Analyzing Real-Time Systems[1]

Jürgen Ruf and Thomas Kropf
Wilhelm-Schickard-Institute
University of Tübingen, Sand 13, 72076 Tübingen, Germany
{ruf,kropf}@informatik.uni-tuebingen.de
http://www-ti.informatik.uni-tuebingen/~ruf

Abstract

Temporal logic model checking is a technique for the automatic verification of systems against specifications. Besides the correctness of safety and liveness properties it is often important to determine critical answer and delay times of systems, especially if they are embedded in a real-time environment. In this paper we present an approach which allows the verification as well as the timing analysis of real-time systems. The systems are described as networks of communicating time-extended finite state machines (I/O-interval structures). We use a compact symbolic representation to obtain efficient analysis algorithms.

1. Introduction

Formal verification has become an important task in the design of software and hardware. Especially automatic techniques like temporal logic model checking are used to support or even to replace standard test and simulation methods. In the area of embedded reactive systems, the verification of timing constrains plays an important role. E.g. minimal and maximal reaction times of a controller have to be guaranteed, setup and hold times of flipflops have to be kept or wait times of work pieces in pause stations of a production automation system should be minimized.

One possibility to verify such timing properties is to use a real-time temporal logic model checker. In this case, the designer has to specify the timing properties with temporal logic formulas and the system as a state transition graph. After the automated verification process, the model checker answers yes or no. This means the system satisfies the specification or not, e.g. the reaction time lies under a certain time bound or not. But the designer obtains no statement about the actual value of the reaction time.

Therefore Campos and Clarke developed an analysis tool (VERUS) which allows the verification of real-time properties as well as the quantitative analysis of real-time systems [1]. This tool gets an description of a system as a synchronous program, the specifications in temporal logic and the analysis queries. The system description is translated into ROBDDs [2] representing the state transition graph. The disadvantage of this approach lies in the representation of time. Since time progress is represented by counters, there exists for every time step a separate state in the underlying graph. Especially if many counters with large values exist, the state space may explode.

Other approaches for real-time model checking use timed automata [3, 4, 5]. Timed automata allow a very detailed description of real-time systems through a fixed number of clocks defined over a dense time model. However, these approaches have very complex model checking algorithms (they are PSPACE-complete) and there exist no analysis algorithms.

In [10] we have presented a new formalism called I/O-interval structure. This formalism expands finite state machines by timed state transitions. We developed efficient model checking algorithms based on a representation of these structures with extended characteristic functions [6]. These functions are implemented with MTBDDs [7,8]. In [9] we have presented a method for the composition of I/O-interval structures completely working on the MTBDD representation, including two minimization heuristics. This approach allows the integration of untimed FSMs (e.g. controller) and I/O-interval structures (e.g. timed environment).

In this paper we develop analysis algorithms working on the symbolic representation of interval structures and which make optimal use of this MTBDD-Based representation. These algorithms determine critical times of specified systems, e.g. wait times of production systems, reaction times of embedded systems or maximal delay times of communication protocols. Together with the already developed verification algorithms, we now have a powerful tool for the verification and the analysis of real-time systems out of many domains.

1. This work is supported by the German Research Grant (DFG project GRASP)

Section 1 introduces the real-time model checking approach which is the basis of the analysis algorithms. First the modeling formalism (I/O-interval structures) and the specification logic (CCTL) are introduced. Afterwards we present the symbolic representation with extended characteristic functions. In Section 3 the different analysis algorithms are introduced. Section 4 describes some optimizations accelerating the analysis algorithms. Experimental results are shown in Section 5. Section 6 concludes this paper.

2. Real-time model checking

2.1. Interval structures and I/O-interval structures

Structures are state-transition systems modeling HW- or SW-systems. The fundamental structures are Kripke structures (unit-delay structures, temporal structures) which may be derived from FSMs. The basic models for real-time systems are interval structures, i.e., state transition systems with additional labelled transitions. We assume that each interval structure has exactly one clock for measuring time. The clock is reset to zero if a state is entered. A state may be left if the actual clock value corresponds to a delay time labelled at an outgoing transition. The state must be left if the maximal delay time of all outgoing transitions is reached (Fig. 2.1). One clock tick is the lowest granularity for the time modeling.

