Copyright © 1995, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

A METHODOLOGY FOR FORMAL VERIFICATION OF REAL-TIME SYSTEMS

by

Felice Balarin, Robert K. Brayton, Szu-Tsu Cheng,
Desmond A. Kirkpatrick, Alberto L. Sangiovanni-Vincentelli,
and Ephrem C. Wu

Memorandum No. UCB/ERL M95/11

27 February 1995

A METHODOLOGY FOR FORMAL VERIFICATION OF REAL-TIME SYSTEMS

by

Felice Balarin, Robert K. Brayton, Szu-Tsu Cheng,
Desmond A. Kirkpatrick, Alberto L. Sangiovanni-Vincentelli,
and Ephrem C. Wu

Memorandum No. UCB/ERL M95/11

27 February 1995

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A Methodology for Formal Verification of Real-Time Systems

Felice Balarin Robert K. Brayton Szu-Tsu Cheng
Desmond A. Kirkpatrick Alberto L. Sangiovanni-Vincentelli
Ephrem C. Wu
Dpt. of EECS
University of California, Berkeley

Abstract

We propose a methodology for formal verification of real-time systems with the following components:

Formal Model The use of a formal model eliminates ambiguities and inconsistencies that often arise in informal or simulation-based descriptions. We define a powerful model that includes as special cases several models that can be automatically verified.

Intermediate Format We have developed BLIF_MVT, an intermediate format to describe our formal model. Thus, to support different system description languages one only needs to provide a translator to BLIF_MVT.

System Specification To specify real-time systems in a form suitable for synthesis, simulation and verification we interpret in our model a large subset of Verilog, which includes delay and event operators.

Property Specification To specify timed properties and timing constraints in a form natural for the designer, we define TESLA, an annotation language for embedding in Verilog.

Automatic or Directed Verification Both TESLA and the subset of Verilog that we consider describe systems that can be automatically verified. However, we have observed that the efficiency of the verification can be significantly improved if the designer provides guidelines describing *why* the system should satisfy the property.

Together, these components provide a full path from high level descriptions of real-time systems and properties to powerful real-time verification capabilities. We have developed a set of tools to support this methodology. We give some preliminary results on a set of small examples.

1 Introduction

The complexity of electronic systems being designed today requires new ways of verifying functionality and performance. In particular, verification of the design at an early phase may reduce substantially design time and yield better quality of the final implementation. Verification at an early stage implies an abstract description of the design in terms of its behavior. At this level of abstraction it is relatively easy to express in a formal way properties the system must have to behave correctly. In this case, it is possible to use formal techniques to check whether the system

has the desired property. This methodology is particularly appealing since using standard verification techniques such as simulation (even at the behavioral level) may require very long time and not guarantee the same accuracy. However, we strongly believe that simulation is indispensable to verify some behavior that cannot be described in abstract form. Thus, we believe, the formal verification may be successful only if integrated in a design flow.

In more traditional design flows, timing specifications enter relatively late in design refinement. However, more and more often timing requirements are most difficult to meet and lengthen the design time. Hence, we believe timing models and specifications have to be given at a higher level of abstractions and possibly be formally verified.

In this paper, we present a timing-driven formal verification framework consisting of a language to describe the system being verified, of a formalism to express properties, of a mathematical model of timed systems, and of algorithms for formal verification of timed properties.

The framework has been developed with a design flow based on successive refinement and hierarchical verification in mind. In this methodology, a design is described at a high level of abstraction together with a set of timed and untimed properties. Extensive verification using formal methods as well as simulation is carried out. Successive refinement is used to bring the description to a level of detail that allows implementation on a given architecture or set of gates. If the successive refinement process follows a formal scheme (for example see [Kur90]), then the properties verified at the entry level are guaranteed to hold. Hence, only properties that relate to lower levels of hierarchy need to be verified. In case a formal scheme is not followed, all properties should be re-verified at all levels, making the verification process quite expensive.

The verification paradigm based on the previous description is agreed upon by the formal verification community and by some high-level designers. However, much remains to be done to bring formal verification in the hands of a larger design community, especially in the case of timing verification.

Most of the existing formal verification systems (like SMV [McM93], MurPhi [DDHY92] and Auto/Autograph [RdS92]) do not support quantitative timing constraints. Only a few systems for formal timing verification are available:

- RT-COSPAN [CDCT93] is based on *timed automata* [AD90]. Unfortunately, both the system and the property to be verified are described in a dedicated language (S/R), so it is hard to integrate in the design flow, and significant user's expertise is necessary to specify the properties to be verified. Furthermore, there is no intermediate format, so developing an interface to different high-level languages is difficult.

- EPSILON [ČGL93] is based on *timed modal specifications* (a formalism similar to timed automata), and it suffers from similar drawbacks as RT-COSPAN.
- KRONOS [NSY92] automatically verifies whether a description in an intermediate format represents a model of a given formula of *timed computational tree logic* (TCTL). The intermediate format used in KRONOS can express structures that are essentially equivalent to timed automata. Translators from a higher level languages are available, but they are either not supported by other tools (ARGOS), or are rarely used in the design community (LOTUS-ET). Furthermore, the English-like syntax of TCTL may be appealing to designers on first sight, but its precise semantics is intricate, and specifying properties (particularly those involving sequences of events) is quite error-prone.

Based on the drawbacks of existing systems, we have formulated a set of requirements that we have followed in developing our framework.

