*Journal of Universal Computer Science, vol. 7, no. 1 (2001), 89-104*
*submitted: 1/9/00, accepted: 13/10/00, appeared: 28/1/01* 

# RAVEN: Real-Time Analyzing and Verification Environment

**Jürgen Ruf**
(University of Tübingen, Germany
ruf@informatik.uni-tuebingen.de)

**Abstract:** In this paper we present the real-time verification and analysis tool RAVEN. RAVEN is developed for verifying timed systems on various levels of abstraction. It integrates a real-time model checker for real-time specifications, it offers algorithms for analyzing critical delay times, for inspecting data values and event occurrences and for detecting dead-locks and live-locks. The counter example generator provides helpful information for error recovering by printing system execution paths (failing a given specification) to the integrated wave-form browser. All included algorithms are based on a common data structure enabling a compact representation and possibilities for acceleration. By some examples we show that our approach outperforms some state-of-the-art verification tools.
**Key Words:** formal verification, model checking, analysis, real-time systems
**Categories:** F.3.1, I.6.4, C.3

## 1 Introduction

Formal verification has become an important task in the design of systems. Techniques like equivalence checking and symbolic model checking have reached industrial applicability. These techniques are well suited for fully synchronous systems modeled with a qualitative notion of time. If systems are embedded in real-time environments and upper or lower bounds for reaction times are important to guarantee a proper and save functionality, the verification of real-time properties becomes very important. We target at this application area with our tool RAVEN.

Various efforts have been undertaken to extend temporal logics and proof algorithms to timed systems (i.e. systems containing quantized timing information). Two main approaches have to be distinguished here: those based on timed automata [ACD90] and extensions of symbolic CTL model checking using ROBDDs (reduced ordered binary decision diagrams) [BCM+90]. For both finding efficient algorithms and implementations is still an active area of research as real-time model checking bears additional challenges compared to standard model checking:

- the model checking algorithms have to cope with time, i.e. with natural or real numbers which makes the application of propositional logics based on ROBDD techniques hard
- adding timing to state transition systems worsens the state space explosion problem, especially if a composition of timed transition systems is necessary and if time intervals, i.e. non-deterministically varying transition times are allowed.

To cope with these challenges our tool (RAVEN) uses MTBDDs for a symbolic representation of the systems [RK97]. This data structure results in compact date structures and efficient verification algorithms. On some examples we will show that this approach outperforms some state of the art tools for the verification of timed systems.

RAVEN is a real-time model checker extended by analysis algorithms. The system description is specified as a network of communicating parallel working real-time processes. Each process is a time extended finite state machine. The properties are specified in the quantitative temporal logic CCTL. The queries for the timing analysis cover minimal and maximal delay time computation as well as minimal and maximal stability computation. Also data values and event occurrences may be analyzed. RAVEN is able to generate counter examples and witnesses for CCTL formulas. Analysis results can be visualized by traces. All traces are graphically presented in an integrated waveform browser. Moreover, RAVEN offers additional checks. For instance, it can detect dead- and live locks and computes traces to the locking system states. It is also possible to generate random simulations of the composed system.

The next section gives a short overview to the state-of-the-art in real-time verification. Afterwards we present the necessary theoretical background in Section 3. Section 4 describes the architectural set-up of the RAVEN system and the major processing steps for the verification. In Section 5 we present the input language of RAVEN: RIL. This language is used to describe parallel working processes and to specify formal properties to prove and timing queries for analysis. The description of two case studies and their modeling in RIL is presented in Section 6. Afterwards we present some experimental results in Section 7. Section 8 will conclude this paper.

## 2 State-of-the-Art

### 2.1 Timed Automata

A formalism which has been created to model real time systems are *timed automata* presented in [ACD90]. In timed automata time is represented by clocks carrying real numbers and time passes in the states. A transition is chosen based on clock predicates on the edges and input events. Specifications are given in TCTL, an extension of CTL. As an arbitrary number of clocks is possible, this is a very powerful approach. Different tools based on this theory have been presented like KRONOS [DOTY96] or UPPALL [BLL+95]. Composition of timed automata is easily possible leading to a new automaton carrying the sum of all the clocks of the original automata.

However, although deriving the composed structure is simple, the state explosion problem is only delayed to the point of model checking. Then in each state the product of the values of all clocks have to be considered. To solve this state explosion problem on-the-fly model checking techniques can be used. However, if specifications are to be proven correct which require the traversal of the complete reachable state space (e.g. mutual reachability of states) the efficiency gain is low. Moreover the underlying proof algorithms are different from standard CTL fixed point computations and hence it is more difficult to find efficient representations. Thus only recently first efforts have been presented on how to use BDD like techniques for a symbolic state set representation. Therefore in practice the resulting runtimes especially for model checking composed structures may be high even when using these symbolic techniques [BMPY97].

### 2.2 Extensions to CTL Model Checking

A different approach is taken by people who try to extend the well-known CTL model checking techniques to real-time. It is based on the observation that in very many applications the expressive power of timed automata is not necessary. This comprises e.g. all those systems which may be described using global time, i.e. only one clock is necessary. Usually these approaches attribute edges of the transition system with delay times (mostly natural numbers) and enables quantized timing parameters in the temporal operators, leading to CTL extensions like RTCTL (real-time CTL, [EMSS92]) or QCTL (quantized CTL, [FGK96]). To retain the efficient BDD representation, delay times are represented by a binary encoding, added to the transition relation or by representing all transitions with a certain delay by a separate transition relation. This can be seen as a special case of timed automata where only one clock carrying natural numbers is allowed which is reset after each state transition. A tool based on this approach is VERUS [CCM97]. A timed model checking algorithm based on multi-terminal BDDs (MTBDDs) has been proposed [KR97]. The main advantage is that the efficient implementation techniques of standard CTL model checking can be used.