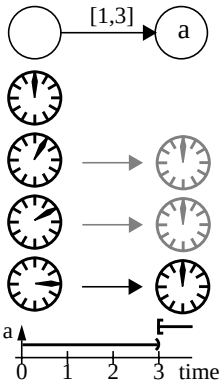

Fig. 2.1. Example IS

Definition 2.1. *An interval structure (IS)* $\Im$ *is a tuple* $\Im = (P, S, T, L, I)$ *with a set of atomic propositions* P*, a set of states* S *(i-states), a transition relation between the states* $T \subseteq S \times S$ *such that every state in* S *has a successor state, a state labeling function* $L: S \to \wp(P)$ *and a transition labeling function* $I: T \to \wp(\mathbb{N})$ *with* $\mathbb{N} = \{1, \ldots\}$ *and* $\wp$ *is the potential set operator.*

Every state of the IS must be left after the *maximal state time*.

Definition 2.2. *The maximal state time of a state* s $\mathrm{MaxTime}: S \to \mathbb{N}$ *is the maximal delay time of all outgoing transitions of* s*, i.e.*

$$MaxTime(s) = max\{t \mid \exists s'.(s, s') \in T \wedge t = max(I(s, s'))\}. \quad (1)$$

Besides the states, we also have to consider the currently elapsed time to determine the transition behavior of the system. Hence, the actual state of a system, called the *generalized state*, is given by an i-state s and the actual clock value v (the elapsed time).

Definition 2.3. *A generalized state (g-state)* $g = (s, v)$ *is an i-state* $s \in S$ *associated with a clock value* $v \in \mathbb{N}_0$ *(*$\mathbb{N}_0 = \{0, 1, \ldots\}$*). The set of all g-states in* $\Im = (P, S, T, L, I)$ *is given by:*

$$G = \{(s, v) \mid s \in S \wedge 0 \le v < \mathrm{MaxTime}(s)\} \quad (2)$$

The semantics of ISs is represented by runs.

Definition 2.4. *Given the IS* $\Im = (P, S, T, L, I)$ *and a starting g-state* g_0*. A run is a sequence of g-states* $r = (g_0, g_1, \ldots)$*. For the g-states* $g_j = (s_j, v_j) \in G$ *of the sequence holds either*

- $g_{j+1} = (s_j, v_j + 1)$ *with* $v_j + 1 < MaxTime(s_j)$ *or*
- $g_{j+1} = (s_{j+1}, 0)$ *with* $(s_j, s_{j+1}) \in T$ *and* $v_j + 1 \in I(s_j, s_{j+1})$.

To expand interval structures by a possibility for communication, we extend them to I/O-interval structures. These structures carry additional input labels on each transition. Such an input label is a Boolean formula over the inputs. We interpret this formulas as input conditions which have to hold during the corresponding transition times. Input-insensitive edges carry the formula $true$.

Definition 2.5. *An I/O-interval structure (I/O-IS) is a tuple* $\Im_{I/O} = (P, P_I, S, T, L, I, I_I)$*. The set of all input valuations is:* $Inp := \wp(P_I)$.

- *The components* P, S, L *and* I *are defined as in IS.*
- P_I *is a finite set of atomic input propositions.*
- *The transition relation connects pairs of states and the destination inputs:* $T \subseteq S \times S \times Inp$.[1]
- $I_I : T \to \wp(Inp)$ *is the transition input labeling.*

The execution semantics of I/O-interval structures are similar to interval structures. The only difference lies in the input restrictions. A transition may only be taken if the corresponding input restriction holds until the delay time is reached and the state is left.

