,\mathbf{when}\,b)$$

and we write $\mathbf{N}^k$ where k is a constant to indicate k applications of the next time operator.

As a somewhat larger example, consider a correlator with inputs `x` and `y` and output `out`. The output should be true if and only if the previous three inputs on `x` matched the corresponding inputs on `y`. This behaviour is specified by the assertions:

$$\begin{array}{ll} & (\texttt{x}\,\mathbf{is}\,a) \wedge (\texttt{y}\,\mathbf{is}\,a) \wedge (\mathbf{N}(\texttt{x}\,\mathbf{is}\,b) \wedge \mathbf{N}(\texttt{y}\,\mathbf{is}\,b)) \wedge (\mathbf{N}^2(\texttt{x}\,\mathbf{is}\,c) \wedge \mathbf{N}^2(\texttt{y}\,\mathbf{is}\,c)) \\ & \Rightarrow\; \mathbf{N}^3(\texttt{out}\,\mathbf{is}\,1) \\ \text{and} & (\texttt{x}\,\mathbf{is}\,a) \wedge (\texttt{y}\,\mathbf{is}\,\neg a) \;\Rightarrow\; \mathbf{N}(\texttt{out}\,\mathbf{is}\,0) \wedge \mathbf{N}^2(\texttt{out}\,\mathbf{is}\,0) \wedge \mathbf{N}^3(\texttt{out}\,\mathbf{is}\,0) \end{array} \tag{4}$$

The first assertions states that if `x` and `y` are the same for three consecutive cycles, then the output must be `1` on the next cycle. The second assertion states that if `x` and `y` differ on some cycle, then the output must be `0` on the next three cycles. No initial state is specified with a trajectory assertion, therefore, verifying such an assertion shows that it holds everywhere along a trajectory. For example, the specification for the correlator describes its operation for all times (although the value of `out` is not fully specified for the first four time steps).

2.2 Specification with High-Level Models

The previous section presented a variety of approaches for specifying a system in terms of *desired properties*. An alternative style of specification uses a *high-level model* to stipulate the allowed behaviours of a system. In this framework, verification entails reasoning about the relationship between the high-level model M_S, also referred to as the *specification*, and a lower-level model M_I, the *implementation*.

An important aspect of this approach to specification is the notion of *abstraction*, which permits unnecessary detail to be hidden from the high-level model. Furthermore, specifications may be given in an hierarchical fashion; starting from a very abstract model at the highest level, one proceeds through a series of abstractions to a detailed description of the implementation. The view of the design at some level k assumes the role of the implementation with respect to the specification at level $k-1$ as well as the role of the specification for level $k+1$.

To facilitate formal reasoning about the implementation and the specification, it is required that a formal interpretation can be given for their descriptions. There are three main approaches to the formal specification of systems and their implementations. A device's behaviour can be formalized in terms of functions and predicates of standard first or second-order predicate logic. Alternatively, its behaviours can be described by a state transition system, whose definition in turn can be expressed in languages such as CSP or UNITY. Finally, the description may be in terms of languages recognized by automata on infinite objects.

In this section, we first review the role of abstraction; we then describe how specifications are described both in logic and as transition systems. We then survey a number of formalizations of what it means that "M_I implements M_S", and review automata-theoretic approaches.

2.2.1 *Abstraction Mechanisms.* Melham [1988] identifies four different types of abstraction which are relevant in hardware verification. We will illustrate each type of abstraction with a simple, 3-bit, up-down counter.

Structural abstraction suppresses details about the implementation's internal structure in the specification. The specification should only reflect a system's externally observable behaviour; it gives a "black-box" view of a device without placing constraints on its internal design. For example, a 3-bit, up-down counter can be described as a component with inputs for the clock and the direction control, and an output for the value of the count. A structural refinement of this description could describe the counter using flip-flops and gates.

Behavioural abstraction suppresses details about what the component does under operating conditions that should never occur. For example, a behavioural description of the three-bit counter may omit a specification of what the counter does if it is at its maximum count and is clocked with the direction input set to "up," or if it is at its minimum count and is clocked with the direction input set to "down'. Behavioural abstraction may also be used to indicate "don't care" conditions. For example, if several devices are requesting access to a bus and they all have the same priority, the bus controller may be allowed to grant the requests in an arbitrary order. This gives the designer greater flexibility to optimize implementation. The specification is more abstract in the sense that its behaviours are a superset of the implementation behaviours.

Data abstraction relates signals in the implementation to signals in the specification when they have different representations. For example, the specification of the counter may specify that the output is an integer between zero and seven, and the implementation may have an output that consists of three, Boolean valued signals. *Data abstraction* requires a mapping which determines how the states or signals of the implementation are to be interpreted in the specification's semantic domain.

Temporal abstraction relates time steps of the implementation to time steps of the specification. For example, the counter may be specified in terms of what happens for each clock cycle while the implementation uses a two-phase clock. In this case, every other implementation state maps to a specification state. Frequently, specifications of a microprocessor use the execution of a single instruction as the basic unit of time. An implementation however, would base its notion of time on the execution of a microcode instruction, a clock cycle, or a clock phase. Temporal abstraction requires that corresponding execution states at the two time scales are identified with each other. This is complicated by the possibility that one specification time unit does not necessarily correspond to always the same number of implementation steps.

2.2.2 *Specifications in Logic.* Gordon [1985] has argued that it is not strictly necessary to resort to specialized hardware description languages; rather, formal logic is sufficient to specify hardware systems. In the following, we will assume that the reader is familiar with first and second order predicate logic (see e.g. [Kleene 1967]). We use the usual notation for predicate logic, plus the following notation for conditional expressions,

$$(a \rightarrow b \mid c) \triangleq \begin{cases} b \text{ if } a, \\ c \text{ otherwise,} \end{cases}$$

where a is of Boolean type and b and c share the same type.

The external behaviour of a device can be modeled as a predicate over the device's external connections or its externally visible state. If the device's timing behaviour is irrelevant, a first order predicate suffices and the external connections are modeled as values in an appropriate domain, such as Booleans or integers. For example, the definition

$$\mathit{NAND}(i_1, i_2, o) \triangleq \big(o = \neg(i_1 \wedge i_2)\big)$$

specifies a delay-less NAND gate by constraining its external connection o (the output) to be equal to the NAND of its (input) wires i_1, i_2.

If time has to be taken into account, and a discrete model of time suffices, it is convenient to represent signals as sequences of values, i.e. as functions from natural numbers into the underlying domain. Then, for some integer, *delta*, a NAND gate with a switching delay of δ time units can be described with the following second-order predicate:

$$\mathit{NAND}(i_1, i_2, o) \triangleq \forall_{t \in \mathbf{N}}\; o(t + \delta) = \neg(i_1(t) \wedge i_2(t)).$$

This predicate is second-order because it is an assertion about functions from time to Booleans (i.e. i_1, i_2, and o) and not about simple Boolean variables. The composition of devices thus specified into more complex devices is formalized by identifying the internal connections and hiding them by existential quantification. For example, we can describe an AND gate built from two NAND gates as shown in figure 2 with the predicate

$$\mathit{AND}(i_1, i_2, o) \triangleq \exists_q\; \mathit{NAND}(i_1, i_2, q) \wedge \mathit{NAND}(q, q, o).$$

A synchronous device can be specified in terms of a predicate over the sequence of its states and the sequence its environment's states. Let $\mathcal{S}_D$ denote the device's state space, and $\mathcal{S}_E$ the state space of its environment. A sequence of states, or *state stream* [Windley 1995a], can be represented by a function from natural numbers into states.

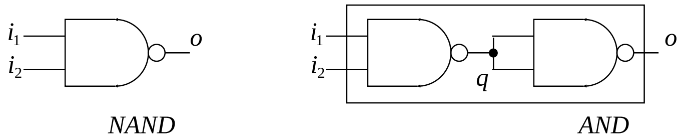

Fig. 2. (a) NAND gate, (b) AND gate

As an example, we will consider the specification of a modulo-eight counter. The counter's state space is given by

$$\mathcal{S}_D \triangleq \{i \in \mathbf{N} \mid 0 \leq i < 8\}.$$

The counter's environment $\mathcal{S}_E = \{\mathit{true}, \mathit{false}\}$ consists of a reset signal; a value of *true* on this signal causes the counter to reset to zero. Otherwise, the counter increments modulo 8 at each time instant.

Since the counter is a deterministic device, we can give its next-state function $N : (\mathcal{S}_D, \mathcal{S}_E) \to \mathcal{S}_D$ as follows:

$$N(c, res) \triangleq res \to 0 \mid (c = 7 \to 0 \mid c + 1).$$

We can now define a predicate which constrains a state stream $s : \mathbf{N} \to \mathcal{S}_D$ and an environment stream $e : \mathbf{N} \to \mathcal{S}_E$ to behaviours which are consistent with the counter's next state function:

$$\mathit{Counter}(s, e) \triangleq \forall_{t \in \mathbf{N}}\ s(t+1) = N(s(t), e(t)).$$

The use of second order logic in the above specification is not essential; the relevant aspects of the specification can be captured in the definition of the next-state function N which does not depend on higher order logic [Hunt 1989].

The use of a general-purpose logic as a specification language has the advantage that one can immediately associate a formal interpretation to the specification based on the formal semantics of the logic. This has to be contrasted with the use of specialized hardware specification languages which are often not defined with formalization in mind; for instance the definition of a formal semantics for VHDL is an area of active research [Van Tassel 1993; Kloos and Breuer 1995].

However, specification in logic may require greater discipline on the part of the specifier to produce a readable specification. This can be aided by specification frameworks which for example constrain the specification of the next-state function to have a certain structure (see section 4.1.1).

2.2.3 *Specification using Transition Systems.* It is clear that we can specify the allowed behaviours of a device in terms of a state transition system whose transitions correspond to the behaviours we have in mind. For example, the state transition system for the traffic light controller in section 2.1.1.2 can be viewed as an abstract specification for a "real" controller which could for instance be be implemented as a synchronous design.

Formally, a state transition system is a tuple $M = (S, S_0, R)$, where S is a set of states, $S_0 \subseteq S$ the set of initial states, and $R \subseteq S \times S$ is the transition relation. We have described the traffic-light specification in terms of a state transition graph, which immediately gives rise to a transition system as defined above, if the initial

$$
\begin{array}{l}
\textbf{program } \textit{TrafficLight} \\
\quad \textbf{declare} \\
\quad\quad ns, ew \; : \; \{red, grn, yel\} \\
\quad \textbf{initially} \\
\quad\quad ns, ew \; = \; red, red \\
\quad \textbf{assign} \\
\quad\quad \begin{array}{llcll}
 & ns := grn & \text{if} & (ns = red) \wedge (ew = red) \\
[] & ns := yel & \text{if} & (ns = grn) \\
[] & ns := red & \text{if} & (ns = yel) \\
[] & ew := grn & \text{if} & (ns = red) \wedge (ew = red) \\
[] & ew := yel & \text{if} & (ew = grn) \\
[] & ew := red & \text{if} & (ew = yel)
\end{array} \\
\textbf{end } \textit{TrafficLight}
\end{array}
$$

Fig. 3. UNITY program for the traffic light controller

states are identified. However, for more realistic applications graph representations become difficult to handle and a textual representation is often more appropriate. Nevertheless, graphical representations such as Statecharts, which facilitate a hierarchical design description [Harel 1988], can contribute to a better understanding of a system and its properties.

A variety of languages have been proposed in the literature which are suitable for specifying transition systems. Here, we will present a specification of the traffic light in UNITY as a representative example [Chandy and Misra 1988].

The basic building block of a UNITY program is the guarded multi-assignment. For example, the assignment

$$x, y := y, x \text{ if } x > y$$

is enabled if the variable x has a greater value than y, and if so, will swap the values of the two variables. The assignment may be executed only if it is enabled, and if so it is executed as a single atomic action, in other words, the all of the expressions on the right side of the assignment are evaluated first, and then the variables on the left side of the assignment are updated. Although other statements may be concurrently enabled, executions must only reach states that can be reached by performing one statement at a time. UNITY is different from sequential PASCAL-like languages in that it neither provides sequential composition of assignments nor control structures such as loops or branches.

Figure 3 shows a UNITY program for the traffic-light controller. The **declare** section declares two variables, ns and ew, of an enumerated type with the three different values red, grn and yel. The two variables represent the states of the lights in the two directions. The state space induced by a program is the cross-product of the domains of all its variables. The **initially** section specifies that the initial values of both variables are red. Variables not explicitly mentioned in this section have arbitrary initial values. Finally, the **assign** section of the program declares a list of assignments which are combined with the asynchronous combinator []. The execution semantics of a UNITY program consisting of []-composed assignments are such that for a given state, any one of the assignment statements enabled in that state is selected non-deterministically and executed, which results in a state change. This process is repeated indefinitely. For example, in state $(ns = red, ew = red)$, a non-deterministic choice between the first or the fourth assignment statement takes place. This corresponds to this state having two successor states in the graph representation in figure 1.

A Unity program P denotes a state transition system $M = (S, S_0, R)$ whose state space S is given by the cross-product of the domains of the variables declared in P, whose initial states S_0 are as specified in the **initially** section, and whose transition relation is given by the guarded multi-assignments of P. Note that a UNITY program can also be seen as denoting a temporal structure by augmenting the corresponding transition system with atomic propositions of the form $[v = a]$, where v is a variable of P and a is a value in v's domain. This facilitates a textual notation for systems whose specification is given in terms of temporal formulae as well.

Other languages which provide a suitable notation for high-level design descriptions as well as temporal structures include CSP [Hoare 1978], Murφ [Dill et al. 1992; Dill 1996], SMV [McMillan 1992], SPL [Manna and Pnueli 1992; Manna et al. 1994] and Synchronized Transitions [Staunstrup 1994]. Traditional HDLs such as Verilog or VHDL can be used as well; as mentioned above, assigning formal semantics to them is however not without problems.

In formal verification, non-determinism is frequently used for behavioural abstraction, leaving aspects of a design unspecified at a certain level of abstraction. For example, the traffic light controller depicted in figure 1 does not specify whether the (r, g) or (g, r) state will follow the (r, r) state. The specification only requires that when a light cycles, it must go from red to green to yellow and back to red. In particular, this specification is satisfied both by an implementation which strictly alternates between the two roads and by one which can show green to the same road more than once based on input from a sensor loop.

2.2.4 *Refinement.* The notion that M_I implements M_S is formalized in terms of a *refinement* relation $\sqsubseteq$. In this section we will explore a number of definitions of this relation which have been used in the literature. It is generally accepted that any definition of $\sqsubseteq$ should at least satisfy the property that $M_I \sqsubseteq M_S$ implies that each observable behaviour of M_I is also an observable behaviour of M_S [Gerth 1989]. Generally, the equivalence of observable behaviours is to be understood with respect to an appropriate abstraction mapping which relates corresponding aspects of M_I and M_S.

Abadi and Lamport [1991] describe a very general notion of refinement based on trace inclusion with respect to the externally visible system states. Let E denote the set of externally visible states of both the implementation and the specification; it is assumed that any data abstraction between the two levels has already been dealt with by lifting external implementation states into their corresponding specification states through an appropriate mapping. Let I_I and I_S denote the internal states of M_I and M_S, and let $S_I = E \times I_I$ and $S_S = E \times I_S$ be the state spaces of the two models.

Let X_I and X_S denote the sets of all sequences of states allowed by M_I and M_S. Then, M_I is defined to be a refinement of M_S *iff*

$$\begin{aligned} &\forall_{\sigma \in X_I} \ (\sigma = (e_0, q_0), (e_1, q_1), (e_2, q_2), \ldots \\ &\qquad \Rightarrow \ \exists_{\sigma' \in X_S} \ \sigma' = (e_0, r_0), (e_1, r_1), (e_2, r_2), \ldots) \end{aligned} \tag{5}$$

Thus, $M_I \sqsubseteq M_S$ *iff* for each behaviour of the implementation, there exists a specification behaviour with the same externally visible states. Temporal abstraction is accommodated by allowing *stuttering*; it is assumed that for every $\sigma \in X_S$, all sequences obtained by replacing states with finitely many repetitions thereof, or by removing repetitions, are also in X_S. For example, consider the implementation behaviour

$$\sigma_1 = (e_0, q_0), \ldots, (e_0, q_{k-1}), (e_1, q_k), \ldots$$

whose first k states differ only in their internal part. Assume that the first k transitions implement a single transition at the specification level, from state (e_0, r_0) to (e_1, r_1). Then σ_1 can be matched up with the "stuttered" specification behaviour

$$\sigma_2 = (e_0, r_0), \ldots, (e_0, r_0), (e_1, r_1), \ldots$$

Maretti [1994] introduces a generalization of trace inclusion where the externally visible state of the system is subject to an interface protocol. Consider for example a device which outputs data on a bus, and indicates the availability of valid data on the bus by setting a signal *valid* to *true*. An interface protocol can be defined which specifies that the bus is only to be observed (i.e. is externally visible) if *valid* = *true*. An implementation of the device which places arbitrary values on the bus while *valid* = *false* can still be regarded a refinement of the specification with respect to the interface protocol, even though the values on the bus are not consistent with the specification if the interface protocol is disregarded.

In a framework where the specification and the implementation are described in predicate logic, refinement is usually expressed in a form which is equivalent to the following [Melham 1988; Windley 1995b]:

$$\forall_{s:\mathbf{N} \to \mathcal{S}_I} \ \forall_{e:\mathbf{N} \to \mathcal{S}_E} \ Imp(s, e) \Rightarrow Spec(A_s(s), A_e(e)) \tag{6}$$

This assertion states that for each pair s, e of implementation-level state and environment streams which are consistent with the implementation, the streams $A_s(s)$ and $A_e(e)$ obtained by subjecting s and e to their respective abstraction functions are consistent with the specification. This notion of refinement implies trace inclusion, it is however more restrictive because it requires the abstraction mapping to be a function and does not allow more general relations. corresponding specification trace, but rather requires that this trace can be expressed as a function of the specification trace. In addition to modeling data abstraction, this approach can model temporal abstraction by mapping a sequence of implementation states to the same specification states. Thus, the finer representation of time in the implementation appears as stuttering moves for the specification. With this sense of abstraction, every safety property of the specification has a corresponding property for the implementation. The same cannot be said of liveness properties as the abstraction mapping does not ensure fairness. For a more detailed exposition on abstraction, the reader is referred to [Abadi and Lamport 1991; Windley 1995a].

2.2.5 *Automata on Infinite Objects and Language Containment.* An ω-automaton (see e.g. [Thomas 1990]) is a tuple $\mathcal{A} = (\Sigma, Q, \bar{q}, \Delta, F)$, where Σ is a finite alphabet, Q a finite set of states, $\bar{q} \in Q$ the initial state, $\Delta \subseteq Q \times \Sigma \times Q$ a transition relation, and F a fairness constraint. Let $\Sigma^\omega \triangleq \{\alpha_0\alpha_1\alpha_2 \ldots \mid \alpha_i \in \Sigma, i \geq 0\}$ denote the set of infinite sequences (or ω-words) over Σ. A *run* of $\mathcal{A}$ on an ω-word $\alpha_0\alpha_1\alpha_2 \ldots \in \Sigma^\omega$ is a sequence of states $q = q_0q_1q_2 \ldots$ with $q_i \in Q$ such that $q_0 = \bar{q}$ and $(q_i, \alpha_i, q_{i+1}) \in \Delta$, for $i \geq 0$. A run q is called *successful* if it satisfies the fairness constraint. $\mathcal{A}$ is said to *accept* a word $\alpha \in \Sigma^\omega$ *iff* there exists a successful run of $\mathcal{A}$ on α. Let $\mathcal{L}(\mathcal{A}) \triangleq \{\alpha \in \Sigma^\omega \mid \mathcal{A} \text{ accepts } \alpha\}$ denote the *language* of $\mathcal{A}$.