Requirement 1 *The formal verification framework must support a language to describe real-time systems at a high level of abstraction.*

Formal verification is most valuable if it can help find design flaws early in the design cycle. Unfortunately, the higher a level of abstraction is, the more complex are the timing constraints, and the more difficult is the verification problem. At a lower-level of abstraction, assigning fixed delays to gates may be sufficient to capture timing constraints. Then, simple longest path techniques can be used to verify the system. At a higher level of abstraction, it is not that simple, e.g. the speed by which a software task progresses might change in time depending on the load of the system. Many formalisms that can capture complex timing constraints have been proposed, but most are not suitable for automatic verification, because the associated verification problems are undecidable. Two notable exceptions are timed automata [Dil89, AD90] and alternating RQ automata [LB93, LB94]. Since both of these models can describe some systems that the other one cannot, we decided to support them both. To provide a uniform framework for both models, we define a new model called TALE (Timed Automata with Linear inEqualities), which includes both timed and alternating RQ automata as special cases. We base our methodology on this formal model.

Requirement 2 *Properties to be verified must be expressed in a form "natural" to the designer.*

The answers provided by the formal verification tool are only as good as questions asked. We can never guarantee that all property descriptions are error-free, but it is plausible to assume that the property is more likely to be correct if it is simple and expressed in a form natural to the person who wrote it down.

For property specification we define TESLA, a simple annotation language for embedding in Verilog. TESLA is capable of describing sequences of events, as well as timing constraints between them. TESLA can also be used to place some restrictions on a system's specification.

Requirement 3 *Formal verification must be integrated in the design flow.*

If verification is done after the design, its value is diminished. If verification is done in parallel but not integrated with the design, inconsistencies between verification and design descriptions of the system are likely. Therefore, formal verification should be integrated in the design flow. To achieve this goal we define formal semantics for a subset of Verilog that includes subsets supported by popular synthesis tools, plus delay and event operators.

We want a verification methodology that can be easily adapted to any design flow, not only those using Verilog. For this purpose, we define BLIF_MVT, an intermediate format for compactly describing TALE's. To adapt our methodology and tools to different system description languages only a translator to BLIF_MVT needs to be provided.

Requirement 4 *Designer's inputs must be allowed to make the verification process feasible.*

Both timed and alternating RQ automata can be verified automatically, but we have observed that the efficiency of the verification can be significantly improved if the designer provides guidelines describing *why* the system should satisfy the property. Since these guidelines also provide a useful documentation of the system, a verification tool supporting the methodology should be able to take advantage of any such guidelines.

To support our methodology we provide the following tools:

- procedures for formal verification of timed and alternating RQ automata implemented in the verification system HSIS [ABB+94]; both procedures can be guided by user-provided *hints*, describing timing constraints that are critical for the property to be verified,
- vl2mvt, a compiler from a subset of Verilog to BLIF_MVT,
- a compiler from TESLA to the subset of Verilog supported by vl2mvt.

Figure 1 provides an overview of the proposed methodology. Starting from the Verilog description annotated with TESLA, we use the TESLA compiler to translate it Verilog. If the TESLA property includes eventualities (e.g. a request must eventually be acknowledged), it is not possible to express it completely either in Verilog, or in BLIF_MVT, so for this purpose we define the *property interchange format* (PIF). PIF can also be used for specifying so-called *fairness constraints* on the system, that arise naturally at the higher level of abstractions (e.g. an internal computation takes some unknown time, but always terminates). The Verilog description is then translated to BLIF_MVT by vl2mvt. Finally, HSIS reads in both BLIF_MVT and PIF descriptions. At this point the user can choose between the timed and alternating RQ automata verification algorithms (provided the descriptions satisfy corresponding constraints), and optionally provide hints that can speed up the computation. If the property to be verified does not hold, HSIS generates a failure trace that is an example of the behavior of the system that violates the property.

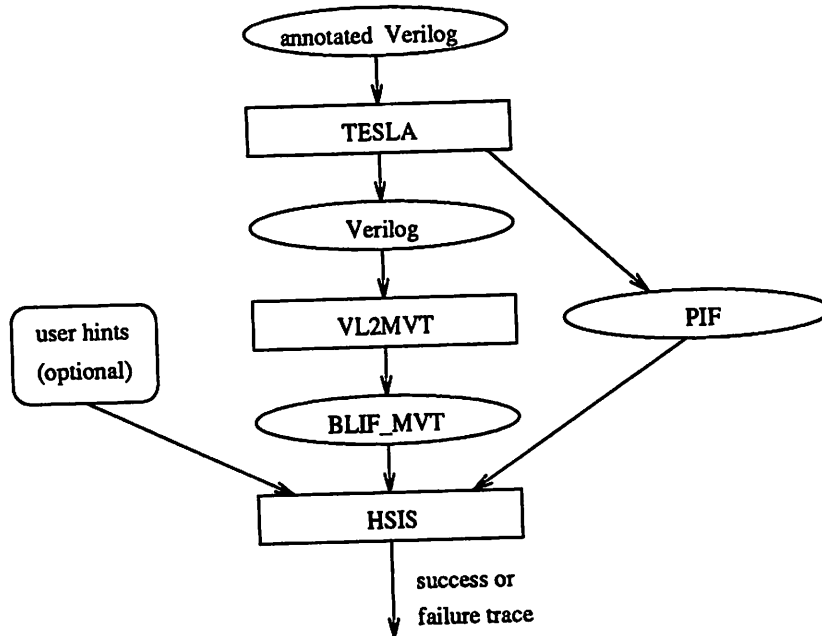

Figure 1: Overview of the proposed methodology.

The rest of the paper is organized as follows. In section 2 we define *timed automata with linear inequalities*, a semantic model of real-time systems used in our methodology. This intermediate format BLIF_MVT is presented in section 3. For system specification, we provide a compiler from a subset of Verilog to BLIF_MVT. This subset is described in section 4. The TESLA language for property specification is introduced in section 5. Verification algorithms for alternating RQ and

timed automata are reviewed in section 6. Initial experimental results are discussed in section 7.