2.2 The logic CCTL

CCTL [9] is a temporal logic extending CTL with quantitative bounded temporal operators. Two new temporal operators are introduced to make the specification of timed properties easier. The syntax of CCTL is shown in (3); where $p \in P$ is an atomic proposition, $a \in \mathbb{N}$ and $b \in \mathbb{N} \cup \{\infty\}$ are time bounds. All interval operators can also be accompanied by a single time-bound only. In this case the lower bound is set to zero by default. If no interval is specified, the lower bound is implicitly set to zero and the upper bound is set to infinity. If the X-operator has no

1. In the most cases, the input valuations of the target states are irrelevant [10].

time bound, it is implicitly set to one. The semantics of CCTL is given in [10].

$$\varphi := \begin{cases} p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \varphi \leftrightarrow \varphi \\ \mid \mathsf{EX}_{[a]}\varphi \mid \mathsf{EF}_{[a,b]}\varphi \mid \mathsf{EG}_{[a,b]}\varphi \mid \mathsf{E}(\varphi\ \mathsf{U}_{[a,b]}\varphi) \\ \mid \mathsf{E}(\varphi\ \mathsf{C}_{[a]}\varphi) \mid \mathsf{E}(\varphi\ \mathsf{S}_{[a]}\varphi) \\ \mid \mathsf{AX}_{[a]}\varphi \mid \mathsf{AF}_{[a,b]}\varphi \mid \mathsf{AG}_{[a,b]}\varphi \mid \mathsf{A}(\varphi\ \mathsf{U}_{[a,b]}\varphi) \\ \mid \mathsf{A}(\varphi\ \mathsf{C}_{[a]}\varphi) \mid \mathsf{A}(\varphi\ \mathsf{S}_{[a]}\varphi) \end{cases} \quad (3)$$

2.3 Symbolic representation of interval structures

In contrast to an explicit representation, the symbolic representation of FSMs uses characteristic functions. These functions map states or transitions to true if they belong to the represented set. This representation avoids the explicit enumeration of states or transitions and is therefore able to represent sets of states with more than 10^{20} elements [11].

The representation of g-states in IS need to store the elapsed time besides the actual state. Therefore we use extended characteristic functions (ECF, [12]) for a symbolic representation of IS. The following definition of ECFs is adapted to the representation of IS.

Definition 2.6. *Given a set of elements U (the universe, e.g. the set of states S) and a subset $A \subseteq U$, where every element in A is associated with a set of natural numbers (the clock values, also called the attribute set). An extended characteristic function representing A is given by: $\Lambda_A : U \rightarrow \wp(\mathbb{N})$ with*

$$\Lambda_A(s) := \begin{cases} \alpha & \text{if } s \in A \text{ and } \alpha \subseteq \mathbb{N}_0 \text{ assoc. with s} \\ \varnothing & \text{otherwise} \end{cases} \quad (4)$$

I.e. we accumulate all clock values of g-states in the clock set which have the same i-state. The standard set operations like union or intersection may be extended to this kind of characteristic functions.

ECFs may also be used to represent the transition relation. In this case we map pairs of states (the transition) to the valid delay times. Also I/O-IS may be represented with ECFs.

2.4 Model checking and composition

The main idea for the model checking algorithm in [6] is to associate each subformula of the specification with the set of g-states holding it. E.g. the formula "true" is associated with the set of all g-states. The atomic proposition $p \in P$ will be associated with the following g-state set:

$$\{g \mid g \in G \wedge \exists s \in S.\exists v \in \mathbb{N}_0.g = (s, v) \wedge p \in L(s)\} \quad (5)$$

The formula $\mathsf{EX}\varphi$ will be associated with the set of predecessor g-states of the g-states associated with φ. Other temporal operators may be computed by special fixed point iterations based on the predecessor computation. This computation may be split in two main operations: the local and the global predecessor computation. The local predecessor computation takes an ECF representing the actual g-state set and decrements all attribute sets by one. A zero value will be removed:

$$\forall s.local\_pre(\Lambda)(s) = \{v - 1 \mid v \geq 1 \wedge v \in \Lambda(s)\} \quad (6)$$

For a set A of natural numbers, we identify the set where each value is decremented with: $A - 1$.

