RAVEN: Real-Time Analyzing and Verification Environment[1]

Jürgen Ruf
Wilhelm-Schickard-Institute
University of Tübingen, Sand 13, 72076 Tübingen, Germany
ruf@informatik.uni-tuebingen.de
http://www-ti.informatik.uni-tuebingen/~ruf/raven.html

1 Introduction

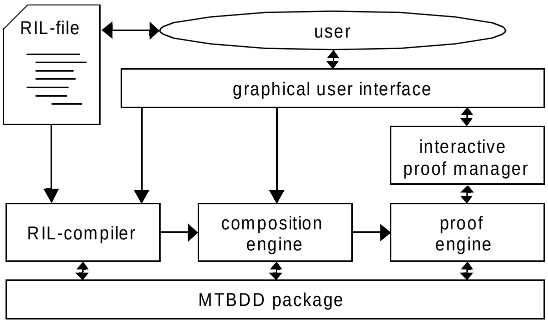
Formal verification has become an important task in the design of systems. Techniques like symbolic model checking have reached industrial applicability. These techniques are well suited for fully synchronous systems modeled with qualitative time (clock cycles). If systems are embedded in a real-time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

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 (I/O-interval structure [1,2]). The properties are specified in the quantitative temporal logic CCTL. **RAVEN** provides different algorithms to determine critical delay times of the design. The queries for the analysis capabilities cover minimum, maximum and maximal stability computations. **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 wave form browser. Moreover, **RAVEN** offers additional checks. For instance, it can detect dead- and live locks and visualizes traces to these "locks" in its integrated wave form browser tool. It is also possible to generate random simulations of the composed system.

RAVEN uses MTBDDs [5,6] for a symbolic representation of the systems [8]. This data structure results in a compact system representation and efficient verification algorithms. All algorithms are alternatively implemented for an ROBDD [7] representation.

2 Architecture and information flow

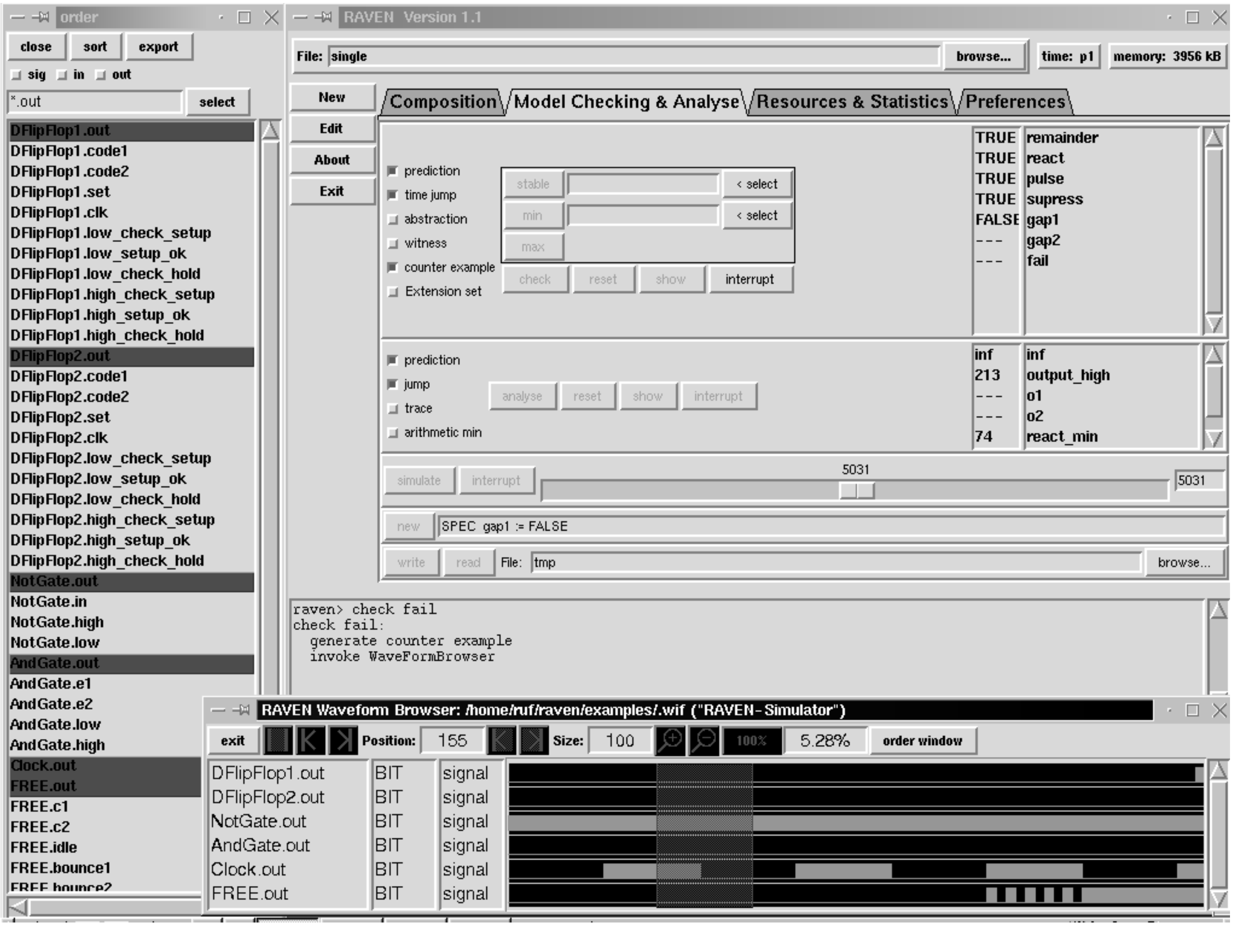
The main tasks of **RAVEN** after parsing the input file, is the construction of the MTBDDs for each process, the composition and synthesis of the MTBDD for the system transition relation. The resulting MTBDDs are then used for checking specifications and for answering timing queries. After the composition, **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 the figure on the right.



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 started. After the composition is completed, **RAVEN** activates the window of the interactive proof manager. A screen shot showing the proof manager window, the

[1] This work is sponsored by the German Research Grant (DFG-Project GRASP)

wave form browser and the wave-form order window is printed below. The proof manager window shows all specifications and their proof state. Also the analysis queries and their computed values are shown. New specifications or queries may be typed in this window or read in from an external file.



□ sig □ in □ out

*.out

DFlipFlop1.out
DFlipFlop1.code1
DFlipFlop1.code2
DFlipFlop1.set
DFlipFlop1.clk
DFlipFlop1.low_check_setup
DFlipFlop1.low_setup_ok
DFlipFlop1.low_check_hold
DFlipFlop1.high_check_setup
DFlipFlop1.high_setup_ok
DFlipFlop1.high_check_hold
DFlipFlop2.out
DFlipFlop2.code1
DFlipFlop2.code2
DFlipFlop2.set
DFlipFlop2.clk
DFlipFlop2.low_check_setup
DFlipFlop2.low_setup_ok
DFlipFlop2.low_check_hold
DFlipFlop2.high_check_setup
DFlipFlop2.high_setup_ok
DFlipFlop2.high_check_hold
NotGate.out
NotGate.in
NotGate.high
NotGate.low
AndGate.out
AndGate.e1
AndGate.e2
AndGate.low
AndGate.high
Clock.out
FREE.out
FREE.c1
FREE.c2
FREE.idle
FREE.bounce1
FREE.bounce2

File: single

time: p1

memory: 3956 kB

☒ prediction
☒ time jump
□ abstraction
□ witness
☒ counter example
□ Extension set

TRUE	remainder
TRUE	react
TRUE	pulse
TRUE	supress
FALSE	gap1
---	gap2
---	fail

☒ prediction
☒ jump
□ trace
□ arithmetic min

inf	inf
213	output_high
---	o1
---	o2
74	react_min

5031

5031

SPEC gap1 := FALSE

File: tmp

```
raven> check fail
check fail:
  generate counter example
  invoke WaveFormBrowser
```

Position: 155

Size: 100

100%

5.28%

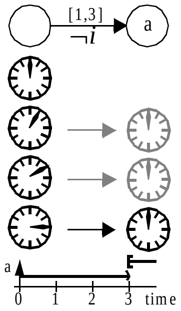
DFlipFlop1.out	BIT	signal
DFlipFlop2.out	BIT	signal
NotGate.out	BIT	signal
AndGate.out	BIT	signal
Clock.out	BIT	signal
FREE.out	BIT	signal

In the following section, we introduce the **RAVEN** input language **RIL**. First we briefly describe I/O-interval structures since each **RIL**-module represents one I/O-interval structure. Afterwards we introduce the temporal logic CCTL and describe the syntax of CCTL in **RIL** together with the analysis querying syntax. In Section 4 we present the modeling of an timed bus arbitration protocol (J1850) and the implementation in **RIL**.Then we introduce step by step the windows of the graphical user interface in Section 5. Together with the graphical user interface, **RAVEN** supports the user by showing counter examples or simulation runs in a wave form browser. This tool is presented in Section 6. For advanced users or for batch jobs, **RAVEN** provides a textual user interface which is described in Section 7. Section 8 shows the BNF grammar of **RIL**.

