Modelling and Analysing a Railroad Crossing in a Modular Way

Dirk Beyer, Claus Lewerentz and Heinrich Rust

Software and Systems Engineering Team, Technical University Cottbus
Postfach 10 13 44
D-03013 Cottbus, Germany
Email:{db|cl|rust}@informatik.tu-cottbus.de

Abstract. One problem of modelling hybrid systems with existing notations of hybrid automata is that there is no modular structure in the model. We introduce an extended modelling notation which allows the modelling of a system as a hierarchical structure of modules. The modules are capable of communicating through the elements of an explicitly defined interface. The interface consists of signals and variables declared with different access modes. This paper describes a model of the railroad crossing example and how to verify it. The current version of a tool for reachability analysis using the double description method to represent symbolically the sets of reachable configurations is presented.

Keywords. Automata, continuous systems, formal verification, real-time systems

1 Introduction

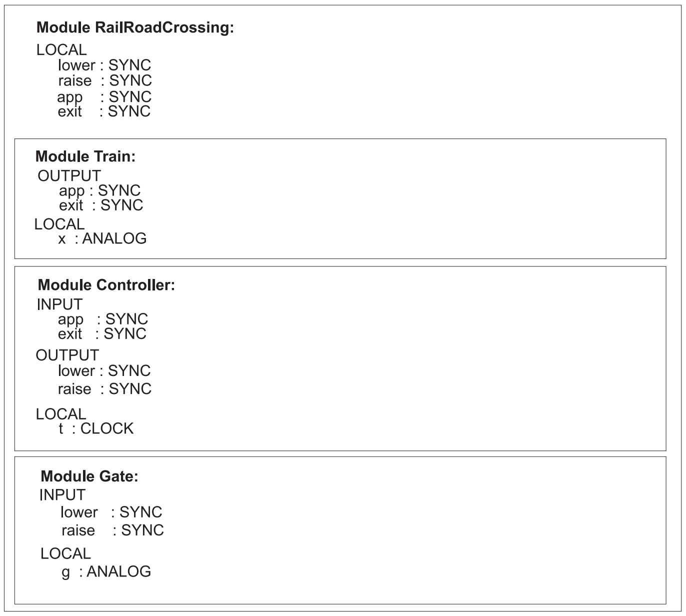
The programming of embedded systems which have to fulfil hard real-time requirements is becoming an increasingly important task in different application areas, e.g. in medicine, in transport technology or in production automation. The application of formal methods, i.e. of modelling formalisms and analysis methods having a sound mathematical basis, is expected to lead to the development of systems with less defects via a better understanding of critical system properties (c.f. [Lev95]).

Beyer and Rust presented the modelling notation CTA which allows to model hybrid systems in a modular way [BR98]. It builds on the theoretical basis used in tools like UppAal [BLL$^+$96], Kronos [DOTY96] and HyTech [HHWT95]. In these formalisms and tools, finite automata are used to model the discrete control component of an automaton. Analogous variables which may vary continuously with time are used to model the non-discrete system components of a hybrid system. Component automata of a larger system communicate via CSP-like synchronisation labels (cf. [Hoa85]) and via common variables. Algorithms for the analysis of these kinds of models have been presented in [ACD93] and [HNSY94].

To provide features for modelling and verifying modular hybrid systems we need a new formalism and tool which includes, in difference to the existing formalisms and tools, the following concepts:

- Compositional semantics: By using two predicates for the transition assignments in our formalism we preserve the information we need to define the semantics of a CTA module on the basis of the semantics of its parts.
- Hierarchy: Subsystem descriptions can be grouped. Interfaces and local components are separated.
- Explicit handling of different types of communication signals: It is possible to express explicitly that an event is an input signal for an automaton, an output signal, or a multiply restricted signal.
- It is possible to express explicitly that an analogous variable is accessed by an automaton as output, as input, or that it is multiply restricted, which means that each module has read and write access to that variable.
- Automatic completion of automata for input signals. Input signals are events which an automaton must always admit. If there are configurations of an automaton in which the reaction to an input signal is not defined, it is understood that the automaton enters an error state.
- Replicated subsystem components do not have to be multiply defined. They are instantiated from a common module type.

The differentiation between different roles of signals in an automaton has been used in the definition of IO-automata [LT87] and extended to hybrid systems [LSVW96]. [AH97] use different access modes in interfaces to describe modular hybrid systems, too. They build on reactive modules [AH96] and extend them with continuously changing variables.



```
Module RailRoadCrossing:
LOCAL
    lower : SYNC
    raise  : SYNC
    app    : SYNC
    exit    : SYNC

  Module Train:
  OUTPUT
      app : SYNC
      exit  : SYNC
  LOCAL
      x  : ANALOG

  Module Controller:
  INPUT
      app   : SYNC
      exit   : SYNC
  OUTPUT
      lower : SYNC
      raise  : SYNC
  LOCAL
      t  : CLOCK

  Module Gate:
  INPUT
      lower   : SYNC
      raise    : SYNC
  LOCAL
      g  : ANALOG
```

Fig. 1. Interfaces of components of the railroad crossing model

The CTA method provides a complete modelling notation and a tool for reachability analysis of those models regarding the features mentioned above.

In section 2 the example of the railroad crossing is described. Section 3 gives an informal description of the formalism. Section 4 gives an overview on the syntax of the modelling language and the analysis commands. Section 5 gives an overview on the CTA tool. The last section explains open questions to be dealt with.

2 Model for railroad crossing

2.1 The problem

The railroad crossing example deals with the following situation: The gate for a railroad crossing must be closed when the train reaches the crossing. There are two sensors on the railroad: One switching on if the train is at position 1000 m before the gate, and the other sensor is situated 100 m after the gate which indicates that the train has passed the gate. The gate can be moved between the angle 0 degree (which means the gate is closed) and the angle 90 degree (open) by a motor. The problem is to create a controller which fulfils the safety condition that the gate is closed before the train is nearer than in a 250 m distance to the crossing.

2.2 A modular, automaton-based approach

Our model consists of three subsystems: The train, the gate, and the controller. The controller coordinates the actions of the other components. The environment of the controller consists of one part for the train behaviour and one part for the gate behaviour.

Fig. 1 displays the structure of the modules: The main module ’RailRoadCrossing’ has four synchronisation signals, which are locally defined because the system is modelled as a closed system. ’app’ models the sensor at the 1000 m-position and ’exit’ models the sensor at the 100 m-position for communication between the train and the controller. The signals ’lower’ and ’raise’ are used by the controller to control the action of the gate. They all have to be declared in this module because they are used by more than one contained module. The other modules contained in the ’RailRoadCrossing’ are described in the following paragraphs.