A Büchi automaton [Thomas 1990] is an ω-automaton with a fairness constraint $F \subseteq Q$. A run q of a Büchi automaton is successful *iff* there is at least one state $p \in F$ such that p appears infinitely often in q. It can be shown that the class of languages accepted by Büchi automata is exactly the class of ω-regular languages, which have the form $\mathcal{L} = \bigcup_{i=1}^{n} U_i.V_i^\omega$, where U_i, V_i are regular languages, and $V^\omega \triangleq \{v_0v_1v_2 \ldots \mid v_i \in V, i \geq 0\}$. There are several variations of ω-automata, including Muller, Rabin, Streett, L- and $\forall$-automata (see [Clarke et al. 1990] for a comparison) as well as Edge-Street/-Rabin automata [Hojati et al. 1994], with the essential differences lying in the acceptance condition F. All of the aforementioned automata are equivalent in expressiveness to Büchi automata in that they all accept precisely the class of ω-regular languages. However, the more expressive fairness constraints of some types of automata allow for a more concise representation. For example, Hojati et al. [1994] have shown that one can construct a class of Edge-Street automata whose translation into L-automata incurs an exponential increase in size.

We can view an automaton $\mathcal{A}_M$ as a specification of a set of behaviours or traces of a system M if we identify its input alphabet with the externally observable states of M, and view its language $\mathcal{L}(\mathcal{A}_M)$ as the set of possible externally observable behaviours of M. Given the automata $\mathcal{A}_{M,I}$ and $\mathcal{A}_{M,S}$ for an implementation and a specification system M_I and M_S, respectively, trace inclusion between M_I and M_S amounts to language containment between the corresponding automata's languages, i.e. $\mathcal{L}(\mathcal{A}_{M,I}) \subseteq \mathcal{L}(\mathcal{A}_{M,S})$.

It can be shown that the expressive power of ω-automata is equivalent to the so-called *monadic second-order theory of one successor* or S1S (see [Thomas 1990]). Moreover, ω-regular languages are strictly more expressive than LTL; for example, one can show that the property "predicate q holds in every other state" cannot be expressed in LTL [Wolper 1983], whereas one can easily construct a Büchi-automaton which recognizes the language $\mathcal{L} = (q\Sigma)^\omega$. Wolper [1983] and Vardi and Wolper [1994] propose extensions to LTL where the temporal connectives are ω-automata; they show this class of temporal logics to be equivalent in expressiveness to ω-regular languages.

3. VERIFICATION TECHNIQUES AND TOOLS

Having introduced a number of frameworks for describing systems and asserting their conformance with a specification, we will now focus our attention on methods for *verifying* such assertions.

An important aspect is the degree of automation offered by a verification technique; the "ideal" verification tool would, given a system description and a specification, decide within an "acceptably short" amount of time whether or not the specification is met. Unfortunately, creating such a tool is often unrealistic in practice, if not impossible in theory due to incompleteness results [Duffy 1991]. One can in fact argue that it is unreasonable to always expect completely automatic verification of designs which result from an intellectual process involving sophisticated engineers.

Currently, available techniques range from completely automatic methods for verifying temporal logic specifications (section 3.1) and language containment (section 3.2) of finite state transition systems, to interactive theorem proving using logical calculi (section 3.3). Finally, section 3.4 surveys some approaches which integrate automatic and deductive approaches.

3.1 Model Checking

In section 2.1, we have introduced temporal logic as a framework for the specification of temporal properties of a design. In this section, we will turn our attention to the question of how to decide whether or not a given design satisfies a temporal formula. There exist algorithms which decide this question for structures with a finite state space completely automatically.

For a branching-time logic, a structure $M = (S, R, L)$ is said to be a *model* for a formula f if there exists a state $s \in S$ such that $M, s \models f$. For a *finite* structure M (one whose state space S has a finite number of elements), the question whether M is a model for a given formula is decidable. More generally, one can phrase the *model checking problem* as follows [Emerson 1990]: For a finite structure M and a formula f, label each state $s \in S$ with f *iff* $M, s \models f$ holds, or equivalently, compute the set $S_f \subseteq S$ such that $M, s \models f$ holds *iff* $s \in S_f$. The model checking problem for linear time logic can be phrased similarly in terms of paths. In many applications, a simpler problem statement is sufficient: "Does $M, s \models f$ hold for *all* states $s \in S$" for BTL, and "Does $M, x \models f$ hold for *all* paths $x \in X$" for LTL.

Even though the model checking problem for finite structures is decidable in principle, there does not necessarily exist a model checking algorithm with an acceptable time complexity (in terms of the sizes of the state space and the formula) for a given temporal logic. Generally, there is a tradeoff between the efficiency of decision procedures and the expressiveness of the underlying logic. For example, model checking for both propositional linear time logic and for CTL* has been shown to be PSPACE-complete [Sistla and Clarke 1985; Clarke et al. 1986], whereas algorithms with time complexity that is polynomial in the cardinality of S (the model's state space) exist for CTL. Worse, S tends to be very large in non-trivial applications. For a structure whose states are tuples of k Boolean variables, $|S| = 2^k$, i.e. the size of the state space is exponential in the number of state-holding elements.

In this section, we will first introduce the basic concepts behind model checking using a decision procedure for CTL as an example. In verification practice, an important factor is the choice of a "good" representation for sets of states. We will present an algorithm based on the symbolic representation of state sets as Binary Decision Diagrams, which are a suitable heuristic in many applications. Finally, we introduce a decision procedure for trajectory assertions over partially ordered

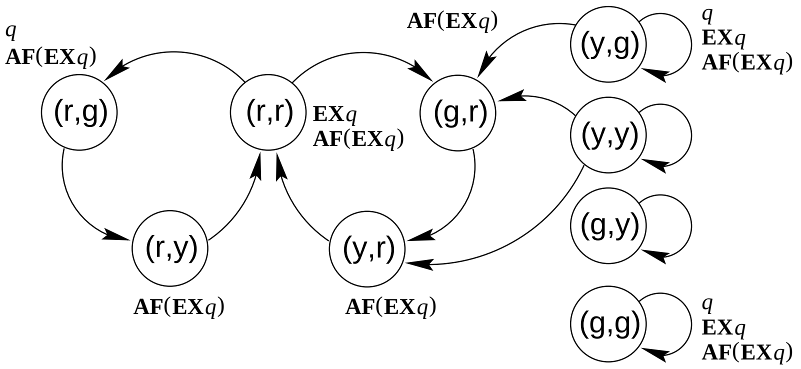

Fig. 4. State Transition Graph Labeled with Sub-formulas

state spaces. Section 3.2 presents algorithms for model checking based on language containment.

3.1.1 *Explicit State Model Checking.* Clarke et al. [1986] present a model checking algorithm for CTL. A CTL formula f is divided up into its sub-formulas $f_1, \ldots, f_k$, and the states of the state-transition graph associated with a structure $M = (S, R, L)$ are labeled with the sub-formulas which hold for a particular state. Let the size $|f|$ of a formula be the number of it's sub-formulas (i.e. the number of nodes in its parse tree, ignoring parentheses). The algorithm proceeds by successively labeling the states with sub-formulas of size $i = 1, \ldots, |f|$. Sub-formulas of size one must be atomic propositions; therefore L provides the initial labeling for $i = 1$.

For $i > 1$ and a sub-formula g of size $|g| = i$, we know that the state graph has already been labeled with the sub-formulas corresponding to the operands of the outermost operator of g, which must be of size smaller than i. We can determine if a state s needs to be labeled with g as follows. In the case $g = \neg g_1$, s is labeled with g if s is not labeled with g_1; the case $g = g_1 \wedge g_2$ is treated analogously. For $g = \mathbf{AX}\, g_1$ ($g = \mathbf{EX}\, g_1$), s is labeled with g if all (some) successors of g are labeled with g_1. For $g = \mathbf{E}[g_1 \,\mathbf{U}\, g_2]$, first every state that is labeled with g_2 is labeled with g. Second, any state that is labeled with g_1 and has a successor labeled with g is labeled with g. The second step is repeated until no further nodes can be labeled with g. This reachability computation can be performed in $O(|S| + |R|)$ time. A similar approach is used for $g = \mathbf{A}[g_1 \,\mathbf{U}\, g_2$. This yields a time complexity of $O(|f|(|S| + |R|))$ for checking a CTL formula algorithm.

It is only necessary to consider states that can be reached from a valid initial state. Let S_{init} be the set of valid initial states. Let S_{reach} be the set of states reachable states in S_{init}, and R_{reach} be the state transition relation restricted to $S_{reach} \times S_{reach}$. S_{reach} and R_{reach} can be computed from S_{init} and R in $O(|S_{reach}| + |R_{reach}|)$ time, and a CTL formula can be checked in $O(|f|(|S_{reach}| + |R_{reach}|))$ time. As S_{reach} is often *much* smaller than § this makes explicit model checking practical for much larger systems that could otherwise be considered. and R_{reach} can be computed from S

To illustrate this with an example, consider figure 4, which shows the state labeling that results from applying this algorithm to the traffic light controller from section 2.1.1.2 and the CTL formula

$$p \equiv \mathbf{AF}\,(\mathbf{EX}\,[ew = g]) \equiv \mathbf{A}[\mathit{true}\;\; \mathbf{U}\;\; \mathbf{EX}\,[ew = g]].$$

This formula has the sub-formulas true, $[ew = g]$ and $\mathbf{EX}\,[ew = g])$. First, all states are labeled trivially with true, and all states s with $[ew = g] \in L(s)$ are labeled with $[ew = g]$. In the figure, we use the abbreviation $q \triangleq [ew = g]$ and omit the *true* labels. In the next step, all states which have at least one successor labeled with q are labeled with $\mathbf{EX}\,[ew = g])$. This is the case for state (r, r) because of the transition $(r, r) \to (r, g)$, and for the states (y, g) and (g, g) because of their self-loops. In the third and final iteration, we label states with $\mathbf{A}[\mathit{true}\,\mathbf{U}\,\mathbf{EX}\,[\text{ew=g}]]$. For a state s, we have to verify that on all paths starting in s, all states on a (possibly empty) finite prefix are marked with *true*, and that this prefix is followed by a state marked $\mathbf{EX}\,[ew = g]$. Note that this is in particular not the case for state (y, y) because of its self-loop which gives rise to the path $(y, y), (y, y), \ldots$.

3.1.2 *Symbolic Model Checking.* The model checking technique described in the previous section requires that the entire state transition graph be constructed. Thus, the space requirements are at least linear in the size of the model's reachable state-space. However, the latter is in often exponential in the number of state-holding elements (latches) of a design. For instance, a device with only two 32-bit registers would already have ca. 10^{20} states.

An alternative to explicit enumeration of the reachable state space is to use a symbolic representation. Consider a design with n state holding elements (e.g. 1 bit registers). A state assigns a Boolean value to each of these registers and can be represented by the Boolean vector $V = (v_1, \ldots v_n)$. A *set* G of states can be represented by a Boolean function $G(V)$ which is *true iff* V represents a state in G. The design's state-transition relation $R \subseteq S \times S$ can be represented symbolically in a similar fashion [Burch et al. 1994]. Let $V' = \{v' | v \in V\}$ be the set of *next-state variables*. Then, represent R by a Boolean function $R(V, V')$ in V and V', such that $R(V, V')$ is *true iff* $(s, s') \in R$ and V, V' have valuations corresponding to states s and s', respectively.

When using a synchronous design style, state transitions correspond to the functions of combinational circuitry that maps the current state (i.e. the outputs of the latches/registers) to new values for each state variable. In other words, $v'_i = f_i(V)$. This leads to the next state relation for state variable v_i,

$$R_i(V, V') \;\equiv\; (v'_i = f_i(V)). \tag{7}$$

Combining these yields the symbolic representation of the state transition relation for the entire device:

$$R(V, V') \;\equiv\; \bigwedge_{i=1}^{n} R_i(V, V'). \tag{8}$$

When using an asynchronous design style, each component has conditions under which it is enabled to change the value of its outputs. Other than these conditions, components operate independently. The next state relation for an asynchronous design is typically obtained by taking the union of the relations for each component. This gives an "interleaving" semantics for concurrently enabled components. It also provides a convenient decomposition of the next state relation that can improve the efficiency of model checking algorithms.

3.1.2.1 *Model Checking Algorithm.* In section 2.1.4 we have seen how the denotational semantics of a temporal logic (the μ-calculus) can be given in terms of sets of states and fixpoints of operations thereon. The idea behind symbolic model checking is to use the symbolic representation for sets of states from the previous section to "implement" these semantics [Burch et al. 1990]. Operations on sets (e.g. intersection) correspond to Boolean operations on the characteristic functions, while the fixpoint operators are implemented by iterations (which converge because of model's finiteness).

In the following, we describe symbolic model checking for CTL [Burch et al. 1994]. A model checker for μ-calculus can be implemented analogously [Burch et al. 1990]. Let $S[f] \subseteq S$ denote the set of states for which a CTL formula f holds, and let $S[f](V)$ denote the Boolean characteristic function representing this set. We assume that the set of atomic propositions is the set of variables V. Then, for an atomic proposition $v \in V$, the set of states for which v holds is characterized by $S[v](V) = v$.

For a non-atomic formula f, the characteristic function $S[f](V)$ is computed recursively from the characteristic functions for the sub-formulas of f. This is straightforward for Boolean operations: For example, for $f = f_1 \wedge f_2$, we have $S[f] = S[f_1] \cap S[f_2]$ and thus $S[f](V) = S[f_1](V) \wedge S[f_2](V)$.

The next-state operators are expressed in terms of quantification over the transition relation[1]: for $f = \mathbf{AX}\, f_1$, we have $S[f](V) = F_{\mathbf{AX}}\,(S[f_1](V))$ with

$$F_{\mathbf{AX}}\,(Q(V)) \;=\; \forall_{v_1'} \cdots \forall_{v_n'}\; R(V, V') \Rightarrow Q(V'). \tag{9}$$

This is precisely equation (2) rephrased in terms of characteristic functions.

Similarly, we can use (3) to obtain the characteristic function for $f = \mathbf{AG}\, f_1$,

$$S[f](V) \;=\; \mu Q(V)\;.\; S[f_1](V) \wedge F_{\mathbf{AX}}\,(Q(V)).$$

The $\mathbf{A\,U}$ operator can be treated similarly while the other modalities are expressible in terms of the latter and the two above ones.

As indicated before, this approach is applicable to the construction of model checkers for the full μ-calculus [Burch et al. 1990], CTL^F [Burch et al. 1994]. LTL model checking is also possible, either directly [Burch et al. 1990], or via reduction to CTL^F model checking [Clarke et al. 1994].

An aspect of model checking which is extremely important in practice is the ability to generate counter-examples. Suppose we attempt to check that that $M, S_0 \models f$ for some CTL formula f, where S_0 denotes the set of initial states, and the verification fails. A counter-example is a path of M starting in a state $s \in S_0$ that violates f. For instance, in the case $f = \mathbf{AG}\, p$, a counter-example can be found essentially by starting with the set $S[\neg p]$, performing backwards reachability analysis until the set S_0 is reached, and picking states from the sets of states arising from

[1] Note that we have chosen to present the cases for the $\mathbf{AX}$ and $\mathbf{AG}$ operators; in [Burch et al. 1994], $\mathbf{EX}$ and $\mathbf{EG}$ is used.

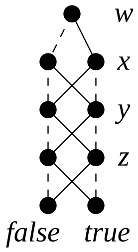

Fig. 5. Binary Decision Diagram for Odd-Parity of four variables

the iterative reachability computation. Hojati et al. [1993] describe a method for finding counter-examples for CTL^F formulae.

3.1.2.2 *Representation of Boolean Functions.* It is crucial for practical applicability of symbolic model checking that the Boolean characteristic functions involved are represented efficiently. Ordered Binary Decision Diagrams (OBDDs) [Bryant 1986; Bryant 1992] often work well in practice.

A Binary Decision Diagram is a directed, acyclic graph whose interior nodes are labeled with the variables which the Boolean function represented depends on, and which has two leaf nodes labeled *true* and *false*. Each interior node has two outgoing arcs labeled *true* and *false*, respectively, which correspond to the two possible valuations of the node's variable. For a given valuation of the variables, the value of the function represented is determined by the label of the leaf node which is reached by a traversal of the graph from the root along arcs whose labels correspond to the valuation.

An *ordered* binary decision diagram (OBDD) has the additional property that on every path between a root node and leaf nodes labeled *true* or *false*, no variable appears more than once, and on all such paths the variables appear in the same order. Subject to a reduction step [Bryant 1992] and for a fixed variable ordering, OBDDs are a canonical representation for Boolean functions in the sense that the OBDDs for two functions f_1 and f_2 are isomorphic *iff* $f_1 = f_2$.

Common sub-expressions are represented by multiple incoming edges to the same vertex. This sharing can make a BDD much more compact than other representations (e.g. sum-of-products form). For example, figure 5 shows the BDD for the function that computes the parity of four variables. "true" edges are drawn with solid lines and "false" edges are marked with dashed lines. The parity parity of n variables is represented by an OBDD with $2n + 1$ nodes, whereas the sum-of-products representation has 2^{n-1} minterms.

Operations on OBDDs can be implemented efficiently: The OBDD of a function f_1 *<op>* f_2, where *<op>* is a Boolean operation, can be computed from the OBDDs of f_1 and f_2 in time linear in the product of the sizes of the two OBDDs. Other operations such as quantification over a variable can be implemented efficiently as well.

Although OBDD operations are logically sound for any variable ordering, the choice of variable ordering can critically affect performance. For example, the size of the OBDD for a the most significant bit of the sum of two integers represented as bit-vectors is linear in the word size if the bits of the operands are interleaved (essentially, the OBDD represents the carry chain of a ripple-carry adder). On the other hand, if all of the bits of one operand word precede the bits of the other word, then the size of the OBDD will be exponential in the number of bits in the words. In this case, the OBDD must have a separate node for each integer value of the first word to "memorize" the value of this word so it can be added to the other argument. It practice, there are two main options for finding a suitable ordering: Intuition about the function to be represented indicates a good ordering. If this is not applicable, heuristics which dynamically adjust the variable order may still lead to an acceptable ordering [Rudell 1993].

There are functions such as integer multiplication for which no "good" variable ordering exists[Bryant 1991b]. In fact any representation of Boolean functions must use exponential space for "most" functions. The success of OBDDs is that they work well for many Boolean functions that occur when verifying real designs. Many researchers have explored variants of OBDDs to find representations that are more efficient for functions where OBDDs are impractical. These include EVBDDs [Lai and Sastry 1992], snDDs [Jain et al. 1996], MDGs [Corella et al. 1994], and HDDs [Clarke et al. 1995]. [Bryant 1995] provides a good overview of recent work in this area.

3.1.2.3 *Practical Issues in BDD-based Model Checking.* The time complexity of operations on BDDs is polynomial in the size of the BDDs involved. However, the BDD sizes themselves can be exponential in the number of BDD variables for ill-suited problems. However, there are several techniques which achieve a significant reduction in BDD size for certain applications.

An approach which aims at reducing the size of the BDD for the state transition relation R is to represent R by a list of implicitly conjoined BDDs for the $R_i(V, V')$ defined in equation (7) [Burch et al. 1994]. Alternatively, the relation R can be partitioned into simpler relations or functions [Lee et al. 1994a]. Even though the aggregate size of the individual BDDs is generally larger than the monolithic BDD for R, the individual BDDs tend to be small and are therefore more efficiently manipulated.

The second potential area of improvement is the size of the BDD used to represent intermediate results in the iterative computation of fixpoints. Hu et al. [1994] apply the idea of using implicitly conjoined BDDs to the characteristic functions of the intermediate sets of states arising from an iterative reachable state set computation. The method employs heuristics to recombine some of the implicitly conjoined BDDs to prevent the list from growing excessively, and applies a simplification procedure to pairs of BDDs which is based on the observation that one BDD defines a "don't care" set for second BDD with which it is conjoined.