The global predecessor computation computes the predecessor of zero clocked g-states. This operation works similar to the predecessor computation of symbolic FSM traversal techniques. This operation selects the g-states with zero clock values and computes together with the transition relation the predecessor g-states in predecessor i-states. This operation is more complex than the local predecessor computation. More details may be found in [9]. The function pre is the union of the local and the global predecessors. The result of this function is exactly the set of predecessors of a given g-state set. The function $succ$ works similar and computes the successor g-states.

Model checking as described above works exactly on one IS, but real-life systems are usually described in a modular way by many interacting components. In order to make model checking applicable to networks of communicating components, it is necessary to compute the product structure of all submodules. This product-computation is called composition. The composition is described in detail in [9].

The composition algorithms are also applicable to networks of I/O-IS if we work with closed systems, where every free variable is bounded by a structure. After composition, the efficient model checking algorithms for IS may be used for verification. These algorithms are described in detail in [6]. Composition as well as the model checking algorithms use the symbolic representation of g-state sets and transition relations with ECFs.

3. The analysis algorithms

The analysis technique should help a designer to extract important time bounds from his formal system description. An often arising problem is to compute the maximal stability of a signal. E.g. if we want to examine a switch circuit with delay times, it may be very important to determine the number of time steps, the out put signal stays stable high. This stability analysis is also useful to compute the maximal number of time steps, a work piece waits on a buffer until it will be processed.

Other typical problems are minimal and maximal delay times between events, e.g. how long does it minimal/maximal take until the first work piece leaves the production cell. In the following two subsections we present the analysis algorithms.

3.1 The STABLE algorithm

The analysis algorithms do also use the traversal techniques (predecessor computation) of the model checking algorithms. The program 1 shows the implementation of the STABLE-algorithms in an imperative (C- or Pascal-like) programming language.

```
1 int stable(ecf f)
2   act := f
3   old := ∅
4   res := 0
5   while (act ≠ old ∧ act ≠ ∅) do
6     old := act
7     act := pre(act) ∩ act
8     res := res + 1
9   if act = ∅ then return res
10  else return ∞
```

Program 1. The STABLE algorithm

The algorithm gets a set of g-states to examine (`f`). This set may be specified as CCTL formula and may be computed by the model checking algorithm (see Section 2.4).

The algorithm determines the maximal number of steps (`res`) such that there exist a run in which the first `res` g-states are members of the set represented by `f` but the g-state at position `res`+1 is not a member of `f`.

The computation of the algorithms initializes a set of actual g-states (`act`) with the set of the g-states to examine (`f`). Then it starts shrinking this set successively by intersecting it with its predecessors. If the set `act` is empty, the maximal stability is computed. If the algorithm reaches a fixed point, i.e. `act=old`, then there exist a cycle in `f`.

The figure 3.1 illustrates the computation of the STABLE algorithm. The interval structure to examine is shown at the top. The lower part displays the g-states and the predecessor relation (dotted arrows). The analysis set is given by the CCTL formula $\mathsf{AF}_{[2]}a$ which means, that on all paths the signal a must be reachable within two time steps. The grey oval includes the actual g-states (`act`). The numbers on the left shows the actual iteration (equal to `res`).

3.2 The MIN/MAX algorithms

The following two algorithms compute the minimal resp. maximal number of unit time-steps between a start and an aim set of g-states. Campos and Clarke introduced algorithms based on an ROBDD representation [13].

