Timing Analysis in COSPAN

Rajeev Alur and Robert P. Kurshan

AT&T Bell Laboratories, Murray Hill

Abstract. We describe how to model and verify real-time systems using the formal verification tool Cospan. The verifier supports automata-theoretic verification of coordinating processes with timing constraints. We discuss different heuristics, and our experiences with the tool for certain benchmark problems appearing in the verification literature.

1 Introduction

Model checking is a method of automatically verifying concurrent systems in which a finite-state model of a system is compared with a correctness requirement. This method has been shown to be very effective in detecting errors in high-level designs, and has been implemented in various tools. We consider the tool Cospan that is based on the theory of ω-automata (ω-automata are finite automata accepting infinite sequences, see [Tho90] for a survey, and [VW86, Kur94] for applications to verification). The system to be verified is modeled as a collection of coordinating processes described in the language S/R [Kur94]. The semantics of such a model M is the ω-language $L(M)$ corresponding to the infinite executions of the model. The property to be checked is described as another process T whose acceptance conditions classify the executions of M in "good" and "bad," and all the good behaviors are removed from the language. The model M satisfies the property T if the language of the product of M and T is empty. The language-emptiness test can be performed algorithmically. The verifier supports a variety of alternatives for performing this test including on-the-fly enumerative search and symbolic search using binary decision diagrams (BDDs) (see [McM93] for applications of BDDs to verification). In this paper, we describe the extension of Cospan to model and verify real-time systems.

In recent years, model-checking methods based on finite-state machines have been extended to real-time and hybrid systems [AD94, ACD93, HNSY94, ACH+95]. In S/R, real-time constraints are expressed by associating lower and upper bounds on the time spent by a process in a local state. An execution is timing-consistent if its steps can be assigned real-valued time-stamps that satisfy all the specified bounds. The semantics of a timed S/R model M with a table B of bounds is, then, the set $L(M, B)$ of its timing-consistent executions. The timing verification problem corresponds to checking emptiness of the language $L(M \otimes T, B)$ for a suitably chosen process T. A variety of correctness requirements such as invariants, absense of deadlocks, liveness, and bounded response, can be modeled in this way. The expressiveness of timed S/R is the same as that of timed automata [AD94]. While we do not support a direct modeling of hybrid systems, verification of hybrid systems can sometimes be reduced to, or approximated by, verification of timed automata (see, for instance, [PV95]).

For checking emptiness of the language $L(M, B)$, the verifier automatically constructs another automaton A_B, also as a S/R process, which when composed with the original model, rules out behaviors that do not satisfy the timing constraints: $L(M \otimes A_B)$ equals $L(M, B)$. The existence of such a finite-state constraining automaton A_B follows from the so-called region construction for timed automata [AD94]. Our implementation augments the region construction with a variety of heuristics. The states of the constraining automaton correspond to sets of values for timers, and may be either regions or convex unions of regions called zones. Zones can be defined by difference-bound-matrices [BD91, Dil89], and are used in many other tools for timing verification such as KRONOS [DOY94], VERITI [Won94], ORBITS [Rok93], and UPPAAL [LPY95]. Once the definition of the constraining automaton has been generated, the original COSPAN verifier can be used as a black-box, and the actual search can be done either enumeratively or symbolically using BDDs. All these choices are coupled with an iterative solution which involves generating successive approximations to the constraining automaton (the theory underlying this approximation method is described in an earlier paper [AIKY95], which also reports an earlier implementation of timing verification in COSPAN. The current implementation with its combination of heuristics is typically one or two orders of magnitude faster than the one reported in [AIKY95]). Furthermore, the underlying continuous semantics of time can be approximated, in a conservative way, by the integers, and this also provides a heuristic simplification of the timing analysis (approximation of the dense semantics by the fictitious-clock semantics has been studied earlier, for instance, in [Bur89, HMP92]).