3 The input format **RIL**

RIL (**RAVEN** input language) is a simple format for specifying networks of communicating time extended finite state machines (I/O-interval structure), property specifications as temporal logic formulas and analysis queries. Each **RIL** module contains one I/O-interval structure. The structures are defined as state transition graphs. The transitions are labeled with time intervals and input restrictions. Inputs are functional connected to output variables of other modules. The following paragraph introduces the I/O-interval structures.

3.1 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. Our 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. 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* .

The I/O-interval structure of the figure above may be expressed by the following **RIL**-description:

```
MODULE structure
  SIGNAL a : BOOL
  INPUT i := other_module.output
  DEFINE
    s0 := !a
    s1 := a
  INIT s0
  TRANS s0 & !i & s1’ :[1,3]
END
```

An I/O-interval structure is introduced by the keyword **MODULE** (alternatively the keywords **FSM** or **MODEL** may be used) followed by the module name. The next part of the module description introduces the signals. Since the actual version only supports Boolean signals, the type identifier may be left out. The definition of several signals is a white space separated list of signal definitions. Each module has to have at least one signal.

The next section defines the input signals of a module. These definitions have the syntax (details about the BNF syntax may be found in Section 8):

$$\text{ident} := \text{formula} \qquad (1)$$

Input definition is a Boolean formula over the signals (or definitions resp. states) of other modules (it is not allowed to use local signals or other input variables here). The identifier of other modules are specified by the module name and the signal name separated by a dot. This is the only interface of a module to other modules, i.e. the remaining module description may only access local identifiers. The input definition may access all signals (states, definitions) of all other modules, i.e. there exist no hiding of local variables.

The following section describes definitions. A definition is an abbreviation of a boolean equation by an identifier. These definitions have the same syntax like the input definition with the exception, that only local signals, input signals and local definitions may be used in the formula. The section is opened by the keyword **DEFINE**. This keyword is followed a white space seperated list of assignments:

$$\text{ident} := \text{formula} \qquad (2)$$

All identifiers used for signals, inputs and definitions have to be unique in a module. Different modules may use identifiers with the same name.

The abbreviations of the define section may be used to define the states of an I/O-interval structure. Therefore the section may be preceded by the keyword **STATE** instead of **DEFINE**. Since states (in the symbolic representation used in **RAVEN**) are composed by the valuations of all signals, it is important to ensure that different states have different valuations of the signals. As an shortcut for the definition of states, the following syntax will also be accepted by **RAVEN**:

ident **:= {** *signal-list* **}** (3)

The signals enumerated in the comma separated list (*signal-list*) are interpreted by **RAVEN** as positive signals, while non-existing signals are interpreted as negative. For instance, if we have a module with the signals i1, i2 and i3, then the following definition:

`idle := {i1}` (4)

is equivalent to

`idle := i1 & !i2 & !i3` (5)

The signal list may only contain local signals, but no input signals or definition identifiers. In contrast to this signal enumeration, the conventional definition syntax may use any local identifiers in the definition formula.

The next part of the module definition specifies the initial states of a module. This definition is a Boolean formula. In the example, it is the state s0. If several states are initial states, they may be connected by the Boolean disjunction. It is also possible to define the initial values of every signal, e.g.

INIT `i1 = 0 & i2 = 1 & i3 = 0` (6)

If no initial section appears in the module, all states are initial. **RAVEN** sets the clock of all initial states implicitly to zero, i.e. there exist no possibility to specify initial states with an other clock value than zero.

The last part of the module description is the specification of the state transition possibilities. This part is introduced by the keyword **TRANS**. The transitions are specified by a (white space separated) list of boolean formulas extended by time bounds. The formulas may use all local signals, all input signals and all definitions. Additionally there exist a quote operator which allows the formula to access the value of a signal (input, definition) in the successing state. An example is shown in the module description above. If the quote operator is applied to a definition (or an input signal), all signals of the definition are quoted.

The time bounds are comma separated lists of intervals or single expressions over natural numbers. All specified values are interpreted as possible delay times. The expressions may use the addition, the substraction and the multiplication. There exists also the possibility to use global time constants (see next subsection). All specified values have to be greater than zero.

RAVEN allows to mix timed modules with full 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 with no timing information. The usual way to specify these modules is by transition functions for each signal. The following description shows an example of a synchronous module. All state changes take implicitly one unit time step.

```
MODULE sync
  SIGNAL
    a : BOOL
    b : BOOL
  INPUT i := om.output
  INIT a & b
  NEXT a′ = i & b
       b′ = i & a
END
```

The module description is terminated by the keyword **END**.

3.2 Global definitions in RIL

Besides the specification of temporal logic formulas and the analysis queries (which are described in the following subsections), there exist two kinds of global definitions.

The fist one are time definitions. **RAVEN** can only work with constant time values, but for an easier instantiation of modules with different times there exist the possibility to define global time constants which may be used in the delay time specification of the modules. The time constant definition has to be the first part of a **RIL**-description. This part is introduced by the keyword **`TIME`**. Then there follows a white space separated list of definitions of the form:

$$\text{ident} \;\texttt{:=}\; \text{nat-expression} \tag{7}$$

The expression over natural numbers may contain the addition, the substraction, the multiplication, constant values and other (preceeding) time constants.

The second kind of global definitions are boolean functions associated with identifiers. This part of a **RIL**-description follows the global time definitions and has to appear before the modules are described. These global functions can be interpreted as modules without signals (and therefore without a transition relation and without states). This definition part is preceded by the keyword **`DEFINE`**. This keyword is followed by a white space separated list of definitions of the form:

$$\text{ident} \;\texttt{:=}\; \text{formula} \tag{8}$$

The Boolean formula uses other global definitions or local signals (definitions) of modules. In the latter case, the signals (definitions) are accessed by the dot operator, the first identifier specifies the module name and the second identifier represents the local signal (definition):

$$\text{ident}\texttt{.}\text{ident} \tag{9}$$

All global defined identifiers (time-constants, global definitions, module names, specification names and analysis query names) have to be unique, but the same identifiers may be used inside a module definition.

3.3 Property specification in RIL

CCTL [1] is a temporal logic extending CTL with quantitative bounded temporal operators. It is used to describe the real-time specifications. Two new temporal operators are introduced to ease the specification of timed properties. The syntax of CCTL is shown in (10); where $p \in P$ is an atomic proposition, $a \in I\!N$ and $b \in I\!N \cup \{\infty\}$ 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 time bound, it is implicitly set to one. The semantics of CCTL is given in [2].

$$\varphi := \begin{cases} p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \varphi \leftrightarrow \varphi \\ \mid \mathsf{EX}_{[a]}\varphi \mid \mathsf{EF}_{[a,b]}\varphi \mid \mathsf{EG}_{[a,b]}\varphi \mid \mathsf{E}(\varphi\ \mathsf{U}_{[a,b]}\varphi) \\ \mid \mathsf{E}(\varphi\ \mathsf{C}_{[a]}\varphi) \mid \mathsf{E}(\varphi\ \mathsf{S}_{[a]}\varphi) \\ \mid \mathsf{AX}_{[a]}\varphi \mid \mathsf{AF}_{[a,b]}\varphi \mid \mathsf{AG}_{[a,b]}\varphi \mid \mathsf{A}(\varphi\ \mathsf{U}_{[a,b]}\varphi) \\ \mid \mathsf{A}(\varphi\ \mathsf{C}_{[a]}\varphi) \mid \mathsf{A}(\varphi\ \mathsf{S}_{[a]}\varphi) \end{cases} \tag{10}$$

The property specification part of a **RIL** description appears after the global definitions and the module definitions. It is introduced by the keyword **`SPEC`**. A white space separated list of specifications is following the keyword. Each CCTL formula is specified through:

$$\text{ident} \;\texttt{:=}\; \text{cctl-formula} \tag{11}$$

The formulas are build upon the signals, definitions and inputs of modules as well as the global definition names. For identifying local signal names, they are preceded by the module name and separated by a dot. Detailed information about the syntax may be found in Section 8.

3.4 Analysis queries in RIL

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 **RAVEN** supports three different algorithms:

- **MIN** requires two sets of configurations: the start and the destination configurations. A configuration is a pair if an I/O-interval structure state and a time value of the clock. Then this algorithm computes the minimal delay time which is necessary to reach a configuration of the destination starting in a configuration of the start set.
- **MAX** analogously computes the maximal delay time which may appear between a configuration of the destination starting in a configuration of the start set.
- **STABLE** requires one set of configurations. This algorithm computes the length of the longest path which do not leave the given set.

