

# Model Checking Semi-Continuous Time Models Using BDDs

Sérgio Campos, Márcio Teixeira,  
Marius Minea, Edmund Clarke and Andreas Kuehlmann

bla

**Abstract.** Nao concreto

## 1 Introduction

Computers are frequently used in applications where failures can have severe consequences, such as in the control of industrial machinery or transportation equipment. In these applications, the computer system must not only produce the correct result, but must do so in timely fashion. For example, a command to apply the brakes of a car or to turn an airplane to a certain direction cannot be late otherwise an accident may occur. Such failures cannot be tolerated, making the correctness of these systems is an extremely important issue.

However, its verification is a very complex problem, made even harder because of its timing requirements. Several methods have been proposed to accomplish this task. One method that has obtained significant success is model checking [7, 8]. In this technique the system being verified is modeled as a state-transition graph and properties of the system are expressed as temporal logic formulas. The verification procedure consists of a search on the state space of the graph to determine which states satisfy the properties.

Original model checkers do not provide a direct way of describing timing characteristics. Several extensions have been proposed that allow such properties to be expressed and verified. The first and simplest is to associate to each transition the passage of one time unit and to count the number of transitions taken to determine the time elapsed between events. This technique assumes a *discrete time* model. The main advantage of this method is that it is simple and can be implemented extremely efficiently, particularly in BDD-based symbolic model checkers such as SMV [16] or Verus [4].

Another approach is to use a *continuous time* model, in which events can happen at any moment in the dense time domain, e.g. timed automata [1, 10]. Since in this case the state space is inherently infinite, model checking entails constructing a finite equivalent model, the complexity of which can be quite high. These models, as well as the verification algorithms are considerably more complex than in the discrete time case. Initial tools were unable to handle models with more than hundreds or thousands of states. Current tools are significantly more efficient [11, 15, 18] but the verification of timed automata is still much more expensive than the verification of discrete time models.

However, discrete time models have one major disadvantage over continuous time models: their limitation in expressing the semantics of event sequences that happen in short periods of time. For example if the occurrence of an event  $a$  triggers an alarm  $b$  and an immediate response  $c$  we can model these events as happening simultaneously or taking at least two time units to occur. This may not correspond to reality, however. It may be the case that after event  $a$  has occurred but before alarm  $b$  another event  $d$  occurs that would change response  $c$ . But if  $a$ ,  $b$  and  $c$  happen at the same time this possibility would not be present. On the other hand, if it takes 1 time unit between  $a$  and  $b$  it would not be possible for  $d$  to occur *between*  $a$  and  $b$ . For this reason discrete time models cannot be used in some applications where accuracy is essential.

The method proposed overcomes this problem using *zero-length* transitions to model the occurrences of events without time passing. The passage of time then occurs in discrete steps using unit-length transitions. The advantage of this new model is that it removes the limitation on event orderings for the discrete time model. For example, it is now possible to let events  $a, b, c$  and  $d$  described previously occur in time zero preserving their order, and only let time elapse after all events have occurred. We argue that this enables the verification of many systems that have been previously thought to require dense time models for verification.

In order to determine the time between events in the semi-continuous time model we use quantitative timing analysis as described in [6, 3, 4]. Of particular interest are the *condition counting* algorithms that count the minimum and maximum number of occurrences of a specific event in a given set of intervals. In this work these algorithms are used to count the minimum and maximum number of unit transitions on paths of interest, computing the time elapsed between events. We propose new condition counting algorithms that are significantly more efficient than the previous one. These algorithms allow verification to be done as efficiently as for the simple discrete time case. It is very similar to the fixpoint computation used in model checking for untimed systems, and as such can be implemented efficiently using BDDs.