We describe our experiences with the tool for certain benchmark problems appearing in the verification literature. These include timing-based distributed protocols, embedded controllers, and timed circuits. The iterative option is sometimes effective in synthesizing requirements on the parameter values for a timed system to satisfy its specification. The search using zones is less sensitive to the constants in the bounds, and can analyze complex timing constraints. On the other hand, symbolic search using BDDs and regions can sometimes handle state-spaces that are a few orders of magnitudes larger.

2 Modeling

We give a brief and informal introduction to the relevant features of the S/R modeling language (see [Kur94] for a detailed and formal description).

2.1 Selection-Resolution Model

An S/R model M consists of a collection of coordinating components called *processes*. Each process has two variables: a *state* variable and a *selection* (or output) variable. We use $P.\$$ to denote the state variable of P and $P.\#$ to denote the selection variable of P. A state of P is a value of its state variable, and a selection of P is a value of its selection variable. A *global state* consists of states of all the processes and a *global selection* consists of selections of all the processes.

The declaration of a process P contains the type declaration of $P.\$$ and $P.\#$, a set of possible initial values for $P.\$$, and update commands for $P.\$$ and $P.\#$ specifying the transition relation. The update command for $P.\#$ specifies the set of possible selections as a function of the state of P and selections of some other processes. The update command for $P.\$$ specifies the set of possible next states as a function of the current state of P and the current global selection.

The execution of an S/R model proceeds in rounds. In the round 0, initial values of all the state variables are chosen. This is followed by an alternation of *selection* and *resolution* phases. Consider a round i such that the global state for the round i is determined. The selection of each process for round i is chosen in the selection phase. The selection of P in round i may depend on, apart from its state, selections of other processes in round i. The dependencies among the selection variables are required to be acyclic, and the S/R compiler determines a total order for the update of selection variables. In the following resolution phase, the global state for round $i+1$ is determined. The state of P in round $i+1$ depends upon its state in round i and the global selection for round i. We assume that the model is *complete*, that is, there is at least one choice for the new global state, and the execution can continue forever.

Thus, an execution of the model M produces an infinite *chain* r of the form $s_0 \xrightarrow{\sigma_0} s_1 \xrightarrow{\sigma_1} s_2 \xrightarrow{\sigma_2} \cdots$ of global states s_i and global selections σ_i. The acceptance conditions are used to rule out uninteresting runs. The S/R model allows two types of acceptance conditions, namely, *cycle-sets* and *recur-edges* [1]. If a set C of states of P is declared to be a cyset, then the chain r is removed (or excepted) if the state of P eventually always stays inside C. A local transition of P is a pair of its states. If a set E of local transitions of P is declared to be a *recur-condition*, then the chain r is removed if some transition in E repeats infinitely often. With the model M, we associate the set $L(M)$ consisting of all the chains that are not excepted by any of the cyset or recur declarations.

As an example, consider Figure 1 that describes a simple model of a train communicating with a gate. Each state is labeled with the set of selections possible from that state, and edges are labeled with predicates over selections. Observe that when ST selects *enter*, its state is updated to *near*, and simultaneously, the state of SG is updated to *closed*. The cyset specifies that any infinite chain in which ST stays forever in the state *inside* or *near* is excepted.

2.2 Asynchrony

While the S/R model is synchronous, asynchrony can be modeled using nondeterminism. Intuitively, whenever an asynchronous process changes its state, it waits a nondeterministic number of rounds before proceeding. Every asynchronous process P has an additional state variable called *the pause-bit*, denoted

[1] Processes with these two types of acceptance conditions are called L-processes, and are studied in [Kur94].