Train. The train component (Fig. 2) models the position of the train (local variable x) and its speed (the time derivation of x). Because the sensor on the railroad which switches on when the train reaches

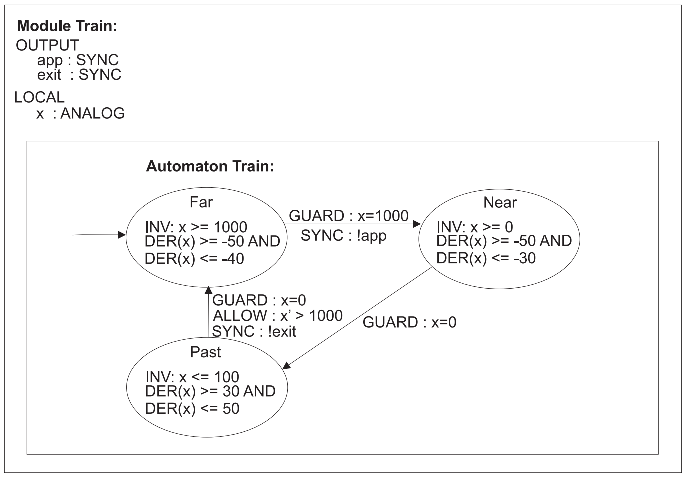


Fig. 2. The train model

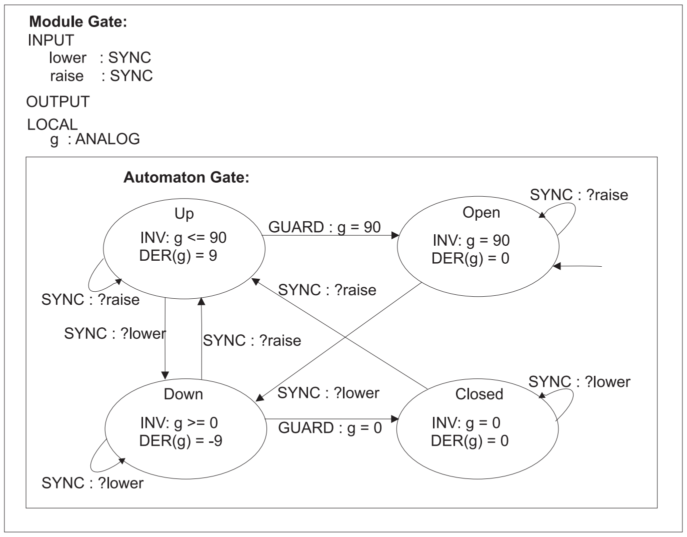


Fig. 3. The gate model

the position at 1000 meters distance to the gate is controlled by this module the signal ’app’ is declared as ’OUTPUT’. This means that all modules in the environment are only allowed to read the signal. The same is true for the signal ’exit’.

The contained hybrid automaton has three discrete states: To model that the train comes from far away, if the distance to the gate is greater than 1000 m we have the state ’Far’. The invariant of this state is that the position of the train is greater than or equal to 1000. The derivation of ’x’ has to be between -50 and -40 to model the possible speed of the train. When the train is passing the first sensor (x=1000) it switches to the state ’Near’ by synchronising with the label ’app’. Following the CSP concept it means that all other automata which know this label must also take a transition with this label. The speed is modelled by the derivation of ’x’ again and the invariant with the corresponding guard at the transition models that the train passes the gate, and the automaton switches to the state ’Past’. After another 100 m the invariant and the guard of the next transition forces the automaton to leave the state by firing this transition by forcing synchronising with the output signal ’exit’. It also sets the variable ’x’ to a value greater than 1000 to be ready for the next cycle in the state ’Far’.

Gate. An illustration of the gate model is given in Fig. 3. This component models the angle of the gate by the analogous variable ’g’. The module has to react on the two input signals ’lower’ and ’raise’. The

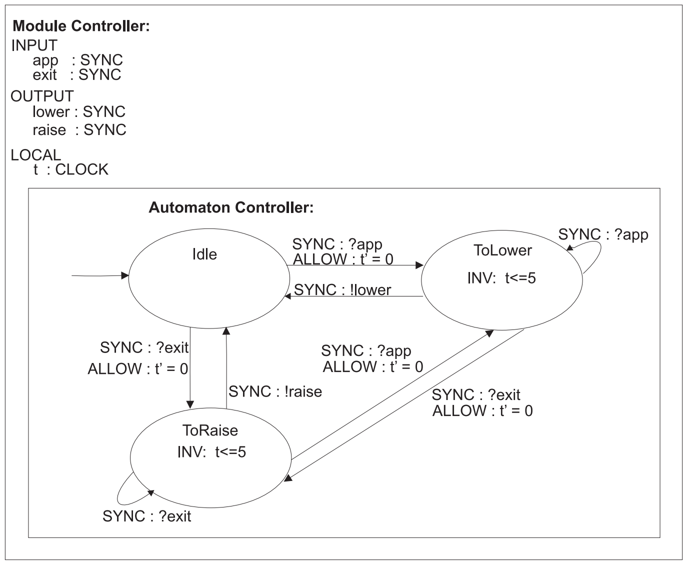


Fig. 4. The controller model

automaton starts in state ’Open’ (g=90). When it receives the signal ’lower’ it switches to the state ’Down’, which models the situation that the gate is lowering but has not finished the task. The speed of the angle change is modelled by the derivation of ’g’. If the angle of 0 degrees is reached the automaton has to switch to state ’Closed’. In this state it waits for the signal ’raise’, which forces to take the transition to the state ’Up’ for starting the opening process. If the angle 90 is reached the automaton switches to the state ’Open’ and it can start from beginning.

Controller. The controller models the component (the electronic device or the controller software) for the coordination of the other parts of the system (Fig. 4). It has two input signals to react on signals from the ’Train’ module and two output signals to control the module ’Gate’. The local clock variable ’t’ is used to measure the reaction time of the controller.

The automaton starts in the ’Idle’ state waiting for the train-is-coming signal from the train (1000 m-position). This signal forces the controller automaton to switch to the state ’ToLower’ and the transition resets the clock ’t’. That state models the situation that the controller consumes the signal, needs some time to react, and then to perform an action. The upper bound of the reaction time ’α’ (in the example it is set to 5) is an important parameter for satisfying the safety condition of the system. At least after time α it must send the ’lower’ signal to the gate and goes back to the ’Idle’ state. If the train has passed the 100 m position after the gate then it sends the ’exit’ signal to the controller. After some reaction time the controller sends the raise-signal to the gate and after this it is waiting again in state ’Idle’.

3 Theoretical background

The following section contains a short description of the CTA formalism. The complete formal definition of the CTA semantics is given in [BR99].

3.1 Formalism for modelling

A CTA system description consists of a set of modules. One of them is designated as the top module. It models the whole system. The other modules are used as templates. They can be instantiated several times in other modules. This makes it possible to express a hierarchical structure of the system, and to define replicated components of a system just once.