To demonstrate the expressive power and efficiency of the method we have verified two examples of systems in which high accuracy is necessary to achieve the correct results. The first is the steam boiler example described in [14]. This example, while small, demonstrates that the proposed model can be used to verify systems which are not usually considered in the realms of discrete time verification. We have then verified an automotive engine controller developed for Magneti-Marelli that has been previously verified by HyTech [17]. We have modeled the controller that identifies that the driver has released the accelerator and regulates the reduction of fuel injection. This identification is a complex time critical function of the position of several sensors. If the timing of the events that take place during its execution is wrong, the algorithm may not converge and the controller can malfunction. We have been able to model both examples in Verus and to verify their correctness, demonstrating the effectiveness of the proposed method.

## 2 Related Work

A precursor to the presented analysis method has been developed in the real-time model checker Verus [4]. This tool implements *quantitative timing analysis* algorithms that determine the timing characteristics of a system by counting the time between events or the number of occurrences of events in given intervals. The method has been used to verify large and complex timed systems such as an aircraft controller [6], a robotics controller [5] and the PCI local bus [3]. However, the condition counting algorithms used in that context require the augmentation of the state space with an additional integer time variable which added a significant overhead to verification. The new algorithms do not require this construct and are efficiently implemented using BDDs.

The occurrence of events without the passage of time has been discussed in [9]. But that work does not consider a symbolic implementation using BDDs and is not as efficient. It also does not use quantitative analysis algorithms and is not able to generate the same type of information as the method proposed.

A significant body of research exists on continuous-time models. One of the most widely used models are the *timed automata* [1, 10], which add real-valued *clock variables* to represent time. Clocks evolve at the same rate, modeling passage of time, and formulas can refer to the value of the clocks to express timing properties. Verification is then performed on a finite-state quotient model, such as the region graph [1] or the *zone automaton* [13]. However, the expressive accuracy comes with a significant increase in complexity, and a significant effort in the development of continuous-time model checkers [18, 15] has been devoted to dealing with the state explosion problem.

The expressivity and efficiency trade-offs between discrete and continuous time raise the question when a discrete-time approximation is sufficient to model all continuous-time behaviors of a system. This problem is analyzed, e.g., in [12]. This work introduces the notion of *digitizability* and proves that such a reduction is possible for timed transition systems, for verification of properties such as time-bounded invariance and time-bounded response. More recent work [2] shows that a reduction to discrete time can be performed for acyclic combinational circuits, but not for all cyclic ones. These can only be reduced under the constraint that no strict inequality is used in its design.

## 3 Condition Counting Algorithms

Our method relies on the ability to count *some* transitions on a path but not necessarily all of them. In order to accomplish this, we will use the algorithms described in this section. The original algorithms used in our method to verify real-time systems determine the length of a path leading from a set of starting states to a set of final states [6, 3, 5]. But to verify semi-continuous time models we also need to compute the minimum and maximum number of times a given condition holds on any path from *start* to *final*. In [6] we have presented algorithms that compute this information. However, these algorithms required

an augmentation of the state space with a counter to store intermediate results. This made the algorithms very expensive in some applications. The algorithms described in this section do not suffer from this limitation.

To simplify the algorithms, we assume that any path beginning in *start* must reach a state in *final* in a finite number of steps. This requirement is necessary to ensure that the minimum (maximum) is well-defined. It can be checked using the maximum algorithm described in [6]. Finally, we assume that all computations involve only reachable states. This can be achieved by intersecting *start* with the set of reachable states computed a priori.

### Minimum Condition Counting

The minimum condition count algorithm computes the minimum number of states satisfying a given condition *cond* over all paths that start in a state in *start* and end in a state in *final*. Any paths starting in *start*, but which do not reach *final* in a finite number of steps are excluded from this computation. In particular, if no path from *start* ever reaches *final*, the algorithm will return the special value **NOPATH**.

The algorithm searches forward beginning in *start*. It looks for paths with an increasing number of occurrences of *cond*. Each iteration consists of two phases: The first is a forward traversal through states that do not satisfy *cond*. This traversal is performed until all states (not satisfying *cond*) reachable from the current frontier are found. If *final* has not been reached yet, the frontier is expanded by one step to states that satisfy *cond* and the condition counter is incremented. The algorithm iterates until *final* is found, or all reachable states are visited.