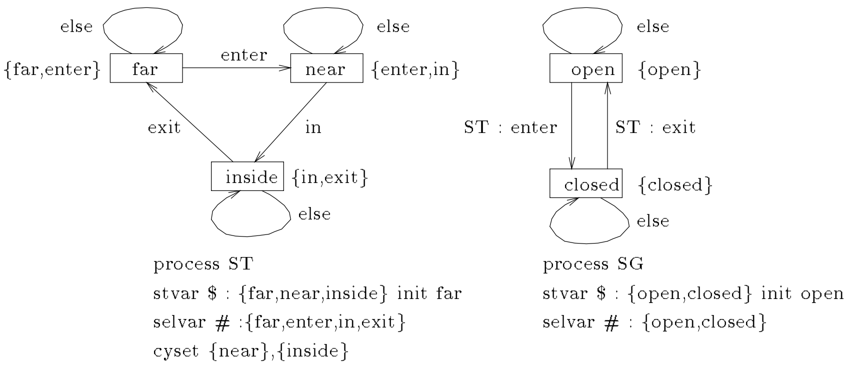

Fig. 1. Synchronous train-gate

P.pbit. The state of P is described by the value of both $P.\$$ and $P.pbit$. The process P is said to be *pausing* when its *pbit* is set to true. For a value v of $P.\$$, a reference to $v@$ indicates that the value of $P.\$$ is v and $P.pbit$ is true. The selection of P depends only on its state. For every value v of $P.\$$, a unique selection is specified corresponding to the case when the *pbit* is true, and a set of selections is specified corresponding to the case when the *pbit* is false. Whenever the value of $P.\$$ changes, the *pbit* is set to true. Every pausing state $v@$ is required to have a self-loop corresponding to the intuition that the pausing states capture delays. There is an implicit nondeterministic transition that changes the *pbit* to false.

The asynchronous mode of S/R is best understood by an example. Figure 2 shows the modified model of the train and the gate. The asynchronous process AG has 4 states: $AG.\$$ can be *closed* or *open*, and $AG.pbit$ may be true or false. When $AG.\$$ is *open* its selection is *up* if the *pbit* is set, and *open* otherwise. The selection *wait* models the delay in opening the gate. Initially, $AG.\$$ is *open* with its *pbit* false. When the selection of AT is *enter*, $AG.\$$ changes to *closed*, and the *pbit* is set to true. The state of AG stays *closed@* for a nondeterministic number of rounds, and then the *pbit* changes to false nondeterministically. Now the selection is *closed*, and the process awaits AT to select *exit*. The cyset *closed@* models the assumption that the *pbit* is unset after a finite number of rounds. The asynchronous process AT is interpreted similarly.

A global state s of an asynchronous model M is said to be *stable* if it is possible that every asynchronous process does not change its state in the next resolution phase. For instance, a global state with the *pbit*s of all the asynchronous processes set to true, is stable. In Figure 2, if the *pbit* of AT is false, then the state is unstable. In the S/R semantics for asynchronous processes, time is spent only in stable states, and unstable states are assumed to be instantaneous. Along a chain in $L(M)$, it is required that every instantaneous state is followed by a stable state. If the transition structure forces two instantaneous states to follow each other, a stability failure is reported.

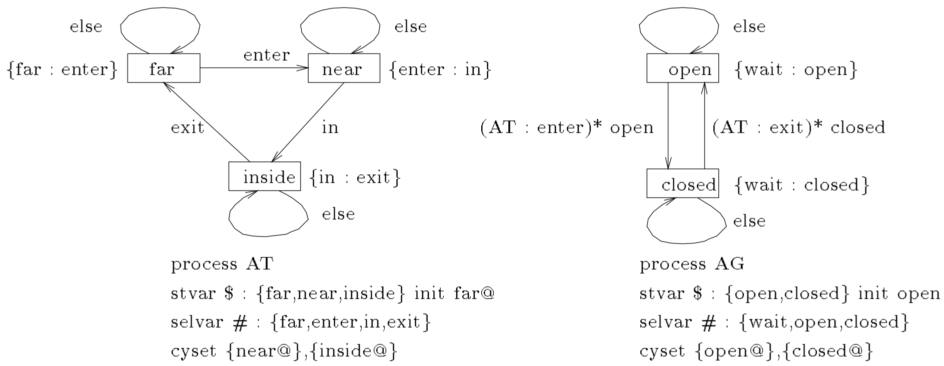

Fig. 2. Asynchronous train-gate

2.3 Timing

Timing is added to an asynchronous model M by specifying bounds on the time spent in the pausing states of individual processes. A *bounds-table* B for M consists of two functions lb and ub: the function lb maps each value v of $P.\$$, for an asynchronous process P, to a nonnegative integer, and the function ub maps each value v of $P.\$$ to a nonnegative integer or to ∞. The two functions lb and ub specify the lower and the upper bounds, respectively, on the time spent in each pausing state. A sample bounds-table for the asynchronous model of Figure 2 is shown in Figure 3. A *timed* S/R model consists of an asynchronous S/R model together with a bounds-table.