Another technique is based on the elimination of state variables which can be expressed as functions of other state variables [Hu and Dill 1993]. During the fixpoint calculation, the functional dependencies are substituted for the eliminated variables. To ensure soundness, it is also necessary to verify at each iteration that the functional dependencies actually hold.

3.1.3 *Refinement and Model Checking.* Recall from section 2.2.4 that verifying refinement between an implementation M_I and a specification M_S generally requires showing that for every observable behaviour of M_I, there exists a corresponding observable behaviour of M_S. In many applications, this condition is verified using deductive methods (which will be described in section 3.3). However, under certain conditions, refinement can be expressed as a safety property, which permits treatment with automatic model checking if the systems have finite state.

Essentially, this is the case if the specification behaviour corresponding to a given implementation behaviour can be obtained through a state-by-state mapping. Assume the implementation and specification are given as transition systems $M_I = (S_I, S_{0,I}, R_I)$ and $M_S = (S_S, S_{0,S}, R_S)$. Assume further that there exists an abstraction function (or refinement mapping) $\mathcal{A} : S_I \to S_S$, such that the specification trace p_S corresponding to an implementation trace $p_I = s_0 s_1 s_2 \ldots$ is given by $p_I = \mathcal{A}(s_0)\mathcal{A}(s_1)\mathcal{A}(s_2)\ldots$. Then, M_I is a refinement of M_S, $M_I \sqsubseteq M_S$, *iff*

$$\begin{aligned} &\mathcal{A}(S_I) \subseteq S_M \\ \wedge\ &\forall_{s \in S_I} \forall_{s' \in S_I}\ (s, s') \in R_I \Rightarrow ((\mathcal{A}(s) = \mathcal{A}(s') \vee (\mathcal{A}(s), \mathcal{A}(s')) \in R_S)\,. \end{aligned} \tag{10}$$

These conditions (10) are satisfied *iff*

$$\mathcal{R}(s) = \forall_{s' \in S_I}\ (s, s') \in R_I \Rightarrow ((\mathcal{A}(s) = \mathcal{A}(s') \vee (\mathcal{A}(s), \mathcal{A}(s')) \in R_S)$$

is a safety property of the implementation M_I. Lee et al. [1994b] use safety properties of this form to verify refinement between programs written in ST [Staunstrup 1994], a Unity-like guarded command language. In this case, the transition relation is given as the union of transition relations corresponding to the individual guarded multi-assignments. The refinement predicate has the form of a product of clauses, one for each implementation transition, and each of which a sum of clauses for specification transitions. The BDD for $\mathcal{R}$ is built directly from the program texts for M_I and M_S and the expression for the abstraction mapping without explicit construction of the transition relation.

Abadi and Lamport [1991] provide conditions on M_I and M_S which ensure that, if M_I implements M_S in the sense of (5), there exists an abstraction function. Their framework also extends to the case where M_I and M_S are augmented with fairness properties.

An interesting question is, given a $M_I \sqsubseteq M_S$ and a temporal formula φ which holds for M_S, $M_S \models \varphi$, does φ also hold for M_I, i.e. does refinement preserve the truth of φ? If φ is a simple safety property as introduced in section 2.1.5, this is indeed the case and can be shown by a simple induction argument. Using a slightly different definition of refinement, Clarke et al. [1994] show that properties stated in $\forall$CTL* (CTL* restricted to universal path quantifiers only) are preserved as well.

3.1.4 *Symbolic Trajectory Evaluation.* In section 2.1.6, we introduced Trajectory Formulas (TF), which form a temporal logic of restricted expressiveness. Seger and Bryant [1995] describe an efficient model checking approach for TF called Symbolic Trajectory Evaluation (STE). STE uses a lattice representation of circuit states; we consider the case nodes have binary valued and are represented by the lattice $\mathcal{T} \triangleq \{X, 0, 1, \top\}$. Here, 0, and 1 denote low and high voltages respectively, X denotes an unknown or indeterminate value, and $\top$ represents an over-constrained value [2]. A circuit with n nodes is represented by a value on $\mathcal{T}^n$.

[2]This model is used in current versions of the Voss verification tool [Seger 1993; Hazelhurst and Seger 1995]. [Seger and Bryant 1995] refers to the previously used ternary model.

Values on $\mathcal{T}$ are partially ordered: $X \sqsubseteq 0, 1 \sqsubseteq \top$. This ordering is reflexive: for any value $a \in \mathcal{T}$, $a \sqsubseteq a$. Furthermore, $a \sqsubseteq b$ and $b \sqsubseteq a$ *iff* $a = b$. The partial order for $\mathcal{T}^n$ is the element-wise extension of the order for $\mathcal{T}$: $\alpha \sqsubseteq \beta$ *iff* $\alpha_i \sqsubseteq \beta_i$ for each component α_i of α and the corresponding component β_i of β. For any two elements of $\mathcal{T}^n$ there is a unique least upper bound: $\mathrm{lub}(\alpha, \beta) = \gamma$ *iff* $\alpha \sqsubseteq \gamma$, $\beta \sqsubseteq \gamma$, and for any δ with $\alpha \sqsubseteq \delta$ and $\beta \sqsubseteq \delta$, $\gamma \sqsubseteq \delta$. We extend the partial order $\sqsubseteq$ element-wise to sequences over $\mathcal{T}^n$ in the natural way.

An execution of the circuit model assigns a Boolean value (0 or 1) to each node at each time step. This can be represented by a sequence elements of $\mathcal{T}^n$. Let g be such a sequence. The fundamental observation behind STE is that for a given TF f, there exists a unique *defining sequence* δ_f such that g satisfies f *iff* $\delta_f \sqsubseteq g$.

We first construct δ_f formulas, f, without symbolic variables. If f is a trajectory formula, we write δ_f to denote the defining sequence for f.

. Simple predicates: $\delta_{\mathtt{a}_i \ \mathtt{is} \ v} = U(i, v), X_n^\omega$
where $U(i, v)$ is the lattice element whose i^{th} element is v and all other elements have value X, and X_n denotes the lattice element with every component equal to X.

. Conjunction: $\delta_{f \wedge g} = \mathrm{lub}(\delta_f, \delta_g)$
A trajectory that satisfies $f \wedge g$ must satisfy the constraints of both f and g. If f specifies that a node must be 0 on a time step when g specifies a value of 1 for the same node, then $f \wedge g$ will specify a value of $\top$ for that node on that time step. This indicates that f and g are *inconsistent* and can be flagged as a probable error in the specification.

. Next time: $\delta_{\mathbf{N}f} = X_n, \delta_f$
Intuitively, $\mathbf{N}f$ says nothing about the head of the defining sequence and makes the assertions of f about the tail of the sequence.

. Domain restriction: $\delta_{f \,\mathbf{when}\, 0} = X_n^\omega$, and $\delta_{f \,\mathbf{when}\, 1} = \delta_f$.
In words, $f \,\mathbf{when}\, 1$ is equivalent to f, and $f \,\mathbf{when}\, 0$ gives an unrestricted defining sequence.

Two observations are in order. First, conjunction has a natural interpretation as a least upper bound of lattice elements. Disjunction, on the other hand, does not correspond to the greatest lower bound, which is why disjunction and negation are not allowed in trajectory formulas. Second, even though defining sequences are infinite, after a finite prefix, every element of the sequence is X_n. The length of this prefix is one greater than the maximum nesting of next time operators in the trajectory formula. Accordingly, defining trajectories can be represented by a finite structure.

Symbolic formulas can be handled by representing each node, `nd`, with two symbolic Boolean formulas $\mathtt{nd}_0$ and $\mathtt{nd}_1$ which indicate when `nd` must be 0 and when `nd` must be 1 respectively. A valuation of the symbolic variables that satisfies neither $\mathtt{nd}_0$ nor $\mathtt{nd}_1$ corresponds to a X value, and a valuation that satisfies both corresponds to a $\top$ value. Simple predicates and next time operations have straightforward extensions to symbolic formulas. Conjunction is performed by defining lub in terms of logical operations over the $\mathtt{nd}_0$ and $\mathtt{nd}_1$ formulas for each node. Symbolic domain restriction, $f \,\mathbf{when}\, E$ is performed by computing the conjunction of each $\mathtt{nd}_0$ and $\mathtt{nd}_1$ formula for the defining trajectory of f with E. In practice, these manipulations are performed using BDDs as described in section 3.1.2.2.

Consider a trajectory assertion, $A \Rightarrow C$. If $\delta_C \sqsubseteq \delta_A$, then any trajectory that satisfies A also satisfies C, and the assertion holds regardless of the circuit's behaviour. To perform meaningful verification, the behaviour of the circuit must be considered. In particular, we only want to consider trajectories that satisfy A and are consistent with the circuit model. The circuit's behaviour is modeled with a function, $Y : \mathcal{T}^n \to \mathcal{T}^n$. This function must be monotonic: if $\alpha \sqsubseteq \beta$ then $Y(\alpha) \sqsubseteq Y(\beta)$. Intuitively, $Y(\alpha)$ may have X values if α has X values. If β can be obtained from α be replacing some X values with 0 or 1, then this must not result in more X values in $Y(\beta)$ than were present in $Y(\alpha)$.

In practice, Y can be obtained implicitly through symbolic simulation on a circuit model. The main distinction between a symbolic simulator and a conventional logic simluator is that it allows variables to appear in the inputs. The values of the circuit's internal nodes and outputs then become functions of these variables (see [Bryant 1991a]). With STE, the variables have domain $\mathcal{T}$; the values of nodes and outputs can be represented by pairs of BDDs as indicated above.

For concreteness, we have presented STE assuming that states are represented on the lattice $\mathcal{T}^n$, More formally, a model for STE is a pair, $\mathcal{M} = [\langle \mathcal{S}^n, \sqsubseteq \rangle, Y]$, where $\mathcal{S}$ denotes a set of states, $\langle \mathcal{S}, \sqsubseteq \rangle$ is a complete lattice and $Y : \mathcal{S} \to \mathcal{S}$ is a monotonic successor function. Intuitively, the partial order $\sqsubseteq$ represents an information ordering on $\mathcal{S}$, and Y places constraints on the successor state $Y(s)$ of a state s in the sense that $Y(s)$ denotes the "least specified" state which $\mathcal{M}$ can reach from r.

Given a defining sequence δ_A, the corresponding *d*efining trajectory is

$$\tau_A^i \triangleq \begin{cases} \delta_F^0 & \text{for } i = 0, \\ \mathrm{lub}(\delta_F^i, Y(\tau_F^{i-1})) & \text{for } i > 0. \end{cases} \tag{11}$$

The i^{th} element of τ_A corresponds to the least specified state that the circuit can reach at step i given that A holds in all states up through state i.

We can now state the central theorem of STE:

THEOREM 1. *For a model structure $\mathcal{M}$ and trajectory formulas A, C,*

$$\models_{\mathcal{M}} [A \Rightarrow C] \quad \Leftrightarrow \quad \delta_C \sqsubseteq \tau_A.$$

This theorem implies that to verify a trajectory assertions, it is sufficient to compute the sequences τ_A and δ_C and compare them with respect to the partial order $\sqsubseteq$. The sequence δ_C can be computed efficiently [Seger and Bryant 1995]; τ_A is computed according to (11), requiring a single symbolic simulation. As noted above, every element of δ_C is X_n after a finite prefix; therefore it is sufficient to compute and compare only these prefixes of τ_A and δ_C.

There are two serious limitations in the STE framework: Negation is not trajectory formulas, and the next-state function is essentially restricted to be deterministic. Addressing the first restriction, ?] has extended STE to a richer logic supporting negation and an "until" operator; however, efficient model checking is not always possible for this logic. ?] have developed a framwork which allows the application of STE to certain classes of speed-independend circuits.

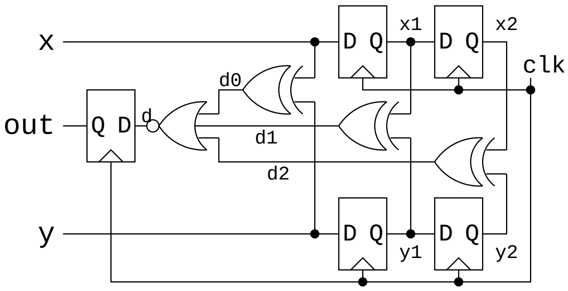

Fig. 6. Implementation of a Simple Correlator

3.1.4.1 *An Example.* Consider the correlator specified in equation 4. Figure 6 shows an implementation of this correlator. A state of this circuit can be represented by the tuple $(\mathtt{x}, \mathtt{y}, \mathtt{out}, \mathtt{x1}, \mathtt{y1}, \mathtt{x2}, \mathtt{y2},)$. Note that `d0`, `d1`, `d2`, and `d` are combinational functions of the other nodes. Let A denote the antecedent of the first assertion from equation 4:

$$A \;=\; \mathrm{x}\,\mathbf{is}\,a) \wedge (\mathrm{y}\,\mathbf{is}\,a) \wedge (\mathbf{N}(\mathrm{x}\,\mathbf{is}\,b) \wedge \mathbf{N}(\mathrm{y}\,\mathbf{is}\,b)) \wedge (\mathbf{N}^2(\mathrm{x}\,\mathbf{is}\,c) \wedge \mathbf{N}^2(\mathrm{y}\,\mathbf{is}\,c))$$

The consequent, C, is $\mathbf{N}^3(\mathtt{out}\,\mathbf{is}\,1)$. The defining sequence for C, δ_C is

$$\begin{array}{l}
0: ((F,\ F),(F,\ F),(F,\ F),(F,\ F),(F,\ F),(F,\ F),(F,\ F))\\
1: ((F,\ F),(F,\ F),(F,\ F),(F,\ F),(F,\ F),(F,\ F),(F,\ F))\\
2: ((F,\ F),(F,\ F),(F,\ F),(F,\ F),(F,\ F),(F,\ F),(F,\ F))\\
3: ((F,\ F),(F,\ F),(F,\ T),(F,\ F),(F,\ F),(F,\ F),(F,\ F))
\end{array}$$

Note that we only need to consider the first four elements of the defining sequence as all others describe completely unconstrained states. Where the first element of each pair is the equation for the condition under which the corresponding node is 0 and the second is the condition under which the node is 1. This asserts $\mathtt{out}\,\mathbf{is}\,1$ at step 3. The defining sequence for A, δ_A, is:

$$\begin{array}{l}
0: ((\neg a, a\ ),(\neg a, a\ ),(F,\ F),(F,\ F),(F,\ F),(F,\ F),(F,\ F))\\
1: ((\neg b, b\ ),(\neg b, b\ ),(F,\ F),(F,\ F),(F,\ F),(F,\ F),(F,\ F))\\
2: ((\neg c, c\ ),(\neg c, c\ ),(F,\ F),(F,\ F),(F,\ F),(F,\ F),(F,\ F))\\
3: ((F,\ F),(F,\ F),(F,\ F),(F,\ F),(F,\ F),(F,\ F),(F,\ F))
\end{array}$$

which asserts that `x` and `y` both have value a on step 0, both have b on step 1, and both have c on step 2. Their values are unconstrained on step 3. Again, we only need to consider the first four elements of the defining sequence as the consequent places no constraints on subsequent states. To compute the defining trajectory for A, we need the next state function Y, for which we assume the obvious interpretation of figure 6. This yields the defining trajectory of A, τ_A, shown below:

$$\begin{array}{l}
0: ((\neg a, a\ ),(\neg a, a\ ),(F,\ F),(F,\ F),(F,\ F),(F,\ F),(F,\ F))\\
1: ((\neg b, b\ ),(\neg b, b\ ),(F,\ F),(\neg a, a\ ),(\neg a, a\ ),(F,\ F),(F,\ F))\\
2: ((\neg c, c\ ),(\neg c, c\ ),(F,\ F),(\neg b, b\ ),(\neg b, b\ ),(\neg a, a\ ),(\neg a, a\ ))\\
3: ((F,\ F),(F,\ F),(F,\ T),(\neg c, c\ ),(\neg c, c\ ),(\neg b, b\ ),(\neg b, b\ ))
\end{array}$$

It is straightforward to show that $\delta_C \sqsubseteq \tau_A$, verifying the first assertion of equation 4. The second assertion can be verified in a similar manner.

3.1.5 *Model Checking Tools.* In this section, we give a brief overview of available tools which implement the methods presented in the previous sections.

EMC (Extended Model Checker) was one of the first model checkers to be implemented [Clarke et al. 1983; Clarke et al. 1986]. It constructs an explicit representation of the state graph from a program written in a subset of CSP, and supports model checking of formulae in CTL and CTL^F.

Murφ [Dill et al. 1992; Dill 1996] is both a Unity-like description language and an explicit state model checking system. Specifications in Murφ are given as simple safety properties. The verifier constructs the reachable state space by storing already visited states in a hash-table, and checks safety properties as it goes along. Extensions to the Murφ system exploit techniques to reduce the size of the representation of the reachable state space [Ip and Dill 1993; Ip and Dill 1996a; Ip and Dill 1996b]. Ever [Hu et al. 1992] is a BDD-based symbolic model checker which can act as a back-end for Murφ; however, it is suitable for use as a stand-alone tool as well. The Ever language supports high-level constructs such as array and record types, as well as imperative statements such as sequential composition.

SMV [McMillan 1992; Clarke et al. 1996] is probably the most widely used symbolic model checker to date. System descriptions in the SMV language are given in terms of a set of equations which determine the next-state relation; programs may be structured into parameterized modules. SMV model checks specifications given in CTL and CTL^F. A recent version supports word-level model checking [Clarke et al. 1996] by using Hybrid Decision Diagrams (HDDs) [Clarke et al. 1995] as the underlying data structure, which permits model checking of properties involving words, i.e. bit vectors interpreted as integers.

CVE [Bormann et al. 1995] is a symbolic model checking environment. It supports model checking of designs described in a subset of VHDL or EDIF against specifications given in a temporal logic called CIL, which is equivalent to a subset of CTL. RuleBase [Beer et al. 1996] is an industry-oriented model checking tool which uses SMV as its core, and provides a graphical user interface, a simpler temporal logic built on top of CTL, support for VHDL and Verilog, and powerful debugging support. VIS [Brayton et al. 1996a; Brayton et al. 1996b] is a tool integrating model checking with other verification techniques such as combinational and sequential equivalence checking (see section 3.2.2). VIS accepts design descriptions in a synthesizable subset of Verilog, and supports CTL^F model checking. Interaction with the SIS synthesis tool [Sentovich et al. 1992] is provided through a common intermediate format.

Voss [Seger 1993] is a verification system based on symbolic trajectory evaluation. Trajectories are evaluated using a symbolic simulator which can operate either on a switch-level model [Bryant et al. 1987] of a transistor netlist, or a state machine extracted from a gate-level netlist or a description in a subset of VHDL. Voss provides a meta-language front-end in the form of `fl`, an ML-like but fully lazy functional language.

3.2 Automata-Theoretic Approaches

This section describes two aspects of automata-theoretic approaches for formal verification. The first is concerned with the verification of containment between languages recognized by automata on infinite words. Here, automata are mainly a tool to capture the properties of the languages in question. The second aspect originates from the observation that deterministic finite state automata are a natural model for sequential circuits. In this context, the verification goal is usually to show observational equivalence between two automata.

3.2.1 *Automata on Infinite Objects and Language Containment.* Recall that the language containment problem involves verifying that the language recognized by an ω-automaton $\mathcal{X}$ is a subset of the language recognized by a second ω-automaton $\mathcal{Y}$, i.e. $\mathcal{L}(\mathcal{X}) \subseteq \mathcal{L}(\mathcal{Y})$ (see section 2.2.5).