## 3 Theoretical Background

Temporal logic model checking takes a structure (representing the system behavior) and a formula and automatically checks if the structure meets the specification. Structures are state-transition systems modeling hardware or software systems. The fundamental structures are Kripke structures (unit-delay structures, temporal structures) which may be derived from finite state machines. Our basic model for real-time systems is the interval structure, i.e., a state transition system with 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. One clock tick is the lowest granularity for the time modeling. To expand interval structures by a possibility for communication, we have extended 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. For instance, input-insensitive edges carry the formula $true$. [Figure 1] shows an I/O-interval structure with two states and one transition. The transition is enabled indeterministically between 1 to 3 time steps after the structure has entered the top state. Since this transition is labeled with the input condition $\neg i$ it may only be activated if this condition is true up to the activation time. The new value of the signal $a$ is assigned at the time of the transition activation.

The semantics of interval structures is defined over runs. A run is a sequence of configurations. A configuration is an association of an interval structure state with a clock value: $g \in S \times \mathbb{N}_0$. For the configurations of a run $r = (g_0, g_1, \dots)$ holds either:

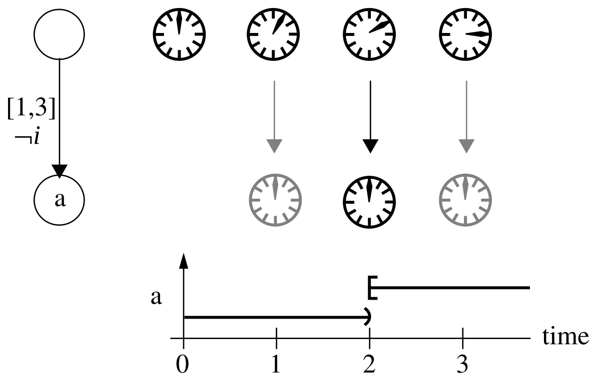


***Figure 1:*** *Example I/O-interval structure*

- that the system remains in its state: $g_i = (s_i, v_i)$ and $g_{i+1} = (s_i, v_i + 1)$ and $v_i <$ maximal delay time
- or the system changes its state according to the transition relation and the corresponding delay times: $g_i = (s_i, v_i)$ and $g_{i+1} = (s_{i+1}, 0)$ and the transition $(s_i, s_{i+1})$ is labeled with the delay time $v_i + 1$ .

This definition guarantees, that exactly one time step passes between two adjacent configurations. The semantics of I/O-interval structures is more complex and may be found in [RK99].

Properties to check are specified by temporal logic formulas. CCTL is a temporal logic extending CTL [CES83] with quantitative bounded temporal operators. Two new temporal operators are introduced to ease the specification of timed properties. It is used to describe real-time specifications. The syntax of CCTL is shown in the following definition, where $p \in P$ is an atomic proposition, $a \in I\!N$ and $b \in I\!N \cup \{\infty\}$ are time bounds.

$$\varphi := \begin{cases} p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \varphi \leftrightarrow \varphi \\ \mid \mathsf{EX}_{[a]}\varphi \mid \mathsf{EF}_{[a,b]}\varphi \mid \mathsf{EG}_{[a,b]}\varphi \mid \mathsf{E}(\varphi\ \mathsf{U}_{[a,b]}\psi) \mid \mathsf{E}(\varphi\ w\mathsf{U}_{[a,b]}\psi) \\ \mid \mathsf{E}(\varphi\ \mathsf{B}_{[a,b]}\psi) \mid \mathsf{E}(\varphi\ w\mathsf{B}_{[a,b]}\psi) \mid \mathsf{E}(\varphi\ \mathsf{C}_{[a]}\psi) \mid \mathsf{E}(\varphi\ \mathsf{S}_{[a]}\psi) \\ \mid \mathsf{AX}_{[a]}\varphi \mid \mathsf{AF}_{[a,b]}\varphi \mid \mathsf{AG}_{[a,b]}\varphi \mid \mathsf{A}(\varphi\ \mathsf{U}_{[a,b]}\psi) \mid \mathsf{A}(\varphi\ w\mathsf{U}_{[a,b]}\psi) \\ \mid \mathsf{A}(\varphi\ \mathsf{B}_{[a,b]}\psi) \mid \mathsf{A}(\varphi\ w\mathsf{B}_{[a,b]}\psi) \mid \mathsf{A}(\varphi\ \mathsf{C}_{[a]}\psi) \mid \mathsf{A}(\varphi\ \mathsf{S}_{[a]}\psi) \end{cases} \tag{1}$$

All temporal operators are preceded by a run quantor (A universal, E existential), i.e. each temporal operator refers to one run and the quantor determines if the temporal operator is true for every run (universal quatification) or for one run (existential quantification) starting in the actual configuration. [Table 1] gives an informal description of the temporal operators

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 time bound, it is implicitly set to one. The complete semantics of CCTL is given in [RK99]. For instance, the semantics of the EF-operator is formally defined through:

$$\mathfrak{I}, g_0 \models \mathsf{EF}_{[n]}\varphi \quad :\Leftrightarrow \quad \text{there ex. a run } r = (g_0, \ldots) \text{ and an } i \leq n \text{ such that } \mathfrak{I}, g_i \models \varphi \quad (2)$$