Consider a chain r consisting of global states s_i and global selections σ_i. Consider an asynchronous process P. If $P.pbit$ is set in state s_0, then P begins a pause at step 0. Consider $i > 0$.

- The process P *begins* a pause at step i if (1) the state of P in s_i is a pausing state v@, and (2) the state of P in s_{i-1} is not v@.
- The process P *ends* a pause at step i if (1) $P.\$$ is the same in both s_{i-1} and s_i, (2) $P.pbit$ is true in s_{i-1}, and (3) $P.pbit$ is false in s_i.
- The process P aborts a pause at step i if (1) the state of P in s_{i-1} is a pausing state v@, and (2) $P.\$$ is not v in state s_i.

The step i of the chain r is instantaneous iff some process ends a pause at step i and some process begins a pause at step $i+1$. The asynchronous semantics of S/R ensures that if the step i is instantaneous, then the step $i+1$ is not instantaneous. A *timing assignment* for the chain r is a sequence $t_0, t_1, \ldots$ of real-valued time-stamps such that for every step i, if the step i is instantaneous then $t_{i+1} = t_i$ else $t_{i+1} > t_i$.

The timing assignment $t_0, t_1, \ldots$ for the chain r of an asynchronous model M is said to be *consistent* with the bounds-table B if it satisfies the constraints imposed by B:

State Variable	Value	Lower bound	Upper Bound
AT.$	*far*	10	∞
AT.$	*near*	5	6
AT.$	*inside*	1	1
AG.$	*open*	1	2
AG.$	*closed*	1	2

Fig. 3. A sample bounds-table

- The duration of each pause does not exceed its upper bound: Suppose a process P begins a pause at step i with state $v@$. Then for every index $j > i$ such that P has not ended or aborted the pause at all steps $i < k < j$, the difference $t_j - t_i$ is at most $ub(v)$.
- Each pause can end only after the lower bound has elapsed: Suppose a process P begins a pause at step i with state $v@$, and ends the pause at step j without an end or abort at all steps $i < k < j$. Then the difference $t_j - t_i$ is at least $lb(v)$.

The chain r of an asynchronous model M is said to be *timing consistent* with the bounds-table B if it has a timing assignment consistent with B. The set of all chains of M that are timing consistent with B is the language of the timed model, and is denoted $L(M, B)$. Observe that $L(M, B)$ is a subset of $L(M)$.

Timed automata [AD94] offer an alternative way of introducing time in finite-state machines. It is straightforward to show that a timed S/R model can be translated to a timed automaton that uses one clock variable for every asynchronous process of the S/R model. Conversely, every timed automaton with k clocks can be translated to an S/R model with one process synchronously communicating with k asynchronous timed processes. Thus, timed automata and timed S/R are expressively equivalent formalisms to model real-time systems.

3 Verification

The verifier Cospan checks whether the language of an S/R model is empty. Given a model M, if the language $L(M)$ is nonempty, Cospan outputs a chain in $L(M)$, called the *error-track*. The error-track is a periodic chain that consists of a prefix followed by a cycle – called the *post-martem track*:

$$s_0 \xrightarrow{\sigma_0} s_1 \xrightarrow{\sigma_1} s_2 \xrightarrow{\sigma_2} \cdots s_m \xrightarrow{\sigma_m} (u_0 \xrightarrow{\nu_0} u_1 \xrightarrow{\nu_1} u_2 \xrightarrow{\nu_2} \cdots u_k \xrightarrow{\nu_k} u_0)^\omega$$

Since all processes are finite-state, the language $L(M)$ is ω-regular, and whenever the language $L(M)$ is nonempty, an error-track of the above form exists.

