A Methodology for Formal Verification of Real-Time Systems

1 Introduction

Presently, simulation is a dominant approach to design verification. But, as the complexity of design increases, simulation is able to cover a decreasing portion of a system's functionality, at an increasing cost of computer resources. Therefore, a significant interest is developing in formal verification techniques to supplement simulation. Many believe that in order to be successful, formal verification must be integrated in the design flow. This leads to the following methodology:

- The initial design is described at a high level of abstraction. Initial simulations are performed and some properties are formally verified.
- The description of the model is iteratively refined until it reaches the level where it can be implemented.
- At every level of abstraction more simulations are performed, and properties specific to that level are verified. Also, if the refinement is not guaranteed to preserve the properties from the higher level, they must be re-checked.

Traditionally, functionality and timing were verified separately. This is insufficient for real-time systems, i.e. systems that must meet strict timing constraints. For example, it may be perfectly acceptable that a car control unit adjusts the temperature in the cabin within ten seconds, but but the ignition timing have to be adjusted much faster. Thus, timing properties must be an integral part of system descriptions from the highest level of abstraction.

1.1 Our Approach

The abstract methodology described above is widely accepted, but to make it practical several choices have to be made regarding methods to specify systems and properties, as well verification algorithms and formalisms. We describe our choices and the requirements that guided us in making them.

Requirement 1 *It should be possible to describe real-time systems at a high level of abstraction, and then efficiently verify them.*

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, and then simple longest path techniques can be used to verify the system. On 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. The problem is that making real time a part of the system's state makes the state space infinite and verification undecidable in general. Two notable exceptions are timed automata [Dil89, AD90] and alternating RQ automata [LB93, LB94]. In both cases, a real-time system is modeled as a combination of a finite-state system and real-valued time measuring devices called *timers*. Also in both cases, the authors have shown that under certain constraints the verification of real-time systems can be reduced to the verification of some finite-state system, and thus is decidable. However, the constraints are different, so both models can describe some systems that the other cannot. Therefore, we define a new model called TALE (Timed Automata with Linear inEqualities), which includes both timed and alternating RQ automata as special cases, and base our methodology on this formal model.

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, the verification tool supporting the methodology should take advantage of any such guidelines.

Requirement 2 *Formal verification should be integrated in the design flow, but it should not limit the choice of synthesis and simulation tools.*

If the verification is done after the designed is done, its value is diminished. If the verification is done in parallel but not integrated with the design, it is likely that inconsistency between verification and design descriptions of the system will arise. Therefore, formal verification must 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 3 *Properties to be verified must be expressed in a way 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 the system's specification.

Requirement 4 *A methodology must be supported by tools.*

To support our methodology we have:

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

1.2 Related Work

Most of the recent developments in formal verification of real-time systems stem from the Alur's and Dill's work on timed automata [Dil89, AD90] and related work on model checking [ACD90]. They reduce the problem to the verification of a finite-state system called a *region automaton*. A region automaton is finite but typically too large to verify with existing resources. Thus, several verification procedures that try to avoid building the full region automaton have been developed [HNSY92, ACD+92, BSV93, AIKY93]. In HSIS we provide iterative algorithms based on [BSV94] and [LB93], but our methodology is not limited to these choices.

Despite these advances, the verification of real-time system remains a hard problem. Thus, it is not surprising that only a few such systems are available. We mention real-time extensions of COSPAN [CDHT93] and EPSILON [ČGL93] all based on automata approaches and KRONOS [NSY92] based on model checking. None of these systems supports all requirements we have placed on the methodology. For example COSPAN has a fixed input language that is not used in synthesis and simulation, KRONOS has an intermediate format, but there were no attempts to interpret any simulation languages in it and EPSILON lacks a simple way of specifying properties to be verified.