The idea of the algorithm in program 2 is to initialize a set of g-states (`act`) with the set of start states and then to expand this set successively by its successors. The algorithm counts the number of predecessor computations while iterating (the delay time between a g-state and its successor is exactly one time step). This procedure will be repeated until at least one aim state is reached or the set

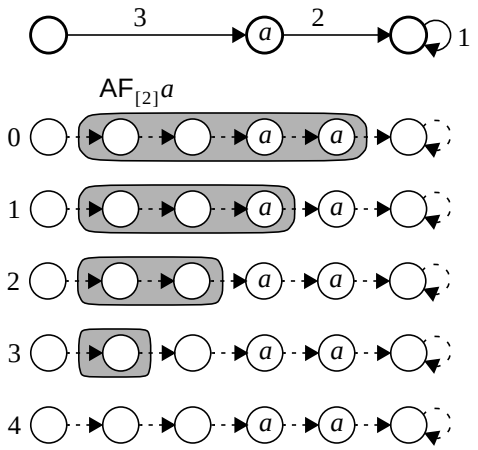

Fig. 3.1. Illustration of the STABLE-computation.

will not change anymore. In the latter case, there exist no path from the start states to the aim states.

```
1 int min(ecf start, aim)
2   act := start
3   old := ∅
4   res := 0
5   while (act ≠ old ∧ act ∩ aim = ∅) do
6     old := act
7     act := act ∪ succ(act)
8     res := res + 1
9   if act ∩ aim ≠ ∅ then return res
10  else return ∞
```

Program 2. The MIN algorithm

The algorithm computing the maximal delay time is shown in program 3. It initializes a set with all g-states except the the `aim` set (`act`). Then it shrinks this set by intersectin `act` with its predecessors until no start g-state is left or a fixed point is reached. In the latter case, there exists either no connection between the start set and the aim set or there exist a loop such that the maximal delay time is infinite.

```
1 int min(ecf start, aim)
2   act := ¬aim
3   old := ∅
4   res := 0
5   while (act ≠ old ∧ act ∩ aim = ∅) do
6     old := act
7     act := act ∩ pre(act)
8     res := res + 1
9   if act=old then return ∞
10  else return res
```

Program 3. The MAX algorithm

4. Time prediction and time jumps

As already mentioned in Section 2.4, the global predecessor computation is more expensive than the local predeces-

sor computation. But on the other hand, the set of global predecessor g-states may stay constant during some iteration steps. Upon this observation, we present an optimization technique which computes global predecessors only if they change. This technique (called time prediction) accelerates the analysis algorithms enormously.

The idea is to predict the number of computations for which the global predecessors stay constant. For this number of steps, the iteration will be performed locally on the attribute sets of the ECFs. We show this technique exemplarily for the STABLE algorithm.

The prediction examines all attribute sets of the actual state set (Λ_A) and the global predecessors (Λ_G), computes the number of steps and returns the minimum:

$$predict(\Lambda_A, \Lambda_G) := min_{s \in S}\ l\_pred(\Lambda_A(s), \Lambda_G(s)) \quad (7)$$

After the prediction is computed, the iteration of line 5 to 9 in program 1 may be performed locally on the attribute sets. Therefore we introduce a local iteration function (`l_iter`) which preforms this iteration on the actual clock values (A) and the actual global predecessors (G) of one state:

```
1 set l_iter(set A, G, int n)
2   res := A
3   old := ∅
4   i := 0
5   while i ≤ n ∧ res ≠ old do
6     old := res
7     res := res ∩ (res-1 ∪ G)
8   return res
```

Program 4. Local iteration function

As already defined, the decrementation in line 7 affects all values in the set. To apply this attribute oriented function to an ECF, we apply it to every attribute set:

$$\forall s.iteration(\Lambda_A, \Lambda_G, n) = l\_iter(\Lambda_A(s), \Lambda_G(s), n) \quad (8)$$

With the knowledge of the local iteration, we may define the local prediction function. The function l_pred receives two sets of natural numbers, the actual clock values (A) and the actual global predecessors (G) of the examined state. The local prediction has to distinguishe two cases:

$$l\_pred(A, G) := \begin{cases} n+1 & \text{if } \exists n.\{0, \ldots, n\} \subseteq A \wedge n+1 \notin A \wedge \\ & \quad \forall c \in G.c \notin 0, \ldots, n \\ \infty & \text{otherwise} \end{cases} \quad (9)$$