The test for emptiness is performed by searching through the reachable state-space of the model. Cospan allows two types of search: an on-the-fly enumerative search, and a symbolic BDD-based search.

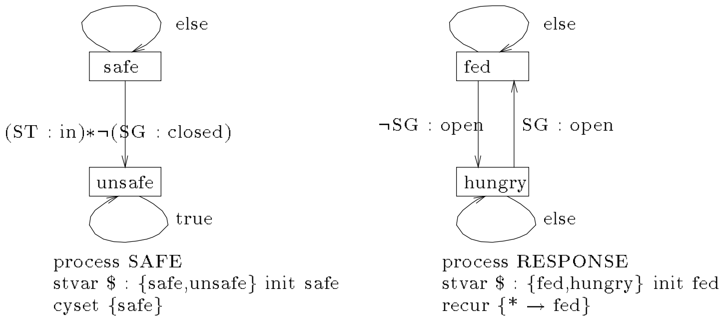

Fig. 4. Properties of train-gate

3.1 Checking Correctness

To check a property of an S/R model M, we add another process T, called *task*, to the model. The update of $T.\$$ depends upon the global selection of M. The behavior of the processes in the model M does not depend upon the task process, but the transition relation of the task process depends upon the behavior of M. The acceptance conditions of T specify which chains should be removed from the language. Formally, adding the task process T to the model M corresponds to the product operation (i.e. language intersection), and the restricted model is denoted $M \otimes T$. We have the language inclusion property: $L(M \otimes T) \subseteq L(M)$. The task is written so that the chains in $L(M \otimes T)$ correspond to undesirable behaviors. The model M satisfies the task T if the language $L(M \otimes T)$ is empty.

As an example, consider the tasks of Figure 4. The process SAFE specifies the requirement that the gate should be closed when the train is inside the gate. If the gate is not selecting *closed* when the train selects *in*, the state of SAFE is updated to *unsafe*. In the chains where this condition never holds, SAFE.$ is always *safe*, and is removed from the language. Given the model $ST \otimes SG$ and the task SAFE, the verifier correctly concludes that the resulting language is empty. The process SAFE can be used as a specification for the asynchronous train-gate model also (in Figure 4, the references to ST and SG need to replaced with AT and AG, respectively).

The task RESPONSE specifies the progress property that requires that the selection of the gate is *open* infinitely often (this implies that a closed gate eventually opens). The acceptance condition is specified using *recur* declaration; every chain that contains infinitely many transitions to the state *fed* is removed from the language.

3.2 Timing Verification

Recall that a timed model consisting of an asynchronous model M and a bounds-table B, specifies the language $L(M, B)$. As before, the correctness can

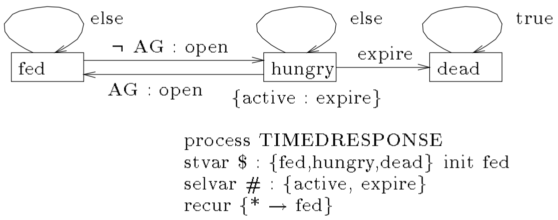

Fig. 5. Real-Time Property

be specified by adding a task process T such that the chains in $L(M \otimes T, B)$ specify the undesirable behaviors. In particular, if we use the asynchronous train-gate model of Figure 2, the bounds-table of Figure 3, and the task SAFE of Figure 4, the resulting language is empty.

The task itself can be an asynchronous process, and lower and upper bounds on the states of the task can be used to check timing properties of the model. For instance, to verify that the gate always selects *open* within 10 units, use the task TIMEDRESPONSE (Figure 5) with the entry

TIMEDRESPONSE.$ hungry 10 10

added to the bounds-table of Figure 3. This is a typical real-time property requiring response within a bounded period.