$\mathfrak{I}$ is an interval structure, $g_0$ is a configuration of $\mathfrak{I}$ and $\models$ is the model relation. RAVEN can automatically investigate if $\mathfrak{I}, g_0 \models \varphi$ holds for all initial configurations $g_0$, i.e. if the given structure satisfies the given specification.

| | |
|---|---|
| $\mathsf{X}_{[a]}\varphi$ | The formula $\varphi$ has to hold after exactly $a$ time steps. |
| $\mathsf{F}_{[a,b]}\varphi$ | The formula $\varphi$ has to hold at least once within the interval $[a, b]$. |
| $\mathsf{G}_{[a,b]}\varphi$ | The formula $\varphi$ has to hold at all time steps of the interval $[a, b]$. |
| $\varphi\ \mathsf{U}_{[a,b]}\psi$ | The formula $\psi$ has to become true within the interval $[a, b]$ and all time steps before, the formula $\varphi$ has to be valid. |
| $\varphi\ w\mathsf{U}_{[a,b]}\psi$ | If $\psi$ becomes true within the interval $[a, b]$ then $\varphi$ has to be true for all preceding time steps. Otherwise $\varphi$ has to hold up to time $b$. |
| $\varphi\ \mathsf{B}_{[a,b]}\psi$ | If $\psi$ becomes true within the interval $[a, b]$ then $\varphi$ has to be valid at one time instance before this event. Otherwise $\varphi$ has to be valid at least once up to the time $b$. |
| $\varphi\ w\mathsf{B}_{[a,b]}\psi$ | If $\psi$ becomes true within the interval $[a, b]$ then $\varphi$ has to be valid before this event. Otherwise there is to condition to $\varphi$. |
| $\varphi\ \mathsf{C}_{[a]}\psi$ | If the formula $\varphi$ is true on the current run up to the time $a-1$ then the formula $\psi$ has to hold at time $a$. |
| $\varphi\ \mathsf{S}_{[a]}\psi$ | From time zero up to time $a-1$ the formula $\varphi$ has to hold and at time $a$ the formula $\psi$ has to become valid. |

***Table 1.*** *Informal description of the temporal operators*

## 4 Architecture

The main tasks of RAVEN after parsing the input file is the construction of the MTB-DDs for each process and the composition and synthesis of the MTBDD for the complete system transition relation. The resulting MTBDD is then used for checking specifications and for answering timing queries. After the composition, RAVEN can be switched to an interactive mode allowing the user to manipulate his specifications and queries and to add new ones. The architecture of RAVEN is shown in figure [Figure 2].

After calling `xraven`, the graphical user interface appears. In this window the user specifies the input file and chooses some global options. Afterwards, the RIL-compiler (RAVEN input language, see Section 3) and the composition engine are launched. If the composition is completed, RAVEN activates the window of the interactive proof manager. A screen shot showing the proof manager window, the wave-form browser and the wave-form order window is printed below. The proof manager window shows all specifications and their proof states. Also the analysis queries and their computed

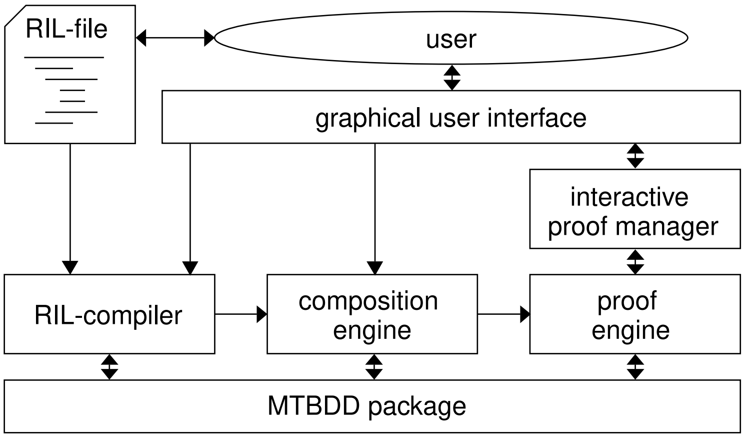


***Figure 2:*** RAVEN*'s architecture*

values are shown. New specifications or queries may be typed in this window or read in from an external file A screenshot is shown in [Figure 3].