The only reason to recompute the global predecessor is when a zero clock value disappears. A new zero clock value may never appear. The zero clock value disappears after $n+1$ steps, when the values 0 to n are in the actual g-state and the value $n+1$ is not an actual clock value. Moreover there may not exist a value in G smaller than $n+1$ because this value supports the zero value. In this situation every iteration step removes one element in A. The first step removes the value n, the second step removes the value $n-1$ and so on. This means after $n+1$ steps the zero clock value will be removed. In program 5 the complete algorithm with time prediction is shown.

```
1 int stable_predict(ecf f)
2   act := f
3   old := ∅
4   res := 0
5   while (act ≠ old ∧ act ≠ ∅) do
6     old := act
7     g := global_pre(act)
8     n := predict(act,g)
9     act := iteration(act,g,n)
10    res := res + n
11  if act = ∅ then return res
12  else return ∞
```

Program 5. Stable computation with prediction

An additional technique to accelerate the algorithms is called time jump. This technique tries to improve the local iteration. If we regard the set of clock values in different iteration steps in figure 4.1, we realize, that the actual clock value set shrinks until it will be supported by a value in G.

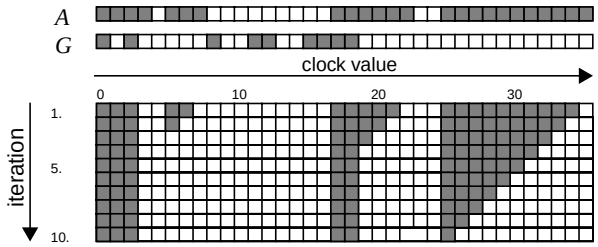

Fig. 4.1. The behavior of the g-state sets

Therefore it is not necessary to perform every iteration step explicitly, we exploit this regular behavior to execute in one complex operation n iteration steps. Formally we may define the time jump for the stability analysis through:

$$jump(A, G, n) := \left\{ v \mid \begin{array}{l} \{v, \ldots, v+n\} \subseteq A \vee \\ \exists w \geq v.w \in G \wedge \{v, \ldots, w\} \subseteq A \end{array} \right. \quad (10)$$

This operation is very implementation dependent, therefore we will not show further details.

Figure Fig. 4.2 shows the computation of figure 3.1with time prediction and time jumps. On the right side of this figure, the result of the prediction for the next jump is shown. Because of the time prediction technique only 2 global iteration steps are necessary.

5. Experimental results

All presented algorithms are implemented in our real-time verification tool **RAVEN**. The ECFs are implemented by

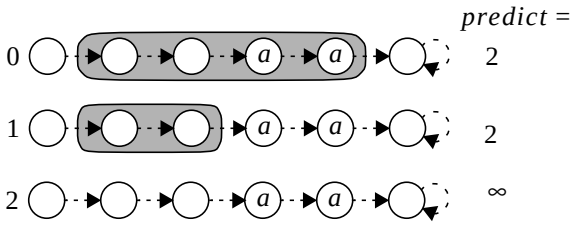

Fig. 4.2. Optimized STABLE-computation

multi terminal binary decision diagrams (MTBDDs). We compared our implementation with VERUS [1]. We modeled timed transitions in VERUS by loops which wait one time unit and decrement a counter. We compared only the min/max algorithms because VERUS do not support the stable query but on the other hand VERUS offers two queries mincount and maxcount which are actually not supported by **RAVEN**. We have chosen the same variable ordering for the signals.

The examined case studies are the single pulser circuit enriched by timed gates (SP), a production cell (PC) and the arbitration mechanism of the J1850 bus protocol. The first two systems are widely used to compare formal methods. Details of the examined systems may be found in [10].