Each module consists of the following components:

– An **identifier.** A system description might contain several modules. Identifiers are used to name them.

– An **interface.** The interface consists of declarations of variables and signals used by the components of the module. It defines the access modes for the variables and signals and the data types for variables.
  - **Signals.** Signals are used for communication between modules running in parallel. Signals are modelled as CSP-like events.
  - **Variables.** Variables are used to model the (predominantly) continuously changing components of a hybrid system. CTA variables are real valued, they may change continuously with time, and they may change discretely.

– A **hybrid automaton.** This automaton consists of the following components:
  - **S**: A finite set of discrete states.
  - **G**: A finite set of signals.
  - **V**: A finite set of analogous variables.
    Variables are used in **value assignments**. A value assignment for a set of variables is a member of the set of functions $A(V) = V \to \mathbb{R}$.
    At every point in time the situation of an automaton is described by its current discrete state and the current value assignment of all variables of the automaton. A **configuration** c of a CTA is defined as the pair $c = (s, a)$, where s is the current discrete state and a is the current value assignment ($c \in C = S \times A(V)$). A **region** $r \in R = \mathcal{P}(C)$ is a set of configurations.
  - **I** $\subseteq$ **C**: An initial condition, described as a region.
  - **T**: A finite set of transitions.
  - **inv** $: S \to 2^{A(V)}$: A function associating an invariant to each state. The invariant is a set of value assignments. As long as the invariant of a state is true, the system may stay in the state. It may leave the state earlier, but the latest moment is just after the invariant has become false.
  - **deriv** $: S \to 2^{A(V')}$: A function associating a set of admissible derivatives to each state. In the finite set of variables V' there is, for each variable $v \in V$, a corresponding element v' which is used to define admissible time derivatives of the variable v. While the automaton remains in a state, the continuous changes of a variable v are defined by their first time derivative v'.
  - **trans** $: T \to S \times S$: A function associating a starting state and a target state to each transition.
  - **guard** $: T \to 2^{A(V)}$: A function associating a guard with each transition. The guard is a set of value assignments. One condition for the transition to be taken is that the guard is true.
  - **sync** $: T \to G \cup \{*\}$: A function associating a signal or no signal to each transition. $*$ is not a signal; it is the value of sync(t) for transitions without a signal.
  - **allowed** $: T \to (A(V) \to 2^{A(V)})$: A function associating with each transition a function which transforms a value assignment into one of a set of value assignments. The aim of this function is to restrict changes which may occur in any environment.
  - **initiated** $: T \to (A(V) \to 2^{A(V)})$ with $\forall t \in T, a \in A(V)$: initiated(t)(a) $\subseteq$ allowed(t)(a): A function associating with each transition a function which transforms a value assignment into one of a set of value assignments. In contrast to 'allowed' the aim of this function is to restrict changes which occur *without* an environment.
    For each $t \in T$ and $a \in$ guard(t), the set initiated$(t)(a)$ must be nonempty. This condition ensures that the 'initiated' or 'allowed' component can not inhibit a discrete transition to be taken.
    A further restrictions is: For each $s \in S$, there is an element t_s of T with the following properties:
    * trans$(t_s) = (s, s)$
    * guard$(t_s) =$ true
    * sync$(t_s) = *$
    * $\forall a \in A(V)$: initiated$(t_s)(a) = \{a\}$
    * $\forall a \in A(V)$: allowed$(t_s)(a) = A(V)$

    These transitions are no-op transitions. The subset of T consisting of all no-op transitions is referred to by 'noop'. The 'allowed' function of no-op transitions does not exclude any resulting value, and 'initiated' defines that no variable value changes. □

– **Instances.** A module may contain instances of previously defined modules. This is used to model systems containing subsystems, and it is especially helpful if a subsystem occurs several times in a system. An instance consists of the following components:
  - An **identifier** is used to give a name to the instance.
  - A reference to a **module** defines which module is instantiated.
  - An **unification** of interface components of the instantiated module with declared components of the containing module defines how the instance is connected to the containing module. This may connect interface signals and interface variables of the instantiated module to signals and variables of the containing module.

Notational convention: A typical case to use the ’initiated’ predicate is to restrict variables which are not restricted by any parallel transition. If there is a variable which does not occur ticked in at least one inequation, then this does not mean that the whole range of $\mathbb{R}$ is possible. For a variable which does not occur ticked in an inequation the meaning is that this transition does not change the value of the variable. Transitions of environmental automata are allowed to restrict the variable. But if no automaton restricts the variable x in its transition in the same point in time, then we use the information of the ’initiated’ set which contains typically the additional restriction $x' = x$. To express that the whole range of $\mathbb{R}$ is possible for x after a transition, one might use the clause $x' > 0$ OR $x' \leq 0$.

In our notation we only use the typical case described above. Perhaps there are other useful aspects for a more general use of the ’initiated’ set, but we did not yet find them, and thus we restrict our notation to have an easy to use syntax. Thus, in a syntactical INITIATE clause would be only restrictions of the form $x' = x$ for each variable $x \in X$ (set of variables known by the automaton), if x does not occur ticked in any inequation of the syntactical clause for ’allow’. The consequence for us is to generate the ’initiated’ set automatically, i. e. we have not a syntactical clause for ’initiate’ in our notation.

Beside the additional concepts, the main difference to existing formalisms are the two assignments ’allowed’ and ’initiated’ and the semantics of the invariant. The splitting of the assignments leads to a compositional semantics. The fact that an invalid invariant not leads to a forbidden state but that a discrete transition is forced has the following advantage: If the invariant of a state becomes invalid because another automaton changes the values of the variables then the automaton can immediately fire a discrete transition to another state.

In the CTA formalism each of the interface components has an **restriction type** to control the access on the component. There are four different restriction types for variables and signals:

– **INPUT** The declaration of a variable as input variable for a module means that this module can only read this variable:
  - The derivation for an input variable may not be restricted in the deriv-set of any state of the automaton.
  - The value of an input variable after a transition may not be restricted in the allowed-set of any value assignment in a transition.
  - In difference to ’allowed’, input variables can be restricted in ’initiated’ to reflect that the value of the input variable is not changed by this automaton.

  For a signal the declaration as input means the following: For each input signal and each state of the automaton, some transition labelled with the signal can always be taken. In this way the automaton does not restrict the input signal and thus it is not to blame for a time deadlock. Thus, it is a guarantee for the environment that the module do not change that component .
– **OUTPUT** the declaration of a variable or signal as OUTPUT is an assumption, that the variable or signal is used only as INPUT in all other modules in the environment.
– **MULTREST** The multiply restricted components are available for all access modes. A module as well as the environment for which a signal or variable is declared as multiply restricted can restrict the component in any way.
– **LOCAL** The declaration of a variable or signal as LOCAL means that it is not visible outside the module and thus no other module can access such a variable or signal.