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 $\sum_{i=1}^{n} \mathrm{a}_i x_i \sim \mathrm{a}_{n+1}$ where $\mathrm{a}_1, \ldots, \mathrm{a}_{n+1}$ are real numbers, and $\sim$ is $<$, $\leq$, $=$, $\geq$, 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 $\sum_{i=1}^{n} \mathrm{a}_i x_i \sim \mathrm{a}_{n+1}$ if $\sum_{i=1}^{n} \mathrm{a}_i \tau(x_i) \sim \mathrm{a}_{n+1}$.

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.

We say that 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 is *accepting* if it initialized, and *fair*. 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 all runs are fair. For more details the reader is referred to [ABB+94].

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 $q_0q_1 \ldots$ such that $q_0q_1 \ldots$ is an accepting run of $\sigma_0\sigma_1 \ldots$ and $\delta_0, \delta_1, \ldots$ is a consistent timing of $q_0q_1 \ldots$. 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 in Figure 1 which is a slight modification of the example in [ACD+92], the system has three components: the train, the gate, and the controller. 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.

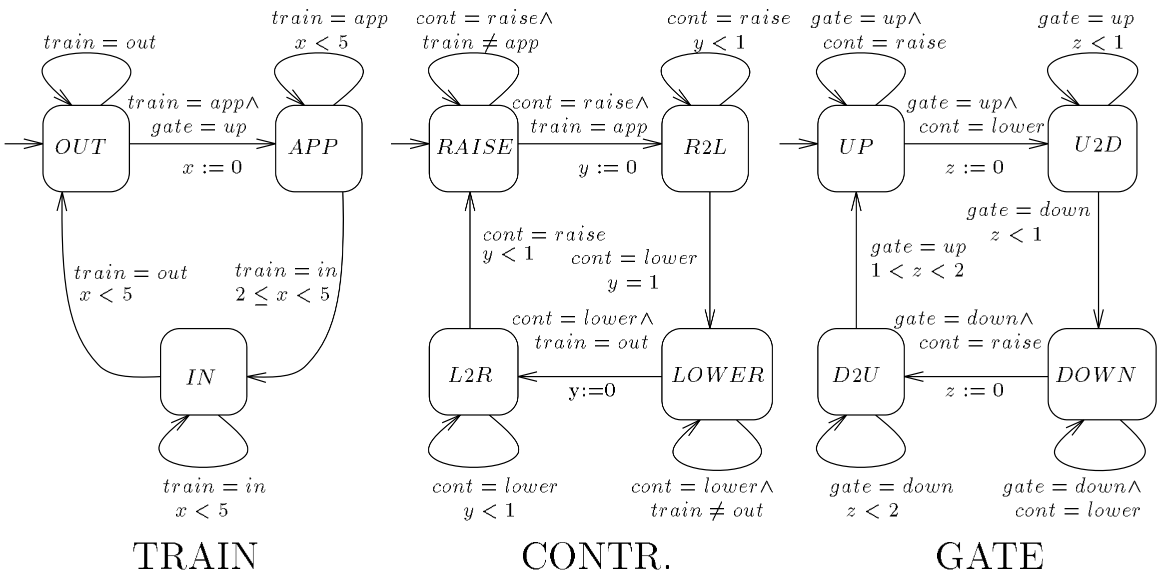

Figure 1: Railroad crossing example.

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, at most 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.

Formally, the common set of I/O values of TALE's in Figure 1 is $\{app, in, out\} \times \{lower, raise\} \times \{up, down\}$ where $\{app, in, out\}$ is the domain of the variable $train$, indicating whether the train is approaching, in, or out of the crossing, $\{lower, raise\}$ is the domain of the variable $cont$, indicating whether the controller instructs the gate to close or open, and $\{up, down\}$ is the domain of the variable $gate$, indicating whether the gate is up or down.

It is easy to check that all automata in Figure 1 are TA's. With a few changes they can also be made ARQA's. First, the references to the timer y (respectively z) on edges incident to the state $L2R$ ($D2U$) are renamed to some new timer y_1 (z_1). This change does not affect the language. The more serious problem is that the automata allow runs on which query edges appear infinitely often, and reset edges appear only finitely many times. However, in this example all such runs are terminated with a self-loop that has an enabling constraint of the form $t < k$. Such runs are possible only if time converges, i.e. an infinite number of events happens in finite time. Many models of real-time exclude these runs by definition. Our model does not, but the user always has an option to use fairness constraints to exclude such runs; then, the automata in Figure 1 are ARQA's.

