Concepts of Cottbus Timed Automata

Dirk Beyer and Heinrich Rust

Lehrstuhl für Software Systemtechnik, BTU Cottbus
Postfach 10 13 44
D-03013 Cottbus, Germany
Tel. +49(355)69-3803, Fax.:-3810, Email:{db|rust}@informatik.tu-cottbus.de

1 Introduction

Today, many industrial production cells are controlled by software. Many such systems have to deal with requirements which the developer has to guarantee. Because of the complexity of the implementation one of the main problems for developing the software for reactive systems is to be sure that such properties are fulfilled. One way to handle the problems is to use formal methods: This means to develop a formal model which is used to prove the properties of the specification with tool support.

There are many different methods to model such reactive systems. Some of these abstract from real-time aspects of the system. We chose a problem area where we have real-time requirements, for example the throughput of the modelled production cell. So we have to use formal methods which support models of real-time systems.

In the past we looked for automata-based approaches. We used the concepts of timed and hybrid automata, for which there exist a well developed theoretical base ([AD94], [Hen96]) and some tools implementing these concepts as well as supporting reachability analysis.

We used one of the tools, HyTech, for collecting experience in the domain of automata-based modelling techniques [Rus99]. For use in software development processes, this tool has some disadvantages, for example:

- There is only one name space, i. e. variable names are global.
- A system has to be modelled as a flat set of communicating automata, modular structure can not be expressed.
- There is no template mechanism to parametrize general parts of the system.

We extended the hybrid automaton theory to include the missing concepts by introducing CTAs [BR98]. We presented a refined notation which introduces the following concepts:

- Hierarchy: Subsystem descriptions can be grouped. Interfaces and local components are separated.
- Explicit handling of different types of communication signals. We allow to express explicitly that an event is an input signal for an automaton, an output signal, or a multiply restricted signal.
- We allow to express explicitly that an analogue variable is accessed by an automaton as output, as input, or that it is multiply restricted.
- Automatical 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.

– Recurring subsystem components do not have to be multiply defined. They are instantiated from a common module type.

In the following sections of this paper we extend CTAs for answering the following questions:

– How do we handle restrictions for value assignments in transitions of composable automata?
– How do we create the product automaton for a set of automata?
– How do we use proved properties of parts of the system to prove properties of the whole system?

1.1 CTA module description

The following section contains an informal description of CTA modules:

A Cottbus Timed Automaton consists of a set of modules. One of these is designated as the main module. It models the whole system. The other modules are used as types. They can be instantiated several times in other modules. This makes it possible to express a hierarchical structure of the system, and to define recurring components of a system just once.

Each module consists of the following components:

– An **identifier.** A system description might contain several modules. We use identifiers to name them.
– **Signals.** Signals are used for communication between modules running in parallel. Signals are modeled after 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.
– **Automaton.** The current module contains an automaton. This automaton consists of a finite set of states, a finite set of transitions between these states, and a signal alphabet.
  With each state, we associate the following:
  • An **invariant**. This is a predicate over analogue variables. 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.
  • A condition for the **derivatives** of analogue variables. As long as the system stays in a state, the derivatives of the variables must fulfill the condition.

  With each transition, we associate the following:
  • A **guard**. Like the invariant of a state, this is a predicate over the variables of the module. One condition for the transition to be taken is that the guard is true.
  • A **signal**. For the transition to be taken, all other modules having the signal in their alphabet must do a transition which is labeled with this signal.
  • A set of **allowed** assignments to analogue variables. When the transition is taken, the variables may get new values. If several transitions involving assignments to the same variables are performed synchronously, an assignment is performed which fits all components. If there is no such assignment, the transition may not be taken.
  • A set of **initiated** assignments to analogue variables. The automaton use this restriction for variables which are not restricted by any other automaton in a transition executed in parallel.

- **Initial condition.** An initial condition is another component of a module. This is a predicate over the module variables and the states of the module's automaton.
- **Instances.** The current 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 **identification** 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.

A formal definition is described in [BR99].