Variable restrictions in transitions. To express nondeterminism the value of a variable after a transition has been performed is selected from a set of possible values. These possible values are described by linear expressions over the variables, which can be denoted with the names for the value before the assignment and with the ticked name for the value of the variable after the assignment (for example $x' \geq 3 * x + 7$).

The **product automaton** construction uses the standard technique for composition of automata, with CSP synchronisation as described in [HHWT95]. We construct the product automaton of two hybrid automata in the following way:

– The new set of **states** is the cross product of the state sets from the two automata.
– The new set of **variables** and the set of **signals** are the union of the corresponding sets from the automata.
– The **initial condition** for the new automaton is the conjunction of initial conditions of the automata.
– The **transition** set is the subset of the cross product from the automata, which consists of the combination of two transitions with the same signals or one of the combined transition is a no-op transition, which do not change the values of the variables and has no synchronisation label.

- **Invariants** and **derivatives** are intersected.
- **Guards** are intersected with the additional condition that the set of allowed assignments must not be empty.
- The new **signal** of a transition is the signal used by one of the parallel transitions.
- The new set of **allowed assignments** is the intersection of the two sets of allowed assignments.

The **semantics of CTA** is understood as a labelled transition system.

Notation. For a real u and two value assignments a and $a' \in A(V)$, let $u * a$ denote the function $\lambda(v : V) : u * a(v)$, and let $a + a'$ denote the function $\lambda(v : V) : a(v) + a'(v)$.

time : $C \times \mathbb{R} \times A(V) \rightarrow C$ is a function describing how the passage of some time u changes a configuration (s, a) when a time derivative $d \in A(V)$ for the variable values is fixed:

$$\text{time}((s, a), u, d) = (s, a + u * d)$$

□

A hybrid automaton can perform time transitions and discrete transitions.

Definition 1. (Time transitions and discrete transitions of a hybrid automaton) Let $\mathcal{H}$ be a hybrid automaton.

time$(\mathcal{H})$ is the set of **time transitions of** $\mathcal{H}$. It is defined as the following set:

$$\begin{array}{l} \{ \quad ((s, a_1), (s, a_2)) \in C \times C \\ \quad \mid \exists d \in \text{deriv}(s), u \in \mathbb{R}, u > 0 : \\ \qquad (\quad (s, a_2) = \text{time}((s, a_1), u, d) \\ \qquad \wedge \quad \forall (u' : 0 \leq u' \leq u) : \quad \text{time}((s, a_1), u', d) \in \text{inv}(s) \\ \qquad) \\ \} \end{array}$$

discrete$(\mathcal{H})$ is the set of **discrete transitions of** $\mathcal{H}$. It is defined as the following set:

$$\begin{array}{l} \{ \quad ((s_1, v_1), (s_2, v_2)) \\ \quad \mid \exists (t : T) : \\ \qquad (\quad \text{trans}(t) = (s_1, s_2) \\ \qquad \wedge \quad v_1 \in \text{guard}(t) \\ \qquad \wedge \quad v_2 \in \text{initiated}(t)(v_1) \\ \qquad) \\ \} \end{array}$$

□

Note. For discrete transitions the invariant is irrelevant. An invariant which is identically false can be used to construct urgent states, i. e. states in which time cannot pass.

The state component of the configuration may not change in a time transition.

In the definition of the set of discrete transitions, the signals and the 'allowed' function are not used. Their meaning is defined later, when we consider the parallel composition of automata. □

We will define a transition system semantics for hybrid automata.

Definition 2. (Transition system) A **transition system** consists of the following components:

- A (possibly infinite) set S of states, with a subset S_0 of initial states.
- A set $T \subseteq S \times S$ of transitions.

□

Notation. We use the point notation $A.x$ to address the component x of A. □

The transition system corresponding to a hybrid automaton is defined in the following way:

Definition 3. Let $\mathcal{H}$ be a hybrid automaton. The **transition system** ts$(\mathcal{H})$ corresponding to $\mathcal{H}$ is defined in the following way:

```
Region $reachable$ := $r$;
WHILE ($post(reachable) \setminus reachable \neq \emptyset$)
    $reachable$ := $reachable \cup post(reachable)$;
```

Fig. 5. Algorithm for fixed point computation of reachable regions

- ts$(\mathcal{H}).S =_{\text{def}} \mathcal{H}.C$.
- ts$(\mathcal{H}).S_0 =_{\text{def}} \mathcal{H}.I$.
- ts$(\mathcal{H}).T =_{\text{def}}$ time$(\mathcal{H}) \cup$ discrete$(\mathcal{H})$.

□

Note. The state space of the transition system consists of the configurations of the hybrid automaton. The set of starting states in the transition system is defined via the initial condition of the hybrid automaton. The transitions of the transition system are all time transitions and all discrete transitions of the transition system. □

3.2 Reachability analysis

System properties which are to be proved in the verification are described as expressions containing reachability operators.

The most important operators for reachability analysis are the operator $post(c)$ for the region of all the configurations reachable from configuration c by using a time transition or discrete transition, and the operator $pre(c)$ for the configurations from which c is reachable in one time transition or one discrete transition. To handle regions as arguments, $post(r)$ is defined for a region r as $\bigcup_{c \in r} post(c)$ and $pre(r)$ is defined as $\bigcup_{c \in r} pre(c)$.

Using the algorithm in Fig. 5 we can define another operator $reachable(r)$ as notation for the fixed point of collecting reachable configurations starting with the region r. For backward reachability one can define an analogous reachability operator for $pre(c)$.

For the railroad crossing in the example the safety condition which the modelled system has to fulfil is: '$error \cap reachable(initial) = \emptyset$', where *error* is the region where the gate is not closed and the train is within the 250 m distance to the gate, and *initial* is the starting region of the system where the train is far away (and x>=1000), the controller is in the idle-state, and the gate is open (and g=90).

In general, the algorithm for the fixed point computation does not necessarily terminate after a finite number of steps, but in the example it does so.

3.3 Abstraction and implementation relation

The CTA modelling language is capable of modelling a modular system by using a composition hierarchy. Several different levels of such a hierarchy can be used to express different abstraction levels of the system.

In each refinement step of a modelling process working in a top-down manner the more abstract modules are successively replaced by more specific modules with a deeper hierarchy or a more specific behaviour. In Fig. 6 an embedded system consists of an environment and a controller. There are two different versions of each component: an abstract module and a detailed module.

The aim of the modularity concept is not only to have a nice modelling technique but to have a technique for 'modular verification', too. Many real software systems are very large, so that a reachability analysis even with use of symbolic representation is not possible because of time and space complexity.