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 shows automata which accept all timed sequences of I/O values that *violate* these properties. A property is satisfied if the composition of the corresponding automaton with the automata in Figure 1 has the empty language.

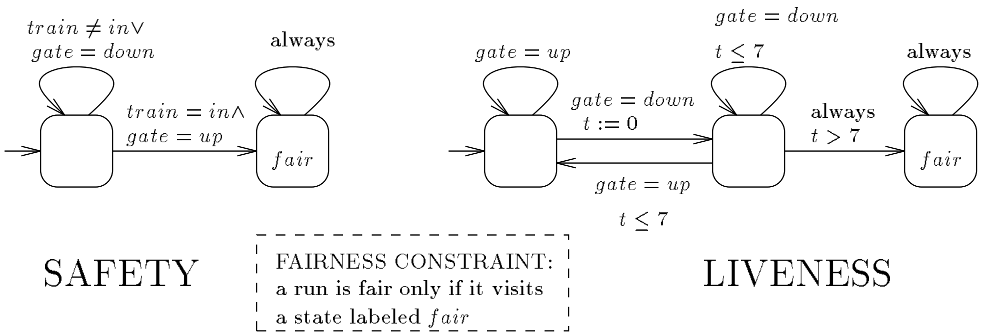

Figure 2: Properties for the railroad crossing example.