The algorithm must differentiate between states that do not satisfy *cond* and those that do, and similarly, between transitions leading to these states. We use subscripts 0 and 1 respectively for the two types of states and transitions. For example,  $start_0$  is the set of initial states that do not satisfy *cond*, and  $start_1$  is the set of initial states that satisfy *cond*:

$$start_0 = start \cap \neg cond \quad start_1 = start \cap cond$$

Furthermore, if  $N(s, s')$  is the transition relation, we denote by  $T_0(S)$  and  $T_1(S)$  the set of transitions from a state in  $S$  that lead to states not satisfying *cond* and to states satisfying *cond*, respectively:

$$T_0(S) = \{s' \mid \exists s \in S. N(s, s') \wedge s' \notin cond\}$$

$$T_1(S) = \{s' \mid \exists s \in S. N(s, s') \wedge s' \in cond\}$$

The argument about the correctness of the algorithm follows from invariants stating that  $R^i$  at the  $i^{th}$  iteration contains the set of all states that can be reached as endpoints of finite paths starting in *start* and having  $i$  or less states satisfying condition. The proof can be found in the full version of the paper.

### Maximum Condition Counting

The maximum condition count algorithm computes the maximum number of states satisfying a given condition *cond* over all paths that begin in a state in

```

proc mincount(start, cond, final)
i = 0; R =  $\emptyset$ ; R' = start0;
do
  do
    if (R'  $\cap$  final  $\neq$   $\emptyset$ ) return i;
    R = R';
    R' = T0(R')  $\cup$  R';
  while (R'  $\neq$  R);
  R' = T1(R')  $\cup$  R';
  if (i = 0) R' = R'  $\cup$  start1;
  i = i + 1;
while (R'  $\neq$  R);
return NOPATH;

proc maxcount(start, cond, final)
i = 0; R' = cond;
do
  R1 = R';
  do
    R = R';
    R' = R'  $\cup$  B0(R');
  while (R'  $\neq$  R);
  if (R'  $\cap$  start =  $\emptyset$ ) return i;
  R' = B1(R');
  i = i + 1;
while (R'  $\neq$  R1);
return  $\infty$ ;

```

Fig. 1. Minimum and maximum condition count algorithms

*start* and end in a state in *final* without previously traversing a state in *final*. If there is a path beginning in *start* that goes through *cond* infinitely often without reaching *final*, the algorithm returns infinity. The basic idea behind the algorithm is to find paths with increasing condition count whose states are all within  $\neg$ *final*. The condition count of the longest path satisfying this condition and starting in *start* is the desired maximum.

The algorithm assumes that all states are reachable from *start*. This can be enforced by performing a reachability computation from *start* and restricting the state space to reachable states. Moreover, we require that every state has at least one outgoing transition.

Similarly to the mincount algorithm, we consider transitions into states that satisfy *cond* and that do not satisfy *cond* separately. This algorithm, however, performs a backward search, and we must define the reverse image of the transition relation. In this case  $B_0(S')$  is the set of states satisfying neither *cond* nor *final* that lead to a state in  $S'$  in one step. Similarly  $B_1(S')$  is the set of states satisfying *cond* but  $\neg$ *final* that lead to a state in  $S'$  in one step. Note that *final* only appears implicitly in the algorithm, in the definitions of  $B_0$  and  $B_1$ .

$$\begin{aligned}
 B_0(S') &= \{s \mid \exists s' \in S'. N(s, s') \wedge s \notin \text{final} \wedge s \notin \text{cond}\} \\
 B_1(S') &= \{s \mid \exists s' \in S'. N(s, s') \wedge s \notin \text{final} \wedge s \in \text{cond}\}
 \end{aligned}$$

Again, we argue about the correctness of the algorithm using an invariant similar to the previous one. It states that at the  $i^{\text{th}}$  iteration  $R'$  is the set of all states that are the start of a finite path of length  $i$  which has no states in *final* (except possibly the last one), and which has  $i$  states that belong to *cond*. The proof can be found in the full version of the paper.