One solution for this problem is to use modular proving methods. For example, there is a system implementation which consists of two system components named CONTROLLER_IMPLEMENTATION and ENVIRONMENT_IMPLEMENTATION (denoted as CONTR_IMPL || ENV_IMPL) and we have to prove the safety property P. If the whole system is too complex for automatic analysis, it might be possible to prove the properties with the system CONTR_IMPL || ENV_ABST where ENV_ABST is a more abstract model of the environment than ENV_IMPL. Now, we can use for proving that the safety property P of the system CONTR_IMPL || ENV_ABST is valid if the following two proofs are valid:

- System CONTR_IMPL || ENV_ABST has safety property P and

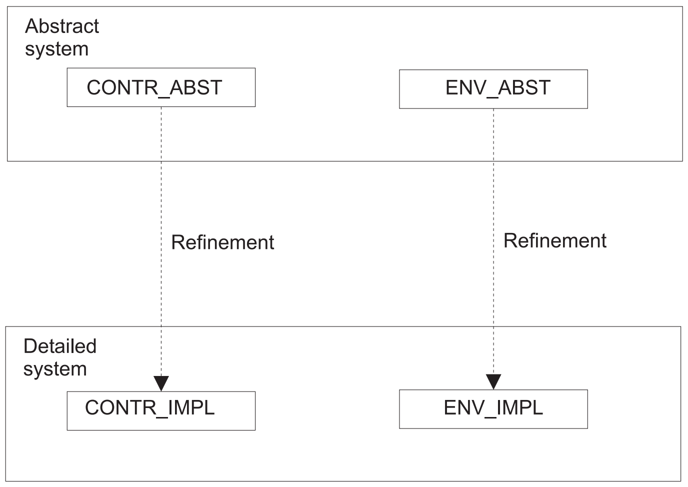


Fig. 6. Modelling by Refinement

– ENV_IMPL implements ENV_ABST.

It is expected that these two steps are easier to compute than the complete proof in one step if the abstractions are selected sensibly. For the first step we use reachability analysis and for the second step we use the method described in the following paragraph.

The intuition behind our implementation concept is an assumption/guarantee principle. We describe it with respect to our formalism: An implementation relation (m_2 implements m_1) for hybrid modules m_1 and m_2 has to fulfil the following properties (G set of signals, V set of variables, I input, O output, MR multiply restricted):

– $m_1.GI \subseteq m_2.GI$. The occurrence of a signal g as input in a module m means that m guarantees that g is not restricted in m. This clearly is a guarantee. Thus each input signal of the specification should be an input signal of the implementation. The same is sensible for input variables $m.VI$.
– $m_2.GO \subseteq m_1.GO$. The occurrence of a signal g as output in a module m means that m assumes that g is not restricted in the environment. The implementation should not make more assumptions than the specification, thus each output signal of the implementation should also be an output signal in the specification. The same is sensible for the output variables $m.VO$.
– $m_1.G - m_1.GL = m_2.G - m_2.GL$. The signals of a module m can be partitioned into a set of interface signals ($m.GI \cup m.GO \cup m.GMR$), and a set of local signals ($m.GL$). Interface signals are those via which m can communicate with the environment. The same is sensible for the variables $m.V$.
– The external trace set S_2 (defined below) of the transition system generated by m_2 is a subset of the external trace set S_1 of the transition system generated by m_1. We use the set theoretical conceptualisation of implementation of Abadi and Lamport [AL91]. They use an implementation relation for sets of traces. They consider a set of traces S_2 to be an implementation of the set of traces S_1 if and only if $S_2 \subseteq S_1$. Their intuition is that the occurrence of a trace t in S_1 means that S_1 *allows* the system behaviour t, and they consider that the implementation should not allow more behaviours than the specification.

Let $L = m.G \cup \mathbb{R}$ be the set of transition labels of the transition system of a module m. $m.G$ is the set of synchronisation labels of the module m. $\mathbb{R}$ is used for the time values. A **trace** of a given transition system is an infinite alternating sequence $< c_0, l_1, c_1, \ldots >$ of configurations and elements of L, starting with an initial configuration c_0. For each (c_i, l_{i+1}, c_{i+1}) the following must hold: If l_{i+1} is a synchronisation label then a discrete transition of the transition system leads from c_i to c_{i+1}; otherwise l_{i+1} is an element of $\mathbb{R}$, the time width between the two configurations, and a time transition of the transition system of that time was taken.

To introduce the notion of an external trace for the behaviour visible outside the module we have to define a function $hide : C \rightarrow E$ which maps a configuration which contains also the information about local variables and the discrete state to a non-local configuration which contains only the value assignments

```
MODULE RailRoadCrossing {
  LOCAL
    app  : SYNC;
    exit : SYNC;
    lower: SYNC;
    raise: SYNC;
  INST Process_Train FROM Train WITH {
    app AS app;
    exit AS exit;
  }
  INST Process_Gate FROM Gate WITH {
    lower AS lower;
    raise AS raise;
  }
  INST Process_Controller FROM Controller WITH {
    app AS app;
    exit AS exit;
    lower AS lower;
    raise AS raise;
  }
}
```

Fig. 7. The model of the rail road crossing system

of the interface variables (input, output, and multiply restricted) by hiding the local components ($C = S \times A(VI \cup VO \cup VMR \cup VL); E = A(VI \cup VO \cup VMR)$).

An **external trace** of a given transition system is an infinite alternating sequence $< e_0, l_1, e_1, \ldots >$ of non-local configurations and elements of L which is constructed by the $hide$ function from a trace of the transition system:
$< \ldots, e_i, l_{i+1}, e_{i+1}, \ldots > = < \ldots, hide(c_i), l_{i+1}, hide(c_{i+1}), \ldots >$.

4 System description

An introduction to the CTA languages is given in this section. The first subsection describes the language to define the model and the second one describes the language for the analysis commands.

4.1 Modelling language

A module contains four components: declaration of variables and signals, an initial configuration of the module, a set of instantiations and a set of hybrid automata.

One of the modules defined in a system description is the so called 'top module' which contains all the stuff needed to model the system. In the top module of the example model all the variables are declared as LOCAL because it is a closed system. (Open system have at least one variable or signal of restriction type INPUT or MULTREST.)

To provide an intuition for the notation, the textual CTA version of the example system is shown in some figures. The top module is the module 'RailRoadCrossing' (Fig. 7), which models the system consisting of the train (Fig. 8), the gate (Fig. 9), and the controller (Fig. 10).