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 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, a BLIF_MV description is much like a 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 (or inputs),
- 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 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 relations defined by the lines. If assigning q to latch outputs and r to latch inputs can satisfy (with some assignment to 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 linear inequalities over timer variables (written in a simple C-like language), and expressions of the form `x=0`, where `x` is some timer variable. 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 1.

```
.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 1.

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 (register) port has no (one) storage element associated with it.

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 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 simulator treats 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 seven time unit interval, which may be unbounded.

4.1 System Modeling

Execution of a vl2mvt-Verilog program consists of a sequence of two alternating phases, *computation* and *idling*. The purpose of the computation phase is to compute the new hardware state while halting the progression of time. In the computation phase, next states of `reg` variables are computed by executing "active statements" – the segments of program that will be executed between the current pause and the next one. This is done in the following way. First, continuous assignments calculate their outputs according to their operands until a stable value is reached. Then procedural blocks compute the next states for `reg` variables that are touched by the active statements during the execution.

In TALE interpretation all activities in the computation phase are collapsed into a single state transition. If delayed assignment is encountered in the computation phase, the so-called timing machines are activated. A generic timing machine is shown in Figure 4. It waits in the initial state until it receives the *activate* signal, and then in no more than *up* and no less than *low* time units it responds with the signal *done*. Each delayed assignment requires one timing machine.

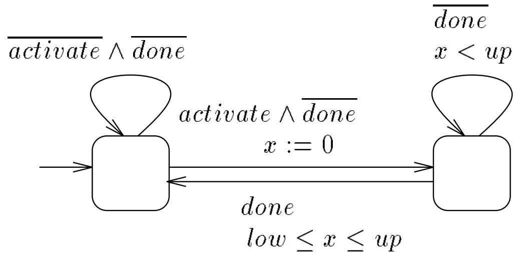

Figure 4: A generic timing machine.

When a procedural block reaches a pause statement, it enters the idling phase. Here, it remains in the same state until the occurrence of the event it is waiting for. Loosely speaking, hardware status progression and time progression are alternated and not overlapped.

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.

To specify timing constraints, designers often use waveform diagrams. These are useful because they contain both relative sequencing of signal transitions, as well as relative timing constraints which bound their occurrence. Often natural language descriptions of machine interfaces include timed sequences as part of a case-by-case description: *"if we have condition A followed by condition B, then we should eventually obtain condition C within time window X"*.

Timed automata allow us to constrain timing between arbitrary edge transitions of a state machine. While these constraints are fairly general, entering timing constraints in this form is tedious; there is a need for combining the description of both timing and event sequencing using a high level of abstraction. Specifically, we propose including timing constraints within property specification for formal verification by focusing on the subset of properties described by sequences with time bounds.

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. consecution 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 acknowledge 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 `lb < 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`.

5.2 Compiling TESLA

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 the HSIS Property Interchange Format (PIF) indicating that every successful completion of a trigger sequence must eventually be followed by a successful completion of an acknowledge sequence.

We can think of the automaton accepting the language of an assertion as a composition of two automata, one recognizing the language of the trigger sequence and the other recognizing the language of the completion sequence. Our language links sequencing primitives or, recursively, subsequences, to form complete sequences. Thus a natural translation of a sequence is into recursively linked automata whose composition forms a machine recognizing the sequence language. We carry this decomposition down to the sequence primitive level, defining an automaton for each primitive, and compositions for every other operator. This dramatically simplifies translation of sequences into automata as the translation follows the recursive syntactic structure of the sequence.

To constrain a sequence in time, we introduce a timing primitive similar to the timed automaton in Figure 4. The primitive monitors the *enable* input and the *success* output of an automaton associated with some sequence, and accepts it only if *success* follows *enable* within a given time bound. The timing primitive 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.

Each constraint of our language translates into a timed automaton which accepts this trigger/completion language. In verification, the automata associated with `restrict` constraints are added to the system to be verified to restrict its behavior. This is useful for system descriptions with a high degree of non-determinism. For `assert` constraints, the language of the system to be verified must be contained in the languages of all the automata associated with `assert` constraints. HSIS can be used to perform this task.

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,
- a fair cycle is traversable infinitely often if and only if it is traversable once.