## 4 Semi-Continuous Time

The basic idea of the method proposed is to allow zero-length transitions to model the occurrence of events without time passing, allowing events to occur independently of the passage of time. To allow zero-length transitions we have

created a special variable  $t$  in the model of the system being verified. Time passage is controlled by enabling unit-length transitions only when  $t$  is *true*, and enabling zero-length transitions only when  $t$  is *false*.

Parallel composition of processes under the new model is defined as follows. Unit transitions have to occur synchronously, that is, all processes execute unit transitions at the same time. Zero-length transitions, on the other hand occur asynchronously, when a process performs a zero-length transition all others are blocked. As a consequence of this, zero-length transitions are always enabled, since there is no need for coordination between processes in this case. Unit transitions however, are only enabled when there is at least one unit transition enabled in each process. Notice that this parallel composition model satisfies one important invariant: time is the same in all processes at all times. This is essential because it means that time is well defined in any state reachable in the model.

A symbolic implementation of this parallel composition model is straightforward given the traditional parallel composition algorithms used in BDD based tools: given two processes  $P_a$  and  $P_b$  defined by their respective transition relations  $TR_a$  and  $TR_b$ , their synchronous composition is the transition relation  $TR_a \wedge TR_b$ , and their asynchronous composition is  $TR_a \vee TR_b$ .

Under the new model we must first differentiate between unit and zero-length transitions. Given  $TR_a$  we define  $TR0_a$  ( $TR1_a$ ) as the transition relation for zero-length (unit) transitions in  $P_a$ . We can then define that the global transition relation for a model with processes  $P_a$  and  $P_b$  is:

$$TR = (TR1_a \wedge TR1_b) \vee (TR0_a \vee TR0_b)$$

Whenever unit transitions are enabled in all processes, the expression above guarantees that they are also enabled in the composed model. This expression also guarantees that zero-length transitions enabled in some process are also always enabled in the composed model. It is easy to see that neither type of transition can block the other type. The only other condition that must be imposed in this model is that time always change. This can be ensured by forbidding zero-length loops. It is a simple condition to enforce because transitions are always marked with unit or zero-length and a syntactic check can be identify zero-length loops.

To determine how much time has elapsed between events, we use the condition counting algorithms. For example,  $\text{MINCOUNT}[a, t = 1, b]$  determines the minimum time between events  $a$  and  $b$ . Similarly the  $\text{MAXCOUNT}$  algorithm can be used to determine the longest time between  $a$  and  $b$ .

## 5 Expressive Power of the Method Proposed

The method proposed does not have the same expressive power as a continuous time model. Our method uses a different “discretization” of dense time, but the final model is still discrete. It has been proven [2] that there exist systems which cannot be discretized without changing their behavior. In [2] it is shown that the following circuit has behaviors that cannot be captured by any discretization. It

has four signals  $x_0, x_1, x_2$  and  $x_3$ , and transitions which assign values to them as:  $x_1 = \neg x_0, x_2 = \neg x_0$  and  $x_3 = \neg x_0$ . Each transition takes time between 0 and 1 units to occur. Let  $t_1, t_2$  and  $t_3$  be the times when each transition occurs. A possible behavior of the circuit could have transitions times satisfying  $0 \leq t_1 < t_2 < t_3 \leq 1$ . In a discrete time model the only values allowed for  $t_i$  are 0 or 1, and so it is impossible to assign three different values for  $t_1, t_2$  and  $t_3$ . Consequently, this behavior cannot exist in a discrete model.

The conclusion in [2] is that only models without strict inequalities can be guaranteed to be discretized correctly. Only a weaker notion of behavior preservation can be maintained during discretization: It is possible that events that occur at different time instants in the dense time model occur at the same time instant in the discretized model. Our model does not overcome this restriction. It is frequently argued that because of this problem systems modeled using discrete time cannot capture the essential properties of a design. We argue, however, that the key feature is not the accuracy with which  $t_1, t_2$  and  $t_3$  are represented, but rather their ordering. In fact, it is often the case that the constants used in specifying properties in continuous time tools can only be integers. Consequently, it is not possible to determine the exact value for the  $t_i$ s, only the order in which the transitions have occurred.