At every point in time we describe the situation of a CTA with its current state (one element from the state set S) and the current value assignment of all variables of the automaton. A **configuration** c of a CTA is defined as the pair $c = (s, a)$ from the set of all configurations $C = S \times A(V)$, where $s \in S$ is the current state and $a \in A(V)$ is the current value assignment. From the real-valuedness of the variables follows the infinity of the set of configurations of any automaton with at least one analog variable. Thus, for the analysis we have to use symbolic representation of configuration sets. A **region** $r \in R = \mathcal{P}(C)$ is a set of configurations. In the CTA notation these regions are described by conjunctions of (in-) equations, i. e. all CTA regions are convex and limited by hyperplanes. In this way an equation system defines a polyhedron in $\mathbb{R}^{|V|}$.

Variable restrictions in transitions. The value of a variable in the CTA notation is decribed by a set of possible values to allow nondeterminism. When a transition of the automaton is performed, then the variables change their values. Such value changes 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 $x' \geq 3 * x + 7$).

2 Extended concepts used by CTA

In this section we describe informally the concepts we have developed over the last months.

2.1 Double transition predicates

The value changes in transitions are restricted by two predicates called 'allowed' and 'initiated'. These two predicates are conjunctions of linear inequalities containing ticked (for the value after the transition) and unticked variables (for the value before the transition). Both predicates are defined with a single syntactical predicate introduced with the keyword 'ALLOW' in a transition description.

- **allowed:** The ALLOW clause in a transition contains a set of inequalities to describe the possible value changes. That means the values of the variables before and after the transition have to fulfill the inequalities. If there are environmental automata then all the value changes of the transitions which are taken in parallel (same point in time) must be consistent with this predicate. From another point of view this predicate describes which value changes are allowed to be performed by the environment.

– **initiated:** This predicate contains all restrictions of 'allowed' and additionally some more restrictions. 'Initiated' describes the value changes initiated by this transition. A typical case to use this 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 of the ALLOW clause, then this does not mean that the whole range of $\mathbb{R}$ is possible. For a variable which does not occur ticked in 'ALLOW' 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 would use the clause ALLOW $\{x' > 0$ AND $x' \leq 0\}$.

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 the 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 ALLOW. The consequence for us is to generate the 'initiated' set automatically, i. e. we do not have a syntactical clause for INITIATE in our notation.

Note. All variables which occur in an 'ALLOW' clause must be declared as OUTPUT, LOCAL, or MULTREST.

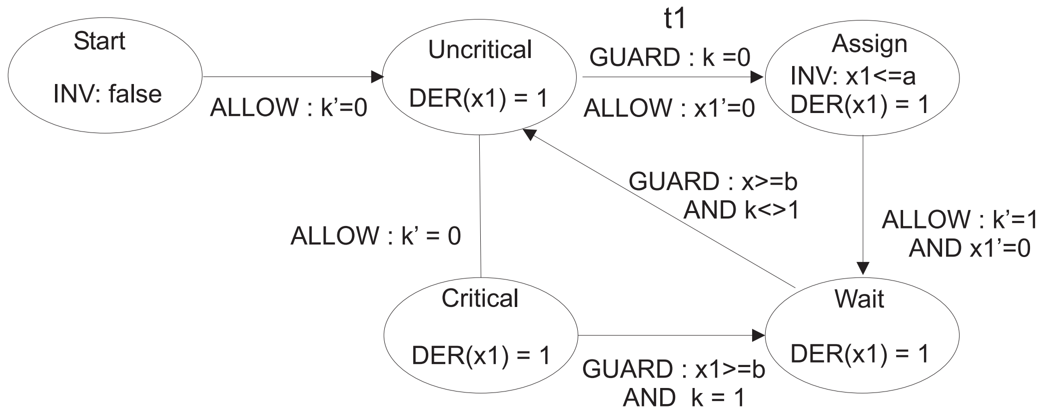
To illustrate the intention of our double transition predicates, we display the automata for Fisher's timing-based mutual exclusion protocol in Fig. 1. In our Fisher automaton a process is modelled by five states. 'Start' is the initial state for the automaton. From here it takes a transition initializing the shared variable k to the state which models the uncritical section. From this state only one transition is possible: If the shared variable signifies that no process is in the critical section then the process can try to go to critical section. It goes to the state modelling the 'Assign' statement. The clock x_i (with time derivation 1 in all states) measures the time staying in this state, and the invariant forces to leave the state after time a, which models the maximal time needed by the assign statement of the process. Then the transition to the 'Wait' state sets the variable k to the number of the process. In this state we have to wait at least time b to give other processes a chance to set k to its process number. After time b the process can decide to enter the critical section if $k = i$. Otherwise it goes back to the uncritical section (to try it again). Leaving the critical section the automaton sets k to value 0 to signify that the resource is free again.