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:

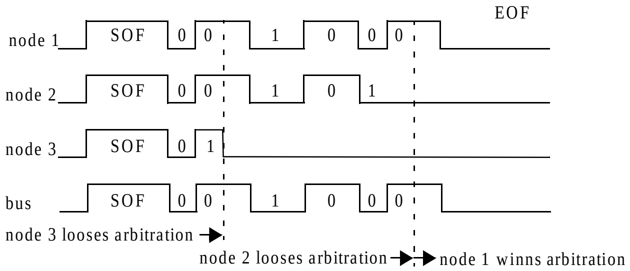
$$q := \mathsf{MAX}(\neg input \wedge \mathsf{EX}input, output) \tag{12}$$

The analysis queries appear at the end of a **RIL** file. They are introduced by the keyword **ANALYSIS**. The syntax is equivalent to the property specifications, with the exception, that the right side of a definition is a query.

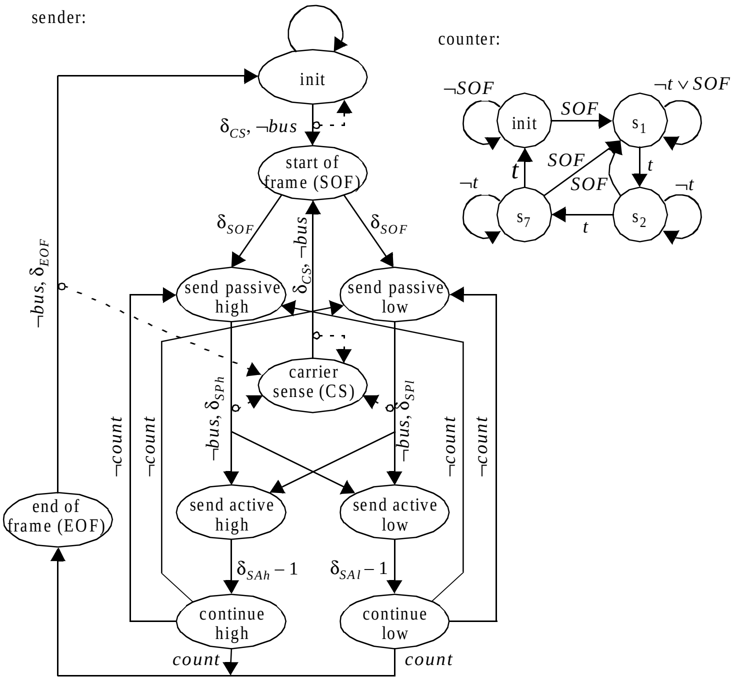
The analysis algorithms are described in [4]. A formal definition of the query operators is given in [10].

4 An example: the J1850 bus arbitration

As an case study we want to examine the arbitration mechanism of a bus protocol. We modeled the J1850 protocol arbitration [9] 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, while sending, every node listens to the bus and compares the received signals to the send signals. If they divide, it looses 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 active and passive. The bits to be send are encoded by a variable pulse width: a passive zero has a pulse width of $64\mu\mathrm{sec}$, a passive one bit takes $128\mu\mathrm{sec}$, an active zero bit takes $128\mu\mathrm{sec}$ and an active one bit takes $64\mu\mathrm{sec}$. 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 $200\mu\mathrm{sec}$. In the following figure 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 $280\mu\mathrm{sec}$, the end of frame (EOF) signal.



One bus node is modeled by two sub-modules: a sender/receiver and a counter. Initially, all modules are in their initial states. If the node decides to send (indeterministically) the sender/receiver listens to the bus. If the bus stays low for δ_{CS} time units, the module changes to the SOF state. The counter is triggered by the continue high/low states of the sender. In the initial state, the counter module sends the *count* signal. 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.



The following **RIL** program describes the J1850 arbitration as described in the diagrams.

```
TIMES
  EOF_time := 280
  SOF_time := 200
  CS_time := 300
  SP0_time := 64
  SP1_time := 128
  SA0_time := SP1_time
  SA1_time := SP0_time
  min := 8*SP0_time+EOF_time
  max := SOF_time + 8*SP1_time + EOF_time-1+
         2*(SOF_time+8*SP1_time+ EOF_time)

DEFINE
  extern_bus := node1.active | node2.active | node3.active


MODULE node1
  SIGNALS
    send : BOOL
    active : BOOL
    alternate : BOOL
    high : BOOL

  INPUTS
    BUS := extern_bus
    count := counter1.init

  STATES
    init := {}
    CS := {alternate}
    SOF := {send,high,active,alternate}
    SP0 := {send}
    SP1 := {send,high}
    SA0 := {send,active}
    SA1 := {send,active,high}
    continue := {send,active,alternate}
    EOF := {send,alternate}

INIT
  init

TRANS
  init & init’ : 1
  init & init’ & BUS : 1
  init & init’ & !BUS & BUS’ : [1,CS_time-1]
  init & SOF’ & !BUS : CS_time

  // the following three transitions cover a typical situation:
  // starting in state CS, if the BUS-signal is low for
  // CS_time time steps, then the system changes to the state SOF
  // (transition three)
  // But, if at one time point before the time CS_time the BUS
  // signal becomes high, the condition for changing to SOF fails
  // and the system remains in state CS. Now the clock is reset and
  // the BUS signal has to stay low again for CS_time time steps
  // before the system may change to state SOF.

  // condition fails at time zero
  CS & CS’ & BUS : 1

  // condition fails between time 1 and CS_time-1
  CS & CS’ & !BUS & BUS’ : [1,CS_time-1]

  // condition holds for CS-Time time steps
  CS & SOF’ & !BUS : CS_time

  // the following two transitions starting in SOF
  // create an indeterminism
  SOF & SP0’ : SOF_time
  SOF & SP1’ : SOF_time

  SP0 & !BUS & SA0’ : SP0_time
  SP0 & !BUS & SA1’ : SP0_time
  SP0 & !BUS & CS’ & BUS’ : [1,SP0_time-1]
  SP0 & BUS & CS’ : 1

  SP1 & !BUS & SA1’ : SP1_time
  SP1 & !BUS & SA0’ : SP1_time
  SP1 & !BUS & CS’ & BUS’ : [1,SP1_time-1]
  SP1 & BUS & CS’ : 1

  SA0 & continue’ : SA0_time - 1

  SA1 & continue’ : SA1_time - 1

  continue & SP0’ & !count : 1
  continue & SP1’ & !count : 1
  continue & EOF’ & count : 1
  EOF & init’ & !BUS : EOF_time
  EOF & CS’ & !BUS & BUS’ : [1,EOF_time-1]
  EOF & BUS & CS’ : 1
END

MODULE counter1
  SIGNALS
  c0 : BOOL
  c1 : BOOL

INPUTS
  start := node1.SOF
  trigger := node1.continue

STATES
  init := {}
  s0 := {c0}
  s1 := {c1}
  s2 := {c1,c0}

INIT
  init

TRANS
  init & init’ & !start : 1
  true & s0’ & start : 1
  s0 & s0’ & !trigger & !start : 1
  s0 & s1’ & trigger & !start : 1
  s1 & s1’ & !trigger & !start : 1
  s1 & s2’ & trigger & !start : 1
  s2 & s2’ & !trigger & !start : 1
  s2 & init’ & trigger & !start : 1

END

// The models of all bus nodes and counters are equivalent
// for the specifications, we assume three bus nodes



...
SPEC
  // these specifications show, that it might happen, that a node
  // which wants to send a message will never be able to do that
  not_fairnes1 := !(AG (node1.SOF -> AF node1.init))
  not_fairnes2 := !(AG (node2.SOF -> AF node2.init))
  not_fairnes3 := !(AG (node3.SOF -> AF node3.init))

  // If only one node is sending a message, it will finish this action
  // within a certain time bound
  live1 := AG(( node1.SOF & !node2.SOF & !node3.SOF ) ->
           AF[min,max] (node1.init))
  live2 := AG((!node1.SOF & node2.SOF & !node3.SOF ) ->
           AF[min,max] (node2.init))
  live3 := AG((!node1.SOF & !node2.SOF & node3.SOF ) ->
           AF[min,max] (node3.init))

  // if two nodes are sending parallel, one of both will win the
  // arbitration and will finish this action within a certain time bound
  live4 := AG(( node1.SOF & node2.SOF & !node3.SOF ) ->
           AF[min,max] (node1.init | node2.init))
  live5 := AG(( node1.SOF & !node2.SOF & node3.SOF ) ->
           AF[min,max] (node1.init | node3.init))
  live6 := AG((!node1.SOF & node2.SOF & node3.SOF ) ->
           AF[min,max] (node2.init | node3.init))

  // if three nodes are sending parallel, one of them will win the
  // arbitration and will finish this action within a certain time bound
  live7 := AG(( node1.SOF & node2.SOF & node3.SOF ) ->
           AF[min,max] (node1.init | node2.init | node3.init))

  // If one node is sending a message, the bits on the bus are
  // equivalent to the bits it sends until the node terminates the message
  cor1 := AG(( node1.SOF & !node2.SOF & !node3.SOF ) ->
           A((node1.active <-> extern_bus) U node1.init))
  cor2 := AG((!node1.SOF & node2.SOF & !node3.SOF ) ->
           A((node2.active <-> extern_bus) U node2.init))
  cor3 := AG((!node1.SOF & !node2.SOF & node3.SOF ) ->
           A((node3.active <-> extern_bus) U node3.init))

  // If two nodes sending parallel a message, the bits on the bus are
  // equivalent to the bits one node is sending
  // until the node terminates the message
  cor4 := AG(( node1.SOF & node2.SOF & !node3.SOF ) ->
           (A((((node1.active <-> extern_bus) |
           A((node2.active <-> extern_bus) U node2.init) ) &
           ((node2.active <-> extern_bus) | A((node1.active <->
           extern_bus) U node1.init) )) U (node1.init | node2.init))))
  cor5 := AG(( node1.SOF & !node2.SOF & node3.SOF ) ->
           (A((((node1.active <-> extern_bus) | A((node3.active <->
           extern_bus) U node3.init) ) & ((node3.active <-> extern_bus) |
           A((node1.active <-> extern_bus) U node1.init) )) U
           (node1.init | node3.init))))
  cor6 := AG((!node1.SOF & node2.SOF & node3.SOF ) ->
           (A((((node2.active <-> extern_bus) | A((node3.active <->
           extern_bus) U node3.init) ) & ((node3.active <-> extern_bus) |
           A((node2.active <-> extern_bus) U node2.init) )) U (node2.init |
           node3.init))))
  // If three nodes sending parallel a message, the bits on the bus are
  // equivalent to the bits one node is sending
  // until the node terminates the message
  cor7 := AG(( node1.SOF & node2.SOF & node3.SOF ) ->
           A((((node2.active <-> extern_bus) | A((  ((node1.active <->
           extern_bus) | A((node3.active <-> extern_bus) U node3.init) ) &
           ((node3.active <-> extern_bus) | A((node1.active <->
           extern_bus) U node1.init) ))  U (node1.init | node3.init)) ) &
           ( ((node1.active <-> extern_bus) | A((((node2.active <->
           extern_bus) | A((node3.active <-> extern_bus) U node3.init) ) &
           ((node3.active <-> extern_bus) | A((node2.active <-> extern_bus)
           U node2.init) ))  U (node2.init | node3.init)) ) &
           ((node3.active <-> extern_bus) | A((((node2.active <->
           extern_bus) |  A((node1.active <-> extern_bus) U node1.init) ) &
           ((node1.active <-> extern_bus) | A((node2.active <->
           extern_bus) U node2.init) )) U (node2.init | node1.init)) ))) U
           (node2.init | (node1.init | node3.init))))

ANALYSE
  // The queries check the minimal and maximal times from the
  // time a node starts sending a message until it stops
  // sending a message
  m1 := MIN(node1.init & EX node1.SOF,node1.EOF & EX node1.init)
  m2 := MAX(node1.init & EX node1.SOF,node1.EOF & EX node1.init)
  m3 := MIN(node2.init & EX node2.SOF,node2.EOF & EX node2.init)
  m4 := MAX(node2.init & EX node2.SOF,node2.EOF & EX node2.init)
  m5 := MIN(node3.init & EX node3.SOF,node3.EOF & EX node3.init)
  m6 := MAX(node3.init & EX node3.SOF,node3.EOF & EX node3.init)
```