2 TALE: a Formal Model of Real-Time Systems

Let $V = \{x_1, \ldots, x_n\}$ denote a set of *timer variables*. *Timing constraints* are formulas of the form $\sum_{i=1}^{n} a_i x_i \sim a_{n+1}$ where $a_1, \ldots, a_{n+1}$ are real numbers, and $\sim$ is one of $<$, $\leq$, $=$, $\geq$, or $>$. Let Φ denote the set of all timing constraints and let 2^{Φ} be the set of all *finite* subsets of Φ. A *timer valuation* $\tau : V \rightarrow R$ assigns a real value to every timer variable. We say that a timer valuation τ satisfies a timing constraint $\sum_{i=1}^{n} a_i x_i \sim a_{n+1}$ if $\sum_{i=1}^{n} a_i \tau(x_i) \sim a_{n+1}$.

A *timed automaton with linear inequalities* (TALE) is a 7-tuple $(\Sigma, Q, I, TR, V, TO, R)$ where Σ is a finite set of *I/O values*, Q is a finite set of *states*, $I \subseteq Q$ is a set of *initial states*, $V = \{x_1, \ldots, x_n\}$ is a set of timer variables, $TR \subseteq Q \times \Sigma \times Q$ is a *transition relation*, $TO : Q \times Q \rightarrow 2^{\Phi}$ is a *timing obligation*, and $R \subseteq Q \times Q \times V$ is a *reset relation*.

Given a TALE $(\Sigma, Q, I, TR, V, TO, R)$ we say that (q, q') is a *reset edge* of a timer $x \in V$ if $(q, q', x) \in R$. If there exists a timing constraint in $TO(q, q')$ in which the coefficient of x is different from 0, we say that (q, q') is a *query edge* of x.

A sequence of states $q_0 q_1 \ldots \in Q^{\omega}$ is a *run* of a sequence of I/O values $\sigma_0 \sigma_1 \ldots \in \Sigma^{\omega}$ if for all $i \geq 0$: $(q_i, \sigma_i, q_{i+1}) \in TR$. A run is *initialized* if $q_0 \in I$. A sequence $\delta_0, \delta_1, \ldots$ of positive real numbers is a *consistent timing* of $q_0 q_1 \ldots$ if there exists a sequence of timer valuations $\tau_0 \tau_1 \ldots$ such that for all $i \geq 0$:

the timing obligation is fulfilled: τ_i satisfies *all* timing constraints in $TO(q_i, q_{i+1})$,

the reset relation is obeyed: for all timer variables $x \in V$:

$$\tau_i(x) = \begin{cases} \tau_{i-1}(x) + \delta_i & \text{if } i > 0 \text{ and } (q_{i-1}, q_i, x) \notin R, \\ \delta_i & \text{otherwise, (i.e. } i = 0 \text{ or } (q_{i-1}, q_i, x) \in R). \end{cases}$$

Thus all timers progress at the same rate.

a run r is *fair* if the set of states that occurs infinitely often in r satisfies *fairness constraints*. For the purpose of this paper it suffices to say that if no fairness constraints are given, then all runs are fair. For more details the reader is referred to [ABB+94]. A run is *accepting* if it initialized, and *fair*.

The *language* of a TALE is the set of all pairs $((\sigma_0 \sigma_1 \ldots), (\delta_0, \delta_1, \ldots))$ for which there exists a run $q_0 q_1 \ldots$ such that $q_0 q_1 \ldots$ is an accepting run of $\sigma_0 \sigma_1 \ldots$ and $\delta_0, \delta_1, \ldots$ is a consistent timing of $q_0 q_1 \ldots$. It is possible to define a composition relation on TALE's such that the language of the

composition is equal to the intersection of the languages of the components. We think of the language as all possible behaviors of the system. The task of formal verification is to determine whether all these behaviors are acceptable, i.e. whether the language of the system is contained in some language $\mathcal{L}_p$ defining acceptable behaviors. We restrict ourself to the case where the *complement* of $\mathcal{L}_p$ can be described as a language of some TALE P. The formal verification problem then reduces to deciding whether the language of the composition of the system and P is empty. This problem is undecidable for TALE's in general. Verification algorithms in section 6 solve this problem for two different subclasses of TALE's: timed and alternating RQ automata.

We say that a TALE $(\Sigma, Q, I, TR, V, TO, R)$ is an *alternating RQ automaton* (ARQA) if for each timer $x \in V$ there is only one reset edge of x and along any initialized fair path:

- between any two occurrences of the reset edge of x there must be at least one query edge of x,
- a query edge of x cannot precede the fist occurrence of the reset edge of x,
- if a query edge of x appears infinitely often so must also the reset edge of x.

A TALE is a *timed automaton* (TA) if for every edge (q, q') and every timing constraint $\phi \in TO(q, q')$:

- ϕ is either $x - y \sim k$ or $x \sim k$, where x and y are timer variables, k is an integer, and $\sim \in \{<, \leq, =, \geq, >\}$.

For example, in the railroad crossing example in Figure 2 [ACD+92], the system has three components: the train, the gate, and the controller [ACD+92]. The train approaches from outside of the crossing. After at least two time units of approaching, the train will enter the crossing, and then exit at most five time units from the beginning of the cycle.

Exactly one time unit after the train approaches, the controller commands the gate to lower, and with a delay of at most one time unit the gate will close. Similarly, within one time unit after the train exits the crossing, the controller commands the gate to raise, and with a delay of at least one and at most two time units the gate will open. For simplicity, we only consider the case when there is ample time between trains. Therefore, we require the train to approach only if the gate is up. Two properties to be verified are:

safety: the gate is *down* whenever the train is *in*, and

liveness: the gate is never down for more than seven time units.