With the use of transitions that take zero time to occur, our method provides a way of preserving the same ordering of events as dense time models. It is not possible to determine the exact values for  $t_i$ s in our method, since they will be zero. However, this is not as strong a restriction as it seems, since in dense time models it is frequently the case that the exact values cannot be determined either. We claim then that the essential properties of a design are preserved by our method in a similar way as by methods that use dense time. For example, one property that would capture the behavior above can be written as (where  $e_i$  is the event corresponding to the transition of signal  $x_i$ ):

$$x.(e_1 \rightarrow EFy.(e_2 \rightarrow EFz.(e_3 \wedge 0 \leq x < y < z \leq 1)))$$

It can be expressed in our method by the property ( $t$  is true in unit-length transitions, and false in zero-length ones):

$$(e_1 \wedge \neg e_2) \rightarrow EXE[\neg tU(e_2 \wedge \neg e_3 \wedge \neg t \rightarrow EXE[\neg tUe_3])]$$

In most cases, the fact that the total time elapsed is less than one time unit is not encoded in the formula itself. In this case the formula can be simplified to

$$(e_1 \wedge \neg e_2) \rightarrow EF(e_2 \wedge \neg e_3 \rightarrow EF e_3)$$

One important consideration is that this property can be verified using discrete time models by simply doubling the time quantum. This is implemented by changing all transitions into two consecutive ones, that is, one transition in the new model takes half a unit, instead of one unit. This however, has two serious problems. One it adds a significant overhead to verification. The second one is that it is not possible to know by how much we should decrease the time quantum, because in general there is no way to find out when events that happened in

different times have been considered simultaneous by the model. Because of this we cannot determine when the results of a verification using discrete time would be different if the model was refined. Our method does not suffer from these problems. There is no significant overhead added, since only one additional variable is created in the model, and all possible ordering of events are represented in the model, making refinements in the time quantum unnecessary.

## 6 Examples

### 6.1 Steam Boiler

In order to demonstrate the expressive power of the method we have verified the steam boiler example described in [14]. Steam boilers are mostly used in thermoelctrical power plants. It is extremely important to keep a steam boiler working correctly since any malfunction may cause an accident with serious consequences.

The steam boiler consists of a water tank, two pumps, and sensors that measure the pumping rates, the steam evacuation rate and the water level. A controller oversees the operation of the system. The controller must guarantee that the water level is always between two values  $M_1$  and  $M_2$  at all times, and should try to maintain water level between the normal operating levels  $N_1$  and  $N_2$  as much as possible. The controller and the physical plant communicate in discrete intervals, once every  $\Delta$  seconds. During each communication phase, all units send information to the controller, which responds by sending messages to the units. All communication takes place instantaneously.

The controller decision to turn on or off the pumps is based on the water level  $w$ . The two pumps need five seconds to start pumping water in the tank because of the high pression inside the tank. The pumps are turned off imediately after receiving a message to stop pumping from the controller. Four values are used by the controller to decide how many pumps should be active. Depending on these values and the current water level the controller turns on or off one or both valves.

The most important property of the steam boiler is that the water level is always between  $M_1$  and  $M_2$ , provided it is initially between  $L$  and  $U$ . We also require that the emergency-stop mode is never entered. Therefore unsafe states are states that satisfy the formula  $(w < M_1) \vee (w > M_2) \vee emergency\_stop$ .

We have modeled the the high-level interactions between discrete control decisions and the continuous aspects of the underlying physical plant. We concentrate on the continous aspects of the system and the modeling of these aspects with the semi-continous method described in the previous section.