5 The graphical user interface (gui)

After calling xraven (which is a Tcl/Tk-script using version 8.0), The **RAVEN** window appears. This window consists of three parts

- At the top there is the status information (the file name, time used for **RAVEN** and memory consumption).
- The middle part is the active part, with the context sensitive control sheets and the global buttons:

  New: exits the actual verification session and initializes a new one,

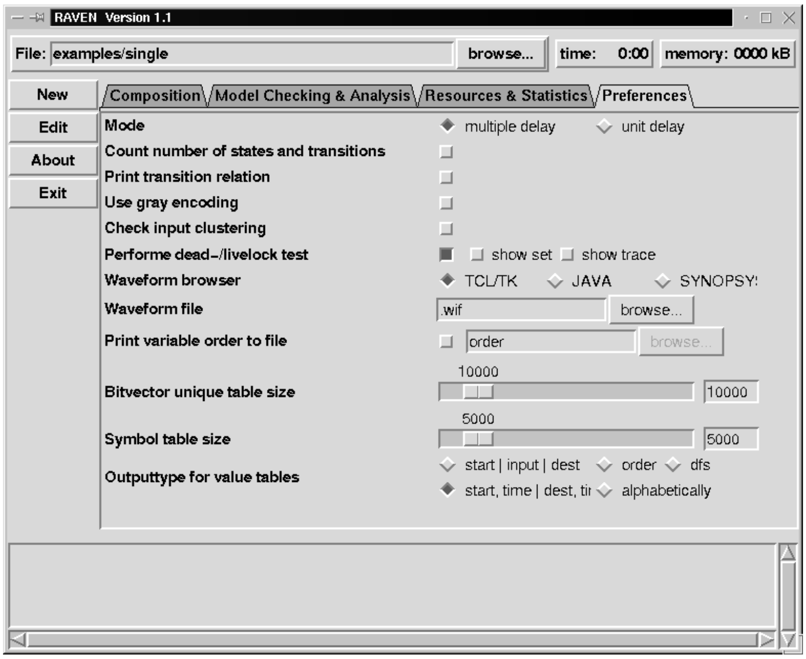
  Edit starts an external editor containing the actual file,

  About: gives some information to theVersion of **RAVEN**,

  Exit: exits the program) .
- The lower part contains the log information, which is the textual information of the communication of the graphical user interface with the **RAVEN** tool.

It is possible to start xraven with a file name of a **RIL**-description. If no file name is specified, it is possible to browse interactively for a **RIL**-file. **RAVEN** comes up with the composition-sheet activated. There exists a sheet containing global Preferences. The settings made in this sheet influence the verification session only if they are done before composition. Therefore this is the first sheet we want to describe in detail.

5.1 The preferences window



File: examples/single

time: 0:00

memory: 0000 kB

Mode ● multiple delay ◇ unit delay

Count number of states and transitions □

Print transition relation □

Use gray encoding □

Check input clustering □

Performe dead-/livelock test ■ □ show set □ show trace

Waveform browser ● TCL/TK ◇ JAVA ◇ SYNOPSY!

Waveform file .wif

Print variable order to file □ order

Bitvector unique table size 10000 10000

Symbol table size 5000 5000

Outputtype for value tables ◇ start | input | dest ◇ order ◇ dfs ● start, time | dest, tir ◇ alphabetically

The following settings may be adjusted:

Mode. **RAVEN** supports two major modes:

1. multiple delay: This mode uses MTBDDs for the transition relation representation. This mode allows the optimization techniques (time prediction and time jumps, and MTBDD minimization)
2. unit delay: This mode uses ROBDDs for the transition relation representation. Time delays are encoded by additional stutter states in the transition relation.

All model checking and analysis algorithms are implemented for both modes.

Count number of states and transitions. This option prints in the log-window the number of states and transitions before (the addition of all modules) and after the composition.

Print transition relation. This option shows a value table of the composed transition relation. It is only useful for small transition relations.

Use Gray encoding. During the composition, additional states will be introduced in the transition relation. These states have to be encoded by additional decision variables. By default, a binary encoding is used. This option enables a Gray encoding.

Check input clustering. For a correct composition, the transitions have to hold a condition called (input clustering condition [2]). This option checks if all modules satisfy this condition.

Perform dead-/live lock test. **RAVEN** may search dead- and live lock states [11]. This option enables the search algorithms. The result is printed at the log window. If the check button "show trace" is activated, **RAVEN** computes a run from an initial state to a dead-/live lock state. This run is displayed in the wave form browser.

Wave form browser. This option allows to switch between three different wave form browsers. The JAVA-browser is not longer supported, the Synopsys wave form browser (waves) is not public domain. The best choice is the tcl/tk browser which is distributed with **RAVEN** (see the following section).

Wave form file. This text entry field allows the specification of a file name to which the wave form output intermediately is printed. The corresponding browser tool is then started with this file as input.