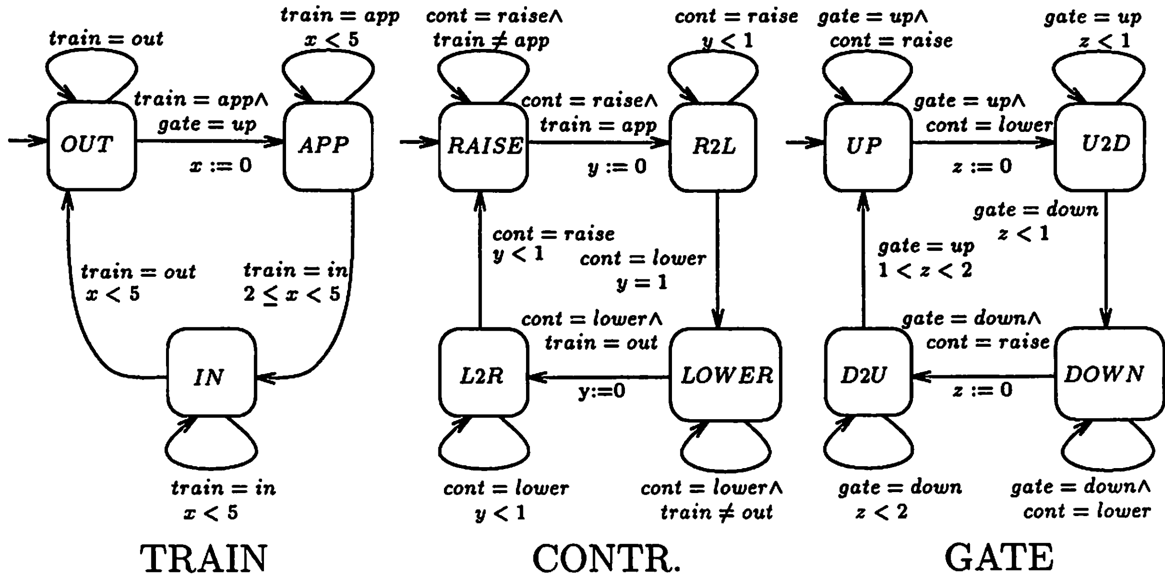

Figure 2: Railroad crossing example.

3 BLIF_MVT Intermediate Format

BLIF_MVT is a timed extension of BLIF_MV, an interchange format used to describe (untimed) automata in HSIS. A BLIF_MV description combines three primitives: variables, tables and latches.

A *variable* (declared by `.mv command`) takes values from some given finite domain. A *table* (declared by `.names`) is a relation defined over a set of variables. The variables of a table are partitioned into inputs and outputs. A *latch* (declared by `.latch`) is a pair of variables: one input one output, both of which must be defined over the same domain. An initial value can also be assigned to latches with the `.r` command. We require that every variable is an output of some unique table or latch.

Intuitively, BLIF_MV descriptions are much like synchronous hardware, with variables corresponding to wires, and tables corresponding to pieces of combinational logic, (and latches corresponding of course to latches). The language associated with a BLIF_MV description is then the set of all possible executions of the hardware. There can be more than one execution because a table can specify more than one output value for the same input (nondeterministic tables).

Formally, we interpret a BLIF_MV description as an automaton as follows:

- the set of states Q is the Cartesian product of the domains of all latch outputs,
- the set of initial states I contains all states consistent with all `.r` commands,

- the set of I/O values Σ is the Cartesian product of the domains of all the variables in the system, including latch inputs and outputs,
- the transition relation TR contains all triplets $(q, \sigma, r) \in Q \times \Sigma \times Q$ such that:
  1. assigning σ to variables in the system satisfies the relations defined by *all* the tables,
  2. the part of σ corresponding to latch outputs is equal to q,
  3. the part of σ corresponding to latch inputs is equal to r.

Tables consist of lines. The relation defined by a table is the union of the relations defined by the lines. If assigning q to latch outputs and r to latch inputs can satisfy (with some assignment to the other variables) a relation defined by a line, we say that a line *covers* an edge (q, r).

BLIF_MVT is an extension of BLIF_MV that allows a declaration of timer variables (by `.timers` command) and a labeling of lines in tables with sentences generated by the following grammar:

```
<label>  ::=  | <label> && <timer> = 0
              | <label> && <lhs> <op> <const>
              | ε
  <lhs>  ::=  | <term> + <lhs>
              | <term>
 <term>  ::=  | <const>*<timer>
              | <timer>
```

where <timer> is a timer variable, <const> is a real constant, and <op> is one of `<`, `>`, `<=`, `>=`, `==`, or `!=`.

We interpret a BLIF_MVT description as a TALE as follows:

- Q, Σ and TR are determined as above.
- the set of timer variables V contains all variables declared by `.timers` command,
- for all states q, r: a linear inequality ϕ is in $TO(q, r)$ if an edge (q, r) is covered by some line labeled with ϕ,
- for all states q, r and all timers x: $(q, r, x) \in R$ if an edge (q, r) is covered by some line labeled with `x=0`.

Figure 3 shows a BLIF_MVT description of the train automaton from Figure 2.

```
.model train
.inputs  gate
.outputs train

.mv ps,ns 3 OUT APP IN
.mv train 3 out app in
.mv gate  2 up down
.timers x

.latch ns ps
.r ps
   out

.names  ps  gate  -> ns  train
        OUT -        OUT out
        OUT up       APP app && x=0
        APP -        APP app && x<5
        APP -        IN  in  && x<5 && x>=2
        IN  -        OUT in  && x<5
        IN  -        OUT out && x<5
.end
```

Figure 3: A BLIF_MVT description of the train automaton from Figure 2.