A decision procedure for the language containment problem can be derived by recasting containment into the language emptiness problem. Given finite automata $\mathcal{X}$ and $\mathcal{Y}$ with input alphabet Σ, a product automaton, $\mathcal{U}$, with $\mathcal{L}(\mathcal{U}) = \mathcal{L}(\mathcal{X}) \cap (\Sigma^\omega - \mathcal{L}(\mathcal{Y}))$ can be constructed [Thomas 1990]. By this construction, $\mathcal{L}(\mathcal{X}) \subseteq \mathcal{L}(\mathcal{Y})$ *iff* $\mathcal{L}(\mathcal{U}) = \emptyset$. Using Büchi acceptance criteria, automaton $\mathcal{U}$ accepts some word *iff* there exists a state $s \in F$ which is reachable from $\bar{q}$ and which is reachable from itself [Thomas 1990]. Because the set of states of $\mathcal{U}$ is finite, the language emptiness problem can be decided by reachability analysis of $\mathcal{U}$'s state graph.

Clarke et al. [1990] propose an elegant way to express the language emptiness problem as a model checking problem, which they use to obtain decision procedures for language containment between different types of ω-automata. The underlying idea is to construct a temporal structure $M = (S, R, L)$ from a given ω-automaton $\mathcal{X} = (\Sigma, Q, \bar{q}, \Delta, F)$, with $S = Q$, $L = \{\{s\} \mid s \in Q\}$ and $R = \{(s, s') \mid \exists_{\alpha \in \Sigma}\ (s, \alpha, s') \in \Delta\}$, i.e. M has the same state space as $\mathcal{X}$, and has transitions where $\mathcal{X}$ has a transition enabled by at least one input symbol. For Büchi acceptance, language emptiness can be expressed in CTL* as shown below:

$$\mathcal{L}(\mathcal{X}) = \emptyset \quad \Leftrightarrow \quad M, \bar{q} \models \neg\mathbf{E}[\mathbf{GF}(\bigvee_{s \in F} s)].$$

Thus, the methodologies developed for model checking problems can be applied to language emptiness as well.

Kurshan [1989] defines L-automata and L-processes. The main difference to other ω-automata is that their transitions are labeled with elements of a Boolean algebra L, rather than simple elements of an input alphabet. A transition is enabled if the current input symbol is contained in the set of symbols of its label. L-processes are defined in a way which facilitates the modeling of coordination of processes as a *tensor*-product. For the details of these definitions, the reader is referred to [Kurshan 1989].

Let $\mathcal{X}$ be an L-automaton, a specification, and let $\mathcal{Y}$ be an L-process, a proposed implementation of $\mathcal{X}$. Verification requires showing that the language of the implementation, $\mathcal{L}(\mathcal{Y})$ is contained in the language of the specification $\mathcal{L}(\mathcal{X})$. In cases where direct verification is not feasible, it may be possible to find simpler automata $\mathcal{X}'$ and $\mathcal{Y}'$ such that showing $\mathcal{L}(\mathcal{Y}') \subseteq \mathcal{L}(\mathcal{X}')$ establishes $\mathcal{L}(\mathcal{Y}) \subseteq \mathcal{L}(\mathcal{X})$ as well. Kurshan formalizes the relationships that must hold between $\mathcal{X}, \mathcal{X}'$ and $\mathcal{Y}, \mathcal{Y}'$ using homomorphisms between the languages and the underlying Boolean algebrae. This provides a formal basis for the simplifying abstractions that are often used with model checking or language containment algorithms and allows larger systems to be verified without resorting to unverified assumptions. These techniques are implemented in the COSPAN verification tool [Hardin et al. 1996].

3.2.2 *State Machine Equivalence.* A special case of the verification condition "device X' implements specification X" arises when both X and X' are deterministic and indeed exhibit *identical* behaviours when operated under the same environment behaviour. For simplicity, we consider Mealy machines where internal states, inputs and outputs are represented by Boolean vectors. Formally, such a machine is given by a tuple $X = (\Sigma, \Omega, S, s_0, \delta, \lambda)$, where $\Sigma = \mathcal{B}^l$ denotes the input alphabet, $\Omega = \mathcal{B}^m$ the output alphabet, $S = \mathcal{B}^n$ the set of states, $s_0 \in S$ the initial state, $\delta : S \times \Sigma \to S$ the state transition function, and $\lambda : S \times \Sigma \to \Omega$ the output function. In a realization of X as a sequential circuit, the state S corresponds to the circuit's latches; Σ and Ω correspond to the circuit's input and output wires; and δ and λ are realized by combinational circuits.

Two machines X and X' with $\Sigma = \Sigma'$ and $\Omega = \Omega'$ are considered equivalent *iff* for each input sequence $(\sigma_1, \sigma_2, \dots) \in \Sigma^*$, both machines generate the same output sequence $(\omega_1, \omega_2, \dots) \in \Omega^*$. A notion of equivalence which also relates the generated sequences of states can be accommodated by extending the output function such that it "forwards" the relevant information about the internal state to the outputs.

The problem of showing that two sequential machines are equivalent can be reduced to the problem of finding the reachable state set of a product machine. To show that machine X and X' are equivalent, one can construct the product machine $M^\times = X \times X' = (\Sigma \times \Sigma', \mathcal{B}, S \times S', (s_0, s_0'), \delta^\times, \lambda^\times)$, where $\delta^\times((s, s'), \sigma) = (\delta(s, \sigma), \delta'(s', \sigma))$ and $\lambda^\times((s, s'), \sigma) = (\lambda(s, \sigma) \equiv \lambda'(s, \sigma))$. Intuitively, $X \times X'$ is a machine that runs X and X' in lock-step on the same inputs, and whose output is 1 *iff* the outputs of X and X' agree. To show that X and X' are equivalent, it is sufficient to compute the set of reachable states Q of $M^\times$. Then, one can verify that $M^\times$ produces output 1 for all states $s \in Q$, and for all possible input vectors [Coudert et al. 1989].

The reachable state set of a sequential machine can be computed using a fixpoint calculation similar to the one used in the model checking algorithm in section 3.1.2. Coudert et al. [1989] present an algorithm for state machine equivalence where they convert the computation of a set's image under the next-state function to a computation of the range of a function derived from the original next-state function.

Their algorithm introduces "constrain" and "restrict" operators that allow them to describe the next-state function with its domain restricted to a particular set. Pixley et al. [1994] uses similar techniques to find synchronizing sequences for sequential circuits.

Alternatively, the next-state function can be viewed as a relation between successive states: a state, s_i is reachable at step i if there is a state, s_{i-1}, reachable at step $i-1$ and input I_{i-1}, such that applying the next state function to s_{i-1} and I_{i-1} leads to state s_i. Direct evaluation of this approach requires creating a formula with a complete set of variables for s_{i-1} and another set for s_i. Touati et al. [1990] observe that properties of the restrict operator allow this existential quantification to be pushed into clauses for small sets of variables of s_i, with an improvement of efficiency arising from the smaller formulas (i.e. BDDs) for the intermediate computations.

3.2.2.1 *Combinational Equivalence.* For two machines X and X' whose state space and initial states are identical, it is a sufficient condition for their equivalence that $\delta = \delta'$ and $\lambda = \lambda'$. In other words, given two sequential circuits using the same state encoding, their equivalence may be established by showing combinational equivalence for the realizations of their next-state and output functions.

Kuehlmann et al. [1995] describe an industrial sequential circuit verification system called Verity which exploits this observation. Verity represents the next-state and output functions as OBDDs and uses dynamic variable ordering (see sec. 3.1.2.2). The functions are extracted from gate- and/or switch-level netlists (i.e. mixed representations are allowed) using a path-based extraction scheme. Verification involves comparing the extracted functions for X and X', subject to a constraint $\mathcal{C} : \mathcal{B}^l \rightarrow \mathcal{B}$ on the input vectors. $\mathcal{C}$ is used to characterize the set of allowed input vectors as a proper subset of $\mathcal{B}^l$. Furthermore, a output constraint $\mathcal{O} : \mathcal{B}^m \rightarrow \mathcal{B}$ can be specified and checked, which characterizes the set of possible generated output vectors as a subset of $\mathcal{B}^m$.

A hierarchical approach is also supported in which the design is specified as a hierarchy of cell (or macro) instances. Leaf nodes in the hierarchy are verified directly. Non-leaf nodes can be handled by abstracting away (some of) their sub-cells. Inputs to the sub-cells are treated as outputs in the comparison of the super-cell, and it is checked that input constraints asserted for the sub-cells indeed hold as output constraints for the super-cell. Conversely, outputs of the sub-cells are treated as input variables for the comparison of the super-cell, and the output constraints asserted for the sub-cells are used as input constraints for the super-cell.

3.2.3 *Language Containment and Model Checking.* There is a close connection between ω-automata and model checking. In section 3.2.1, we have already seen that the language containment problem for two ω-automata can be reduced to a model checking problem. We will now point out that the converse holds as well.

The basis of this approach to model checking is to view a temporal formula as an acceptor of infinite structures, and a temporal structure as a generator thereof [Vardi and Wolper 1986a]. For simplicity, we describe the approach for linear time logic first.

An LTL formula φ over a set of atomic propositions $\mathcal{P}$ can be understood as an acceptor of infinite words over the alphabet $\Sigma = 2^{\mathcal{P}}$ in the sense that φ accepts a word $x \in \Sigma^\omega$ *iff* x is a model of φ. Thus a formula φ gives rise to a language $\mathcal{L}_\varphi$ containing precisely the words accepted by φ. It can be shown that one can construct an ω-automaton $\mathcal{A}_\varphi$ from a formula φ in LTL, such that $\mathcal{A}_\varphi$ accepts exactly the language defined by φ, i.e. $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}_\varphi$ [Manna and Wolper 1984; Kesten et al. 1993; Vardi and Wolper 1994].

On the other hand, a (finite) temporal structure $M = (S, R, L)$ can be interpreted as a generator of ω-words over $2^{\mathcal{P}}$; the language generated is the set

$$\mathcal{L}(M) = \{L(s_0)L(s_1)L(s_2)\ldots | s_0, s_1, \ldots \text{ is a path of } M\}.$$

M can be straightforwardly extended into a Büchi automaton $\mathcal{A}_M$ which accepts exactly $\mathcal{L}(M)$ by labeling the transitions $(s,t) \in R$ with the atomic propositions $L(s)$.

Using these observations, we can now rephrase the model checking problem as a language containment problem [Vardi and Wolper 1986a]: $M \models \varphi$ *iff* $\mathcal{L}(\mathcal{A}_M) \subseteq \mathcal{L}(\mathcal{A}_\varphi)$, or in other words, every sequence generated by M is accepted by φ in the above sense.

In the automata-theoretic framework, model checking can be extended to infinite state spaces by constructing the product automaton for the language containment test incrementally [Manna et al. 1994]. This results in a semi-decision procedure, i.e. the model checker may either terminate with a positive answer or a counter-example, or not terminate at all.

The corresponding automata-theoretic approaches for branching time logics require the use of automata on infinite trees, rather than words, to reflect the branching structure of time (see [Vardi and Wolper 1986b; Emerson 1990; Bernholtz et al. 1994]).

3.3 Deductive Methods

As we have seen in section 2.2.2, specifications and verification conditions can be expressed in general purpose logic. Verifying that an implementation meets its specification in such a framework is equivalent to proving a theorem in the underlying logic. In principle, this proof could be carried out on paper. However, proofs of such theorems are often long and rather tedious in practice, making it likely that they contain errors. Using a mechanized theorem proving system can ensure soundness and reduce tedium by automating parts of the proof.

3.3.1 *Automated Theorem Proving.* A theorem proving program mechanizes a proof system. A *calculus* or *proof system* $\mathcal{C}$ for a logic $\mathcal{L}$ consists of a set of *axioms* $\mathcal{A}$ and a set of *inference rules* $\mathcal{R}$. The axioms are formulae of $\mathcal{L}$ and are generally "elementary" in the sense that they capture the basic properties of the logic's operators. The general form of an inference rule is

$$\frac{\alpha_1, \ldots, \alpha_k}{\beta}.$$

The formulae $\alpha_1, \ldots, \alpha_k$ are called the *premises* of the rule while β is called the *conclusion*. For example, the formulae $p \Rightarrow (p \vee q)$ and $(p \wedge q) \Rightarrow q$ are axioms, and

$$\frac{p, p \Rightarrow q}{q}$$

is an inference rule of the "classical" propositional calculus [Kleene 1967]. Both the axioms and the inference rule are actually axiom and rule *schemas*, i.e. they give rise to an infinite number of axioms and rules obtained by replacing p and q with arbitrary formulae.

A *deduction* in $\mathcal{C}$ of a formula φ_n from a set of assumptions Γ is a finite sequence of formulae $\varphi_1, \ldots, \varphi_n$ such that for all $i = 1, \ldots, n$, either $\varphi_i \in \mathcal{A} \cup \Gamma$, or φ_i can be obtained by applying an inference rule in $\mathcal{R}$ to some subset of $\{\varphi_1, \ldots, \varphi_{i-1}\}$. We say that a formula φ is a *theorem* of Γ, written $\Gamma \vdash \varphi$, if there exists a deduction of φ from Γ. A calculus is said to be *sound* if for all sets of formulae Γ, all theorems of Γ are logical consequences of Γ, and *complete* if all logical consequences of Γ are also theorems (i.e. can be proven in $\mathcal{C}$). For a more complete treatment, see e.g. [Duffy 1991].

The discussion so far pertains to rigorous manual proofs as much as to automated systems. However, there are three aspects which are particular to mechanized proof systems:

(1) It can mechanically *check* a proof, i.e. verify that a given sequence of formulae $\varphi_1, \ldots, \varphi_n$ is indeed a deduction. This is usually not a very difficult task; all that is required is, for each φ_i, to syntactically match the inference rules against the appropriate premises in the sequence and verify that φ_i is indeed obtained by application of the rule.

(2) It can assist in the *construction* of a proof. Given a set of assumptions Γ and a goal β, heuristic search techniques may be able to find a deduction such that $\beta = \varphi_n$. In general this will not succeed completely automatically for "difficult" theorems, but will rather require a certain amount of human guidance to construct a deduction.

(3) It permits the use of *decision procedures*. A decision procedure is an algorithm which decides the validity of a class of formulae. For example, BDDs [Bryant 1992] provide a practical decision procedure for propositional logic. Other decision procedures exists for Presburger arithmetic (essentially integer arithmetic with multiplication by constants only) [Shostak 1979] and even real algebra [Tarski 1951; Harrison 1993]. There are also algorithms for combinations of theories such as real arithmetic, equality with uninterpreted functions and arrays under *store* and *select* [Nelson and Oppen 1979; Shostak 1984].
One can view a decision procedure as giving rise to a class of axioms similar to an axiom schema, except that the class is generally much larger. For example, the axiom schema $p \Rightarrow (p \vee q)$ just generates axioms of this particular form, while a propositional decision procedure gives rise to all propositional tautologies.
Decision procedures are important in practice since they can handle many "uninteresting" cases automatically and thus help to alleviate the tedium of proofs.

A myriad of theorem proving systems have been implemented, and many have been used for hardware verification, including HOL [Gordon 1988], ISABELLE [?], LP [Garland and Guttag 1989], Nqthm and ACL2 [Boyer and Moore 1979; Kaufmann and Moore 1996], Nuprl [?], OTTER [McCune 1994], PVS [Owre et al. 1992], RRL [?], and TLP [Engberg et al. 1992]. These systems are distinguished by, among other aspects, the proof style used, the mathematical logic used, the way automatic decision procedures are integrated into the system, and the user interface. Proofs styles are often characterized as forward or backward. A forward proof starts with the axioms and assumption; then, inferences are applied until the desired theorem has been proven. A backward proof starts with the theorem as a goal and applies the inverses of inference rules to reduce the theorem to simpler intermediate goals. Sufficiently simple goals are discharged by matching axioms or assumptions or by applying built-in decision procedures.

General purpose theorem provers are usually based on some form of first- or higher-order logic. In a first-order logic, variables (e.g. Boolean and integers) are distinguished from functions (e.g. a mapping from integers to Booleans). In a first order logic, quantification is allowed over variables but not over functions. Higher-order logic also allows quantification over functions. The choice of a logic influences the logical idioms used to write specifications and describe designs as well as influencing the proof-techniques and decision procedures that can be used to verify a design. As illustrated by the examples in section 4, both first- and higher-order logic have been successfully employed for hardware verification.

3.3.2 *Theorem Proving Systems.* In this section, we give brief descriptions of some commonly used automated theorem provers.

Higher Order Logic (HOL). The HOL system [Gordon 1988] is a general-purpose theorem prover whose underlying logic, called HOL as well, is a variation of Church's theory of simple types [Church 1940]. HOL is a descendant of the earlier LCF system [Gordon et al. 1979]. In contrast to first order predicate logic, higher order logic allows quantification not only over variables but also higher-order predicates and functions, which can result in a more concise and readable formalization of certain concepts.

HOL supports both forward and goal-directed backward proofs in a natural-deduction-style calculus. HOL does not use decision procedures; ultimately, all theorems are proven in terms of the eight basic inferences of the calculus. ML is used as a meta-language with which the user interacts with the HOL system. The user guides the system by applying *tactics* to proof obligations; a tactic corresponds to a high level proof step and automatically generates the sequence of elementary inferences necessary to justify the step. Tactics can be composed into even larger steps using *tacticals* such as "apply tactics A, B, and C repeatedly until no further simplification is obtained" An important aspect of the system is user defined tactics cannot compromise the soundness of a proof because sequents are implemented as an abstract data type in ML. Therefore, valid theorems can be created only through application of the basic inferences.

PVS. The logic of PVS (Prototype Verification System) [Owre et al. 1992; Owre et al. 1996] is a strongly typed higher order logic with a rich type system which includes dependent types and predicate subtypes. Type checking in this logic is undecidable and requires the assistance of the theorem prover and possibly human intervention to discharge automatically generated type correctness conditions. Definitions and theorems can be grouped into parameterized theories whose parameters may have constraints attached. Specifications are written in a Lisp dialect which offers constructs such as tuples, records, arrays and tables.

Proofs in PVS are backward proofs using a sequent representation for proof obligations. The inference rules operate at a higher level than the primitive inferences of higher order logic and include Skolemization and instantiation, application of theorems, equality rewriting as well as complex propositional rewriting such as case-split and IF-lifting. Higher level proof strategies analogous to HOL tactics can be constructed from the basic inference rules using a specialized strategy language.

To discharge certain classes of obligations, PVS employs an integrated decision procedure for equality reasoning, linear arithmetic, arrays etc. as well as a BDD-based procedure for propositional logic. The decision procedures are integrated with the type checker to make use of additional constraints available from type information. PVS also includes a model checker for finite-state μ-calculus which can be accessed through a formalization of the μ-calculus within the PVS logic [Rajan et al. 1995].

Boyer-Moore. The Boyer-Moore logic is a first order, quantifier-free logic of total, recursive functions with equality and mathematical induction [Boyer and Moore 1979]. The syntax of the logic is that of applicative Common Lisp. *Nqthm* is a mechanization of this logic [Boyer and Moore 1988]; a re-engineered and slightly extended version has been released recently under the name *ACL2* [Kaufmann and Moore 1994; Kaufmann and Moore 1996]. The description here is based on ACL2.

ACL2 contains axiomatizations of primitive data types such as numbers and lists. New functions can be introduced through a definitional principle which requires proof of termination for recursively defined functions and ensures that consistency is maintained.

ACL2 combines backward and forward methods to prove theorems. In the backward phase, a pending obligation is chosen, and the prover attempts to discharge it by applying a sequence of more and more general proof techniques. First, a simplifying stage applies conditional and congruence-based rewrites, a BDD-based propositional decision procedure, a linear arithmetic decision procedure and other simplification techniques to the obligation. If this step fails to reduce the obligation to *true*, ACL2 tries other proof techniques, the most general of which attempts to discover an induction scheme for the obligation.