We have set the values of the system constants as follows: sampling time  $\Delta = 5$  seconds, maximal steam rate  $W = 6$  liters per second, pumping capacity  $P = 4$  liters per second, interval of normal water levels  $[N_1 = 100, N_2 = 150]$  liters, interval of acceptable water levels  $[M_1 = 5, M_2 = 220]$  liters. These constants have the same values as in [14]. Thus, we may compare our results with the

results obtained in that paper. Using Verus, we have been able to verify that the controller with  $L' = 25$ ,  $L = 70$ ,  $U = 170$ , and  $U' = 200$  maintains the water level within the required bounds. This result is the same obtained in [14]. The verification took 2.3 seconds and 1.1 MBytes of memory on a Pentium II system.

We have also verified other properties of the steam boiler using the mincount and maxcount algorithms. For example, an important parameter of the system is the size of  $\Delta$ , the frequency of communication between controller and plant. Using Verus we have been able to determine that  $\Delta = 6$  also satisfies the safety requirements, but  $\Delta = 7$  does not. A counterexample has been produced showing how the longer communication delay can cause problems. This means that if communication between controller and units is delayed by up to one second, safety is maintained, but longer delays can cause safety problems. Other parameters that have been identified include, for example, the minimum and maximum times needed for water to go from the minimum to the maximum level. The interval is  $[20, \infty]$  seconds, meaning that that the water may never reach the maximum water level from the minimum water level, but it never take less than 20 seconds. Several other similar parameters have been computed.

## 6.2 Automotive Engine Controller in Cutoff Model

In order to demonstrate the efficiency of the method we have verified an automotive engine controller in cutoff mode described in [17]. This model was derived from the one verified by HyTech. We have studied the cutoff mode, where the driver releases the accelerator and the controller regulates fuel injection to minimize the oscillation while decelerating.

*Cutoff control problem.* In the cutoff mode we consider control of the engine once the driver has released the accelerator pedal, thereby expressing a desire to have zero torque delivered by the engine. The control objective is to reach injection cutoff while minimizing acceleration discomfort. If fuel injection is abruptly cut off, the vehicle may exhibit very undesirable acceleration oscillations. If fuel injection remains for a long time the car does not decelerate. In order to minimize these problems, the controller makes intelligent decisions about when and how to cut off fuel injection.

The system consists of a the engine, which include the driveline and the cylinders, and its controller. The car's driveline can be basically described by four states variables. The state variable  $\zeta_1$  represents the engine block angle (in radian),  $\zeta_2$  represents the wheel revolution speed (in radians per second),  $\zeta_3$  represents the axle torsion angle (in radians),  $\zeta_4$  represents the crankshaft angle (in degrees). The engine has four cylinders, each of which cycles in lockstep through four phases in the following order: intake (I), compression (C), expansion (E), and exhaust. The controller must make its decision on injection (modeled by the binary output variable  $j$ ) at the beginning of the preceeding exhaust phase. If fuel is injected into a cylinder, the cylinder produces torque on its next expansion phase. Thus the driveline does not react to a control decision until three phases later.

The controller sets the value of  $j$  at each phase change, with the function  $F$  modeling the decision to inject fuel or not. The function  $F$  is defined over a transformed state space (over the variables  $x_1, x_2, x_3, x_4$ ) that helps isolate the fundamental modes related to acceleration oscillations. Powertrain oscillations are due to the pair of complex conjugate poles, which are related to  $x_2$  and  $x_3$  components. Thus we concentrate on the  $x_2 - x_3$  subspace, where the encirclements of the origin correspond to oscillations.

*Requirements.* The automotive engine controller should meet the requirement that for a given initial condition the state is close to the origin (injection cutoff) within a bounded number of phases (convergence).

*Verification.* To prove the convergence requirement using the same parameters described in [] we considered only trajectories of duration up to 30 phases long and showed that all such trajectories is close to the origin within 30 phases.

In general, it has been observed that for systems which involve large time constants, discretization can lead to a large state space representation even when using BDDs, and continuous time models may be more efficient, since the representation is less dependent on time granularity. However, for models whose timing constants are well-behaved, a discrete-time model with a uniform BDD-based representation can present significant savings in efficiency. Our proposed method alleviates some of the expressivity loss associated with discrete time, while performing as efficiently as discrete time verification algorithms.