Print variable order to file. This option may activate the printing of the variables in the order as they appear in the MTBDDs/ROBDDs. The text entry field specifies the file name to which the variables are written.

Bitvector unique table size. The bitvector unique table is important for the MTBDD reduction. This slider defines the size of this table. A larger size may lead to better run times but to a larger memory consumption.

Symbol table size. This table stores all identifiers appearing in the actual **RIL**-description. If **RAVEN** terminates with the error "symbol table overflow", this parameter should be increased.

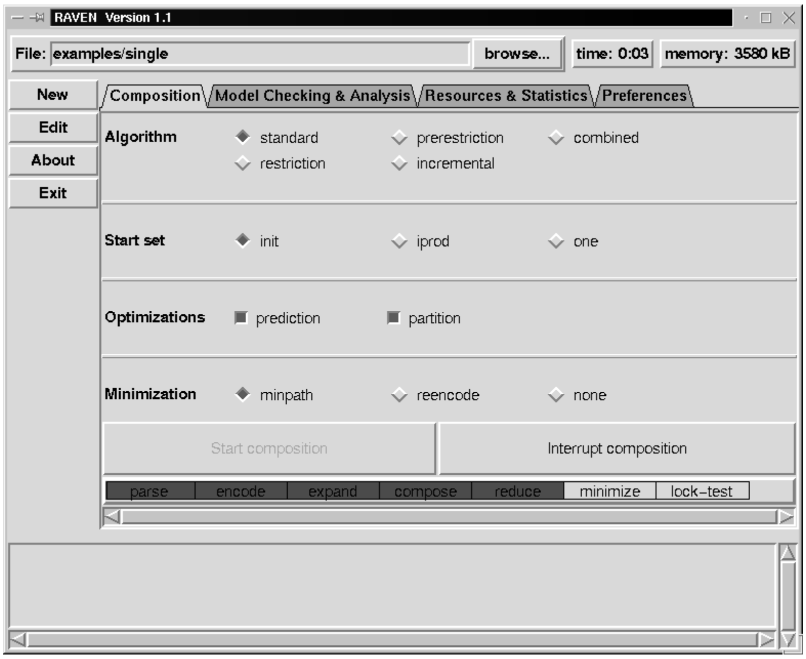
Output type for value tables. This parameter sets the order of the variables in value tables printed to the log window. Value tables appear if the option print transition relation is activated or if the option extension set is activated for model checking. There exists five types for the order:

1. start|input|dest: This option prints the variables of the start state before the inputs and at least the variables of the destination set. This option is interesting for models before composition.
2. order: the variables appear in their variable order they have in the MTBDD.
3. dfs: The variables appear as they are found in a depth first search in the MTBDD.

4. start,time|dest,time: The start variables appear at first, then the additional introduced time encoding variables, then the destination state variables and at least the time encoding variables of the destination states. This is the standard option for the composed transition relation (there exist no input variables anymore).
5. alphabetically

5.2 The composition window

After setting the global options, the next step for a **RAVEN** user is the composition. There exist several different composition algorithms and composition options which may be set in this window.



File: examples/single

time: 0:03

memory: 3580 kB

Algorithm ● standard ○ prerestriction ○ combined ○ restriction ○ incremental

Start set ● init ○ iprod ○ one

Optimizations ■ prediction ■ partition

Minimization ● minpath ○ reencode ○ none

Algorithm. There exist five different composition algorithms:

1. standard: This is the standard algorithm expansion-composition-reduction. This is the preferred algorithm for small or medium sized systems in the multiple-delay mode.
2. restriction: This algorithm computes the set of reachable states after composition and restricts the set of states in every traversal step in the checking/analysis algorithms by this set. Since the standard algorithm in the multi-delay mode always handles only the reachable states, this option has no effect in this mode.
3. prerestriction: This algorithm computes the set of reachable states after composition and cuts the transition relation with it. This option works well for small and medium sized systems in the unit-delay mode. This option has only an effect to the composition in the multiple-delay mode, if not the initial set as start set is chosen (see below). In this case this or the incremental options are recommended.
4. incremental: This algorithms builds the composition structure incrementally by traversing the original structures. The resulting structure contains only the reachable states and transitions. This algorithm should be chosen for large systems in the unit-delay mode.
5. combined: This algorithm performs the composition and the reduction in one combined step. This is the preferred algorithm for large systems in the multiple-delay mode. Since the unit-delay mode needs no reduction, **RAVEN** puts a warning and uses the incremental algorithm.

Start set. The reduction operation of the multiple-delay mode may start in different sets of states:

1. init: start in the initial states. Practical tests have shown, that this set leads in most cases to faster composition results.
2. iprod: This option sets the start set to the product set of all original interval structure sets.
3. one: This option sets the start set to the product of at least one interval structure state and all possible stutter states.

Optimization. These two options may accelerate the composition in the multiple-delay mode in many cases.

1. prediction: This is a technique where local timing informations of the modules is used to find times in which no state change happens. After computation of these times (prediction) this algorithm may increase the time progress while composition.
2. partition: This option partitions the transition relation such that the structure is stored in an array of ROBDDs. Composition operations on ROBDDs are often more efficient than on MTBDDs. This often leads to faster composition but increased memory consumption.

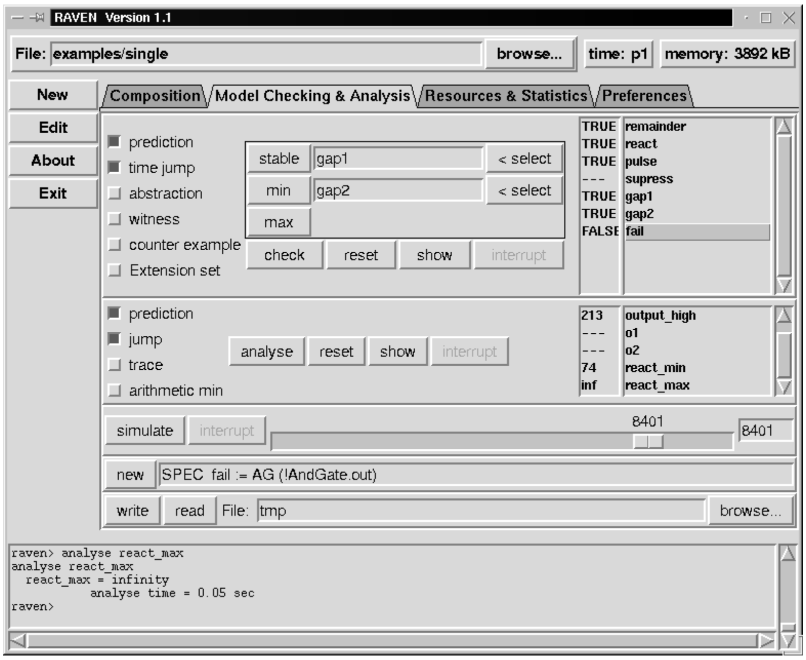
Minimization. There exists two heuristics which try to minimize the MTBDDs computed in the multiple-delay mode:

1. minpath: Build equivalence classes of states which behave same and introduces new transitions. This operation leads in most cases to a reduced number of MTBDD nodes.
2. reencode: This heuristic reencodes the new introduces encoding variables.We have found examples in which this heuristic lead to MTBDDs which are 5-times larger than the original MTBDDs [3].

After choosing the options, the user may start the composition and interrupt it. After starting the composition **RAVEN** shows in the lower bar the status of processing. The red section is the actual action **RAVEN** performs and the green ones are already computed.

5.3 The model checking & analysis window

After composition **RAVEN** switches automatically to the "Model Check & Analysis" window.



File: examples/single

time: p1

memory: 3892 kB

■ prediction
■ time jump
□ abstraction
□ witness
□ counter example
□ Extension set

stable gap1

min gap2

TRUE	remainder
TRUE	react
TRUE	pulse
---	supress
TRUE	gap1
TRUE	gap2
FALSE	fail

■ prediction
■ jump
□ trace
□ arithmetic min

213	output_high
---	o1
---	o2
74	react_min
inf	react_max

8401

8401

SPEC fail := AG (!AndGate.out)

File: tmp

```
raven> analyse react_max
analyse react_max
  react_max = infinity
          analyse time = 0.05 sec
raven>
```

This window is split into three parts.

Model checking. The list box on the right shows the parsed specifications and their proof state (---: unproved). The options on the left side have the following meaning:

- prediction: This is a technique which works together with the multiple-delay mode. It allows very fast model checking of timed specifications.
- time jump: This is also an optimization of the model checking algorithms used in the multiple delay mode. It works only if time prediction is activated.
- abstraction: If a specification has no quantitative timed operators and no next operators, it is possible to remove the timing information in the MTBDD representation and use a more compact ROBDD representation.
- witness: If this option is activated, than all ECCTL (CCTL with only E-path quantors) formulas produce a trace which shows their correctness. The selected wave form browser is started automatically.
- counter example: If the verification of an ACCTL-formula fails, this option produces a counter example showing the error.
- extension set: This option prints the extension sets of checked formulas as value tables on the log window.