The forward aspect of ACL2 is that most of the proof techniques are rule-driven. Previously proved theorems can be turned into rules which are added to a rule database and can be used by later proofs. For example, once one has proved the associativity of a certain operator, this theorem can be converted into a rewrite rule which will then be used by the simplifier rewrite occurrences of this operator in an obligation.

The user guides the theorem prover by starting with simple lemmas that can be proven directly and converted into new rules. These rules are then used to prove successively more difficult lemmas until finally the main theorem is proven. ACL2 allows definitions, lemmas and theorems to be collected in *books*. A book may have local lemmas which are invisible outside the book and whose only purpose is to guide the prover to the proof for the main theorems exported by the book. This feature (which was not available in Nqthm) facilitates the development of reusable theories.

3.3.3 *Proof Strategies for Hardware Verification.* As we have seen in the previous section, proof discovery in most theorem provers is guided by the user through the application of proof strategies or tactics. Standard tactics operate at a level of detail such as quantifier instantiation or rewrite with respect to a set of equalities. However, hardware proofs tend to follow similar general patterns, which suggests the development of proof strategies that automatically discharge more complex obligations.

[Aagaard et al. 1993] report on a hardware proof strategy for both HOL and Nuprl which automatically solves simple goals of the form (6). The tactic automatically expands definitions used in the implementation, then applies heuristics for instantiating quantifiers and for case analysis. Finally, it attempts to to discharge obligations through rewriting.

Cyrluk et al. [1994] describe a similar proof strategy which was implemented in PVS. This strategy applies to correctness conditions similar to (6), where both the specification and implementation are given in terms of a state transition function. A naive strategy would fully expand the definitions of the specification, the implementation and the abstraction mapping, subject the result to case analysis and verify all the cases using the decision procedures. However, the resulting intermediate expression becomes much too large for non-trivial examples. A better strategy first applies a limited form of rewriting, performs a case-split on the number of cycles the implementation has to run and then successively applies first a simple and if this fails an aggressive rewrite strategy, further case-split and decision procedures to the resulting cases. This strategy is capable of completely automatically verifying the Tamarack microprocessor [Joyce 1988], a relatively simple microcoded design often used as a benchmark example. A similar approach is described in [Kumar et al. 1993].

3.4 Combining Model Checking and Deductive Reasoning

In the previous sections, we have surveyed formal verification techniques ranging from highly automated, model-checking-type approaches to theorem-proving-based methods which require a considerable amount of human interaction. A verifier faces the trade-off between choosing an automated technique, which is in general subject to limitations in it applicability (such as being confined to finite systems), and more general, but also more work-intensive approaches.

However, this trade-off can be avoided in certain cases where it is possible to apply automated techniques to subsystems or simpler obligations and then use deductive approaches to combine the results thus obtained into an overall correctness result. In this section, we will survey a number of tools which exploit this approach. Related work has been reported in [Kurshan and Lamport 1993; Joyce and Seger 1993; Rajan et al. 1995; Schneider and Kropf 1996].

Note that it is a non-trivial exercise to add a model-checking decision procedure to an existing general-purpose theorem prover, because this generally requires formalizing the semanitcs of the temporal logic within the logic of the theorem-prover (cf. [Joyce and Seger 1993; Rajan et al. 1995]).

3.4.1 *Compositional Theory for Trajectory Assertions.* Hazelhurst and Seger [1994, 1995] describe a proof system for trajectory assertions consisting of seven inference rules which in general have the form

$$\text{If } \models_{\mathcal{M}} A_i \Rightarrow C_i \text{ for } i = 1, \ldots, n \text{ and } Q_1, \ldots, Q_m, \text{ then } \models_{\mathcal{M}} A \Rightarrow C,$$

i.e. $\models_{\mathcal{M}} A \Rightarrow C$ can be derived from a set of trajectory assertions which are known to hold, given that the additional properties $Q_1, \ldots, Q_m$ are satisfied. Based on assertions verified using STE, more complex properties can be derived using these inference rules. It has been shown by Zhu and Seger [1994] that the addition of one more inference rule renders the system of rules complete in the sense that it is powerful enough to derive all logical consequences of a given set of trajectory assertions.

Based on the above proof system, an LCF-style proof tool called VossProver has been developed on top of the Voss verification tool. Trajectory theorems $\models_{\mathcal{M}} A \Rightarrow C$ which have already been proven are represented as objects of an abstract data type in Voss' meta-language `fl`. For each inference rule, there is a function (proof rule) which takes the theorems corresponding to the rule's premises as arguments, checks whether the rule is applicable, and if so returns a new theorem corresponding to the rule's conclusions. In addition, there is a basic rule which returns a theorem for a given trajectory assertion after verifying this assertion using STE.

On top of the above inference rules, a set of heuristics has been implemented which automatically determine for example an appropriate time shift which aligns two given theorems such that they can be combined using the transitivity rule. These heuristic rules are sound by construction because they are implemented in terms of the basic inference rules; however they may fail to find a proof for a given theorem.

3.4.2 *STeP.* STeP [Manna et al. 1994; Bjørner et al. 1996] is an integrated system for reasoning about reactive systems. The underlying computational model are transition systems with fairness constraints; programs are specified in a programming language called SPL, which provides, in addition to sequential constructs such as loops, constructs for concurrency and nondeterminism. Specifications are given in first-order LTL (i.e. quantification over state variables is allowed).

STeP integrates model-checking and theorem-proving methods for proving that a temporal logic formula φ is valid for a program $\mathcal{P}$. The model checker is based on the construction of the product automaton for $\mathcal{P}$ and $\neg\varphi$ and checking the emptiness of its language (see section 3.2.3). The model checker may be applied to systems with infinite state spaces, in which case termination is not guaranteed.

The theorem proving support in STeP has several aspects. There are proof rules which allow the validity of a temporal formula to be decomposed into so-called *verification conditions* involving only first-order formulae not containing temporal operators. A verification condition has the form $\{\varphi\}\tau\{\psi\}$ and asserts that taking a transition τ from a state satisfying φ must lead to a state ψ. The verification conditions for a program can be visualized using *verification diagrams*, graphs whose nodes and edges are labeled with formulae and transitions, respectively.

An automatic prover for first order logic, which is integrated with decision procedures for Presburger arithmetic, is available to prove the resulting obligations. If the automatic prover is unable to discharge the obligations, an interactive sequent-style prover may be used to simplify proof obligations.

STeP also provides methods for automatically discovering invariants, which are needed e.g. for proofs of safety properties. The system attempts to generate invariants both in a bottom-up fashion based on static analysis of the program, as well as top-down by strengthening given properties.

4. CASE STUDIES

In the previous sections, we have introduced a variety of specification styles and verification methodologies. We now present a selection of case studies where these methods have been applied to actual designs. We have attempted to choose applications to "real-life", mostly industrial designs in order to provide a sense of the state-of-the-art in verification and of how designs can be specified and verified in practice.

The application areas include microprocessors, floating-point hardware, protocols, memory arrays, and communications hardware.

4.1 Microprocessors

This section presents a selection of recent applications of formal methods to microprocessors. Much of the work has used an approach where the specification is given in terms of a state transition function on the programmer-visible state of the processor. This function captures the effect of executing exactly one instruction. Then, theorem-proving techniques are used to show that an implementation, described at a functional unit, RTL, or even gate-level, is a refinement of the specification. Earlier work in this category includes the Tamarack [Joyce 1988] and Viper [Cohn 1988; Cohn 1989a] verification efforts. Section 4.1.5 describes an approach where the specification consists of an RTL description, and state-machine equivalence checking is used to verify an implementation that is partly synthesized and partly hand-crafted.

4.1.1 *The Generic Interpreter Theory and AVM-1.* Windley [1995a] has developed a framework for the verification of refinement between deterministic finite state machines whose transitions can be grouped into classes, such as microprocessors where the classes correspond to instructions. His *Generic Interpreter Theory* (GIT) provides a structured model for the specification of such state machines at different levels of abstraction. The GIT model is based on the common structure of earlier microprocessor verifications, such as Tamarack [Joyce 1988], VIPER [Cohn 1988] and FM8501 [Hunt 1994].

The AVM-1 is a general-purpose 32-bit microprocessor which was devised for use as an example to demonstrate verification in the the GIT framework [Windley 1995a]. AVM-1 was designed to provide a reasonable set of features found in commercial processors and yet be verifiable at the same time. It features a RISC architecture, a large register file, supervisory mode, and supports external and software interrupts.

The generic interpreter framework permits hierarchical specifications where a design is described at a series of more and more detailed levels of abstraction: for instance, the AVM-1 was specified behaviourally at the top-most level representing the programmer's view of the processor, the microcode level, the multi-phase clock level, and with a structural representation at the lowest level [Windley 1995a]. The verification of such a hierarchy proceeds by showing that each of the levels (except for the top-most) implements the next higher level.

In the GIT framework, a deterministic machine is modeled as an *interpreter*. An interpreter $\mathcal{I}[s, e]$ is defined as a predicate over a state stream $s : \mathbf{N} \to \mathcal{S}_S$ and an environment stream $e : \mathbf{N} \to \mathcal{S}_E$ as in section 2.2.2:

$$\mathcal{I}[s, e] \triangleq \forall_{t \in \mathbf{N}}\ s(t+1) = \mathcal{N}(s(t), e(t)), \tag{12}$$

where $\mathcal{N}$ denotes the system's state transition function. One of the most important aspects of the GIT is the representation of this transition function. For a behavioural description of a microprocessor, it is natural to decompose $\mathcal{N}$ into a set $\mathcal{J}$ of transition functions, each of which corresponds to a single machine instruction. In the GIT, $\mathcal{N}$ is expressed in terms of a function which selects the next instruction to be executed in a given state, and the instruction-specific transition functions themselves.

In the GIT, the general condition for refinement (i.e. equation (6) on page 18) is specialized using separate functions for temporal and data abstractions:

$$\exists_{A_S, A_E, T} \quad \forall_{s_{imp}, e_{imp}} \quad \begin{array}{l} \mathcal{I}_{imp}[s_{imp}, e_{imp}] \\ \Rightarrow \mathcal{I}_{spec}[A_S \circ s_{imp} \circ T, A_E \circ e_{imp} \circ T]. \end{array} \tag{13}$$

Here, A_S and A_E map implementation states into specification states, and T maps specification time into implementation time. Note that in fact $\mathcal{I}_{imp}$ need not be an interpreter of the form (12). In particular, if the implementation is described at a structural level, the predicate $\mathcal{I}_{imp}$ can be expressed as the composition of predicates for the registers, gates and functional units which make up the circuit, with internal wires hidden by existential quantification (see section 2.2.2).

The special structure of the state transition function and the abstraction functions allows the correctness statement to be reduced to two minor lemmata and an *instruction correctness lemma* which essentially states that each instruction completes in a specific number of implementation time units and has the effect stipulated by its instruction-specific transition function. This facilitates case analysis over the set of instructions. The definitions of the transition function and the abstraction functions as well as the above lemmata are collected into an abstract theory [Windley 1992]. Instantiating the abstract theory automatically generates proof obligations for the lemmata; once these obligations are proved, instantiations of the theory's theorems, in particular the main correctness result (13), are established automatically.

Windley [1995a] argues that one of the most important benefits of this model is that it provides a template for a specification by making explicit which definitions (implementation, specification instruction set and abstraction functions) have to be made; and that the proof is structured to a large extent by automatically deriving which lemmata (i.e. the theory obligations) have to be proven. In earlier proof efforts, these definitions and lemmata were arrived at in an *ad hoc* manner. Specifying the processor hierarchically also contributes significantly to the management of proof complexity. In addition, it allows theorems about behavioural aspects of a processor, such as the integrity of the supervisory mode, to be proven at an appropriate level of abstraction [Windley 1991].

4.1.2 *Pipelined Microprocessors.* One of the major difficulties in the verification of pipelined or super-scalar processors is that instruction boundaries at the specification and the implementation level do not coincide in general. Instead, several uncompleted instructions may be executed in parallel in each cycle. Thus, the abstraction function must combine temporal and data aspects [Windley 1995b]. This poses two challenges: It is more difficult to find the appropriate abstraction function, and the proof structure becomes more irregular [Windley and Coe 1994].

Burch and Dill [1994] describe an approach to obtaining an abstraction function which is applicable if the implementation of the processor can be stalled. *Stalling* the processor refers to operating it in a mode where no new instructions are inserted into the pipeline, but the instructions which are already in the pipeline continue executing as usual. This can be achieved e.g. by inserting no-op instructions. The pipeline is *flushed* by stalling it sufficiently many cycles to allow all instructions to complete. The abstraction function is then composed of a function which symbolically flushes the pipeline, and a pure data-abstraction function.

The correctness condition they use states that the application of the specification and the implementation state transition function on an arbitrary state commutes with the above abstraction function, that is, in the notation from the previous section,

$$\forall_{s \in \mathcal{S}_{imp}} \exists_m \;\; Abs(\mathcal{N}_{imp}(s)) = \mathcal{N}^m_{spec}(Abs(s)). \tag{14}$$

Here, Abs is an abstraction function which first flushes the pipeline and then applies the data abstraction mapping from implementation states into specification states. The formulation in (14) includes a recent extension to the model [Burch 1996] which accounts for super-scalar execution, where more than one instruction can be issued per implementation cycle. This is reflected in the variable m, which corresponds to the number of instructions issued into the pipeline in a given implementation cycle. The correctness statement used in [Burch and Dill 1994] is obtained by fixing $m = 1$.

Burch [1996] gives a decomposition theorem which breaks the correctness condition for the processor (14) into three simpler properties. Informally, the first property states that the implementation executes instructions correctly from a flushed state; the second confirms that stalling does not affect the specification state; and the third checks that instructions are fetched correctly.

Burch and Dill [1994] and [Burch 1996] have applied this methodology to pipelined and super-scalar implementations, respectively, of a subset of the DLX architecture. The DLX [Hennessy and Patterson 1990] is a generic RISC architecture representing common features of (earlier) commercial RISC architectures such as the MIPS R3000, SPARC1, IBM 801, and Intel i860.

The verification was carried out using a tool based on symbolic simulation and a validity checker. Representations of the state transition functions are compiled from descriptions of the implementation and specification in a simple HDL. The tool is targeted at the verification of pipeline *control*; uninterpreted function symbols are used to represent e.g. ALU operations. The proof obligations are expanded and then discharged using a specially tailored decision procedure for quantifier-free, first-order logic with equality and uninterpreted function symbols [Jones et al. 1995; Barrett et al. 1996]. The tool does not allow for deductive reasoning to discharge obligations; However, the user can suggest case splits to the validity checker, which was indeed necessary to complete the verification of the super-scalar DLX implementation.

Other work on pipelined microprocessors includes [Srivas and Bickford 1990; Tahar and Kumar 1993; Saxe et al. 1993; Windley 1995b].

4.1.3 *FM9001.* The FM9001 is a general-purpose 32-bit microprocessor which was designed with verification in mind by Bishop Brock and Warren Hunt at Computational Logic, Inc. [Brock et al. 1994]. The FM9001 instruction set is based on the instruction set of the earlier FM8502 design [Hunt 1989] and features the usual integer arithmetic and Boolean operations, conditional stores and five addressing modes. Instructions are of fixed sized and use a two address format.

The high-level specification for the FM9001 was formalized in the Boyer-Moore logic. The central part of the specification is a function `FM9001-step` which, given the current programmer-visible state of the processor (i.e. registers, flags and memory), returns the updated state resulting from the execution of one instruction. This function is written in terms of auxiliary functions specifying how the current instruction is fetched from memory, decoded, and executed. The top level-specification is a function which runs `FM9001-step` for n steps.

Unlike earlier verification efforts, Brock et al. [1994] formalize the implementation level description not directly as functions in the logic. Instead, they define a simple netlist HDL whose syntax and semantics were formalized in the Boyer-Moore logic [Hunt and Brock 1992]. The formalization of the HDL consists of a predicate which recognizes well-formed netlists, i.e. checks that requirements with respect to fanout violations, clock distribution, absence of loops etc. are satisfied, and a symbolic simulator which defines the operational semantics of the HDL. The simulator, called `DUAL-EVAL`, is based on a four valued logic which includes the two additional logical values $floating$ and $undefined$. The `DUAL-EVAL` netlist description of the processor has been translated automatically (but informally) into a commercial netlist format for fabrication.

The implementation and specification are related by a proof carried out in the Nqthm prover. The main result states that for an arbitrary user-visible state s and a positive integer n, there exists an integer c such that executing the specification `FM9001-step` n steps starting in state s has the same effect on the user-visible state as simulating the behaviour of the FM9001 netlist under `DUAL-EVAL` semantics for c clock cycles, subject to an appropriate abstraction mapping between state at the specification and the netlist level. In addition, it the proof shows that the FM9001 implementation can be reset from an arbitrary, undefined state. This proof is carried out at the netlist level and draws on monotonicity properties proved about the `DUAL-EVAL` simulator.

The processor has been fabricated from its netlist description and has been subjected to extensive testing; no functional design errors have been reported [Albin et al. 1995].

4.1.4 *AAMP5.* The AAMP5 microprocessor is a commercial microprocessor distributed by Collins Commercial Avionics and is part of a family of avionics processors intended for use in critical applications such as avionics displays. The architecture has been designed as a target architecture for high-level block-structured languages and provides hardware support for run-time system primitives such as procedure-state saving and parameter passing as well as scheduling primitives such as task-state saving, context switching and interrupt handling. The instruction set of the AAMP architecture is large and CISC-like with 209 variable-length instructions.

The implementation-level architecture of the AAMP5 consists of four independent units: a bus interface, an instruction cache, a look-ahead fetch unit, and a microcoded, three-stage-pipelined data-processing unit. The implementation consists of ca. 500,000 transistors.

A partial verification of an RTL description of the AAMP5 against a high-level specification has been carried out in PVS in a cooperation between Collins, NASA

Langley and SRI International [Miller and Srivas 1995; ?]. The formal specification of the AAMP5 is given as usual in terms of next-state functions which describe the effect of each instruction on a state-tuple representing the programmer-visible state of the processor and the memory. 108 out of the 209 instructions were specified; the specification consists of ca. 2,500 lines of PVS. A significant part of the specification effort was invested in validating the specification with domain experts to ensure that the formal specification indeed captures the intended behaviour as described in the programmer's manual.

At the implementation level, the data-processing unit (DPU) is described at the register-transfer-level, while the other three units are specified only in terms of their external behaviour. The processing unit itself is specified as the composition of register-transfer-level components (latches, ROMs, multiplexers, shifters, register files, etc.), and modules such as the ALU described at a more abstract level. The individual components are specified in terms of their outputs as functions of their input signals, in contrast to the predicative style described in section 2.2.2. This functional specification style was chosen because it is more easily treated with the proof strategies used. There is no formal link to the gate-level implementation of the processor; this link has been validated using traditional, informal design reviews.

The correctness condition used in this verification states that each instruction satisfies the following assertion: If the DPU is in a state where it just begins executing the first microinstruction of an instruction, then execution will eventually complete, the sequence of micro-instructions will have the same effect under the abstraction mapping as the macro- (specification-level) instruction, and the DPU will end up in a state where it just begins executing the first micro-instruction of the next macro-instruction. In addition, it has to be shown that the processor can be reset into a valid state. The proofs are complicated by two features of the architecture. First, the pipelined implementation leads to a complex, non-orthogonal abstraction mapping Second the decoupled instruction pre-fetch units can fetch instructions an arbitrary (though finite) number of cycles before the instruction actually enters the execution unit.