Declaration of the interface The modelling language of CTA has two orthogonal type classifications for identifiers. There are four different restriction types to distinguish between different access modes:

- Local variables or signals are not visible outside the current module. They are used for communication between submodules and the automaton of the current module.
- Input variables or signals may not be restricted in the current module. This means for signals that the current module in every configuration must be able to react on such a signal.
- Output components may be restricted in the current module, but not outside the module. This means for signals that the decision when this signal appears is only taken in the current module, not in any other one.
- Multiply restricted variables may be restricted both in the current module and outside it (like normal global components).

```
MODULE Train {
  OUTPUT
    app: SYNC;
    exit: SYNC;
  LOCAL
    x: ANALOG;    // Distance of the train.
  INITIALIZATION {
    STATE(Train) = Far AND x >= 1000; }
  AUTOMATON Train {
    STATE Far {
      INV { x >= 1000; }
      DERIV { DER(x) >= -50 AND DER(x) <= -40; }
      TRANS { GUARD { x = 1000; } SYNC !app; GOTO Near}
    }
    STATE Near {
      INV { x >= 0; }
      DERIV { DER(x) >= -50 AND DER(x) <= -30; }
      TRANS { GUARD { x = 0; } GOTO Past}
    }
    STATE Past {
      INV { x <= 100; }
      DERIV { DER(x) >= 30 AND DER(x) <= 50; }
      TRANS { GUARD { x = 100; } SYNC !exit; DO { x' > 1000; } GOTO Far}
    }
  }
}
```

Fig. 8. The train model

```
MODULE Gate {
  LOCAL
    g: ANALOG;    // Degree of the gate.
  INPUT
    lower: SYNC;  // Lower the gate.
    raise: SYNC;  // Raise the gate.
  INITIALIZATION {
    STATE(Gate) = Open AND g = 90; }
  AUTOMATON Gate {
    STATE Open {
      INV { g = 90; }
      DERIV { DER(g) = 0; }
      TRANS { SYNC ?raise; GOTO Open }
      TRANS { SYNC ?lower; GOTO Down }
    }
    STATE Up {
      INV { g <= 90; }
      DERIV { DER(g) = 9; }
      TRANS { SYNC ?raise; GOTO Up }
      TRANS { SYNC ?lower; GOTO Down }
      TRANS { GUARD { g = 90;} GOTO Open }
    }
    STATE Down {
      INV { g >= 0; }
      DERIV { DER(g) = -9; }
      TRANS { GUARD { g= 0;} GOTO Closed }
      TRANS { SYNC ?raise; GOTO Up }
      TRANS { SYNC ?lower; GOTO Down }
    }
    STATE Closed {
      INV { g = 0; }
      DERIV { DER(g) = 0; }
      TRANS { SYNC ?raise; GOTO Up }
      TRANS { SYNC ?lower; GOTO Closed}
    }
  }
}
```

Fig. 9. The gate model

For example, the controller in Fig. 10 has two input signals to communicate with the train. They should be declared as input because the train forces the controller to react on the train actions. The controller must be able to synchronise at each point in time with such input signals. For the communication with the gate there are two output signals to control the gate. The declaration of a signal as output means that the environment must be able to react on it. The clock ’t’ is used to measure the time to react on a signal in the controller.

```
MODULE Controller {
  LOCAL
    t: CLOCK;    // Timer for the controller.
  INPUT
    app: SYNC;
    exit: SYNC;
  OUTPUT
    lower: SYNC;  // Lower the gate.
    raise: SYNC;  // Raise the gate.
  INITIALIZATION {
    STATE(Controller) = Idle; }
  AUTOMATON Controller {
    STATE Idle {  // Waiting for a signal from train.
      TRANS { SYNC ?app;  DO { t’ = 0; } GOTO ToLower}
      TRANS { SYNC ?exit; DO { t’ = 0; } GOTO ToRaise}
    }
    STATE ToLower {
      INV { t <= 5; }
      TRANS {SYNC ?app; GOTO ToLower }
      TRANS { SYNC !lower; GOTO Idle }
      TRANS { SYNC ?exit; DO { t’ = 0; } GOTO ToRaise }
    }
    STATE ToRaise {
      INV { t <= 5; }
      TRANS { SYNC ?exit; GOTO ToRaise }
      TRANS { SYNC !raise; GOTO Idle }
      TRANS { SYNC ?app; DO { t’ = 0; } GOTO ToLower }
    }
  }
}
```

Fig. 10. The controller model

To provide an expressive syntax, there are five different data types, as in HyTech:

- CONST: A variable with a fixed value.
- DISCRETE: A variable which can be used to store a value. The derivation of a discrete variable is always zero.
- CLOCK: A continuous variable with the fixed time derivation one.
- STOPWATCH: A clock which can be stopped. This means the time derivation can be zero or one.
- ANALOG: A continuous variable without restrictions for the time derivation.

The controller model of the railroad crossing has one clock as continuous variable and four signals.

Instantiations To get a model for the whole system, the module ’RailRoadCrossing’ contains three instantiations of the other modules. Each of the interface variables and signals from the template module has to be identified with one of the variables or signals from the containing module. Local variables or signals are not identified with one of the containing module.

Automata, states and transitions A hybrid automaton consists of a set of states. For each state, there is a restriction to set the first time derivations of all the variables and another restriction to define an invariant which must be fulfilled while the automaton stays in the state.

The transitions are syntactically contained in the source state of the transition. A transition consists of a guard, a synchronisation label, a restriction to determine the values of the variables after the transition, and the follower state. Such a discrete transition can only fire if the guard (a restriction over the variables) is fulfilled and all other automata which also use the synchronisation label perform a transition with this label, too.

Initialisation The starting state of an automaton and the initial value assignment for the variables are set by the INITIALISATION clause.

Linear restrictions Linear restrictions are sets (disjunctions) of inequality systems to restrict the value assignments of the variables. They consist of conjunctions of constraints which use the operators $<$, $<=$,

```
REGION CHECK RailRoadCrossing {
  VAR
    initial, error, reached : REGION;
  COMMANDS
    // We use the initial regions from the modules.
    initial := INITIALREGION;
    error   :=   COMPLEMENT( STATE(Process_Gate.Gate) = Process_Gate.Closed )
               INTERSECT
                 Process_Train.x < 250;
    reached := REACH FROM initial FORWARD;
    IF( EMPTY(error INTERSECT reached) )  {
      PRINT "Safety requirement satisfied.";
    }
    ELSE {
      PRINT "Safety requirement violated.";
      PRINT "The resulting region:" reached " !";
    };
}
```

Fig. 11. The analysis section for the railroad crossing model

$>$, $>=$, $=$, and $<>$. The expressions contained in the constraints must be linear, which means that the expression must be an additive combination of numbers or variables which can be multiplied with a number.

A variable x can occur in three forms in linear restrictions:

- **x** addresses the value of x. In an assignment of a transition it means the value of x before the transition is executed.
- **x'** is used only in assignments of transitions to address the value of the variable after execution.
- **DER(x)** is used only in the DERIV-clause of a state to address the first time derivation of a variable.