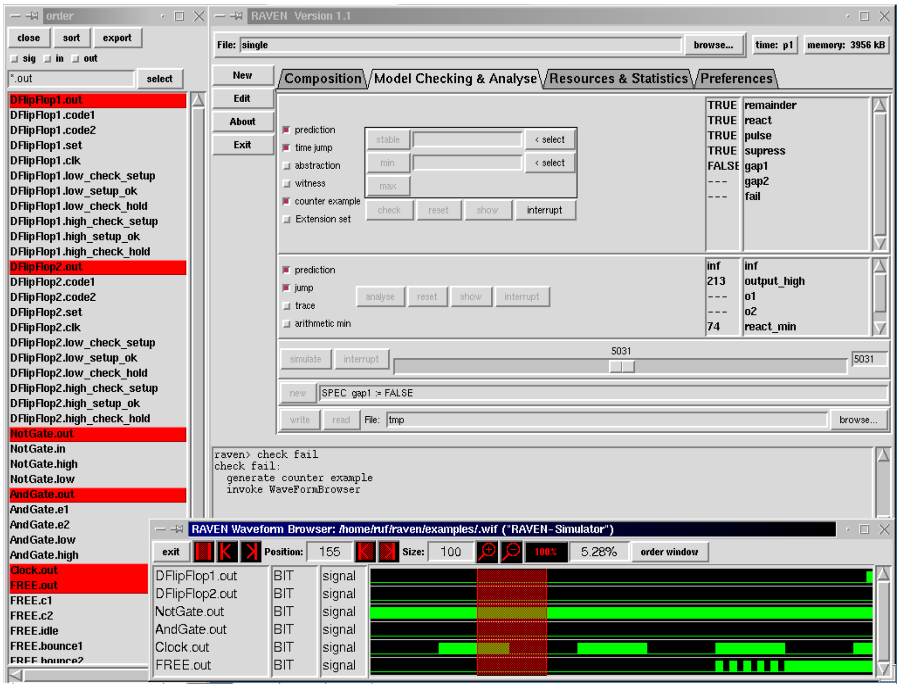


***Figure 3:*** *Screenshot of the graphical user interface of* RAVEN

## 5 The Input Format RIL

RIL (RAVEN input language) is a language for specifying networks of communicating time extended finite state machines. Each RIL module contains one I/O-interval structure. The I/O-interval structure description in [Program 1] shows exemplarily the syntax of RIL. After specifying a name for the module, the local signals are declared. RIL supports the following data types: Boolean, enumeration, ranges of natural numbers and bitvectors. The keyword **INPUT** is followed by the input signals and their connections with other parallel running modules. The **DEFINE** section introduces definitions of formulas which can be used in the remaining system description and as outputs connected to inputs of other modules. The keyword **INIT** is followed by a boolean formula describing the possible initial values of the signals. The remaining part of the module is for the definition of the state transitions. A transition consists of four parts: the start values, the input restrictions, the delay times and the target values specified by signal assignments. The first transition in the example defines a transition from the state `s=wait` to the state `s=fire`. The transition is activated 10 time steps after the `wait` state is entered and only if during the 10 time steps the input `enable` is true. The next line shows a short cut for the case that the input restriction fails. In this situation the system falls back to the state `s=wait` and the local clock is reset to zero, i.e. the system can switch to `s=fire` at the earliest after 10 time steps.

```
MODULE trigger
   SIGNAL s : { wait fire }
   INPUT  enable := controler.enable_trigger
   DEFINE action := (s=fire) & enable
   INIT   (s=wait)
   TRANS
      |- s=wait -- enable: 10 -->s := fire
                              !->s := wait
      |- s=fire --       : 1  -->s := wait
END
```

***Program 1.*** *RIL description of a timed process*

RAVEN allows the user to mix timed modules with fully synchronous modules. The transition relations of these modules are preceded by the keyword **NEXT**. Then the transition relation is defined by a (conjunctive connected) sequence of boolean formulas. A transition function is usually defined for each signal. The description in [Program 2] shows an example of a synchronous module. All state changes consume one unit time step.

Specifications are globally defined for all modules at the end of the RIL-file. Each specification gets an unique name and the corresponding formula. The syntax of the formula is equivalent to equation (1) expressed in ASCII format. The following RIL-code shows exemplarily the definition of specifications:

```
SPEC
   save := AG !( a & b )
   life := AG( req -> AF[10,20] ack )
```

RAVEN also allows the computation of critical time delays of the given system, e.g., minimal reaction times of an embedded system or the maximal wait time of a work piece in a production automation system. For these tasks the current version of

```
MODULE counter
   SIGNAL c : RANGE[0,10]
   INPUT i := trigger.action
   DEFINE carry := (c=10)
   INIT (c = 0)
   NEXT
      c' = CASE   i & c<10  :c+1
                  i & c=10  :0
                  !i        :c
           ENDCASE
END
```

***Program 2.*** RIL *description of a synchronous unit-delay process*

RAVEN supports three different algorithms invoked by the three different timing queries:

- **MIN TIME FROM** *startset* **TO** *targetset*
  This query requires two sets of configurations: the *start* and the *target* configurations. Then the corresponding algorithm computes the minimal delay time which is necessary to reach a configuration of the *target* starting in a configuration of the *start* set.
- **MAX TIME FROM** *startset* **TO** *targetset*
  This query analogously computes the maximal delay time which is necessary to reach a configuration of the *target* starting in a configuration of the *start* set.
- **MIN STABLE TIME OF** *set*
  This query requires one *set* of configurations. This algorithm computes the length of the shortest path crossing the given *set*. This computation can also be described as: finding the minimal time the given CCTL formula stays stable true directly after it changed from false to true.
- **MAX STABLE TIME OF** *set*
  This algorithm computes the length of the longest path inside the given *set*. This computation can also be described as: finding the maximal time, the given CCTL formula stays stable true.

The set of configurations are specified by CCTL formulas, e.g. if we are interested in the maximal delay time from the moment the input signal rises until the output becomes high, we may write this query as follows:

$$\textbf{MAXTIME FROM} \neg input \wedge \mathsf{EX}\ input\ \textbf{TO}\ output \tag{3}$$

Other analysis queries are:

- **MIN-/MAX VALUE OF** *variable* **IN** *set*
  This query investigates the minimal (resp. maximal) value of a *variable* within a given *set* of configurations (this set is specified as a CCTL formula)
- **MIN-/MAX VALUE OF** *variable* **FROM** *startset* **WITHIN** *timebound*
  This query investigates the minimal (resp. maximal) value of a *variable* within the specified *time interval* starting in the given set of configurations (*startset*).