4.1.5 *PowerPC Transistor Level Verification.* Appenzeller and Kuehlmann [1995] report on the application of formal methods in the PowerPC microprocessor design process. In this project, the top-level formal specification consists of the RTL description of the design, written in VHDL. The transistor-level implementation was formally verified to be functionally equivalent to this specification using the Verity tool (see section 3.2.2.1, [Kuehlmann et al. 1995]).

The RTL specification was written with simulation performance in mind and validated to conform with the informal PowerPC architecture specification through extensive simulation. There is no formalization of a specification at a higher level. The RTL specification is structured into hierarchical layers; the structural decomposition was planned such that the individual modules were manageable in complexity both for the designers and the verification tool. There are three main layers: The top-most, chip layer specifies how the entire processor is composed of its functional units; the second layer describes these units in terms of components, which are in turn described at the lowest layer. The average size of such a component corresponds to ca. 7000 lines of VHDL or 60,000 transistors. Each of the main layers were subdivided into further sub-layers if necessary.

The transistor-level implementation was developed using the the structural decomposition given by the RTL specification. The design used a combination of synthesized and custom designed components. Static and pre-charged circuits are both used in the design. The Verity tool was used to show functional equivalence for all synthesized components and 81% of the custom designed components The unverified components either included large storage arrays or they were components that were designed before Verity was introduced, and the project schedule did not accomodate the restructuring that would be needed for verification. The authors noted that much of the verification was possible because Verify was part of the design flow, and circuits were structured with verification in mind. Ninety-five percent of the circuits were verified in less than 800 CPU seconds and 30MB of memory each. With this level of performance, it was possible to integrate Verity into an interactive design environment.

4.2 Floating-Point Units

In the wake of the highly publicized flaw in the Intel Pentium divider, many researchers have looked to using formal methods to verify hardware for floating point operations. The verification of floating-point *algorithms* has also received much attention recently, but is beyond the scope of this paper. Recent efforts in this area include work on SRT division (the algorithm used in the Pentium floating-point unit, see [Cavanagh 1984]) [Miner and Leathrum, Jr. 1996; Ruess et al. 1996], square root [Leeser and O'Leary 1995], and natural logarithm algorithms [Harrison 1995] and the core of the AMD5_K86 floating-point division algorithm [Brock et al. 1996; Moore et al. 1996].

4.2.1 *Intel Extended-Precision FPU.* Chen et al. [1996] apply word-level model checking [Clarke et al. 1996] to the verification of all operations of the floating-point unit of a recent Intel microprocessor[3]. They also demonstrate their approach using the Weitek WTL3170/3171 Sparc floating-point co-processor[4] as an example.

Word-level model checking is not capable of directly verifying that the circuit implements a high-level specification of (e.g. IEEE conformant) floating-point operations. Instead, the overall correctness condition is decomposed into a set of properties about sub-units of the FPU. These properties may be further decomposed into several assertions, which are then expressed as temporal formulae in word-level CTL. For iterative algorithms, such as division and square-root, these individual properties state that the circuit maintains the appropriate loop invariants. In the case of the Intel FPU, over 120 such properties were verified and are claimed to cover all arithmetic operations of the unit. There is no machine-checked proof that the collection of these properties implies that the FPU conforms to a formalized specification.

In order to make the verification tractable, a technique called *property-specific model extraction* was applied to the circuit description before model checking of the individual verification conditions. This step exploits that a given operation

[3]The paper does not specify exactly which model

[4]The architecture of this unit has been documented in the literature [Birman et al. 1990].

(such as multiplication) usually exercises only part of the FPU; thus, the irrelevant parts of the unit can be abstracted away when checking a property pertaining to this operation. The authors do not provide a detailed description of this step, nor do they describe whether the validity of this abstraction is shown formally or informally.

4.2.2 *The ADK IEEE Multiplier.* Aagaard and Seger [1995] report on the design and verification of a IEEE compliant floating-point multiplier called ADK. The design satisfies all requirements stated in the IEEE standard, except that some non-standard rounding modes are not supported and denormalized operands are treated by either considering them as equal to zero (in violation of the standard) or by raising an exception which allows treatment in software.

The ADK is implemented as an array-multiplier with radix-8 modified Booth-recoding, and has a four-stage pipelined architecture. In the first pipeline stage, the operands are prepared, which involves computing the Booth recoding (see [Cavanagh 1984]) of the first operand's significand; in addition, the exponents are added and special cases (such as NaNs) are detected. The second stage consists of the multiplier array of carry-save-adders which performs the actual multiplication of the significands. The third stage adds the resulting carry and sum vectors and also performs normalization of the result. The fourth and final stage implements the rounding and re-normalization of the result and also detects overflow conditions and raises the appropriate exception signals. The design was implemented in structural VHDL and synthesized into a unit-delay gate-level model with ca. 33,000 gates.

The top-level specification of the multiplier was given in the relational style introduced in section 2.1.6 and formalizes the textual description of the IEEE multiplication standard. The relation between input operands and outputs is written in terms of arithmetic operations on integers, i.e. the significand and exponent vectors are interpreted as integer values at the specification level.

The verification proceeded by first showing that the gate-level implementations of the design's functional units (such as the Booth-recoder or the multiplication array) satisfy corresponding TF specifications. This step was done automatically using trajectory evaluation. Then the results for the individual units were composed into specifications of larger circuit components and eventually shown to imply the top-level specification. This step used the proof rules of the compositional theory for TF provided by the VossProver system [Hazelhurst and Seger 1995] (see section 3.4.1).

Thus, it was shown that a gate-level model of the ADK implementation satisfies a high-level specification of the IEEE multiplication standard. The authors estimate the combined effort for design, implementation and verification of the multiplier at around 70 work-days.

4.3 Asynchronous and Distributed Systems

4.3.1 *The Summit Bus Converter.* Harkness and Wolf [1994] apply symbolic model checking to the Summit bus converter, an industrial component of a multiprocessor system. The bus converter provides a communication link between a high-speed processor bus and a low-speed I/O bus. Both the processor bus and the I/O bus components of the converter are designed as collections of interacting finite state machines. The authors informally abstracted away aspects of the design which were considered unimportant to the verification conditions they had in mind. For instance, error handling, data transfer, pipelined link transfer and other performance-enhancing features were not modeled, also addresses were reduced to one bit. Model checking was then applied to the resulting abstract model of the design, which was formalized in SMV, to demonstrate the presence of deadlock on the system's link queues in earlier versions of the design, as well as the absence of deadlock in a revised version. This was accomplished by showing that the system satisfies the CTL formula

$$\mathbf{AG}\,(LQ.Head.Occ \Rightarrow \mathbf{AF}\,(\neg LQ.Head.Occ)),$$

stating that whenever the head buffer of the link queue is occupied with a request, this will eventually be processed.

The potential for deadlock in the incorrect design had already been known, and the abstract model was created with the verification of deadlock-freedom in mind. When validating their model against other putative temporal properties of the system, the authors also discovered a data corruption error in the design.

4.3.2 *Cache Coherence Protocols.* A distributed shared memory system (e.g. [Lenoski et al. 1992]) consists of a collection of clusters, each of which has one or more processor, local memory, caches, memory-mapped I/O devices, a directory, and a DSM controller. The clusters are connected through a communication fabric. Each processor sees the same, shared, linear virtual address space. The DSM controller ensures that each processor sees the same, shared, address space. The cache coherence protocol is supposed to ensure that this memory is *c*oherent: although memory reads and writes may occur concurrently on different nodes, the values obtained from reads appear as if the memory actions occured in a single, global, sequential order. Subtle bugs can appear in an incorrect protocol when messages exchanged between nodes in an order unforeseen by the designer. As there may be an extremely large number of possible event orderings, coherence protocols can be very hard to validate using informal methods such as simulation.

Model checking techniques have been used to verify several such protocols. Cache coherence protocols are typically specified as a collection of communicating state machines, making them natural candidates for model checking. However, a real implementations have extremely large amounts of state including several status bits for each cache block on each node. Model checking is generally applied to an informally constructed abstraction of a small system. In this section, we present two recent applications of this approach, the coherence protocols of a recent Silicon Graphics multiprocessor architecture, and of the Sun S3.mp architecture. Numerous other coherence protocols have been verified including those for the Gigamax [McMillan and Schwalbe 1991], Futurebus+ [Clarke et al. 1993] and SCI [Stern and Dill 1995].

Eiríksson and McMillan [1995] describe the verification of the cache coherence protocol for a Silicon Graphics multiprocessor using the SMV model checker. The state transition tables for the protocal are machine tranlated into SMV representations and composed with a model of the connection network. Then, high-level correctness properties of the protocol, expressed in CTL, are verified. To make the verification tractable, the high-level specification was kept as abstract as possible and dependent variables were eliminated (see section 3.1.2.3). The properties verified included absence of deadlock, requests always receive the correct response, absence of unsolicited responses, and safety properties expressing cache coherence. In one case, a bug which lead to loss of coherence was exposed; the problem occurred after a sequence of 19 events in a particular ordering and would most likely not have been found in simulation [Eiríksson and McMillan 1995].

The verification effort was an integral part of the design process [Eiríksson 1996; Eiríksson and McMillan 1995]. A single abstract specification of the cache coherence protocol was used for formal verification, performance evaluation, documentation, and design. For a number of sub-components, it has also been verified that a Verilog RTL implementation of the component is a refinement or the corresponding high-level FSM. This was accomplished using a translator which extracts an SMV model from the Verilog code and using modelchecking to verify the refinement property. Regression verification was used to automatically repeat the verification as the design evolved. A complete regression run which checks all of the roughly 300 CTL assertions for less than four clusters takes less than two days on a cluster of 200 workstations with up to 2GB memory each.

The Sun S3.mp shared memory multiprocessor [Nowatzyk et al. 1995] also uses a directory-based cache-coherency scheme. The nodes in this system are standard workstations with special interconnect controllers attached to their system buses. Each workstation may itself have more than one processor, consistency within a single workstation is achieved through a snooping protocol on the system bus. In the Sun protocol, each block has a home node, and the DSM controller on that node maintains a list of other nodes that contain copies of the block. The protocol allows one processor to have an exclusive copy of a block or many processors to have read-only copies. The home node is responsible for serializing requests for its memory blocks.

Pong et al. [1995] model the protocol in the Murφ language and system [Dill et al. 1992] (see section 3.1.5). Here, we will focus on the how the multiprocessor system is modeled in order to make the verification tractable. Only one memory block is modeled; it is assumed that other blocks don't interfere. The model contains this single block's home node with its directory entry as well as k other nodes with one cache block each. The nodes are modeled with one processor only based on the assumption that the local snooping protocol works correctly. The network is modeled by non-FIFO buffers for incoming and outgoing messages at each node. The data values held in the cache are represented by five symbolic values reflecting their relation with the other cached copies. For example the symbolic value `GlobalFresh_Hold` denotes a value written to the cache by the local node but which has not yet been propagated to all other nodes; similarly, `Obsolete` denotes a value which is out of date. A required safety property for a protocol which maintains data consistency is that processors can never read an `Obsolete` value. The symmetry of the model is exploited to group equivalent symmetric states [Ip and Dill 1996a]. Using this model, several protocol errors were detected and corrected. Data consistency was then verified for a model with two remote nodes in addition to the home node.

To verify larger configurations, the authors applied a technique called *symbolic state model* (SSM) [Pong and Dubois 1993]. The key idea here is that it is irrelevant precisely how many nodes have (for example) a shared copy of a block; rather, it is sufficient to distinguish states in which either zero, one, or more nodes have caches and network buffers in a particular state. In the SSM for the S3.mp cache coherence, a system state consists of a detailed account of the nodes on the distributed linked list, as well as repetition counts in the above sense for the remaining nodes. Using SSM, the correctness condition is established for a configuration with arbitrarily many nodes, but where at most four nodes (including the home node) can simultaneously share a copy of the block. More errors were found that only surfaced in models with at least three remote nodes.

4.4 Memory Subsystems

Complex memory arrays such as multi-ported register-files and caches are particularly difficult applications for model-checking techniques because of their very large state spaces. Furthermore, their implementations typically have non-trivial timing characteristics which may preclude the extraction of accurate gate-level model from the transistor netlist.

Pandey et al. [1996] apply symbolic trajectory evaluation to two arrays from the PowerPC architecture. The first array is a multi-ported register file with 36 32-bit registers and two write and five read ports. The second unit is a data cache tag (DTAG) with 128 4-way associative sets. Each way in a set has a 20-bit tag plus valid and dirty bits; in addition each set has 6 LRU bits. When a 7-bit index to select one of the sets as well as a tag are presented to the unit, it determines if the tag matches one of four ways. If so, this is indicated and the LRU history of the set is updated. Otherwise, the tag of the least recently used way is output. The implementation of the DTAG has over 12,000 latches.

The specifications for the units were structured into high-level assertions about abstract state transitions (i.e. memory read or write) and an implementation mapping which captures the detailed timing information (see [Beatty and Bryant 1994]). The high-level assertions have the form $A \overset{\text{leadsto}}{\Longrightarrow} C$, where the antecedent A is an assertion about the inputs and internal state before an abstraction transition, and C is a corresponding assertion bout the internal state afterwards. The implementation mapping relates each clause in A or C to a trajectory formula which specifies the detailed timing of the node(s) corresponding to the high-level clause. Assertions on the pre and post values of the registers are expressed in the form $R[i] = u$, where i is a symbolic index [Beatty 1993], and the corresponding circuit nodes are initialized with symbolic expressions rather than constants. This technique is vital to the tractability of the verification in that it reduces the number of symbolic variables to a number approximately logarithmic in the number of registers.

The implementations were given both as flat transistor netlists and at the RTL level in an internal HDL. To identify the state-holding nodes in the transistor netlist, an automated technique was used which exercises a write operation on the design with symbolic input values, and then matches the resulting symbolic expressions on the nodes against symbolic indexing functions for the state holding elements. The switch-level models for the symbolic simulation underlying STE were derived from

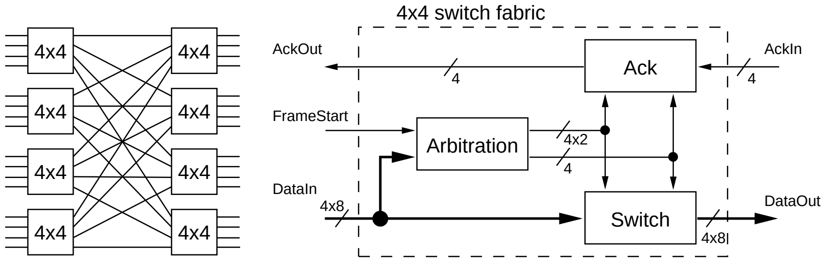

Fig. 7. 4x4 switch fabric and delta network

the netlists. In the case of the DTAG unit, some amount of manual modeling was necessary, which involved annotating the circuit with transistor strengths. Care was taken not to assign directions to transistors.

Both the switch-level and the RTL-level models were checked against the trajectory assertions comprising the circuit specification. For both units, discrepancies between the behaviour of both models appeared, although in the case of the register file, this did not correspond to a bug in the circuit because the corresponding behaviour would not be exercised in the intended operating environment. For the DTAG unit two actual bugs were found, the first of which was due to a transistor 'sneak path' and which therefore was not visible at the RTL-level. Checking the most complicated assertion for the DTAG unit required ca. 10 minutes of processing time and 150MB of memory.

4.5 ATM Switch Fabrics

The Asynchronous Transfer Mode (ATM) is a technology for high-speed local- and wide-area networks [Vetter 1995]. Its main characteristic is that data is transported in small, fixed-size packets called *cells*. The network consists of multi-ported switches interconnected by point-to-point high-speed links. Fairisle [Leslie and McAuley 1991] is an ATM architecture for local area networks (LANs) supporting link-speeds of 100Mbps.

The switch fabric forms the core of the switch; it does the actual routing of cells from one of the sixteen input lines to the desired output lines. The fabric is implemented as a delta network of eight 4x4 switch fabrics as shown in figure 7. The network is self-routing: Port controllers at the input lines attach two routing tags to incoming cells. The 4x4 fabrics read the tags to determine which of their outputs a cell is to be forwarded to, and then strip off the tag. If two cells have the same destination, one of them must be dropped; successful delivery is communicated back to the input via the acknowledgment lines. The decision which of two or more competing cells will be forwarded is made by the arbitration unit, according to a round-robin scheme which also takes priority information from in the routing tags into consideration. The frame start signal indicates that new cells (headed by a routing tag) will appear at the inputs; cells appear in synch on the input lines. In the following, we will discuss two applications of formal methods to the switch fabric.

Tahar et al. [1996] verified safety properties of the 4x4 fabric. They created two models of the design: a gate-level model derived from an HDL description, and an abstract model where data inputs and outputs are abstracted to an abstract sort `wordn`, representing n-bit words, and data-path operations are represented by uninterpreted function symbols. Using a tool based on Multiway Decision Graphs (MDGs, a generalization of BDDs which supports abstract sorts and uninterpreted functions [Anon et al. 1996]), they verified the sequential equivalence of the two models (for fixed $n = 8$) by constructing a product machine which included appropriate encoding/decoding units converting between the abstract sort and 8-bit vectors.

The authors then verified a number of properties of the abstract model. Correct operation is required only in an environment which meets certain constraints; for example, the routing tags must appear on the input lines a certain number of clock cycles after the frame start signal was asserted. The environment was modeled by an environment machine which generates allowed sequences of input signals. The properties verified were phrased as safety properties of the composition of the environment machine and the abstract model. For example, it was verified that proper default values appear at the data and acknowledgment outputs after a reset. Properties regarding the correct forwarding of data require that, during an appropriate time span relative to the beginning of a frame, the appropriate data output equals the corresponding data input from four cycles earlier, subject to arbitration decisions. In order to phrase this condition as a safety property, it was necessary to route copies of the data signals generated by the environment through a four-cycle delay buffer.

Curzon and Leslie [1996] report on the verification of the complete 16x16 fabric in HOL. This verification effort is based on earlier work by the first author on the 4x4 fabric [Curzon 1994]. A hierarchical description of the implementation is given at the gate-level in the specification style introduced in section 2.2.2. A behavioural specification of the switching element is given in terms of a predicate relating the input and output signals over time. The overall correctness statement then takes the form

$$Assumptions \Rightarrow (Imp \Rightarrow Spec),$$

where $Assumptions$ is a predicate capturing constraints on the environment behaviour.

The 4x4 elements of the 16x16 fabric are in fact not identical; there are small differences in timing between the two stages, and a slight difference in the routing behaviour between the top and bottom elements of the first stage. The proofs for these elements were adapted from the original proof of the 4x4 element. The proof for the main correctness theorem for the 16x16 fabric could not be completed because the theorems proven for the first stage elements could not guarantee an environment assumption concerning the timing of cell arrivals which was needed by the second stage. This problem did not arise in [Tahar et al. 1996] because they only considered a single 4x4 element.

Based on this proof effort, several modifications of the original 16x16 design were proposed, resulting in simplifications of the interfaces between functional units and the removal of environment assumptions. In particular, the environment assumption which prevented the completion of the original proof could be removed by adding a small amount of logic to the arbitration unit. All but one of the suggested modifications were considered acceptable by the designer of the original 16x16 fabric. The authors conclude that keeping provability in mind during the design phase (as opposed to verification after design completion) can result in shorter proofs and even improved, simpler designs.

Tahar and Curzon [1996] provide a more detailed comparison of the two verification efforts described above.

5. CONCLUSION

Traditional, simulation-based validation can only check a very small fraction of the behaviours of any non-trivial, digital design. Research in formal methods attempts to verify that designs work for all allowed inputs and all reachable states by using rigorous methods from formal mathematics. This requires a formal, mathematical specification of the design requirements, an accurate model of the implementation, and practical techniques to show that the implementation indeed satisfies the requirements. When a claim is made that a design has been formally verified, it is important to understand what requirements were specified and at what level of detail the design was modeled. This paper has described formal methods for specification and verification, and summarized case studies where these methods have been applied to realistic designs.