4 System Specification with Verilog

For system specification we define a TALE semantics for a subset of Verilog. We have developed vl2mvt, a compiler from that subset to BLIF_MVT. The subset that vl2mvt supports (referred to as vl2mvt-Verilog) includes the subsets supported by popular synthesis tools.

Conceptually, a design in vl2mvt-Verilog consists of a set of modules. All modules run in parallel and communicate through a set of channels (essentially sets of wire variables declared in the modules that these channels belong to). It is assumed that communication through channels takes no time. Within each module, values on channels can be accessed through a set of ports. Ports can be either wires or registers. Through wire (register) ports a module can input/output from/to channels instantaneously. A wire port has no storage element associated with it, while a register has one.

A module consists of a set of *declarations*, *module instantiations*, *continuous assignments* (statements beginning with the key word `assign`), and *procedural blocks* (sequential blocks, sometimes referred to as `always` statements). Continuous assignments are executed in every step. They can be thought of as combinational circuits continuously computing some function of their inputs. Statements within a procedural block are executed sequentially. Module instances, continuous assignments, and procedural blocks within a module run concurrently. Execution of each continuous

```
⋮
always
    begin
    if (ctrl_state != lower) @(lowerE);
    #(1)    gate_state = down;
    if (ctrl_state != raise) @(raiseE);
    #(1:2)        gate_state = up;
    end
⋮
```

Figure 4: Verilog description of a railroad crossing gate gate.

assignment, basic block in a procedural block, and module instance is assumed to be atomic within each instant.

Caution should be taken when there is more than one procedural block in the same module since Verilog simulators treat each statement as atomic instead of each basic block (which may consist of several sub-statements). If there is more than one procedural block in a module and the simulated result depends on how expressions from different blocks are interleaved by the simulator, then the simulated result may not be reproducible by the generated automata.

Delayed assignments (e.g. `# 7 a=b-4`) and event-controls (e.g. `@(go) a=5`) synchronize the transition of `always` statements. The vl2mvt-Verilog language allows delays/event-controls to appear anywhere inside `always` statements, but it requires that every such statement blocks the execution. For example, `#2 a <= b;` is allowed. It says that program execution is halted and the assignment `a <= b;` will be executed after two time units, and then execution restarts from that statement. However, a statement like `a <= #2 b;` is forbidden. It says that, without blocking execution, `a` is scheduled to get the value of `b` two time units later. Thus, in the statement `always @(trigger) a <= #2 b;`, the number of pending assignments (and thus the number of required timers) depends on the number of `trigger` events in a given time interval, which may be unbounded.

For example, Figure 4 shows a portion of the Verilog code describing the gate module from Figure 2.

5 Specifying Timing Properties with TESLA

Verilog is a high-level language for describing system implementations. For verification, there is also a need for an abstract, natural language for describing properties about a system. Currently property specification is rather mathematical, and hence not well suited for designers. We introduce the TESLA language for describing properties and constraints of systems at a level which is natural for designers and useful in formal verification.

5.1 The TESLA Language

TESLA statements are composed of a set of sequencing constraints. A constraint is a specification of the form "*every* **request** *is eventually followed by an* **acknowledge**". However, instead of referring simply to atomic events, we constrain sequences. Specifically, a trigger sequence is related to its completion sequence as an abstraction of a 'request/acknowledge' exchange. Sequences are recursively defined, so a sequence may be composed of subsequences in several ways (e.g. concatenation or conjunction). An assertion also implies a fairness condition; just as in the atomic case we imply that we cannot transmit a request without receiving an acknowledge, every trigger sequence completion must be followed by a complete completion sequence.

The general form of a TESLA statement is:

```
{assert | restrict} <trigger-sequence> => <sequence>;
```

With `assert` constraints, we specify properties we expect the system to adhere to; a verification tool must report a failure if these are violated. With `restrict` constraints, we specify restrictions on the behavior of the system to be verified; a verification tool must not consider any behavior violating them.

A sequence is defined by:

```
<sequence>  ::=  | <ap>                         | eventually <ap>
                 | <ap> until  <ap>             | not <sequence>
                 | <sequence> <op> <sequence>   | <sequence>[<lb>, <ub>]
```

where op is `then`, and, `or`, or `xor`; `lb` and `ub` are two integers satisfying $0 < \texttt{lb} \leq \texttt{ub}$; and an *atomic proposition* ap is an arbitrary predicate on the state and I/O variables.

Atomic propositions combined with the `eventually` and `until` operators form the primitive sequences of the language. All sequences can be composed via concatenation (`then` operator), conjunction (and), disjunction (`or`), mutual exclusion (`xor`), and negation (`not`). In addition, any sequence or subsequence can be time-constrained relative to its start.

For example, in the TESLA constraint:

```
assert trig => (eventually b [4,10] then c then eventually d[6,15]) [3, 20];
```

trigger sequence `trig` is followed by event `b` within 4 to 10 units from trigger, which is then followed immediately by event `c`, which is then followed by event `d` within 6 to 15 time units from the time of event `c`. The entire sequence must complete (i.e. event `d` must be received) within 3 to 20 time units from the completion of `trig`.

The TESLA compiler accepts Verilog descriptions which include TESLA assertions and constraints embedded in Verilog comments. For each sequence, TESLA produces a vl2mvt-Verilog description of an automaton with *reset* and *enable* inputs and *success* and *fail* outputs. In addition, for each assertion, the TESLA compiler produces a fairness constraint in PIF indicating that every successful completion of a trigger sequence must eventually (in a finite but unbounded time) be followed by a successful completion of an acknowledge sequence. To constrain a sequence in time, we introduce a timing primitive which is implemented as a library element available in the vl2mvt compiler. This element is both a TA and an ARQA, so TESLA is compatible with both HSIS verification algorithms.