For the single pulser we computed the minimal and the maximal length of the output impulse. In the production cell we were interested in the minimal and the maximal time when the first work piece leaves the cell. In the J1850 example we checked the minimal and the maximal delay time when a node will leave the sending mode. The following table compares the runtimes of both tools. For the VERUS run-times we tried various options and choose the best results. For **RAVEN** we compared the time prediction (+t) and the prediction together with the time jumps (+tj).

The run-times in the tabel with and without optimizations seems to show, that these techniques cause only a tiny speedup. But the run-times shown in the table contain besides the analyse times also the composition times of the structures. In all three examples the composition consumes the major part of the times (11.73 sec. for the single pulser, 2000 seconds for the production cell and 25 sec. for the J1850). If there will be more than two analysis (as computed in the examples), than the fraction of composition time to analyse time will shrink and the the optimizations will cause a larger speedup.

	SP	PC	J1850
VERUS	118	-[a]	-[b]
RAVEN	12	2314	406
RAVEN +t	12	2146	97
RAVEN +tj	12	2044	95

a. VERUS terminated with an error: „string table overflow“

b. VERUS was terminated due to a memory consumption over 600MB

6. Conclusion

In this paper we presented a new approach for the analysis of real-time systems. This approach works on the same representation as real-time model checking. This allows the verification as well as analysis.

The systems are described by networks of communicating I/O-interval structures. These structures include timed transitions and the labeling of these transitions with input restrictions. After the composition of the substructures, **RAVEN** performs the model checking and the analysis on the resulting interval structure.

Due to the symbolic representation of the structures and g-state sets with extended characteristic functions, this approach works very compact in the memory consumption. Especially if long delay times are specified, the advantages of this technique are obvious. By exploiting the locally stored timing information (in the attribute sets), techniques like time prediction and time jumps accelerate the model checking and analysis algorithms.

Bibliography

[1] S. Campos, E. Clarke, and M. Minea. The verus tool: A quantitative approach to the formal verification of real-time systems. In *CAV*, LNCS. Springer Verlag, June 1997.

[2] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, August 1986.

[3] R. Alur, C. Courcoubetics, and D. Dill. Model Checking for Real-Time Systems. In *LICS*, Washington, D.C., June 1990. IEEE CSP.

[4] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. In *LICS*, Santa-Cruz, June 1992. IEEE Computer Society Press.

[5] M. Bozga, O. Maler, A. Pnueli, and S. Yovine. Some progress in the symbolic verification of timed automata. In *CAV 97*. Springer Verlag, June 1997.

[6] J. Ruf and T. Kropf. Symbolic model checking for a discrete clocked temporal logic with intervals. In *CHARME 97*, Montreal, Canada, Oct. 1997. Chapman and Hall.

[7] E. Clarke, K. McMillian, X. Zhao, M. Fujita, and J.-Y. Yang. Spectral Transforms for large Boolean Functions with Application to Technologie Mapping. In *DAC 93*, Dallas, TX, June 1993.

[8] R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and Their Applications. In *ICCAD*, Santa Clara, CA, Nov. 1993. ACM/IEEE, IEEE CSP.

[9] J. Ruf and T. Kropf. Using MTBDDs for composition and model checking of real-time systems. In *FMCAD 1998*, Palo Alto.Springer.

[10] J. Ruf and T. Kropf. Modeling and Checking Networks of Real-Time Systems. In *CHARME 99*, Bad Herrenalb, Germany. Springer Verlag, Septemper 1999.

[11] J. Burch, E. Clarke, K. McMillan and D. Dill. Symbolic Model Checking: 10^{20} States and Beyond. In *LICS*, IEEE Computer Society press, June 1990.

[12] J. Ruf and T. Kropf. Using MTBDDs for discrete timed symbolic model checking. Multiple-Valued Logic – An International Journal, 1998. Gordon and Breach publisher.

[13] S. Campos, E. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitive charakteristics of finite-state real-time systems. Technical Report, Pittsburgh, May 1994.