Broadly summarizing, specification techniques tend to describe either temporal properties that a design should have or a high-level model of behaviour that the design should implement. For example, a temporal formula could state that all messages that enter a network must eventually be delivered, or that executing an instruction leaves the right symbolic values in the register file. Using high-level models, one could specify the network as a collection of abstract queues with a process that transfers messages between queues according to their routing tags, and the processor could be specified with a high-level model of its instruction set.

Design models can range from very abstract models where the design is divided into a few block down to very detailed descriptions that model each transistor. Verification at a high level of abstraction does not prove that the lower level details of the implementation are correct. On the other hand, formal methods can be very effective for finding errors at high levels of abstraction before a large design effort is invested in implementing an flawed system architecture. The choice of models must reflect both what the designer believes is most likely to reveal errors and what is possible to verify with existing methods.

Verification techniques generally can be divided into two catagories: reachability analysis (e.g. explicit or symbolic state enumeration) and deductive methods. Model checkers, language containment checkers, and state machine equivalence checkers are examples of the first approach. These tools are often applied used to verify requirements specified as temporal properties, but, as noted in section 3.1.3, model checking can also be used to verify refinement of a high-level model. For these to work, the model must be small enough for the verification to complete with the CPU and memory resources available. This usually requires that a design is verified at a relatively abstract level, or that relatively small sub-systems of the design are verified separately. Although these methods offer a high degree of automation, successful verification often requires an expert's insight into the design itself as well as the implementation of the model-checker.

Many different theorem provers have been used for deductive verification. Most examples using theorem provers verify that some model of a design is a refinement of a higher level, although deductive proofs of temporal properties are also possible (e.g. [?; Bjørner et al. 1996]). These methods are guided by an expert's understanding of the design and are not limited to finite structures, which allows deductive verification to address designs that cannot be verified by more automatic techniques. However, using a theorem prover requires a high degree of expertise in mathematical logic, a familiarity with the theorem prover, and an intimate understanding of the design itself.

It is interesting to observe that in several of the larger verification efforts carried out to date, both model-checking and theorem-proving based, the developers of the tool used (or the underlying theory) were involved at least in the initial stages of the project (e.g. [Eiríksson and McMillan 1995; Miller and Srivas 1995; Pandey et al. 1996; Brock et al. 1996]). Although these efforts required the expertise of the tool developer, the successful results on real designs indicates that formal verification is likely to become more widely adopted in the near future.

References

AAGAARD, M. AND SEGER, C.-J. H. 1995. The formal verification of a pipelined double-precision IEEE floating-point multiplier. In *ACM/IEEE International Conference on Computer-Aided Design* (Nov. 1995), pp. 7–10.

AAGAARD, M. D., LEESER, M. E., AND WINDLEY, P. J. 1993. Toward a super duper hardware tactic. In J. J. JOYCE AND C.-J. H. SEGER Eds., *Higher Order Logic Theorem Proving and Its Applications, HUG '93*, Number 780 in Lecture Notes in Computer Science (Vancouver, Aug. 1993), pp. 400–412. Springer-Verlag.

ABADI, M. AND LAMPORT, L. 1991. The existence of refinement mappings. *Theoretical Computer Science 82*, 2 (May), 253–284.

ALBIN, K. L., BROCK, B. C., HUNT, W. A., AND SMITH, L. M. 1995. Testing the FM9001 microprocessor. Technical Report 90 (Jan.), Computational Logic, Inc., Austin, TX.

ANON, K. D., BOULERICE, N., CERNY, E., CORELLA, F., LANGEVIN, M., SONG, X., TAHAR, S., XU, Y., AND ZHOU, Z. 1996. MDG tools for the verification of RTL designs. In R. ALUR AND T. A. HENZINGER Eds., *8th International Conference on Computer-Aided Verification, CAV '96*, Number 1102 in Lecture Notes in Computer Science (July/Aug. 1996), pp. 433–436. Springer-Verlag.

APPENZELLER, D. P. AND KUEHLMANN, A. 1995. Formal verification of the PowerPC^{TM} microprocessor. In *1995 Intl. Conference on Computer Design: VLSI in Computers & Processors* (Oct. 1995), pp. 79–84. IEEE Computer Society Press.

BARRETT, C., DILL, D., AND LEVITT, J. 1996. Validity checking for combinations of theories with equality. In M. SRIVAS AND A. CAMILLERI Eds., *1st Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD '96)*, Number 1166 in Lecture Notes in Computer Science (Nov. 1996), pp. 187–201. Springer-Verlag.

BEATTY, D. L. 1993. A methodology for formal hardware verification, with application to microprocessors. Technical Report CMU-CS-93-190 (Aug.), School of Computer Science, Carnegie Mellon University. Ph.D. Thesis.

BEATTY, D. L. AND BRYANT, R. E. 1994. Formally verifying a microprocessor using a simulation methodology. In *Proc. 31st Design Automation Conference* (June 1994), pp. 596–602. ACM.

BEER, I., BEN-DAVID, S., EISNER, C., AND LANDVER, A. 1996. RuleBase: An industry-oriented formal verification tool. In *33rd Design Automation Conference, DAC '96* (June 1996), pp. 655–660.

BERNHOLTZ, O., VARDI, M. Y., AND WOLPER, P. 1994. An automata-theoretic approach to branching-time model checking. In D. L. DILL Ed., *6th International Conference on*

Computer-Aided Verification, CAV '94, Number 818 in Lecture Notes in Computer Science (June 1994), pp. 142–155. Springer-Verlag.

BIRMAN, M., SAUMELS, A., CHU, G., CHUK, T., HU, L., MCLEOD, J., AND BARNES, J. 1990. Developing the WTL3170/3171 Sparc floating-point coprocessors. *IEEE Micro 10*, 1 (Feb.), 55–64.

BJØRNER, N., BROWNE, A., CHANG, E., COLÓN, M., KAPUR, A., MANNA, Z., SIPMA, H., AND URIBE, T. 1996. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *8th International Conference on Computer Aided Verification*, LNCS (1996). Springer-Verlag. To appear.

BORMANN, J., LOHSE, J., PAYER, M., AND VENZL, G. 1995. Model checking in industrial hardware design. In *32nd Design Automation Conference, DAC '95* (1995).

BOYER, R. S. AND MOORE, J. S. 1979. *A Computational Logic.* Academic Press, New York.

BOYER, R. S. AND MOORE, J. S. 1988. *A Computational Logic Handbook.* Academic Press, New York.

BRADFIELD, J. C. 1992. *Verifying Temporal Properties of Systems.* Birkhäuser, Boston.

BRAYTON, R. K., HACHTEL, G. D., SANGIOVANNI-VINCENTELLI, A., SOMENZI, F., AZIZ, A., CHENG, S.-T., EDWARDS, S. A., KHATRI, S. P., KUKIMOTO, Y., PARDO, A., QADEER, S., RANJAN, R. K., SARWARY, S., SHIPLE, T. R., SWAMY, G., AND VILLA, T. 1996a. VIS: A system for verification and synthesis. In *8th International Conference on Computer-Aided Verification, CAV '96*, Number 1102 in Lecture Notes in Computer Science (July 1996), pp. 428–432. Springer-Verlag.

BRAYTON, R. K., HACHTEL, G. D., SANGIOVANNI-VINCENTELLI, A., SOMENZI, F., AZIZ, A., CHENG, S.-T., EDWARDS, S. A., KHATRI, S. P., KUKIMOTO, Y., PARDO, A., QADEER, S., RANJAN, R. K., SARWARY, S., SHIPLE, T. R., SWAMY, G., AND VILLA, T. 1996b. VIS. In *1st Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD '96)*, Number 1166 in Lecture Notes in Computer Science (Nov. 1996), pp. 248–256. Springer-Verlag.

BROCK, B. AND HUNT, W. A. 1990. Report on the formal specification and partial verification of the VIPER microprocessor. Technical Report 46 (Jan.), Computational Logic, Inc., Austin, TX.

BROCK, B., HUNT, W. A., AND KAUFMANN, M. 1994. The FM9001 microprocessor proof. Technical Report 86 (Dec.), Computational Logic, Inc., Austin, TX.

BROCK, B., KAUFMANN, M., AND MOORE, J. S. 1996. ACL2 theorems about commercial microprocessors. In M. SRIVAS AND A. CAMILLERI Eds., *1st Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD '96)*, Number 1166 in Lecture Notes in Computer Science (Nov. 1996), pp. 275–293. Springer-Verlag.

BRYANT, R., BEATTY, D., BRACE, K., CHO, K., AND SHEFFLER, T. 1987. COSMOS: a compiled simulator for MOS circuits. In *Proc. 24th ACM/IEEE Design Automation Conference* (1987), pp. 9–16.

BRYANT, R. E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers C-35*, 8 (Aug.), 677–691.

BRYANT, R. E. 1991a. A methodology for hardware verification based on logic simulation. *Journal of the ACM 38*, 2 (April), 299–328.

BRYANT, R. E. 1991b. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers C-40*, 2 (Feb.), 205–213.

BRYANT, R. E. 1992. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys 24*, 3 (Sept.), 293–318.

BRYANT, R. E. 1995. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *International Conference on Computer-Aided Design, ICCAD '95* (Nov. 1995).

BURCH, J., CLARKE, E., LONG, D., MCMILLAN, K., AND DILL, D. 1994. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13*, 4 (April), 401–424.

BURCH, J. R. 1996. Techniques for verifying superscalar microprocessors. In *33rd Design Automation Conference, DAC '96* (June 1996), pp. 552–557.

BURCH, J. R., CLARKE, E. M., MCMILLAN, K. L., DILL, D. L., AND HWANG, J. 1990. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th Annual Symposium on Logic in Computer Science* (1990). IEEE Computer Society Press.

BURCH, J. R. AND DILL, D. L. 1994. Automatic verification of pipelined microprocessor control. In D. L. DILL Ed., *6th International Conference on Computer-Aided Verification, CAV'94*, Number 818 in Lecture Notes in Computer Science (June 1994), pp. 68–80. Springer-Verlag.

CAVANAGH, J. J. 1984. *Digital computer arithmetic : design and implementation.* McGraw-Hill, New York.

CHANDY, K. M. AND MISRA, J. 1988. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, Mass.

CHEN, Y.-A., CLARKE, E., HO, P.-H., HOSKOTE, Y., KAM, T., KHAIRA, M., O'LEARY, J., AND ZHAO, X. 1996. Verification of all circuits in a floating-point unit using word-level model checking. In M. SRIVAS AND A. CAMILLERI Eds., *1st Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD '96),* Number 1166 in Lecture Notes in Computer Science (Nov. 1996), pp. 19–33. Springer-Verlag.

CHURCH, A. 1940. A formulation of the simple theory of types. *J. Symbolic Logic 5*, 56–115. **check this**.

CLARKE, E., EMERSON, E., AND SISTLA, A. 1983. Automatic verification of finite-state concurrent systems using temporal logic specifications: A practical approach. In *10th Ann. ACM Symposium on Principles of Programming Languages* (1983), pp. 117–126.

CLARKE, E., EMERSON, E., AND SISTLA, A. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems 8*, 2 (April), 224–263.

CLARKE, E., FUJITA, M., AND ZHAO, X. 1995. Hybrid decision diagrams – overcoming the limitations of MTBDDs and BMDs. In *International Conference on Computer-Aided Design, ICCAD '95* (Nov. 1995), pp. 159–163.

CLARKE, E., GRUMBERG, O., HIRAISHI, H., JHA, S., LONG, D., MCMILLAN, K., AND NESS, L. 1993. Verification of the Futurebus+ cache coherence protocol. In L. CLAESEN Ed., *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and their Applications* (April 1993). North Holland.

CLARKE, E., MCMILLAN, K., CAMPOS, S., AND HARTONAS-GARMHAUSEN, V. 1996. Symbolic model checking. In *8th International Conference on Computer-Aided Verification, CAV '96*, Number 1102 in Lecture Notes in Computer Science (July 1996), pp. 419–422. Springer-Verlag.

CLARKE, E. M., BROWNE, I. A., AND KURSHAN, R. P. 1990. A unified approach for showing language containment and equivalence between various types of ω-automata. In A. ARNOLD Ed., *Proc. 15th Colloquium on Trees an Algebra and Programming, CAAP'90*, Number 431 in Lecture Notes in Computer Science (May 1990), pp. 103–116. Springer-Verlag.

CLARKE, E. M., GRUMBERG, O., AND HAMAGUCHI, K. 1994. Another look at LTL model checking. In D. L. DILL Ed., *6th International Conference on Computer-Aided Verification, CAV '94*, Number 818 in Lecture Notes in Computer Science (June 1994), pp. 415–427. Springer-Verlag.

CLARKE, E. M., GRUMBERG, O., AND LONG, D. E. 1994. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems 16*, 5 (Sept.), 1512–1542.

CLARKE, E. M., KHAIRA, M., AND ZHAO, X. 1996. Word level model checking — avoiding the Pentium FDIV error. In *33rd Design Automation Conference, DAC '96* (June 1996), pp. 645–648.

COHN, A. 1988. A proof of correctness of the Viper microprocessor: The first level. In G. BIRTWISTLE AND P. SUBRAHMANYAM Eds., *VLSI Specification, Verification and Synthesis*, pp. 27–71. Kluwer Academic Publishers.

COHN, A. 1989a. Correctness properties of the Viper block model: The second level. In G. BIRTWISTLE AND P. SUBRAHMANYAM Eds., *Current Trends in Hardware Verification and Automated Theorem Proving*, pp. 1–91. Springer-Verlag.

COHN, A. 1989b. The notion of proof in hardware verification. *Journal of Automated Reasoning 5*, 127–139.

Corella, F., Zhou, Z., Song, X., Langevin, M., and Cerny, E. 1994. Multiway Decision Graphs for automated hardware verification. Research Report RC 19676 (July), IBM.

Coudert, O., Berthet, C., and Madre, J. C. 1989. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis Ed., *Automatic Verification Methods for Finite State Systems*, Number 407 in Lecture Notes in Computer Science (June 1989), pp. 365–373. Springer-Verlag.

Curzon, P. 1994. The formal verification of the Fairisle ATM switching element: an overview. Technical Report 328 (March), University of Cambridge Computer Laboratory.

Curzon, P. and Leslie, I. 1996. Improving hardware design whilst simplyfying their proof. In *3rd Workshop on Designing Correct Circuits (DCC)* (Båstad, Sweden, Sept. 1996).

Cyrluk, D., Rajan, S., Shankar, N., and Srivas, M. K. 1994. Effective theorem proving for hardware verification. In R. Kumar and T. Kropf Eds., *2nd Intl. Conference on Theorem Provers in Circuit Design, TPCD '94*, Number 901 in Lecture Notes in Computer Science (Sept. 1994), pp. 203–222. Springer-Verlag.

Dill, D. L. 1996. The murφ verification system. In *8th International Conference on Computer-Aided Verification, CAV '96*, Number 1102 in Lecture Notes in Computer Science (July 1996), pp. 390–393. Springer-Verlag.

Dill, D. L., Drexler, A. J., Hu, A. J., and Yang, C. H. 1992. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors, ICCD'92* (1992), pp. 522–525. IEEE Computer Society.

Duffy, D. A. 1991. *Principles of Automated Theorem Proving*. John Wiley & Sons, Chichester.

Eiríksson, A. T. 1996. Integrating formal verification methods with a conventional project design flow. In *33rd Design Automation Conference, DAC '96* (June 1996), pp. 666–671.

Eiríksson, A. T. and McMillan, K. L. 1995. Using formal verification/analysis methods on the critical path in system design. In P. Wolper Ed., *7th International Conference on Computer-Aided Verification, CAV '95*, Number 939 in Lecture Notes in Computer Science (July 1995), pp. 367–380. Springer-Verlag.

Emerson, E. A. 1990. Temporal and modal logic. In J. van Leeuwen Ed., *Handbook of Theoretical Computer Science*, Volume B, pp. 995–1072. The MIT Press/Elsevier Science Publishers.

Emerson, E. A. and Halpern, J. Y. 1986. "Sometimes" and "not never" revisited: On branching versus linear time temporal logic. *Journal of the ACM 33*, 1 (Jan.), 151–178.

Engberg, U., Grønning, P., and Lamport, L. 1992. Mechanical verification of concurrent systems with tla. In *4th International Conference on Computer-Aided Verification, CAV '92*, Number 663 in Lecture Notes in Computer Science (1992), pp. 44–55. Springer-Verlag.

Garland, S. J. and Guttag, J. V. 1989. An overview of LP, the Larch prover. In N. Dershowitz Ed., *Rewriting Techniques and Applications, RTA-89*, Number 355 in Lecture Notes in Computer Science (April 1989), pp. 137–151. Springer-Verlag.

Gerth, R. 1989. Foundations of compositional program refinement — safety properties —. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg Eds., *Stepwise Refinement of Distributed Systems*, Number 430 in Lecture Notes in Computer Science (1989), pp. 777–808. Springer-Verlag.

Gordon, M. 1985. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. J. Milne and P. A. Subrahmanyam Eds., *Formal Aspects of VLSI Design* (1985), pp. 153–177. Elsevier Science Publishers.

Gordon, M. J. 1988. HOL: a proof generating system for higher-order logic. In G. Birtwistle and P. Subrahmanyam Eds., *VLSI Specification, Verification and Synthesis*, pp. 74–128. Kluwer Academic Publishers.

Gordon, M. J. C., Wadsworth, C. P., and Milner, A. J. 1979. *Edinburgh LCF: a mechanised logic of computation.* Number 78 in Lecture Notes in Computer Science. Springer-Verlag.

Gupta, A. 1992. Formal hardware verfification methods: A survey. *Formal Methods in System Design 1*, 2/3 (Oct.), 151–238.

Hardin, R. H., Har'El, Z., and Kurshan, R. P. 1996. COSPAN. In *8th International Conference on Computer-Aided Verification, CAV '96*, Number 1102 in Lecture Notes in Computer Science (July 1996), pp. 423–427. Springer-Verlag.

HAREL, D. 1988. On visual formalisms. *Communications of the ACM 31*, 5 (May), 514–530.

HARKNESS, C. AND WOLF, E. 1994. Verifying the Summit bus converter protocols with symbolic model checking. *Formal Methods in System Design 4*, 83–97.

HARRISON, J. 1993. A HOL decision procedure for elementary real algebra. In J. J. JOYCE AND C.-J. H. SEGER Eds., *Higher Order Logic Theorem Proving and Its Applications, HUG '93*, Number 780 in Lecture Notes in Computer Science (Vancouver, Aug. 1993), pp. 426–436. Springer-Verlag.

HARRISON, J. 1995. Floating point verification in HOL. In E. T. SCHUBERT, P. J. WINDLEY, AND J. ALVES-FOSS Eds., *Higher Order Logic Theorem Proving and Its Applications, 8th Intl. Workshop*, Number 971 in Lecture Notes in Computer Science (Sept. 1995), pp. 186–199. Springer-Verlag.

HAZELHURST, S. AND SEGER, C.-J. H. 1994. Composing symbolic trajectory evaluation results. In D. L. DILL Ed., *6th International Conference on Computer-Aided Verification, CAV'94*, Number 818 in Lecture Notes in Computer Science (June 1994), pp. 273–285. Springer-Verlag.

HAZELHURST, S. AND SEGER, C.-J. H. 1995. A simple theorem prover based on symbolic trajectory evaluation and BDDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 14*, 4 (April), 413–422.

HENNESSY, J. L. AND PATTERSON, D. A. 1990. *Computer Architecture : A Quantitative Approach*. Morgan Kaufman.

HOARE, C. A. R. 1978. Communicating sequential processes. *Communications of the ACM 21*, 8 (Aug.), 666–677.