6 Verification Algorithms

In general, it is undecidable whether the language of a TALE is empty. However, algorithms are available for two special cases: timed automata and alternating RQ automata.

The decision procedure for language emptiness for timed automata was developed by Alur and Dill [AD90]. They show how to construct a finite-state automaton called the *region automaton* that accepts only timing consistent sequences. Unfortunately the number of states of the region automaton grows exponentially not only with the number of timers but also in the relative sizes of constants (normalized to integers) appearing in timing constraints.

The verification of alternating RQ automata is based on the following simple-path properties:

- a state is reachable from an initial state if and only if it is reachable through a simple path (one with no loops),
- a fair cycle is traversable infinitely often if and only if it is traversable once.

Using this property, Lam and Brayton [LB94] have proved that the verification of ARQA can be reduced to the verification of untimed automata, but in the worst case exponential blow-up may

occur because all simple paths may have to be enumerated.

6.1 Iterative Algorithms

To speed up the verification of timed automata and ARQA, an iterative procedure is used in HSIS [BSV94, Wu94]. The key idea is to introduce timing constraints only if needed and when they are needed. In this way, the verification process is made as similar to the untimed verification process as possible, and hence we may be able to avoid exponential blow up. A sketch of the procedure follows:

INITIALLY: Ignore all timing constraints.

VERIFY: If the language of the current abstraction of the system is empty, then return *PASS*, otherwise proceed with the next step.

ANALYZE: If some accepting run in the current abstraction of the system contains no violations of timing constraints, then *FAIL*, otherwise proceed with the next step.

MODIFY: Compose the current abstraction of the system with some small abstraction of timing constraints, sufficient to eliminate the reported failure. Proceed with the **VERIFY** phase.

In the case of timed automata, **ANALYZE** reduces to checking whether there exists a negative weighted cycle in the graph. In the **MODIFY** phase the current abstraction of the system is composed with some abstraction of the region automaton. These abstractions are all of a certain type. It is possible to show that there can be only finitely many abstractions of that type, so the procedure will terminate.

In the case of alternating RQ automata, **ANALYZE** reduces to checking satisfiability of a set of linear inequalities. In the **MODIFY** phase the current abstraction of the system is composed with an automaton that blocks all infinite paths related via loops to some simple path on which the associated linear inequalities are not satisfiable. Since there are only finitely many simple paths, the procedure will terminate.

In both cases, there can be exponentially many iterations in the worst case, but in our experience that is rare. Also, since the algorithms build different abstractions it is possible that each is more efficient for a certain class of real-time systems. Defining criteria to recognize these classes is an interesting open problem.

6.2 Hints

Usually, the designer knows which timing constraints are critical for a particular property. Not relaxing these constraints initially can dramatically reduce the number of iterations. Following our methodology, we provide in HSIS an option to the user of hinting which timing constraints *not* to ignore initially. This may increase the size of the initial abstraction, so one must be careful not to list too many constraints. Ideally, the user would list exactly those constraints that are necessary to prove the property at hand. In that case, the algorithm would terminate in a single iteration.

In general, every hint has two parts. The first consists of a lower bound on some timer and a set of transitions which are enabled only if that bound is satisfied. Similarly, the second part consists of an upper bound on some timer and a set of transitions which are enabled only if that bound is satisfied.

In describing hints, we use the expression $(A : a, b)$ to denote the transition of the automaton A from state a to state b. similarly, we use $(A : a, *)$ to denote all transitions of A from state a.

For example, consider the safety property in the railroad crossing. It is satisfied because:

- the train will enter the crossing at least two time units after it approaches, and
- the gate will be *down* in less than two time units after the train approaches (one time unit for the controller to issue the command, and less than one unit for the gate to close after that),

This reasoning can be converted into three hints. The first hint for the safety property of the railroad crossing indicates that $(TRAIN : APP, IN)$ implies $x \geq 2$, and that $(GATE : U2D, *)$ implies $z < 1$. Given such hints, HSIS first checks that indeed in the original description the transitions $(TRAIN : APP, IN)$ and $(GATE : U2D, *)$ are possible only if $x \geq 2$ and $z < 1$ are satisfied. If that were not the case, the hint would be ignored.

The constraints $x \geq 2$ and $z < 1$ cannot be satisfied simultaneously if $x - z \leq 1$. Therefore, HSIS automatically generates a two-state automaton with one state corresponding to values of x and z satisfying $x - z > 1$, and the other corresponding to the values satisfying $x - z \leq 1$. We refer to this automaton as H_1. The automaton H_1 monitors the rest of the system to check whether x and z are reset, and chooses the next state accordingly. For example, if both x and z are reset, the next state must be $x - z \leq 1$.

Finally, HSIS adds a constraint to H_1 disallowing the train to move from APP to IN if the gate is in the $U2D$ state and H_1 is in the $x - z \leq 1$ state. The resulting automaton is shown as H_1 in Figure 5.

The second hint indicates that $(H_1 : x - z > 1, *)$ implies $x > 1$, and that $(CONT : R2L, *)$ implies $y < 1$. Again, after checking the validity of the hint, HSIS generates a two state automaton (say H_2) with states $x - y > 0$ and $x - y \leq 0$, and adds to the transition relation a constraint not allowing the controller to be in the $R2L$ state if H_1 is in $x - z > 1$ and H_2 is in $x - y \leq 0$.