Name spaces Each module has its own name space. All names of variables and signals are only known to the module. To allow communication between different modules the interface components can be identified with some components of the containing module, respecting consistency conditions. Local components must not occur in an identification of interface components.

A module contains several different name spaces: Automata names, state names, variable and signal names, instantiation names. For the module names only one global name space exists.

4.2 Verification language

The analysis language provided by the tool is described in the following section. It is similar to HyTech's analysis language [HHWT95], but extended to handle the different name spaces coming from the hierarchical name space structure.

As in the previous section the example is used for illustration. In Fig. 11 the analysis section for the verification of the railroad crossing model is displayed.

The main idea is to compute with regions. There are variables for storing regions which are needed in the further verification process, and statements to say how to compute the next step.

Declaration of region variables As illustrated in Fig. 11 three regions are declared: 'initial' to have the region for the starting point of reachability analysis, 'reached' for the region representing the state space which is reachable from 'initial', and the region variable 'error' for the region which should not appear in a correct model.

Region expressions A region expression consists of operations to build regions either from existing regions or from linear restrictions, state restrictions, or region variables.

Region restrictions To define a region one can use linear restrictions over the variables of the module on the one hand, e.g. 'Process_Gate.g $>=$ 0' to define a region of all configurations which fulfil the condition that the variable 'g' of the instantiation 'Process_Gate' is greater than zero. On the other hand a region can be defined by a state restriction, e.g. 'STATE(Process_Gate.Gate) = Process_Gate.Open' to define the region where the state of the automaton 'Gate' of the instantiation 'Process_Gate' is the state 'Open'.

Reachability operations For computation of regions which can be reached by using discrete and time transitions from a given region there are four reachability operations: ’POST(<region>)’ and ’PRE(<region>)’ to compute the follower (or precursor) region using one time or discrete transition, and ’REACH FROM <region> FORWARD’ and ’REACH FROM <region> BACKWARD’ to compute the fixed point region using iterative ’POST’ (or ’PRE’) operations.

Set operations To build new regions by set operations the language provides three binary operators and one unary operator: ’INTERSECT’ for the intersection of two regions, ’UNION’ for the union of two regions, ’DIFFERENCE’ for the difference of two regions, and ’COMPLEMENT’ for the complement of a region.

Region variable A region also can be defined by an existing region which is already computed and stored in a region variable.

Initial region To access the initial region defined in the section ’INITIALISATION’ of the module description can be used in the analysis section by the keyword ’INITIALREGION’.

Convention for access to components. Because of the different name spaces, we use the dot notation to address a special component in a ’Region restriction’ in a ’REGION CHECK’ section. In the analysis section all variables and all states are available. The current name space is the name space of the top model used for analysis, i.e. to access a component ’x’ of the module ’Process_Train’ the absolute name $Process\_Train.x$ is used, and the discrete state ’Far’ of the automaton ’Train’ contained in the module instance ’Process_Train’ is accessible by $Process\_Train.Far$.

Boolean expressions A Boolean expression is used as a predicate over regions for statements which use a condition to decide what they have to compute. Such a boolean expression consists of comparisons between and checks of regions as well as boolean combinations of such expressions.

Comparisons To evaluate set relations between two regions there are the operators ’=’ which returns true if both regions are the same sets, and the operator ’CONTAINS’ which returns true if the first region contains the second one. To check whether a region is empty, one can use the keyword ’EMPTY’.

Combinations of boolean expressions Boolean expressions can be combined by the usual operators ’AND’, ’OR’, and ’NOT’.

Statements To formulate the verification tasks different statements can be used. Possible statements are assignments, if-then-statements, loops, and printing states.

Assignment One can assign a computed region to a previously declared region variable for further use with an assign statement ’<regvar> := <regexpr>’.

Conditional statement With the keywords ’IF’, ’THEN’ and ’ELSE’ one can evaluate boolean expressions and depending on the result execute some statements.

Iteration To define iterative verification processes a statement ’WHILE’ is provided with the expected meaning.

Output of results To put messages out on the screen, the statement ’PRINT’ is possible with different arguments. The argument of the print statement can be a string, a region expression or a region expression between two strings. A region is printed out as a set of inequalities over the variables and conditions about the discrete states. The print statement of the code in Fig. 11 illustrates it.

The statements in the example define three regions: the initial configurations, the error region (which the model has to avoid), and the region which can be reached starting from the initial region. The ’IF’ statement evaluates whether the computed region is empty or not.

5 CTA tool environment

The first version of the CTA tool is implemented in C++ and consists of the following components:

- A compiler front-end for the **system description** to build the hierarchy of hybrid modules in memory.
- A library which provides the data structures and algorithms for the double description method (DDM) **symbolic representation of the regions** [FP96].

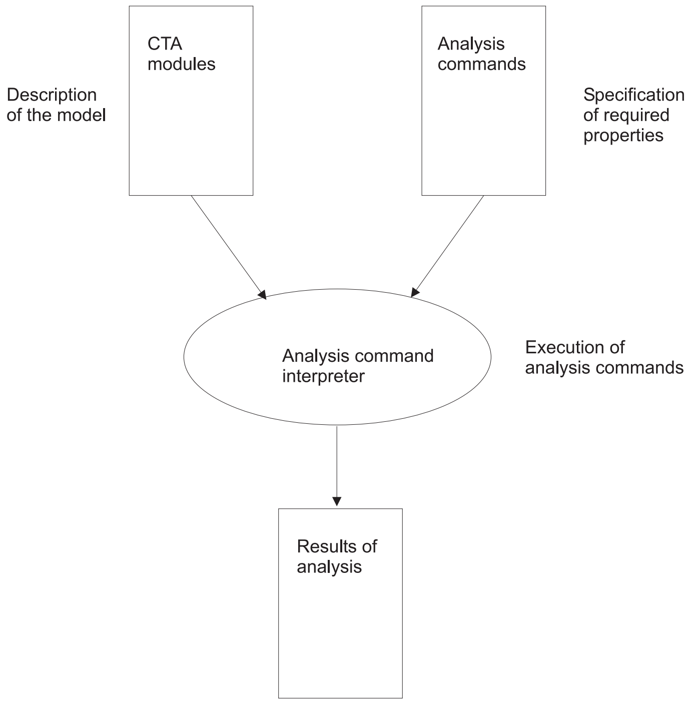

Description of the model
CTA modules
Analysis commands
Specification of required properties
Analysis command interpreter
Execution of analysis commands
Results of analysis

Fig. 12. CTA Tool

– An interpreter for the analysis language which provides several analysis commands for **reachability analysis**. It transforms parts of the model needed in the verification into a DDM representation and invokes the analysis tasks on the data structures.

The tool works in the way depicted in Fig. 12: There are two inputs, a description of the model as CTA modules and a specification of required properties as analysis commands. First, the tool analyses the model regarding context conditions like consistency and compatibility of combined modules.