HOJATI, R., BRAYTON, R. K., AND KURSHAN, R. P. 1993. BDD-based debugging of designs using language containment and fair CTL. In C. COURCOUBETIS Ed., *5th International Conference on Computer-Aided Verification, CAV '93*, Number 697 in Lecture Notes in Computer Science (June 1993), pp. 41–58. Springer-Verlag.

HOJATI, R., SINGHAL, V., AND BRAYTON, R. K. 1994. Edge-Streett/Edge-Rabin automata environment for formal verification using language containment. Memorandum UCB/ERL M94/12, Electronics Res. Lab., University of California, Berkeley.

HU, A. J. AND DILL, D. L. 1993. Reducing BDD size by exploiting functional dependencies. In *Proc. 30th Design Automation Conference* (1993), pp. 266–27.

HU, A. J., DILL, D. L., DREXLER, A. J., AND YANG, C. H. 1992. Higher-level specification and verification with BDDs. In *4th Intl. Workshop on Computer-Aided Verification* (Montreal, Quebéc, July 1992).

HU, A. J., YORK, G., AND DILL, D. L. 1994. New techniques for efficient verification with implicitly conjoined BDDs. In *Proc. 31st Design Automation Conference* (1994).

HUNT, W. A. 1989. Microprocessor design verification. *Journal of Automated Reasoning 5*, 4, 411–428.

HUNT, W. A. 1994. *FM8501: A Verified Microprocessor*. Number 795 in Lecture Notes in Computer Science. Springer-Verlag, Berlin.

HUNT, W. A. AND BROCK, B. 1992. A formal HDL and its use in the FM9001 verification. Technical Report 79 (July), Computational Logic, Inc., Austin, TX.

IP, C. N. AND DILL, D. L. 1993. Efficient verification of symmetric concurrent systems. In *1993 Intl. Conference on Computer Design: VLSI in Computers & Processors* (1993), pp. 230–234. IEEE Computer Society Press.

IP, C. N. AND DILL, D. L. 1996a. Better verification through symmetry. *Formal Methods in System Design 9*, 1/2 (Aug.), 41–75.

IP, C. N. AND DILL, D. L. 1996b. State reduction using reversible rules. In *33rd Design Automation Conference, DAC '96* (June 1996), pp. 564–567.

JAIN, J., ABRAHAM, J. A., BITNER, J., AND FUSSELL, D. S. 1996. Probabilistic verification of boolean functions. *Formal Methods in System Design 1*, 1 (July), 63–115.

JONES, R. B., DILL, D. L., AND BURCH, J. R. 1995. Efficient validity checking for processor verification. In *International Conference on Computer-Aided Design, ICCAD '95* (1995).

JOSKO, B. 1989. Verifying the correctness of AADL modules using model checking. In J. W. DE BAKKER, W.-P. DE ROEVER, AND G. ROZENBERG Eds., *Stepwise Refinement of Distributed Systems*, Number 430 in Lecture Notes in Computer Science (1989), pp. 386–400. Springer-Verlag.

JOYCE, J. J. 1988. Formal verification and implementation of a microprocessor. In G. BIRTWISTLE AND P. SUBRAHMANYAM Eds., *VLSI Specification, Verification and Synthesis*, pp. 129–157. Kluwer Academic Publishers.

JOYCE, J. J. AND SEGER, C.-J. H. 1993. Linking BDD-based symbolic evaluation to interactive theorem-proving. In *Proc. 30th ACM/IEEE Design Automation Conference* (1993), pp. 469–474.

KAUFMANN, M. AND MOORE, J. S. 1994. Design goals for ACL2. Technical Report 101 (Aug.), Computational Logic, Inc., Austin, TX.

KAUFMANN, M. AND MOORE, J. S. 1996. ACL2: an industrial strength version of Nqthm. In *Proc. 11th Annual Conference on Computer Assurance (COMPASS '96)* (June 1996), pp. 23–34. IEEE Computer Society Press.

KESTEN, Y., MANNA, Z., MCGUIRE, H., AND PNUELI, A. 1993. A decision algorithm for full propositional temporal logic. In C. COURCOUBETIS Ed., *5th International Conference on Computer-Aided Verification, CAV '93*, Number 697 in Lecture Notes in Computer Science (June 1993), pp. 97–109. Springer-Verlag.

KLEENE, S. C. 1967. *Mathematical Logic*. John Wiley & Sons, Chichester.

KLOOS, C. D. AND BREUER, P. 1995. *Formal Semantics for VHDL*. Kluwer Academic Publishers, Boston.

KOZEN, D. 1993. Results on the propositional μ-calculus. *Theoretical Computer Science 27*, 3 (Dec.), 333–354.

KUEHLMANN, A., SRINIVASAN, A., AND LAPOTIN, D. P. 1995. Verity – a formal verification program for custom CMOS circuits. *IBM Journal of Research and Development 39*, 1/2 (Jan./March), 149–165.

KUMAR, R., SCHNEIDER, K., AND KROPF, T. 1993. Structuring and automating hardware proofs in a higher-order theorem-proving environment. *Formal Methods in System Design 2*, 165–230.

KURSHAN, R. P. 1989. Analysis of discrete event coordination. In J. W. DE BAKKER, W.-P. DE ROEVER, AND G. ROZENBERG Eds., *Stepwise Refinement of Distributed Systems*, Number 430 in Lecture Notes in Computer Science (1989), pp. 414–453. Springer-Verlag.

KURSHAN, R. P. AND LAMPORT, L. 1993. Verification of a multiplier: 64 bits and beyond. In C. COURCOUBETIS Ed., *5th International Conference on Computer-Aided Verification, CAV '93*, Number 697 in Lecture Notes in Computer Science (June 1993), pp. 166–179. Springer-Verlag.

LAI, Y.-T. AND SASTRY, S. 1992. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *29th Design Automation Conference, DAC '92* (June 1992), pp. 608–613.

LAMPORT, L. 1980. "Sometime" is sometimes "not never" — on the temporal logic of programs. In *7th Ann. ACM Symposium on Principles of Programming Languages* (1980), pp. 174–185.

LAMPORT, L. AND SCHNEIDER, F. B. 1984. The "Hoare logic" of CSP, and all that. *ACM Transactions on Programming Languages and Systems 6*, 2 (April), 281–296.

LEE, T. W., GREENSTREET, M. R., AND SEGER, C.-J. 1994a. Automatic verification of asynchronous circuits. *IEEE Design and Test 12*, 1 (Spring), 24–31.

LEE, T. W., GREENSTREET, M. R., AND SEGER, C.-J. 1994b. Automatic verification of refinement. In *Proceedings of the 1994 International Conference on Computer Design* (Boston, Oct. 1994).

LEESER, M. AND O'LEARY, J. 1995. Verification of a subtractive radix-2 square root algorithm and implementation. In *1995 Intl. Conference on Computer Design: VLSI in Computers & Processors* (Oct. 1995), pp. 526–531. IEEE Computer Society Press.

LENOSKI, D., LAUDON, J., GHARACHORLOO, K., WEBER, W.-D., GUPTA, A., HENNESSY, J., HOROWITSZ, M., AND LAM, M. S. 1992. The Stanford Dash multiprocessor. *IEEE Computer 25*, 3 (March), 63–79.

LESLIE, I. AND MCAULEY, D. 1991. Fairisle: An ATM network for the local area. In *Proc. SIGCOMM-91, Computer Communications Review, 21(4)* (Sept. 1991), pp. 327–336. ACM Press.

MANNA, Z., ANUCHITANUKUL, A., BJØRNER, N., BROWNE, A., CHANG, E., COLÓN, M., DE ALFARO, L., DEVARAJAN, H., SIPMA, H., AND URIBE, T. 1994. STeP: The Stanford temporal prover. Technical Report STAN-CS-TR-94-1518 (June), Computer Science Department, Stanford University.

Manna, Z. and Pnueli, A. 1992. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, Berlin.

Manna, Z. and Wolper, P. 1984. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems 6*, 1 (Jan.), 68–93.

Maretti, N. 1994. Mechanized verification of refinement. In R. Kumar and T. Kropf Eds., *2nd Intl. Conference on Theorem Provers in Circuit Design, TPCD '94*, Number 901 in Lecture Notes in Computer Science (Sept. 1994), pp. 185–202. Springer-Verlag.

McCune, W. 1994. OTTER 3.0. Preprint MCS-P399-1193 (March), Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill. URL: ftp://info.mcs.anl.gov/pub/tech_reports/reports/P399.ps.Z.

McFarland, M. C. 1993. Formal verification of sequential hardware: A tutorial. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 12*, 5 (May), 663–654.

McMillan, K. 1994. Fitting formal methods in to the design cycle. In *31st Design Automation Conference, DAC '94* (1994).

McMillan, K. L. 1992. Symbolic model checking – an approach to the state explosion problem. Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, 1992.

McMillan, K. L. and Schwalbe, J. 1991. Formal verification of the Encore Gigamax cache consistency protocol. In *Proc. of the 1991 Intl. Symposium on Shared Memory Multiprocessors* (April 1991).

Melham, T. F. 1988. Abstraction mechanisms for hardware verification. In G. Birtwistle and P. Subrahmanyam Eds., *VLSI Specification, Verification and Synthesis*, pp. 267–291. Kluwer Academic Publishers.

Miller, S. P. and Srivas, M. 1995. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques* (Boca Raton, FL, April 1995), pp. 2–16.

Miner, P. S. and Leathrum, Jr., J. F. 1996. Verification of IEEE compliant subtractive division algorithms. In M. Srivas and A. Camilleri Eds., *1st Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD '96)*, Number 1166 in Lecture Notes in Computer Science (Nov. 1996), pp. 64–78. Springer-Verlag.

Moore, J. S., Lynch, T., and Kaufmann, M. 1996. Mechanically checked proof of the correctness of the kernel of the $\mathrm{AMD5}_K 86^{TM}$ floating-point division algorithm. URL: http://devil.ece.utexas.edu:80/ lynch/divide/divide.html.

Nelson, G. and Oppen, D. C. 1979. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems 1*, 2 (Oct.), 245–257.

Nowatzyk, A., Aybay, G., Browne, M., Kelly, E., Parkin, M., Radke, W., and Vishin, S. 1995. The S3.mp scalable shared memory multiprocessor. In *ICPP'95* (1995).

Owicki, S. and Lamport, L. 1982. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems 4*, 3 (July), 455–495.

Owre, S., Rajan, S., Rushby, J., Shankar, N., and Srivas, M. 1996. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger Eds., *8th International Conference on Computer-Aided Verification, CAV '96*, Number 1102 in Lecture Notes in Computer Science (July/Aug. 1996). Springer-Verlag.

Owre, S., Rushby, J. M., and Shankar, N. 1992. PVS: A prototype verification system. In D. Kapur Ed., *11th International Conference on Automated Deduction (CADE)*, Number 607 in Lecture Notes in Artificial Intelligence (Saratoga, NY, 1992), pp. 748–752. Springer-Verlag.

Pandey, M., Raimi, R., Beatty, D. L., and Bryant, R. E. 1996. Formal verification of $\mathrm{PowerPC}^{TM}$ arrays using symbolic trajectory evaluation. In *33rd Design Automation Conference, DAC '96* (June 1996), pp. 649–654.

Pixley, C., Jeong, S.-W., and Hachtel, G. D. 1994. Exact calculation of synchronizing sequences based on binary decision diagrams. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13*, 8 (Aug.), 1024–1034.

PNUELI, A. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science* (Providence, Rhode Island, 1977), pp. 46–57. IEEE.

PNUELI, A. 1986. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In J. W. DE BAKKER, W.-P. DE ROEVER, AND G. ROZENBERG Eds., *Current Trends in Concurrency*, Number 224 in Lecture Notes in Computer Science (1986), pp. 510–584. Springer-Verlag.

PONG, F. AND DUBOIS, M. 1993. The verification of cache coherence protocols. In *Proc. of the 5th Annual Symp. on Parallel Algorithm and Architecture* (June 1993), pp. 11–20.

PONG, F., NOWATZYK, A., AYBAY, G., AND DUBOIS, M. 1995. Verifying distributed directory-based cache coherence protocols: S3.mp, a case study. In *Intl. Conference on Parallel Processing, EURO-PAR'95* (Stockholm, Sweden, Aug. 1995).

RAJAN, S., SHANKAR, N., AND SRIVAS, M. 1995. An integration of model-checking with automated proof checking. In P. WOLPER Ed., *Computer-Aided Verification, CAV '95*, Lecture Notes in Computer Science (Liege, Belgium, July 1995), pp. 84–97. Springer-Verlag.

RUDELL, R. 1993. Dynamic variable ordering for Ordered Binary Decision Diagrams. In *International Conference on Computer-Aided Design, ICCAD '93* (Santa Clara, CA, Nov. 1993), pp. 42–47.

RUESS, H., SHANKAR, N., AND SRIVAS, M. 1996. Modular verification of SRT division. In R. ALUR AND T. HENZINGER Eds., *8th International Conference on Computer-Aided Verification, CAV '96*, Number 1102 in Lecture Notes in Computer Science (New Brunswick, NJ, 1996). Springer-Verlag.

SAXE, J. B., HORNING, J. J., GUTTAG, J. V., AND GARLAND, S. J. 1993. Using transformations and verification in circuit design. *Formal Methods in System Design 3*, 3 (Dec.), 181–209.

SCHNEIDER, K. AND KROPF, T. 1996. A unified approach for combining different formalisms for hardware verification. In M. SRIVAS AND A. CAMILLERI Eds., *1st Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD '96)*, Number 1166 in Lecture Notes in Computer Science (Nov. 1996), pp. 202–217. Springer-Verlag.

SEGER, C.-J. H. 1992. An introduction to formal hardware verification. Technical Report TR-92-13, Department of Computer Science, University of British Columbia, Vancouver. URL: http://www.cs.ubc.ca/tr/1992/TR-92-13.ps.

SEGER, C.-J. H. 1993. Voss – a formal hardware verification system, user's guide. Technical Report TR-93-45, Department of Computer Science, University of British Columbia, Vancouver. URL: http://www.cs.ubc.ca/tr/1993/TR-93-45.ps.

SEGER, C.-J. H. AND BRYANT, R. E. 1995. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design 6*, 147–189.

SENTOVICH, E., SINGH, K., LAVAGNO, L., MOON, C., MURGAI, R., SALDANHA, A., SAVOJ, H., STEPHAN, P., BRAYTON, R., AND SANGIOVANNI-VINCENTELLI, A. 1992. SIS: a system for sequential circuit synthesis. Technical Report UCB/ERL M92/41 (May), Electronics Research Lab, Univ. of California, Berkeley.

SHANKAR, A. U. 1993. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys 25*, 3 (Sept.), 225–262.

SHOSTAK, R. E. 1979. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM 26*, 2 (April), 351–360.

SHOSTAK, R. E. 1984. Deciding combinations of theories. *Journal of the ACM 31*, 1 (Jan.), 1–12.

SISTLA, A. P. AND CLARKE, E. M. 1985. The complexity of propositional linear temporal logics. *Journal of the ACM 32*, 3 (July), 733–749.

SRIVAS, M. AND BICKFORD, M. 1990. Formal verification of a pipelined microprocessor. *IEEE Software 7*, 5 (Sept.), 52–64.

STAUNSTRUP, J. 1994. *A formal approach to hardware design.* Kluwer Academic Publishers, Boston.

STERN, U. AND DILL, D. L. 1995. Automatic verification of the SCI cache coherence protocol. In P. E. CAMURATI AND H. EVEKING Eds., *Correct Hardware Design and Verification Methods, CHARME '95*, Number 987 in Lecture Notes in Computer Science (Oct. 1995), pp. 21–34. Springer-Verlag.

TAHAR, S. AND CURZON, P. 1996. A comparison of MDG and HOL for hardware verification. In J. VON WRIGHT, J. GRUNDY, AND J. HARRISON Eds., *Theorem Proving in Higher Order Logics: 9th International Conference*, Number 1125 in Lecture Notes in Computer Science (1996), pp. 415–430. Springer-Verlag.

TAHAR, S. AND KUMAR, R. 1993. Implementing a methodology for formally verifying RISC processors in HOL. In J. J. JOYCE AND C.-J. H. SEGER Eds., *Higher Order Logic Theorem Proving and Its Applications, HUG '93*, Number 780 in Lecture Notes in Computer Science (Vancouver, Aug. 1993), pp. 281–294. Springer-Verlag.

TAHAR, S., ZHOU, Z., SONG, X., CERNY, E., AND LANGEVIN, M. 1996. Formal verification of an ATM switch fabric using Multiway Decision Graphs. In *Great Lakes Symposium on VLSI* (Ames, Iowa, March 1996). IEEE Computer Society Press.

TARSKI, A. 1951. *Decision Method for Elementary Algebra and Geometry*. University of California Press, Berkeley, CA.

THOMAS, W. 1990. Automata on infinite objects. In J. VAN LEEUWEN Ed., *Handbook of Theoretical Computer Science*, Volume B, pp. 133–191. The MIT Press/Elsevier Science Publishers.

TOUATI, H. J., SAVOJ, H., LIN, B., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. 1990. Implicit state enumeration of finite state machines using BDDs. In *International Conference on Computer-Aided Design, ICCAD '90* (Nov. 1990), pp. 130–133.

VAN TASSEL, J. P. 1993. A formalism of the VHDL simluation cycle. In L. CLAESEN AND M. GORDON Eds., *Higher Order Logic Theorem Proving and its Applications* (Sept. 1993), pp. 359–374. North-Holland.

VARDI, M. AND WOLPER, P. 1986a. An automata-theoretic approach to automatic program verification. In *Proc. 1st Annual Symposium on Logic in Computer Science* (1986), pp. 332–344. IEEE Computer Society Press.

VARDI, M. AND WOLPER, P. 1986b. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Sciences 32*, 183–221.

VARDI, M. Y. AND WOLPER, P. 1994. Reasoning about infinite computations. *Information and Control 115*, 1 (Nov.), 1–37.

VETTER, R. J. 1995. ATM concepts, architectures, and protocols. *Communications of the ACM 38*, 2 (Feb.), 30–38,109.

WINDLEY, P. J. 1991. Using correctness results to verify behavioral properties of microprocessors. In *Proceedings of the IEEE Computer Assurance Conference* (June 1991).

WINDLEY, P. J. 1992. Abstract theories in HOL. In L. J. M. CLAESEN AND M. J. C. GORDON Eds., *Proc. IFIP TC10/WG10.2 Intl. Workshop on Higher Order Logic Theorem Proving and Its Applications, HOL '92* (1992). North Holland.

WINDLEY, P. J. 1995a. Formal modeling and verification of microprocessors. *IEEE Transactions on Computers 44*, 1 (Jan.), 54–72.

WINDLEY, P. J. 1995b. Verifying pipelined microprocessors. Technical report (Oct.), Laboratory for Applied Logic, Brigham Young University. URL: ftp://lal.cs.byu.edu/pub/hol/lal-papers/correct.ps.

WINDLEY, P. J. AND COE, M. 1994. A correctness model for pipelined microprocessors. In R. KUMAR AND T. KROPF Eds., *2nd Intl. Conference on Theorem Provers in Circuit Design, TPCD '94*, Number 901 in Lecture Notes in Computer Science (Sept. 1994), pp. 33–51. Springer-Verlag.

WOLPER, P. 1983. Temporal logic can be more expressive. *Information and Control 56*, 1/2 (Jan./Feb.), 72–99.

ZHU, Z. AND SEGER, C.-J. H. 1994. The completeness of a hardware inference system. In D. L. DILL Ed., *6th International Conference on Computer-Aided Verification, CAV '94*, Number 818 in Lecture Notes in Computer Science (June 1994), pp. 286–298. Springer-Verlag.

Contents