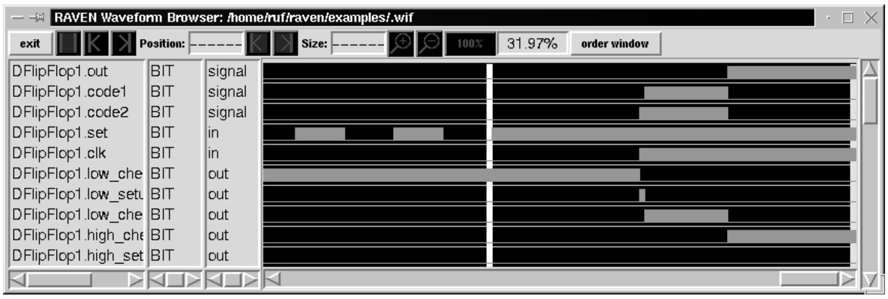
The buttons in the middle have the following meanings:

- check: starts the model checking algorithm with the chosen options for the selected specifications. If counter example or witness is activated, **RAVEN** displays the wave forms directly after checking. Also extension sets are printed automatically to the log window if this option is selected.
- reset: This button resets the proof state to unproved. This might be useful if the same specification will be checked with different options.
- show: This action prints the selected specification to the text entry field in the lower part of the window. In this window, the specification may be edited and parsed again.
- interrupt: This button may be used to interrupt the current model checking activity, e.g. if the chosen options take too much time.

The rectangle above the buttons is for on-line analysis. This kind of analysis does not have a query as usual analysis has. The two select buttons allow to specify two CCTL formulas representing two sets of configurations necessary for analysis. After choosing the specifications, the buttons min, max or stable may be used to start the analysis. The analysis result is not stored and only printed to the log window.

Analysis. The middle part of this window is for analysis of timing queries. There exists four options which affect the algorithms:

- prediction: This is the same optimization as it is used to accelerate the model checking algorithms
- jump: Also the time jump optimization is also applicable to the analysis algorithms.
- trace: This option makes **RAVEN** to display a simulation trace which visualizes the result of the analysis. The following figure shows a screen shot of a min-trace visualized with the wave form browser. The trace is a run starting in an initial state passing the start set of the min-query and ending at the destination set of the min-query. The start and the destination elements of the query are highlighted by using a colored rectangle behind the wave forms (white, yellow).



DFlipFlop1.out	BIT	signal
DFlipFlop1.code1	BIT	signal
DFlipFlop1.code2	BIT	signal
DFlipFlop1.set	BIT	in
DFlipFlop1.clk	BIT	in
DFlipFlop1.low_che	BIT	out
DFlipFlop1.low_setu	BIT	out
DFlipFlop1.low_che	BIT	out
DFlipFlop1.high_che	BIT	out
DFlipFlop1.high_set	BIT	out

- arithmetic min: This is an experimental implementation of the min algorithm, which uses another representation for delay times than the other analysis and model check algorithms do.

The possible actions to perform are:

- analyse: Starts to compute the selected analysis queries. After finishing a computation, the simulation trace will be displayed if the corresponding option is selected.
- reset: works like the reset button of the model checking part.
- show: see model checking part.
- interrupt: see model checking part.

Miscellaneous. The lower part of the window contains some additional functionality:

The first function is the simulation of the specified model. Since I/O-interval structures may contain indeterminisms, the simulation is randomly generated. The simulation is driven by the following elements:

- simulate: starts the simulation for a specified amount of time steps.
- interrupt: stops the simulation. The already computed runs are rejected since the asynchronous interruption may create faulty wave forms.
- scale: This slider allows the specification of the amount of time steps the simulation should be performed.
- entry: This text entry field is an alternative possibility to enter the number of simulation steps.

The second function is the text entry field for specifications and analysis queries. As described above, the show button prints the selected specification (query) into this field. The specifications (queries) may be manipulated or new ones may be typed in. The new button causes **RAVEN** to parse the typed specification (query) and to add them to the corresponding list. If a specification (query) with a same name exists in the list, the old specification (query) will be rewritten. The syntax of entering a new specification (queries) is equivalent to add specifications in **RIL**: first the keyword **SPEC** (**ANALYSIS**) has to appear followed by the name, the assign operator (:=) and the formula (query) in **RIL** syntax.

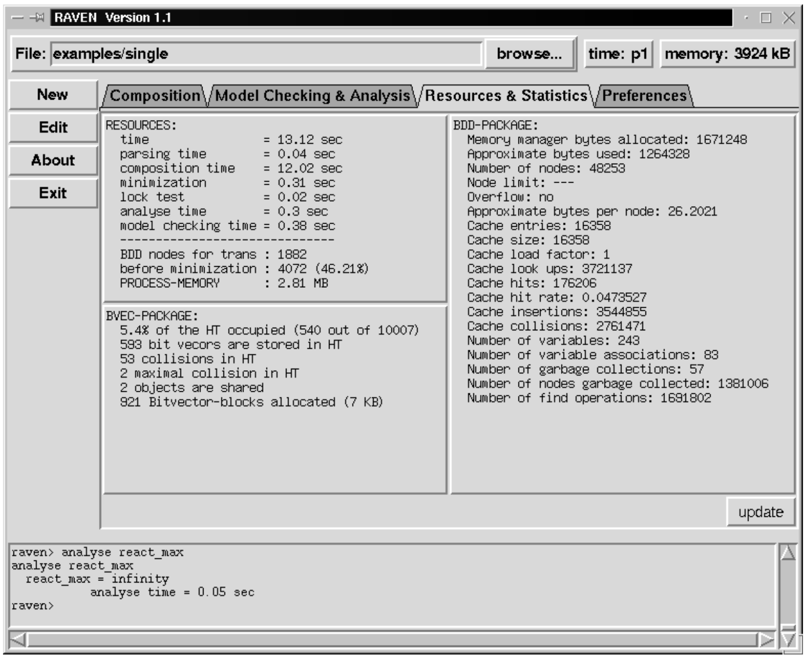
The last functional part of the window is also used to enter new specifications and queries to the corresponding lists. In contrast to the text entry field above, the button read loads a file containing several specifications and analysis queries. The file is parsed and new specifications (queries) are added to the corresponding lists. Existing specifications with the same names are rewritten. The file has one specification part with several specifications and one analysis part with several queries. The syntax is equivalent to the **RIL** syntax.

The write button allows the user to dump the actual set of specifications and queries to a file. If no specifications and queries are selected, all specifications and queries are printed to the file. The text entry field is for specifying the file name to which the lists have to be dumped out. The browse button allows the user to search interactively for a file which should be used to read or write.

5.4 The Resources & Statistics window

The last window of the **RAVEN** user interface is the Resources & Statistics window. This window may be updated by pressing the button in the lower right corner. The information can only be updated, if **RAVEN** is not busy (composition, model checking, analysis or simulation). The information is split into three parts:

- The RESOURCES part prints detailed information about the run times **RAVEN** used for different tasks. Also the number of MT-/ROBDD nodes is printed. If the minimization of MTBDDs is activated, also the number before the application of the corresponding heuristic is shown. At least the amount of memory in use is printed.
- The BVEC-PACKAGE window displays detailed information about the usage of the BVEC-unique table.
- The BDD-PACKAGE window prints detailed information generated by the used BDD package.