Finally, the third hint indicates that $(TRAIN : APP, IN)$ implies $x \geq 1$, and that $(CONT : R2L, *)$ implies $y < 1$. To enforce this hint HSIS does not have to generate any automaton, because an automaton with states $x - y > 0$ and $x - y \leq 0$ has already been generated (namely H_2). HSIS only needs to add a constraint to H_2 that disallows the train to move from APP to IN, if the controller is in the $R2L$ state, and H_2 is in the $x - y \leq 0$ state. The resulting automaton is shown in Figure 5, where R_x, R_y, and R_z respectively denote the transitions on which timers x, y, and z are reset, and C_1, C_2 and C_3 respectively denote transitions disabled by the first, the second, and the third hint above. We invite the reader to check that in the composition of the automata in Figure 5 and Figure 2 (with timing constraints relaxed), the gate is always *down* whenever the train is *in*.

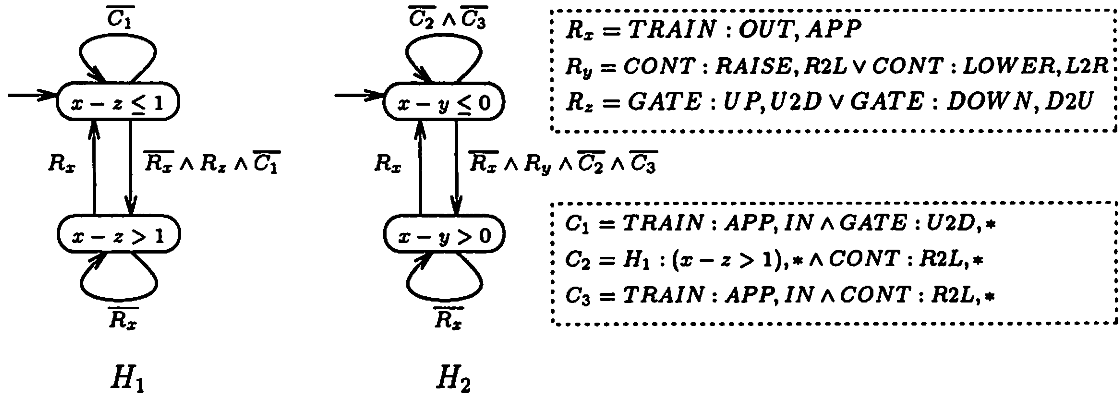

Figure 5: Automata resulting from hints for the railroad crossing.

The mechanics of hints is directly taken from the **MODIFY** phase of the TA verification algorithm. Therefore, hints can be used to eliminate all potential timing violations, and ensure that the verification terminates in one iteration. But usually, this is too big a burden on the designer, because generating the right hints can be time consuming and non-intuitive. In our experience, it is most productive to generate hints interactively, i.e. the designer takes over the **ANALYZE** phase of the verification algorithm. Usually, the designer is a much better analyzer than HSIS, because HSIS cannot distinguish between timing violations that are easily avoided from those that really restrict the behavior of the system. Typically, in the interactive mode only a few iterations are needed to

Table 1: Results for the timed automata algorithm.

	no hints			with hints			
example	iter.	reach. st.	time	hints	iter.	reach. st.	time
csma	5	120	$1.5s$	1	1	48	$0.5s$
fddi-l	2	15	$0.5s$	1	1	15	$0.3s$
fddi-s	38	95	$39.0s$	6	1	29	$0.4s$
fis3	14	826	$26.4s$	6	1	830	$0.8s$
fis4	29	42,998	$1,141.8s$	12	1	44,545	6.7
fis5	space out			20	1	$6 * 10^6$	$168.4s$
belt	space out			5	1	299	$0.8s$
patho	space out			4	1	$3 * 10^6$	$16.5s$
fact	space out			58	3	$8 * 10^7$	$84.7s$
cross-s	4	16	$0.6s$	3	1	11	$0.3s$
cross-l	13	78	$3.6s$	11	1	17	$0.4s$

generate exact hints, while in the direct mode even tens of iterations may not be enough to verify the system.

7 Experimental Results

Experimental results for the timed automata verification algorithm are summarized in Table 1. All experiments were performed on a DEC workstation with 440Mb of physical memory. Most of the examples can be roughly divided into communication protocols: CSMA/CD, FDDI, and Fischer's (denoted in Table 1 with prefix fis), and control examples: railroad crossing (cross), seat-belt alarm (belt), and automated factory (fact). The exception is a model of the PATHO real-time operating system (path) [BPSVV94]. The reported numbers of reachable states corresponds to the abstraction of the system in the last iteration. Reported execution times include the time to read the BLIF_MVT description, the file describing a property to be verified, the time to process hints (if any), and the verification time. Typically, the total time not spent in verification grows much slower than the verification times, which dominate the larger examples. All the examples and properties in Table 1 were written directly in BLIF_MVT and PIF.

The value of hints is obvious from Table 1. Without them, only the smallest examples can be verified. This suggests that better failure analysis techniques are needed. The automated factory examples illustrates how automatic verification can complement hints. After several tries, we were able to develop a set of hints that enforces most of the timing constraints necessary to verify the property. Automatic verification then filled-in the remaining gaps. The user effort in developing the hints varied from example to example, but generating 2–3 hints per hour seems a good rule of

Table 2: Comparison of different methods.

example	BLIF_MVT	VL2MVT	ARQA
fis3	26.4s		115.8s
cross-s	0.6s	16.8s	7.7s
cross-l	3.6s	24.6s	4s

thumb.