After that, the analysis command interpreter executes the analysis commands and writes out the results of the verification process. To avoid problems with the name spaces, the module which will be used for analysis has to be declared in each region section.

At least for reachability analysis we can only use one single hybrid automaton. From this it follows that we have to transform our hierarchically structured system of communicating modules to a flattened normal form. This normal form of CTA consists of a set of hybrid automata without scopes, special data types and restriction types of variables/signals as well as without abstraction layers as a 'flat' system. This is done with the help of our tool after context check and some additional analysis. The second step is to produce a product automaton from the set of hybrid automata.

If we have not modelled all necessary transitions for reacting on input signals in an automaton, the tool adds these transitions automatically, but they lead to a special error state 'INPUT_ERROR'. If such a state exists, we can analyse if this state is reachable. If it is reachable then we have modelled a situation where an input signal is restricted, and thus we have a modelling mistake.

5.1 Region representation

From the real-valuedness of the variables follows the infinity of the set of configurations of any automaton with at least one analogous variable. Thus, for the analysis the tool has to use a symbolic representation of configuration sets. In the CTA tool a region is represented by sets of pairs (s, a), where s is a discrete state and a is a set of convex polyhedra in $\mathbb{R}^{|V|}$, which are limited by hyper-planes. For operations over the polyhedra the tool uses the double description method. A region (as a set of configurations) is represented by a map which assigns to each state the corresponding set of polyhedra (if it is not empty).

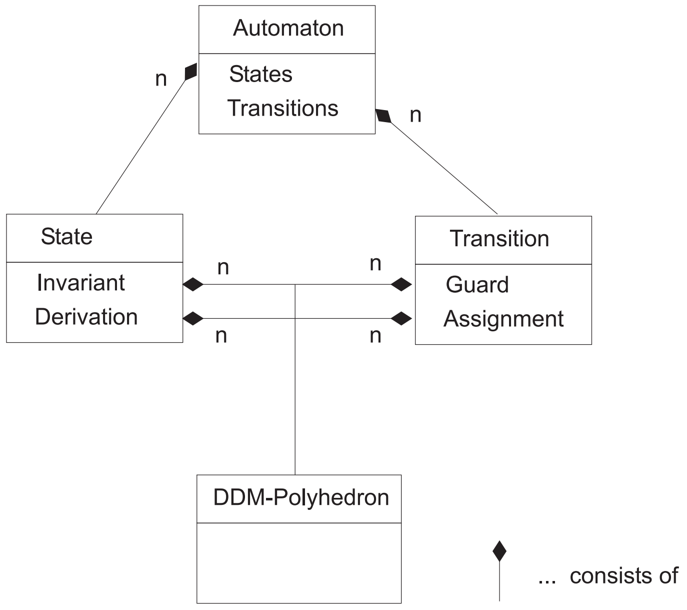


Fig. 13. Structure of automaton representation

Fig. 13 shows a rough overview of the internal representation with focus on the main data structure: The polyhedra. One automaton consists of a set of states and a set of transitions. Each state has two important components, one set of polyhedra for the state invariant and another set of polyhedra for the derivation. We have to use sets of polyhedra because the data structure with the efficient algorithms represents only convex polyhedra. Each transition has two such sets, the guard and the allowed assignments.

The component for the polyhedron representation is the most important one because all the algorithms of automaton components are based on the basic algorithms like intersection, unions and emptiness check for such polyhedra.

5.2 Results of the verification of the example

For analysis of the railroad crossing example we used the code from Fig. 11. Starting with the initial configuration of all automata the tool computes the region of reachable configurations. The region which violates the safety condition of the example is stored in another region. If the intersection of the reached region and the error region is nonempty, the model violates the condition. Using the model given in former sections the tool computes that the intersection is empty and thus the model fits the safety conditions. After execution of an reachability statement the tool gives a message how many steps were used to reach the fixed point.

The verification task of the example which is displayed in Fig. 11 fails if we increase the reaction time 'α' of the controller or if we increase the safety distance in which the train must not be before the gate is closed.

6 Open Questions

The verification of modelled systems with a lot of combined automata (i.e. more than five automata) is an open problem of the current version of the tool. As in other tools (i.e. HyTech), the technique used in the CTA-tool does not use symbolic representation for the discrete part of a configuration.

In our tool the region representation is separated from the rest of the tool through a clear interface. Thus, we are able to plug in other representation techniques. In the next step the research group is going to investigate the possibilities of BDD data structures for representing continuous state spaces and the discrete state space together.

Acknowledgements

We thank the anonymous referees for their comments.

References

[ACD93] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking in dense real-time. *Information and Computation*, 104:2–34, 1993.

[AH96] Rajeev Alur and Thomas A. Henzinger. Reactive modules. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS 1996)*, pages 207–218, 1996.

[AH97] Rajeev Alur and Thomas A. Henzinger. Modularity for timed and hybrid systems. In *Proceedings of the 8th International Conference on Concurrency Theory (CONCUR'97)*, LNCS 1243, pages 74–88, Berlin, 1997. Springer-Verlag.

[AL91] Martin Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[BLL$^+$96] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Petersson, and Wang Yi. Uppaal – a tool suite for automatic verification of real-time systems. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Hybrid Systems III*, LNCS 1066, pages 232–243, Berlin, 1996. Springer-Verlag.

[BR98] Dirk Beyer and Heinrich Rust. Modeling a production cell as a distributed real-time system with cottbus timed automata. In Hartmut König and Peter Langendörfer, editors, *FBT'98: Formale Beschreibungstechniken für verteilte Systeme*, pages 148–159, June 1998.

[BR99] Dirk Beyer and Heinrich Rust. A formalism for modular modelling of hybrid systems. Technical Report 10/1999, BTU Cottbus, 1999.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Hybrid Systems III*, LNCS 1066, pages 208–219, Berlin, 1996. Springer-Verlag.

[FP96] Komei Fukuda and Alain Prodon. Double description method revisited. In *Combinatorics and Computer Science*, LNCS 1120, pages 91–111. Springer-Verlag, 1996.

[HHWT95] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. A user guide to HyTech. In *Proceedings of the First Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 1019, pages 41–71. Springer-Verlag, 1995.

[HNSY94] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model-checking for real-time systems. *Information and Computation*, 111:193–244, 1994.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Hemel Hempstead, 1985.

[Lev95] Nancy G. Leveson. *Safeware*. Addison Wesley, Reading/Massachusetts, 1995.

[LSVW96] N. Lynch, R. Segala, F. Vaandrager, and H.B. Weinberg. Hybrid I/O automata. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Hybrid Systems III*, LNCS 1066, pages 496–510, Berlin, 1996. Springer-Verlag.

[LT87] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151. ACM, August 1987.