Verus and SMV have been used to verify the requirements. The generated code for both tools has been derived from Hytech original code where each state of the hybrid automaton used by Hytech was mapped into a state of both tools following their respective syntax. This mapping process is done automatically by a simple script wrote for this purpose. We divided the  $x_2 - x_3$  state space into 25 x 25 partitions increasing the accuracy of the rectangular approximations. The convergency property verification using Verus took xxx seconds and yyy Mbytes on a Pentium II system.

## 7 Conclusions

In this work we propose a new algorithm to perform quantitative timing analysis of models that is more efficient than its predecessor. This algorithm, called *condition counting*, counts the minimum and maximum number of occurrences of events between two events *start* and *final*. We then use this algorithm to implement an alternative method to represent time that is more expressive than pure discrete time, although less expressive than continuous time. Its verification is, however, as efficient as the verification of discrete time models, and it can be used in symbolic model checkers such as SMV and Verus virtually without changes to the tools.

## References

1. Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for real-time systems. In *Proc. 5<sup>th</sup> Ann. Symp. on Logic in Comput. Sci.*, pages 414–425. IEEE Comp. Soc. Press, June 1990.
2. Eugene Asarin, Oded Maler, and Amir Pnueli. On discretization of delays in timed automata and digital circuits. In D. Sangiorgi and R. de Simone, editors, *CONCUR'98: Concurrency Theory. 8<sup>th</sup> International Conference Proceedings*, volume 1466 of *Lecture Notes in Computer Science*, pages 470–484, Nice, France, September 1998. Springer Verlag.
3. S. Campos, E. Clarke, W. Marrero, and M. Minea. Verifying the performance of the pci local bus using symbolic techniques. In *International Conference on Computer Design*, 1995.
4. S. V. Campos. *A Quantitative Approach to the Formal Verification of Real-Time Systems*. PhD thesis, SCS, Carnegie Mellon University, 1996.
5. S. V. Campos, E. M. Clarke, W. Marrero, and M. Minea. Timing analysis of industrial real-time systems. In *Workshop on Industrial-strength Formal specification Techniques*, 1995.
6. S. V. Campos, E. M. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. In *IEEE Real-Time Systems Symposium*, 1994.
7. E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, 1981*. Springer-Verlag, 1981. *Lecture Notes in Computer Science*, vol 131.
8. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
9. H. De-Leon and O. Grumberg. Modular abstractions for verifying real-time distributed systems. *Formal Methods in System Design*, 2:7–43, 1993.
10. David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 197–212, Grenoble, France, June 1989. Springer-Verlag.
11. T. A. Henzinger, P. H. Ho, and H. Wong-Toi. HyTech: the next generation. In *IEEE Real-Time Systems Symposium*, 1995.
12. Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. What good are digital clocks ? In W. Kuich, editor, *Automata, Languages and Programming. 19<sup>th</sup> International Colloquium Proceedings*, volume 623 of *Lecture Notes in Computer Science*, pages 545–558, Wien, Austria, July 1992. Springer Verlag.
13. Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. In *Proceedings. Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 394–406, Santa Cruz, CA, USA, June 1992. IEEE Computer Society Press.
14. Thomas A. Henzinger and Howard Wong-Toi. Using hytech to synthesize control parameters for a steam boiler. In *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*, pages 265–282. Springer Verlag, 1996.
15. K. G. Larsen, P. Petterson, and W. Yi. Compositional and symbolic model-checking of real-time systems. In *16<sup>th</sup> IEEE Real-Time Systems Symposium*, 1995.
16. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

17. Tiziano Villa, Howard Wong-Toi, Andrea Balluchi, Joerg Preussig, Alberto Sangiovanni-Vincent, and Yosinori Watanabe. Formal verification of an automotive engine controller in cutoff mode. In *CDC98: IEEE Conference on Decision and Control*, Tampa, Florida, December 1998.
18. S. Yovine. Kronos: A verification tool for real-time systems. In *Springer International Journal of Software Tools for Technology Transfer*, volume 1, October 1997.