Thus, the timing verification problem corresponds to checking emptiness of the timing-consistent language: given a model M and a bounds-table B, we need to check whether $L(M, B)$ is empty. Our solution is to construct another S/R process A_B, called the *constraint automaton*, such that $L(M \otimes A_B)$ equals $L(M, B)$. The S/R process A_B monitors the model M, and removes all the chains that are not timing consistent with B.

Let us consider the construction of the process A_B. The process A_B needs to ensure that the order in which the processes begin, end, and abort pauses is timing consistent. For a start, let us assume that A_B has, for every asynchronous process P, a real-valued state variable $P.timer$. Initially, $P.timer$ is 0. In every resolution phase, the process A_B chooses a real-valued increment δ denoting the time spent in the current global state. The value δ needs to satisfy the following constraints: (1) δ equals 0 if the current global state is instantaneous, and (2) for every asynchronous process P in a pausing state $v@$, adding δ to the value of $P.timer$ does not exceed the upper bound $ub(v)$. After choosing δ, the value of $P.timer$ is incremented by δ for every pausing process P. If P is ending a pause at state v, then the updated value of $P.timer$ should be at least $lb(v)$, if not, the resulting chain is removed from the language. Whenever P aborts its pause, or ends its pause, $P.timer$ is reset to 0.

This scheme involves real-valued state variables. If there are n asynchronous processes, then the state of A_B is a point in the n-dimensional space. Let c be the largest constant appearing in the bounds-table. Then, it suffices to consider only those points all of whose coordinates lie within $[0, c]$, this gives a bounded

n-dimensional cube. To obtain a finite-state process, such points need to be clustered together. The current implementation in COSPAN supports the following two clustering constructions:

- *Region Automaton* R_B: Two points in the cube belong to the same *region* iff they agree on the ordering of the fractional parts of the timer values and the integer parts of the timer values. The number of regions is finite, and the partitioning of the space in regions gives the so-called *time-abstract bisimulation*. Details about the theory of regions can be found in [AD94]. The states of the region automaton R_B correspond to these regions. Each region is described by n variables ranging over $0..c$ representing the integer parts of the timer values and n variables ranging over $0..n$ that represent ordering of the fractional parts.
- *Zone Automaton* Z_B*:* The states of the zone automaton correspond to convex unions of regions. Each such zone is described by a set of linear inequalities; each inequality gives a bound on a timer value or a bound on the difference between values of two timers. Such a set of linear inequalities is best represented by the so-called *difference-bounds-matrix* [Dil89]. The zone automaton Z_B has $n(n+1)$ integer variables ranging over $-c..c$ representing the differences, and has $n(n+1)$ boolean variables that represent whether the corresponding bound gives a strict or a nonstrict inequality.

Both Z_B and R_B are constraint automata, that is, $L(M, B) = L(M \otimes R_B) = L(M \otimes Z_B)$. It should be noted that, while the number of states of a constraint automaton grows exponentially with the bounds-table, the size of its definition is polynomial in the size of B. The actual states of the constraint automaton are constructed only on-the-fly, in conjunction with the exploration of M.

3.3 Approximations

Since checking emptiness of $L(M, B)$ is computationally expensive —PSPACE-complete—, we support the following two approximations.

Successive Approximations
The state-spaces of R_B or Z_B depend on the number of timed asynchronous processes and the constants in the bounds-table. Furthermore, not all the constraints in B may be relevant to proving the property. This suggests that the constraints should be added to B in an incremental fashion. In the successive approximation scheme, we wish to construct a sequence of approximations $B_0, B_1 \ldots$ such that

approximation: for each i, $L(M, B_i)$ is a superset of $L(M, B)$,
monotonicity: For each i, $L(M, B_{i+1})$ is a subset of $L(M, B_i)$, and
convergence: $L(M, B_i)$ converges to $L(M, B)$ as i increases.