In Table 2 we compare verification results obtained by applying timed automata algorithm to the description written directly in BLIF_MVT (column BLIF_MVT) with the results obtained by applying the same algorithm to the vl2mvt generated description (column VL2MVT), and the results obtained by the ARQA algorithm. For the same algorithm, hand-generated BLIF_MVT code is approximately an order of magnitude more efficient than code generated by vl2mvt. We are working on some compiler optimizations that should narrow the gap, but it is unlikely that it would ever be completely eliminated. Also, the timed automata algorithm appears to be more efficient than the ARQA algorithm, but the current ARQA implementation is too preliminary and the set of common examples is too small to make definitive conclusions yet.

8 Conclusions and Future Work

We have proposed a methodology for formal verification of timing properties that can be integrated in a design flow based on successive refinement. To support our methodology we have extended the HSIS verification system with timing capabilities and developed interfaces to Verilog (for systems verification) and TESLA (for property specification). Compared to the existing systems mentioned in the introduction, HSIS offers a system-specification language that is widely used and supported by synthesis and simulation, and a more natural property-specification language. HSIS is also the only system that offers a user-guided mode in addition to an automatic verification mode. This may not be important from a theoretic point of view, because it does not contribute to the worthy goal of fully automatic verification, but from a practical point of view the ability to trade off user involvement and capacity of the tool is certainly appealing. Finally, HSIS is the only system that offers state of the art techniques for both timed and untimed verification problems.

Experimentally, the performance of KRONOS is similar to that of HSIS in the automatic mode, but both can verify significantly smaller systems than HSIS in the user-guided mode. Experimental results are unfortunately not available for RT-COSPAN and EPSILON, but since in both cases the verification algorithms are similar to that of KRONOS, it is plausible to expect comparable results.

Experimental results show that further improvement in efficiency are necessary before the methodology is widely accepted. One avenue to explore is a better understanding for which class of systems ARQA or timed automata algorithms are more efficient. Also, identifying essential timing constraints and retaining them in the initial abstraction is shown to speed up the verification process significantly, but presently there are no automatic techniques for doing it. Similarly, failure analysis techniques that can distinguish essential from non-essential timing violations need to be developed.

References

[ABB+94] A. Aziz, F. Balarin, R. K. Brayton, S.-T. Cheng, R. Hojati, S. C. Krishnan, R. K. Ranjan, A. L. Sangiovanni-Vincentelli, T. R. Shiple, V. Singhal, S. Tasiran, and H.-Y. Wang. HSIS: A BDD-based environment for formal verification. In *Proceedings of the 31th ACM/IEEE Design Automation Conference*, 1994.

[ACD+92] Rajeev Alur, Costas Courcoubetis, David L. Dill, Nicholas Halbwachs, and Howard Wong-Toi. An implementation of three algorithms for timing verification based on automata emptiness. In *Proceedings of IEEE Real-time Systems Symposium*, 1992.

[AD90] Rajeev Alur and David L. Dill. Automata for modelling real-time systems. In M.S. Paterson, editor, *ICALP'90 Automata, languages, and programming: 17th international colloquium*. Springer-Verlag, 1990. LNCS vol. 443.

[BPSVV94] Felice Balarin, Karl Petty, Alberto L. Sangiovanni-Vincentelli, and Pravin Varaiya. Formal verification of the PATHO real-time operating system. In *Proceedings of 33rd Conference on Decision and Control, CDC'94*, December 1994.

[BSV94] Felice Balarin and Alberto L. Sangiovanni-Vincentelli. Iterative algorithms for formal verification of embedded real-time systems. In *Digest of Technical Papers of the 1994 IEEE International Conference on CAD*, November 1994.

[CDCT93] Costas Courcoubetis, David L. Dill, M. Chatzaki, and Panagiotis Tzounakis. Verification with real-time COSPAN. In G. v. Bochmann and D.K. Probst, editors, *Proceedings of Computer Aided Verification : 4th International Workshop, CAV '92, Montreal, Canada, June 29-July 1, 1992*. Springer-Verlag, 1993. LNCS vol. 663.

[ČGL93] K. Čerāns, J. Godskesen, and K. Larsen. Timed modal specification - theory and tools. In Costas Courcoubetis, editor, *Computer Aided Verification: 5th International*

Conference, CAV'93, Elounda, Greece, June/July 1993, Proceedings, pages 253–267. Springer-Verlag, 1993. LNCS vol. 697.

[DDHY92] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol Verification as a Hardware Design Aid. In *Proceedings of ICCD*, pages 522–525, October 1992.

[Dil89] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite-State Systems*. Springer-Verlag, 1989. LNCS vol. 407.

[Kur90] R. P. Kurshan. Analysis of discrete event coordination. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems : Models, Formalisms, Correctness*, pages 414–453. Springer-Verlag, 1990. LNCS vol. 430.

[LB93] William K. C. Lam and Robert K. Brayton. Alternating RQ timed automata. In Costas Courcoubetis, editor, *Computer Aided Verification: 5th International Conference, CAV'93, Elounda, Greece, June/July 1993, Proceedings*, pages 237–252. Springer-Verlag, 1993. LNCS vol. 697.

[LB94] William K. C. Lam and Robert K. Brayton. Criteria for the simple path property in timed automata. In David L. Dill, editor, *Computer Aided Verification: 6th International Conference, CAV'94, Stanford, June 1994, Proceedings*, pages 27–40. Springer-Verlag, 1994.

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

[NSY92] Xavier Nicolin, Jospeh Sifakis, and Sergio Yovine. Compiling real-time specifications into extended automata. *IEEE TSE Special Issue on Real-Time Systems*, September 1992.

[RdS92] Valerie Roy and Robert de Simone. Auto/autograph. *Formal Methods in System Design: An International Journal*, 1(2/3):239–250, 1992.

[Wu94] Ephrem Wu. Formal timing verification with simple-path timed automata. Master's thesis, University of California, Berkeley, 1994.