- **MIN/-MAX COUNT OF** *event* **FROM** *start* **TO** *target*
  This query counts the minimal (maximal) number of occurrences of the *event* between *start* and *target*.

## 6 Case studies

We have implemented the model checking and analysis algorithms based on extended characteristic functions [RK97]. For an efficient and compact symbolic representation we have implemented the extended characteristic functions by MTBDDs [RK98a]. The algorithms are based on standard CTL model checking algorithms [BCM+90], but we have introduced new techniques (time prediction, time jump) which take advantage of the MTBDD representation for accelerating the state space traversal [RK97]. Also the analysis algorithms use this representation and the new optimization techniques [RK00]. The composition of I/O-interval structures is symbolically performed by using MTBDDs [RK98b]. We have developed two MTBDD based heuristics for minimizing the representation of the transition relation. Many details about the implementation may also be found in [RRSV01] published in this journal.

The first examined case study is the single pulser circuit [JMC94]. The used gates are modelled with specific timing behavior. Impulses on the inputs which are shorter than the delay time of a gate are suppressed. Only if an impulse stays constant at least for the delay time, the gate may change its outputs. For modeling flip-flops, we assume a setup and a hold time. If the input signals violate these timing constraints, the flip-flop remains in its actual state. The circuit is shown in [Figure 4].

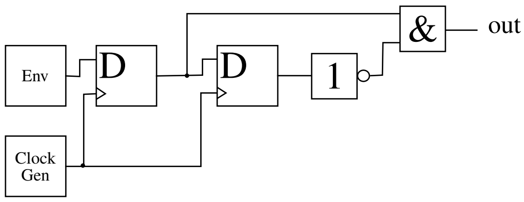


***Figure 4:*** *The circuit of the single-pulser*

[Figure 5] shows the basic gates (initial states are bold). The dotted lines represent transitions which are activated (at any time) if the condition at the "main" transition (solid lines) fails before the delay time is reached. The input signals are $a$, $a_1$ and $a_2$ and the clock input of the flip-flop is $c$. The delay times are $\delta_n$, $\delta_a$ of the NOT- and the AND-gate. The setup time of the flip-flop is $\delta_s$ and the hold time is $\delta_h$.

For modeling the AND gate, input sensitive edges have been used: If the system starts in the state *high*, and the inputs fulfil $\neg a_1 \vee \neg a_2$ for $\delta$ time units then it changes to the state *low*. If the inputs fulfill $a_1 \wedge a_2$ before $\delta$ time units are passed, the structure remains in the state *high*. Here, I/O-interval structures allow a user a very clear and compact modeling of timing constraints and communication behavior. The RIL program code for the AND-gate is shown in [Program 3].

The clock generator is modeled by an I/O-interval structure with two states (*high* and *low*) which toggles between both states every cycle-time. The environment is either a human pressing the button (which should be single pulsed) or it's a bouncing pulse.

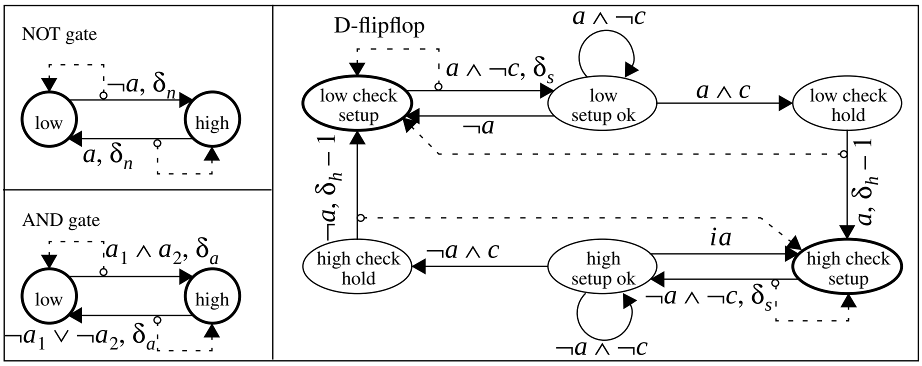


***Figure 5:*** *Basic gates*

```
MODULE AndGate
   SIGNAL s : { low high }
   INPUT  a1 := DFF1.out
          a2 := NotGate.out
   DEFINE out = (s=high)
   INIT   (s=low)
   TRANS
      |- s=low -- a1 & a2 : delta    -->s := high
                                     !->s := low
      |- s=high-- !(a1 & a2) : delta -->s := low
                                     !->s := high
END
```

***Program 3.*** RIL *description of a timed process*

The specification checks that an output signal appears if the input stays high long enough:

$$spec1 := \mathsf{AG}(\neg Env.out \to \mathsf{EX}(\mathsf{A}(Env.out\ \mathsf{C}_{[2\delta_c + \delta_h]}\mathsf{AF}_{[\delta_a - \delta_h]}And.out))) \tag{4}$$

The following specification verifies that the output stays high for one cycle period and then changes to low for a further cycle. Afterwards it remains low at least until the input becomes high:

$$\begin{aligned} spec2 &:= \mathsf{AG}(\neg And.out \to \mathsf{EX}(And.out \to \mathsf{A}(And.out\ \mathsf{C}_{[2\delta_c + \delta_n]}tmp1))) \\ tmp1 &:= \mathsf{AG}_{[2\delta_c - 1]}\neg And.out \wedge \mathsf{A}((\neg And.out)w\mathsf{U}Env.out) \end{aligned} \tag{5}$$