The scheme is useful only if the constraint automata for the approximations B_i are simpler than the constraint automaton for B. To construct these approximations, observe that for two bounds-tables B and B', (1) if there exists a constant c such that each bound in B is c times the corresponding bound in B', then $L(M, B)$ and $L(M, B')$ are identical, and (2) if each lower bound in B is at least the corresponding lower bound in B', and each upper bound in B is at most the corresponding upper bound in B', then $L(M, B')$ is a superset of $L(M, B)$. This suggests that we can simplify a bounds-table by decreasing lower bounds, increasing upper bounds, and dividing by common factors.

As a first approximation B_0, our algorithm sets all lower bounds to 0 and all upper bounds to ∞. Thus, $L(M, B_0)$ equals $L(M)$. If $L(M, B_0)$ is empty then so is $L(M, B)$. If not, an error-track r is reported. If r is in $L(M, B)$ then it is reported as an error-track, else the next approximation B_1 is constructed so that r is inconsistent with B_1 (this ensures that B_1 is more restrictive than B_0). To construct B_1, the original B is simplified as much as possible preserving the inconsistency of r. This is done in a heuristic manner as described in [AIKY95]. Alternative schemes of approximations for timing verification are studied in [Won94] and [BS92].

Fictitious-Clock Approximation

The semantics of asynchronous S/R models requires that nonzero amount of time is spent in stable states. Let us relax this assumption: a *fictitious-clock timing assignment* for a chain r of an asynchronous model M is a sequence $t_0, t_1, \ldots$ of real-valued time-stamps such that $t_i \leq t_{i+1}$. Note that every timing assignment for r is also a fictitious-clock timing assignment. The chain r is said to be *fictitious-clock-timing-consistent* with the bounds-table B if it has a fictitious-clock timing assignment consistent with B, and let $L^f(M, B)$ denote the set of all such chains. The following two facts hold:

1. the set $L^f(M, B)$ is a superset of $L(M, B)$ and a subset of $L(M)$, and
2. if a chain r is fictitious-clock-timing-consistent then there is a fictitious-clock timing assignment $t_0, t_1, \ldots$ consistent with B such that all t_i's are integers.

The first fact says that proving emptiness of $L^f(M, B)$ implies emptiness of $L(M, B)$, and the second fact says that to check fictitious-clock-timing-consistency, it suffices to consider only integer valued time-stamps. The integer-valued time-stamps can be viewed as readings of a fictitious global discrete clock that is incremented at regular intervals; all events between successive increments of this clock get the same time-stamp (such models are studied in [AH94, HMP92]).

This gives the following scheme to check emptiness of $L(M, B)$. The first step is to check if $L^f(M, B)$ is empty. If so then so is $L(M, B)$, else let r be a periodic error-track in $L^f(M, B)$. If r is in $L(M, B)$ then r can be reported as an error-track, else check if $L(M, B)$ is empty by generating R_B or Z_B. Let us consider checking emptiness of $L^f(M, B)$. We construct an S/R process A^f_B such that $L(M \otimes A^f_B)$ equals $L^f(M, B)$. Since we can assume all time-stamps are integer valued, the region automaton R_B can be simplified to R^f_B. A state of R^f_B is a lattice point, and the variables encoding the ordering of fractional parts are not needed. Similarly, the zone automaton Z^f_B is simpler than Z_B since all inequalities can be assumed to be nonstrict.

The above scheme also requires checking whether a periodic error track is in $L(M, B)$. An algorithm for this purpose is described in [AIKY95], and its complexity is strongly polynomial.

4 Examples

We have used COSPAN on various benchmark examples for timing verification appearing in the literature. The performance of different heuristics varies depending on the choice of the example. Following observations hold in most cases.

1. There is no clear choice between region and zone automata. Typically, zone automaton leads to a smaller number of states, and is less sensitive to the actual magnitudes of constants. The BDD representation of the region automaton is smaller than that of the zone automaton. We use zone automata with enumerative search, or region automata with BDD-based search.
2. In most benchmark examples, fictitious-clock model suffices to prove or disprove language emptiness.
3. Iterative option is useful in understanding the behavior of the system being verified by identifying the bounds that are relevant in proving the property.