File: examples/single
time: p1
memory: 3924 kB

```
RESOURCES:
  time                = 13.12 sec
  parsing time        = 0.04 sec
  composition time    = 12.02 sec
  minimization        = 0.31 sec
  lock test           = 0.02 sec
  analyse time        = 0.3 sec
  model checking time = 0.38 sec
  ------------------------------
  BDD nodes for trans : 1882
  before minimization : 4072 (46.21%)
  PROCESS-MEMORY      : 2.81 MB
```

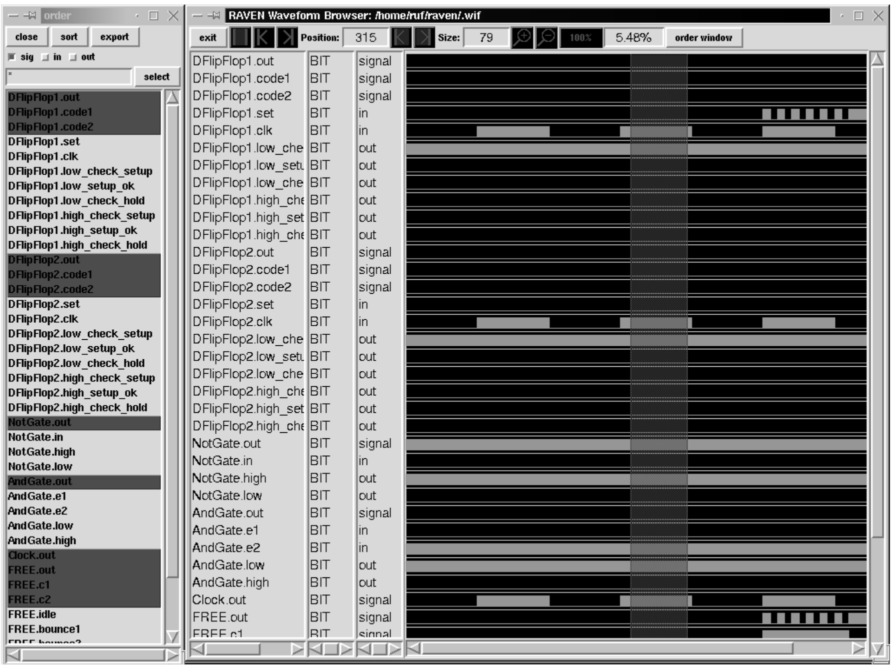
```
BVEC-PACKAGE:
  5.4% of the HT occupied (540 out of 10007)
  593 bit vecors are stored in HT
  53 collisions in HT
  2 maximal collision in HT
  2 objects are shared
  921 Bitvector-blocks allocated (7 KB)
```

```
BDD-PACKAGE:
  Memory manager bytes allocated: 1671248
  Approximate bytes used: 1264328
  Number of nodes: 48253
  Node limit: ---
  Overflow: no
  Approximate bytes per node: 26.2021
  Cache entries: 16358
  Cache size: 16358
  Cache load factor: 1
  Cache look ups: 3721137
  Cache hits: 176206
  Cache hit rate: 0.0473527
  Cache insertions: 3544855
  Cache collisions: 2761471
  Number of variables: 243
  Number of variable associations: 83
  Number of garbage collections: 57
  Number of nodes garbage collected: 1381006
  Number of find operations: 1691802
```

```
raven> analyse react_max
analyse react_max
  react_max = infinity
          analyse time = 0.05 sec
raven>
```

6 The wave form browser

One important feature of **RAVEN** is the generation of counter examples, witnesses and analysis-traces. All these runs are displayed as wave forms in a wave form browser. Together with **RAVEN** a wave form browser is distributed. This tool reads a sub-set of the Synopsys wave form interchange format (wif). The actual version of the browser tool can only handle boolean signals. On the left side of the main browser window all signals are enumerated together with their type and their usage inside the system (input, output, state signal). The right side of the window shows the corresponding wave forms.



*

DFlipFlop1.out
DFlipFlop1.code1
DFlipFlop1.code2
DFlipFlop1.set
DFlipFlop1.clk
DFlipFlop1.low_check_setup
DFlipFlop1.low_setup_ok
DFlipFlop1.low_check_hold
DFlipFlop1.high_check_setup
DFlipFlop1.high_setup_ok
DFlipFlop1.high_check_hold
DFlipFlop2.out
DFlipFlop2.code1
DFlipFlop2.code2
DFlipFlop2.set
DFlipFlop2.clk
DFlipFlop2.low_check_setup
DFlipFlop2.low_setup_ok
DFlipFlop2.low_check_hold
DFlipFlop2.high_check_setup
DFlipFlop2.high_setup_ok
DFlipFlop2.high_check_hold
NotGate.out
NotGate.in
NotGate.high
NotGate.low
AndGate.out
AndGate.e1
AndGate.e2
AndGate.low
AndGate.high
Clock.out
FREE.out
FREE.c1
FREE.c2
FREE.idle
FREE.bounce1

RAVEN Waveform Browser: /home/ruf/raven/.wif

Position: 315 Size: 79 100% 5.48%

DFlipFlop1.out	BIT	signal
DFlipFlop1.code1	BIT	signal
DFlipFlop1.code2	BIT	signal
DFlipFlop1.set	BIT	in
DFlipFlop1.clk	BIT	in
DFlipFlop1.low_che	BIT	out
DFlipFlop1.low_setu	BIT	out
DFlipFlop1.low_che	BIT	out
DFlipFlop1.high_che	BIT	out
DFlipFlop1.high_set	BIT	out
DFlipFlop1.high_che	BIT	out
DFlipFlop2.out	BIT	signal
DFlipFlop2.code1	BIT	signal
DFlipFlop2.code2	BIT	signal
DFlipFlop2.set	BIT	in
DFlipFlop2.clk	BIT	in
DFlipFlop2.low_che	BIT	out
DFlipFlop2.low_setu	BIT	out
DFlipFlop2.low_che	BIT	out
DFlipFlop2.high_che	BIT	out
DFlipFlop2.high_set	BIT	out
DFlipFlop2.high_che	BIT	out
NotGate.out	BIT	signal
NotGate.in	BIT	in
NotGate.high	BIT	out
NotGate.low	BIT	out
AndGate.out	BIT	signal
AndGate.e1	BIT	in
AndGate.e2	BIT	in
AndGate.low	BIT	out
AndGate.high	BIT	out
Clock.out	BIT	signal
FREE.out	BIT	signal
FREE.c1	BIT	signal

On the top of the browser window there are some buttons placed which allow to analyze the wave forms or to change the display parameters:

- exit: terminates the browser.

- slider bar: This button activates the slider bar. This bar is for examining time points or for measuring time delays. Initially the slider bar has the size 1 and appears at the left side of the actual display. The bar may be moved by clicking it with the left mouse button and drag it to the desired position. The size of the slider may be increased by clicking the left or right boundary and drag it to the desired size.
- move to next event left: It is possible to move the slider to an event happens on a signal. It is necessary that one signal is selected. In this case this button moves the left end of the slider to the next event (raising or falling edge of the signal) on the left.
- move to next event right: This button moves the right boundary of the slider to the next event occurring right to the slider.
- position: This field displays the actual position of the left end of the slider bar.
- increase slider to next event left: This button increases the slider such that the right end does not change it position and the left end is resized to the next event (of the selected signal) in the left of the current position.
- increase slider to next event right: This button increases the slider such that the right boundary is moved to the next event and the left boundary stays constant at the actual position.
- slider size: This field shows the size of the slider. For instance, with the resize buttons it is possible to determine the size of an impulse. First move the slider in the middle of the impulse you want to examine. The slider has to be smaller than the impulse. Then activate the signal of the impulse, press both increase slider buttons. The slider is now exactly as wide as the impulse is. The size of the slider is the size (delay time) of the impulse.
- zoom in: This button increases the zoom factor of the x-axis of the wave forms.
- zoom out: This button decreases the zoom factor of the x-axis of the wave forms.
- zoom to original size (100%): This button resets the zoom factor to 100%. This is the value the wave forms appear after launching the browser.
- zoom factor: This field prints the zoom factor of the x-axis of the wave forms.
- order window: This button activates a new window which allows the reordering and en-/disabling of signals.

The order window. The order window has some buttons and one text entry field at the top and a list of all signals in a large list box below. Clicking and dragging signal names in the list box with the middle mouse button moves signals to new positions in the order window. All modifications inside the order window have no direct influence to the ordering of the signals in the browser window. But if the export button is pressed, only the actual selected signals appear in their order in the browser window. The buttons have the following meanings:

- close: This button closes the ordering window. The actual order chosen in the window does not affect the signal order in the browser window.
- sort: Sorts all signals alphabetically by their names.
- export: This button exports the actual selected signals to the browser window. Nonselected signals disappear in the browser window. The order of the selected signals in the order window is also exported to the browser window.
- The three check buttons are for selecting state signals (sig) inputs (in) outputs (out). The selection becomes active, if the select button will be pressed.
- The text entry field allows to specify names of signals which should be selected if the select button will be pressed. Wildcards as for file names may be used (*,?) inside the names. For instance to select all signals, inputs and outputs of the AndGate, write "AndGate.*" inside the text field and press the select button.
- select: A selection via the check buttons and the text entry field becomes active only if this button will be pressed. Only results matching the text entry field and one of the activated check buttons will be selected. Single signals may be directly selected in the list box via the left mouse button.

7 The textual interface

calling raven (instead of xraven) directly starts the textual interface of **RAVEN**. This call is important for batch jobs which do not run with human interaction. The syntax for calling the textual interface of **RAVEN** is:

$$\text{raven <option-list> <filename>[.ril]} \qquad (13)$$

The suffix .ril has not to be specified, **RAVEN** searches automatically for files with this suffix. The following table presents all options. Options are preceded by a "+" to enable them or by a "-" to disable them. Options which have nothing to en- or disable may be preceded by "+" or "-" alternatively (e.g. choosing the composition algorithm). The options may be abbreviated by an unique prefix (write -comp instead of -composition). Calling **RAVEN** with no arguments results in a printing of all options and their meanings.