The next example is the arbitration mechanism of a bus protocol. We modeled the J1850 protocol arbitration [SAE95] which is used in on- and off-road vehicles. The protocol is a CSMA/CR protocol. Every node listens to the bus before sending (carrier sense, CS). If the bus is free for a certain amount of time, the node starts sending. It may happen that two or more nodes simultaneously start sending (multiple access, MA). Therefore, every node listens to the bus and compares the received signals to the send signals. If they differ, it starts arbitration (collision resolution, CR) and waits until the bus is free again. A sender distinguishes between two sending modes, a passive and an active mode. Active signals override passive signals on the bus. Succeeding bits are alternately send actively and passively. The bits are encoded by a variable pulse width: a passive zero has a pulse width of $64\mu sec$, a passive one bit takes $128\mu sec$, an active zero bit takes $128\mu sec$ and an active one bit takes $64\mu sec$. The bus is simply the union of all actively send signals. The arbitration is a bit-by-bit arbitration, since a (passive/active) zero shadows a one bit. Before sending the first bit, the nodes send an SOF (start of frame) signal, which is active and takes $200\mu sec$. In [Figure 6] some examples of arbitration are shown. We assume an exact frame length of 8 bits. After sending the last bit, the sender sends a passive signal of $280\mu sec$, the end of frame (EOF) signal.

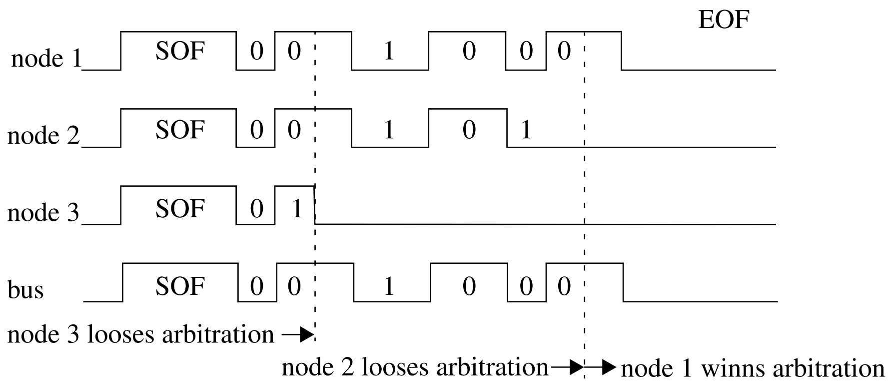


***Figure 6:*** *Some examples of arbitration*

One bus node is modeled by two sub-modules: a sender/receiver and a counter (see [Figure 7]). Initially, all modules are in their initial states. If a node decides to send (nondeterministically) the sender/receiver listens to the bus. If the bus stays low for $\delta_{CS}$ ($300\mu sec$) time units, the module changes to the SOF state. The counter is triggered by the continue high/low states of the sender. After sending the SOF signal, the sender sends alternately passive and active one and zero bits. If the bus becomes active while sending a passive bit, the sender/receiver changes to the CS state and tries sending again later. In the initial state, the counter module sets the $count$ signal to high to indicate that the package is completely send.

A further case study is the Karlsruher production cell [LL94]. This is a more complex system consisting of two robot arms, a robot rotary table, an elevating rotary table, a feed belt, a deposit belt, a press, and the central controlling unit. All physical components are modelled with delay times representing the motion of the physical elements (e.g. the expansion of a robot arm). The controlling unit is modelled by an synchronous process. Details of the examined systems and their modeling with I/O-interval structures are given in [RK99]. Experimental results to these case studies are presented in [Table 2] of Section 7.