Space does not permit us a detailed description of the examples, a few are listed below.

Timed mutual exclusion: Many papers on timing verification have used Fischer's protocol for mutual exclusion to illustrate their approach. As the number of processes grows, the number of reachable states grows exponentially, and effectiveness of BDDs becomes apparent. For verifying the mutual exclusion property for 10 processes using the fictitious-clock region automaton, the number of reachable states is 6×10^{10}, and BDD-based search with regions requires 12 cpu minutes and 24 MB memory (other combinations run out of memory). For the timing-based protocol for fast mutual exclusion (see [AIKY95]), the given model has 3 timed processes, each with 11 pausing states, and the untimed S/R model has 125000 reachable states. When we run our iterative scheme, termination is obtained after 13 iterations of the fictitious-clock model with zone automata (the last iteration is the slowest, and takes about a minute). Out of the original 33 pausing states, the final bounds-table specifies simplified bounds only on 15 state; indeed the simplified bounds correspond to the tightest bounds required to prove the property.

Timed Circuits: A simple application of the tool is to analyze timing diagrams, and this is potentially useful as an industrial application. An interesting example for timing verification is the Seitz timed queue element (see [Bur89] or [Rok93] for details of this example).

Robot Controller: In [PV95], a model of a simple manufacturing plant is considered, and the authors show how to construct a timed automaton from the original description involving differential inclusions. In the COSPAN-model, both enumerative search wih zones or BDD-based search using regions work effectively.

References

[ACD93] R. Alur, C. Courcoubetis, and D.L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[ACH$^+$95] R. Alur, C. Courcoubetis, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[AH94] R. Alur and T.A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1):181–204, 1994.

[AIKY95] R. Alur, A. Itai, R.P. Kurshan, and M. Yannakakis. Timing verification by successive approximation. *Information and Computation*, 118(1):142–157, 1995.

[BD91] B. Berthomieu and M. Diaz. Modeling and verification of time-dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, SE-17(3):259–273, 1991.

[BS92] F. Balarin and A. Sangiovanni-Vincentelli. A verification strategy for timing-constrained systems. In *Proceedings of the Fourth Workshop on Computer-Aided Verification*, LNCS 663, pages 151–163. Springer-Verlag, 1992.

[Bur89] J.R. Burch. Combining CTL, trace theory and timing models. In *Automatic Verification Methods for Finite State Systems: Proceedings of the First CAV*, LNCS 407, pages 197–212. Springer-Verlag, 1989.

[Dil89] D.L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, LNCS 407, pages 197–212. Springer–Verlag, 1989.

[DOY94] C. Daws, A. Olivero, and S. Yovine. Verifying ET-LOTOS programs with KRONOS. In *Formal Description Techniques VII, Proceedings of FORTE'94*, pages 227–242, 1994.

[HMP92] T.A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *ICALP 92: Automata, Languages, and Programming*, LNCS 623, pages 545–558. Springer-Verlag, 1992.

[HNSY94] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model-checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[Kur94] R.P. Kurshan. *Computer-aided Verification of Coordinating Processes: the automata-theoretic approach*. Princeton University Press, 1994.

[LPY95] K. Larsen, P. Pettersson, and W. Yi. Compositional and symbolic model-checking of real-time systems. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, 1995.

[McM93] K. McMillan. *Symbolic model checking: an approach to the state explosion problem*. Kluwer Academic Publishers, 1993.

[PV95] A. Puri and P. Varaiya. Verification of hybrid systems using abstractions. In *Hybrid Systems II*, LNCS 999. Springer-Verlag, 1995.

[Rok93] T. Rokicki. *Representing and modeling digital circuits*. PhD thesis, Stanford University, 1993.

[Tho90] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. Elsevier Science Publishers, 1990.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First IEEE Symposium on Logic in Computer Science*, pages 332–344, 1986.

[Won94] H. Wong-Toi. *Symbolic approximations for verifying real-time systems*. PhD thesis, Stanford University, 1994.