Table 1: the command line options

option	description	default
verbose [scrmp]	There exists five verbose modes: s: print the original structures c: print information while composing r: print information while restricting m: print info while minimization p: print info while proving	off
composition <alg>	specifies the composition method (see gui description): standard, restriction, prerestriction, incremental, combined	standard
startset <set>	specifies the start states for the reduction (see gui description): init, iprod, one	init
order <file>	writes the variable order into file	
minimization <alg>	disactivates or activates minimization heuristics: minp, reenc	minp
unitdelay	use unit delay model for composition and model checking	off
prediction	use time prediction for the composition	on
partition	use transition relation partitioning for the composition	on
locktest	en/disables the dead-lock test before model checking	on
timeprediction	use time prediction for model checking	on
jump	use time jumps for model checking	on
transition	prints the transition relation	off
number	prints number of states and transitions	off
statistics	prints statistic information	off
extensionset	prints the extension sets of the specifications	off
simulation <n>	simulates n steps the system	off
interactive	turns on/off interactive mode	off

Table 1: the command line options

option	description	default
counterexample	turns on/off counter example generation	on
witness	turns on/off witness-, lock-trace- and analysis-trace-generation	on
bvecsize <n>	determines the size of the bitvector unique table	10000
symbolsize <n>	determines the size of the symbol table	5000
tabletype <n>	determines the order of signals in function tables (1,2,3,4,5 see gui description)	2
graycode	use gray encoding for time variables instead of binary encoding	off
abstraction	use stutter state abstraction while model checking untimed specifications	on
background	starts wave form browser asynchronously	off
browser <name>	select the wave form browser: TCL/TK, SYNOPSYS, JAVA	TCL/TK
wif <file>	specifies the file for the wave forms	.wif
icc	this option turns on/off the input clustering check	on
amc	this option turns on/off the alternative min computation	off

If the option "+interactive" is activated, **RAVEN** switches after the composition into an interactive user mode. In this mode, it is possible to check formulas or to compote analysis queries or to add new formulas etc. After finishing the composition the raven prompt appears (raven>). Now the user may interactively enter commands. The possible commands are enumerated in the following table. Commands may also be abbreviated by unique prefixes.

Table 2: interactive RAVEN commands

command	description
help, ?	prints all commands
check [formula | all]	checks the specified formula
analyse [query | all]	starts analysis of the specified query
stable formula	computes stability of the specified formula
min formula1 formula2	computes minimum between the formulas specified as arguments
max formula1 formula2	computes the maximal time between the two formulas
simulate n	starts simulation for n time steps
reset [formula | query | all]	resets the proof state of the specified formula or query

Table 2: interactive RAVEN commands

command	description
new SPEC name := formula	adds a new or rewrites an existing formula
new ANALYSIS name := query	adds a new or rewrites an existing query
read filename	reads an input file with formulas and queries
write [formula | query | all] file	writes the specified formula (query) or all formulas and queries to the specified file
quit	terminates interactive mode
info	prints some information about the system and the proof states
show [formula | query | all]	prints one formula, query or all formulas and queries.
statistics	prints statistics about the used time, the BVEC-PACKAGE and the BDD-PACKAGE
option [+|-] [prediction | jump | extensionset | verbose | winess | counter_example | abstraction | amc]	allows to en-/disable some options (see gui description)
garbage	initiates a garbage collection of the BDD package

8 RIL syntax

In this section we present the syntax of **RIL** in BNF. The BNF symbols have the following meaning:

Table 3: BNF notation

BNF notation	meaning
rilfile	The start symbol
something	The string "something" itself (terminal symbol)
something	something replaced by its definition
(something)	something
something_1 ::= something_2	something_1 is defined by something_2
something_1 something_2	something_1 is concatenated with something_2
something_1 | something_2	something_1 or something_2 (lowest priority in BNF)
[something]	nothing or something
$\{\text{something}\}^*$	nothing or something concatenated arbitrary often

Table 3: BNF notation

BNF notation	meaning
{something}$^{+}$	something {something}*

Comments can be included anywhere in the text. Comments start with // and ends with a new line.

```
rilfile ::=             [time_var_defs]
                        [global_defs]
                        {model}*
                        [specs]
                        [anas]

time_var_defs ::=       TIMES {time_var_def}+

time_var_def ::=        ident := nat-expression

global_defs ::=         DEFINE[S] {global_def}+

global_def ::=          ident := formula

model ::=               ( MODULE | FSM | MODEL ) ident body END

body ::=                SIGNAL[S] {ident [ : BOOL ] }+
                        [ INPUT[S] {inputdecl}+ ]
                        [ ( STATE[S] | DEFINE[S] ) statedecl+ ]
                        [ INIT formula ]
                        [ transdecl ]

inputdecl ::=           ident := formula

statedecl ::=           ident := ( enum | formula )

enum ::=                { {ident}* }

formula ::=             TRUE | FALSE | INIT | ident
                       | ident . ident
                       | ! formula
                       | formula'
                       | formula  | formula
                       | formula & formula
                       | formula -> formula
                       | formula <-> formula
                       | formula xor formula
                       |  ( formula  )

transdecl ::=           TRANS { transition }*
                       | NEXT { formula }*

transition ::=          fornmula  : timebound {, timebound }*

timebound ::=           nat-expression
                       | [ nat-expression ,  nat-expression  ]

nat-expression ::=      number
                       | ident
                       | nat-expression - nat-expression
                       | nat-expression + nat-expression
                       | nat-expression * nat-expression
                       | ( nat-expression  )

specs ::=               SPEC[S] {spec}+
```

spec ::= ident **:=** cctl-formula

anas ::= **ANA**[**LYSIS**] analysis

analysis ::= ident **:=** anatype

anatype ::= **STABLE (** cctl-formula **)**
| **MIN (** cctl-formula **,** cctl-formula **)**
| **MAX (** cctl-formula **,** cctl-formula **)**

cctl-formula ::= atom
| predicate
| compound

compound ::= **!** cctl-formula
| **(** cctl-formula **)**
| cctl-formula **<->** cctl-formula
| cctl-formula **->** cctl-formula
| cctl-formula **xor** cctl-formula
| cctl-formula **|** cctl-formula
| cctl-formula **&** cctl-formula

atom ::= **TRUE**
| **FALSE**
| **INIT**
| ident **.** ident
| ident

predicate ::= **EX** [opttime] cctl-formula
| **EF** [inter] cctl-formula
| **EG** [inter] cctl-formula
| **E (** cctl-formula **U** [inter] cctl-formula **)**
| **E (**cctl-formula **wU** [inter] cctl-formula **)**
| **E (** cctl-formula **B** [inter] cctl-formula **)**
| **E (** cctl-formula **wB** [inter] cctl-formula **)**
| **E (** cctl-formula **S** [opttime] cctl-formula **)**
| **E (** cctl-formula **C** [opttime] cctl-formula **)**
| **AX** [opttime] cctl-formula
| **AF** [inter] cctl-formula
| **AG** [inter] cctl-formula
| **A (** cctl-formula **U** [inter] cctl-formula **)**
| **A (** cctl-formula **wU** [inter] cctl-formula **)**
| **A (** cctl-formula **B** [inter] cctl-formula **)**
| **A (** cctl-formula **wB** [inter] cctl-formula **)**
| **A (** cctl-formula **S** [opttime] cctl-formula **)**
| **A (** cctl-formula **C** [opttime] cctl-formula **)**

optinter ::= **[** nat-expression [**,** inf-expression] **]**

opttime ::= **[** nat-expression **]**

inf-expression ::= nat-expression | **INF**[**INITY**]

ident ::= letter { letter | digit }*

number ::= digit$^+$

letter ::= **a** | ... | **z** | **A** | ... | **Z**

digit ::= **0** | ... | **9**

Bibliography

[1] 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.

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

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

[4] J. Ruf and T. Kropf. Analyzing Real-Time Systems. In *DATE 2000*, Paris, France. IEEE Computer Society Press.

[5] 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.

[6] 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.

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

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

[9] SAE. J1850 class B data communication network interface. *The Engeneering Society for Advancing Mobility Land Sea and Space*; October 1995.

[10] J. Ruf. Formal Verification of Timing Properties of a Holonic Material Transport System, *Technical Report, WSI-2000-3,* Februar 2000.

[11] J. Ruf. A Toolset for the Symbolic Examination of Finite State Transition Systems. In M*ethoden und Beschreibungssprachen zur Modellierung und Verifikation von Systemen. GI/ITG/GMM Workshop*, 2000, Frankfurt, Germany.