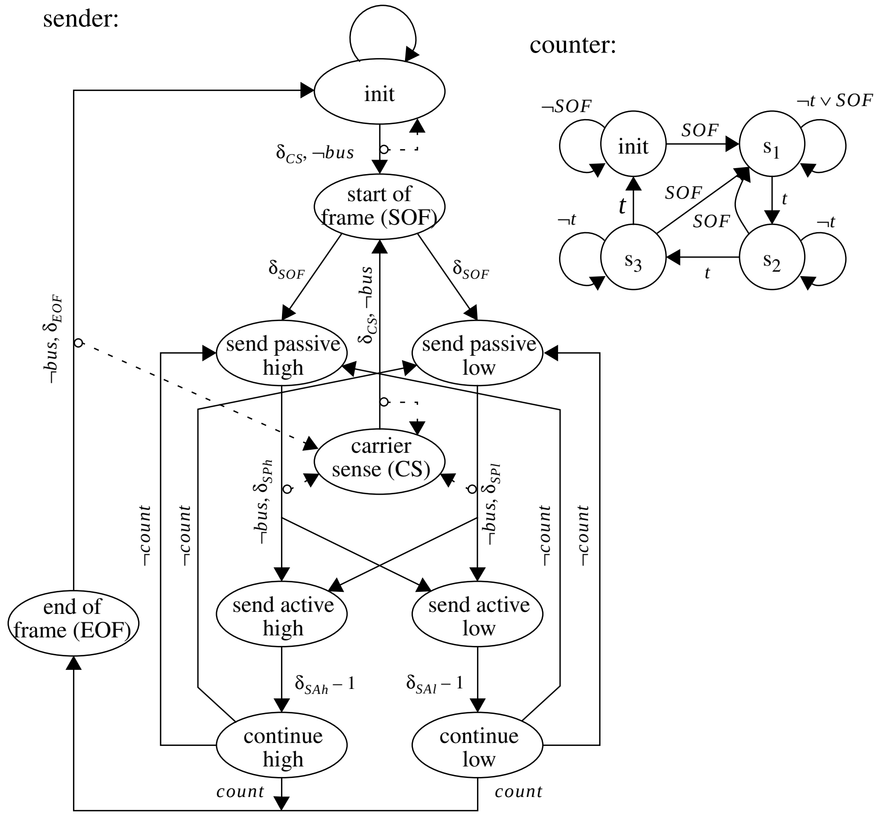


***Figure 7:*** *Two submodules modeling one bus node*

In another case study we have examined a holonic material transport system. This system consists of an input and an output station for workpieces and three processing machines with input and output buffers. The machines are supplied by three autonomous transport vehicles. This case study and the experimental results obtained by using RAVEN are presented in [FMPR01].

## 7 Experimental results

In this section we will investigate our tool in comparison to other free available model checking tools. We compare our approach to SMV (a CTL model checker for finite state machines [McM93]) and KRONOS (a TCTL model checker for timed automata [DOTY96]). The translation of interval structures to timed automata is shown in [Figure 8]. The clock has to be reset explicitly at each transition. The maximal state time has been formalized using a state invariant.

The example we examine is a system consisting of several communicating structures. A simple reader/writer system, where the modules access a shared memory. After writing, the writer module signals the reader processes, that they can start their work on the memory. All reader run in parallel. The runtimes shown in [Figure 9] contain the composition and the checking time. The left part of the [Figure 9] shows

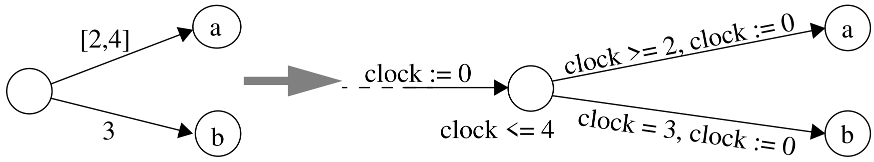


***Figure 8:*** *Timed Automaton modeling an interval structure*

another simple example composed by several toggling structures. A toggling structure consists of two states and two transitions connecting these states. The delay times on the transitions are in the range from 300 to 6000. All delay times are different.

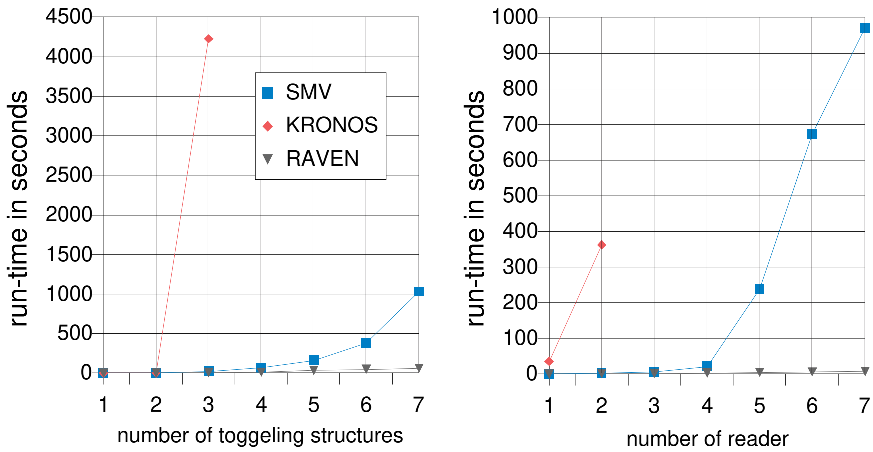


***Figure 9:*** *Runtime and memory comparison of SMV, KRONOS and* RAVEN

Timed automata are a very detailed formalism for describing timed systems. Due to the complex model checking algorithms for dense time models implemented in KRONOS, the run-times are worse than the runtimes of RAVEN. But often it is sufficient to model systems with discrete time, e.g. fully synchronous systems, systems with a central controlling unit and timed environment or systems with a shared bus. Furthermore, the discrete time model can be seen as an abstraction technique decreasing the accuracy of the model but accelerating the model checking algorithms.

Our translation of I/O-interval structures to finite state machines for SMV preserve the model behavior. Since SMV is developed for unit-delay systems RAVEN outperforms SMV for models with long delay times.

The last table compares the analysis algorithms of RAVEN with the analysis algorithms of VERUS [CCMM96]. We have compared the runtimes for one **MINTIME** and one **MAXTIME** computation. In the single pulser example we computed the minimal and the maximal length of the output impulse. In the J1850 example we checked the minimal and the maximal delay time until a node will leave the sending mode. In the production cell we were interested in the minimal and the maximal time elapsing until the first work piece leaves the cell. For the VERUS runtimes we tried various options and choose the best results. The corresponding runtimes are printed in [Table 2].