Figure 5 shows an automaton which *does not* have a simple path property. State s_2 is reachable from s_0 only if the self-loop $s_1 \to s_1$ is traversed exactly 4 times.

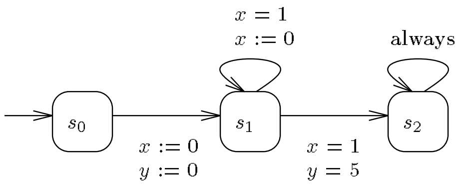

Figure 5: An automaton that is not ARQA.

Lam and Brayton have proved in [LB94] that if X_π is the set of all not traversable ω-simple paths[1] in an alternating RQ automaton A, and A^u is A with timing constraints ignored, then there is a traversable ω path in A if and only if there is an ω simple path in $A^u - X_\pi$.

Thus, 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 ω-simple paths may have to be enumerated. However, the simple path property allows the following: once a simple path π is proved to be not traversable, then all paths related to π by adding loops can be eliminated. In effect, an infinite number of paths can be eliminated at one time.

Our methodology does not limit the choice of verification algorithms. Next section describes algorithms currently implemented in HSIS. Details of the algorithms appear elsewhere. Here we focus on how user guidelines can be used to improve the efficiency.

[1] An infinite path is ω-simple if it consists of a simple path from an initial state to a simple cycle.

6.1 Iterative Algorithms

To possibly avoid an exponential blow-up, the following iterative strategy is used in both TA and ARQA language emptiness algorithms in HSIS:

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. For more details refer to [BSV94].

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 ω-simple path on which the associated linear inequalities are not satisfiable. Since there are only finitely many ω-simple paths, the procedure will terminate. More details are provided in [LB93, LB94].

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.

The hints described in this section were developed for the verification of timed automata, i.e. they describe an abstraction of the region automaton. In the future, we plan to develop hints for the ARQA verification algorithm, i.e. hints that describe untraversable ω-simple paths. If the system to be verified is both a TA and an ARQA, then these hints can be freely mixed to form an starting point for either of the verification algorithms.

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. Given a hint, HSIS first validates it. A hint is valid only if it is implied by the timing constraints in the original description.

After validating a hint, HSIS generates an automaton (say H) with one state (say "good") in which the two bounds can be satisfied simultaneously, and the other (say "bad") in which that is not the case. Finally, HSIS adds a restriction that the two sets of transitions mentioned in the hint can not occur simultaneously if H is in the "bad" state. The set of transitions mentioned in the hint can either be transitions of the original system, or transitions in the automata generated by other hints.

To see how hints work 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. 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.

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 $z < 1$. When given such hints, HSIS first checks that the state transitions mentioned above are really possible only if the corresponding timing constraints are satisfied. For example, to validate the hint above, HSIS needs to check that $x \geq 2$ is implied by some constraint in $TO(APP, IN)$, and that on every transition from the $U2D$ state there exists a constraint $z < 1$.

The constraints $x \geq 2$ and $z < 1$ cannot be satisfied simultaneously if $x - z \leq 1$. Therefore, HSIS 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 in Figure 6.

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 6, 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 to check that in the composition of automata in Figure 6 and Figure 1 (with timing constraints relaxed), the gate is always *down* whenever the train is *in*.

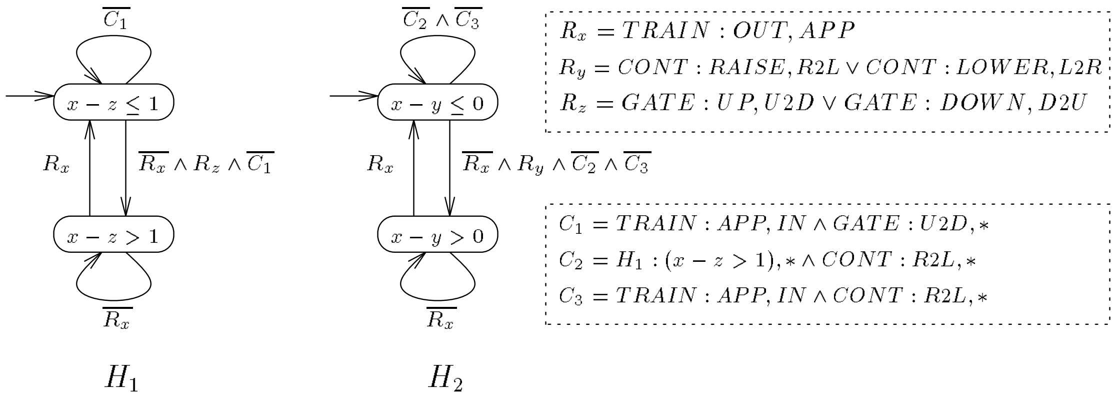

Figure 6: Automata resulting from hints for the railroad crossing.

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	$1.5s$	not avail.			
fddi-l	2	15	$0.5s$	not avail.			
fddi-s	38	95	$39.0s$	not avail.			
fis3	14	826	26.4	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	$0.8s$
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	$0.4s$

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 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. 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 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.

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.

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.

[ACD90] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking for real-time systems. In *Proceedings of 5th LICS*, pages 414–425, June 1990.

[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.

[AIKY93] Rajeev Alur, Alon Itai, R. P. Kurshan, and M. Yannakakis. Timing verification by successive approximation. 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.

[BSV93] Felice Balarin and Alberto L. Sangiovanni-Vincentelli. A verification strategy for timing constrained systems. 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*, pages 151–63. Springer-Verlag, 1993. LNCS vol. 663.

[BSV94] Felice Balarin and Alberto L. Sangiovanni-Vincentelli. An iterative approach to verification of real-time systems. *Formal Methods in System Design: An International Journal*, 1994. to be published.

[CDHT93] Costas Courcoubetis, David L. Dill, M. Hatzaki, 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.

[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.

[HNSY92] Thomas A. Henzinger, Xavier Nicolin, Jospeh Sifakis, and Sergio Yovine. Symbolic model-checking for real-time systems. In *Proceedings of 7th Symposium on Logics in Computer Science*. IEEE Computer Society Press, 1992.

[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 *Computer Aided Verification: 6th International Conference, CAV'94, Stanford, June 1994, Proceedings*. Springer-Verlag, 1994.

[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.