The runtimes of RAVEN in the table with (RAVEN opt) and without optimizations seems to show, that the developed optimization techniques cause only a tiny speedup. But the runtimes shown in the table contain besides the analysis 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 queries (as computed in the examples), then the fraction of composition time to analysis time will shrink and the optimizations will cause a larger speedup.

| runtimes in seconds | single pulser | J1850 | production cell |
|---|---|---|---|
| VERUS | 119.23 | -[a] | -[b] |
| RAVEN | 13.15 | 790.38 | 1766.65 |
| RAVEN opt | 12.90 | 177.63 | 1584.34 |

***Table 2.*** *Comparison of RAVEN and VERUS*

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

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

## 8 Conclusion

In this paper we have presented a tool for the analysis and formal verification of real-time systems: RAVEN. The systems are described by networks of communicating I/O-interval structures. These structures provide timed transitions for modeling real-time systems and the labeling of the transitions with input restrictions for modeling interaction between parallel working components. After the composition of the structures RAVEN performs the model checking and the analysis on the resulting interval structure automatically.

Due to the symbolic representation of the structures and sets of configurations with extended characteristic functions (implemented by MTBDDs), 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 extended characteristic functions) techniques like time prediction and time jumps accelerate the model checking and analysis algorithms. We have shown the applicability of our tool by a variety of real-world examples described on various levels of abstraction.

**Acknowledgement**

This work is sponsored by the German Research Grant (DFG) in the SPP SoftSpec, Project GRASP.

## References

[ACD90] R. Alur, C. Courcoubetis, and D.L. Dill. Model Checking for Real-Time Systems. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 414–425, Washington, D.C., June 1990. IEEE Computer Society Press.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: $10^{20}$ States and Beyond. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–33, Washington, D.C., June 1990. IEEE Computer Society Press.

[BLL+95] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. {sc Uppaal — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, Oct 1995.

[BMPY97] M. Bozga, O. Maler, A. Pnueli, and S. Yovine. Some progress in the symbolic verification of timed automata. In O. Grumberg, editor, *Conference on Computer Aided Verification (CAV)*, volume 1254 of *Lecture Notes in Computer Science*, pages 179–190. Springer Verlag, June 1997.

[CCM97] S. Campos, E. Clarke, and M. Minea. The verus tool: A quantitative approach to the formal verification of real-time systems. In O. Grumberg, editor, *Conference on Computer Aided Verification (CAV)*, volume 1254 of *Lecture Notes in Computer Science*, pages 452–455. Springer Verlag, June 1997.

[CCMM96] S. Campos, E.M. Clarke, W. Marrero, and M. Minea. Verus: A tool for quantitative analysis of finite-state real-time systems. Technical Report CMU-CS-96-159, Carnegie Mellon University, August 1996.

[CES83] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1983.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool {KRONOS}. In *Hybrid Systems III*, volume LNCS. Springer, 1996.

[EMSS92] E.A. Emerson, A.K. Mok, A.P. Sistla, and J. Srinivasan. Quantitative Temporal Reasoning. *Journal of Real-Time Systems*, 4:331–352, 1992.

[FGK96] J. Frößl, J. Gerlach, and T. Kropf. An Efficient Algorithm for Real-Time Model Checking. In *European Design and Test Conference (EDTC)*, pages 15–21, Paris, France, March 1996. IEEE Computer Society Press (Los Alamitos, California).

[FMPR01] S. Flake, W. Müller, U. Pape, and J. Ruf. Analyzing timing constraints in felixible manufacturing systems. In *Intelligent Manufacturing Systems*, Dubai, March 2001.

[JMC94] S.D. Johnson, P.S. Miner, and A. Camilleri. Studies of the single pulser in various reasoning systems. In T. Kropf and R. Kumar, editors, *International Conference on Theorem Provers in Circuit Design (TPCD)*, volume 901 of *Lecture Notes in Computer Science*, pages 126–145, Bad Herrenalb, Germany, September 1994. Springer-Verlag. published 1995.

[KR97] T. Kropf and J. Ruf. Using MTBDDs for discrete timed symbolic model checking. In *European Design and Test Conference (EDTC)*, pages 182–187, Paris, France, March 1997. IEEE Computer Society Press (Los Alamitos, California).

[LL94] C. Lewerentz and T. Lindner. Case study 'production cell', a comparative study in formal software development. Technical report, FZI-Publication, FZI Karlsruhe, January 1994.

[McM93] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[RK97] J. Ruf and T. Kropf. Symbolic model checking for a discrete clocked temporal logic with intervals. In E. Cerny and D.K. Probst, editors, *Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 146–166, Montreal, Canada, October 1997. IFIP WG 10.5, Chapman and Hall.

[RK98a] J. Ruf and T. Kropf. Using MTBDDs for discrete timed symbolic model checking. *Multiple-Valued Logic – An International Journal*, 1998. Special Issue on Decision Diagrams.

[RK98b] Jürgen Ruf and Thomas Kropf. Using MTBDDs for composition and model checking of real-time systems. In *FMCAD 1998*. Springer, November 1998.

[RK99] J. Ruf and T. Kropf. Modeling and checking networks of communicating real-time processes. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods (CHARME)*, volume 1703 of *Lecture Notes in Computer Science*, pages 256–279, Bad Herrenalb, Germany, September 1999. Springer-Verlag.

[RK00] J. Ruf and T. Kropf. Analyzing real-time systems. In P. Marwedel and I. Bolsens, editors, *Design Automation and Test in Europe (DATE)*, pages 243–248, Los Alamitos, CA, March 2000. IEEE Compter Society Press.

[RRSV01] W. Reif, J. Ruf, G. Schellhorn, and T. Vollmer. Correctness of efficient real-time model checking. *Journal on Universal Computer Science (www.jucs.org)*, 2001.

[SAE95] SAE. J1850 class B data communication network interface. *The Engeneering Society For Advancing Mobility Land Sea Air and Space*, Otober 1995.