The ALLOW clause at transition t_1 of process p_1 defines both the 'allowed' set and the 'initiated' set. The restriction for the 'allowed' set is $x' = 0$, which defines the new value of variable x after the transition is taken. The 'initiated' set is additionally restricted by the convention described above: $x' = 0$ AND $k' = k$. Thus variable k is not changed by this transition although k could be changed by a transition executed in parallel to this transition.

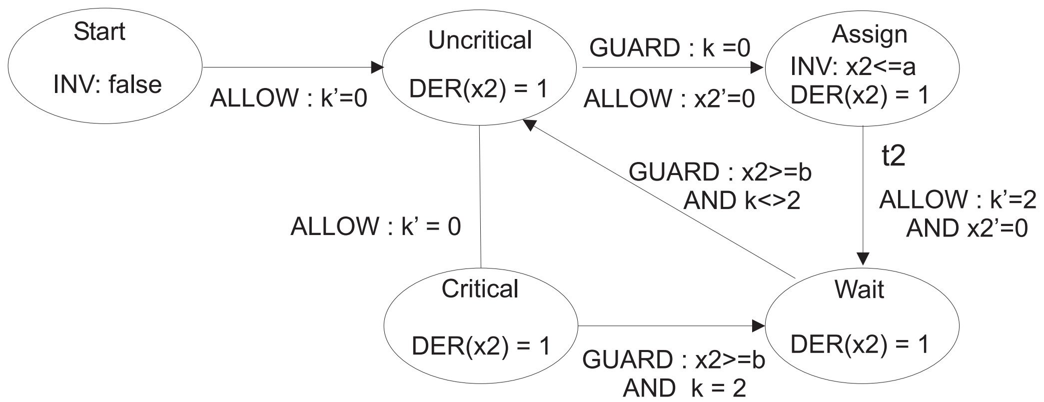
The advantage of the distinction beetween 'allowed' and 'initiated' is that we can express that the value of a variable is not changed by a transition in one automaton but this transition allows a value change by a transition in a parallel automaton. If, for example, p_1 is in state 1 and it takes transition t_1 then we have two possible situations in process p_2:

– p_2 takes t_2 at the same point in time. The only valid value of variable k is 2 because of the ALLOW clause of t_2.

Process 1 (p1):



Process 2 (p2):



INPUT	a: CONST;	b: CONST
LOCAL	x1: STOPWATCH;	x2: STOPWATCH
MULTREST	k: DISCRETE	

Fig. 1. Fisher’s mutual exclusion protocol

– If p_2 does not take t_2 in parallel, then the 'initiated' set of process p_1 forces $k = 0$.

2.2 Normal form and product automaton

At least for reachability analysis we can only use one single hybrid automaton. From this it follows that in a first step we have to transform our hierarchically structured system of communicating modules to a flattened normal form. This normal form of CTA consist 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. 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 intersection 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 transition with same signal or one of the combined transition is a noop transition.
– **Invariants** and **derivatives** are intersected.
– **Guards** are intersected with the additional condition that the 'initiated' set must be not empty.
– The new **signal** of a transition is the signal used by on of the parallel transitions.
– The new **allowed** set is the intersection of the two allowed sets.
– The new **initiated** set is the subset of the initiated set of the first automaton which is allowed by the second automaton united with the subset of the initiated set of the second automaton which is allowed by the first automaton.

Almost everything in this construction is standard, only the handling of 'allowed' and 'initiated' is special.

2.3 Implementation relation

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, we have 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 automatical 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
– 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 fulfill the following properties (G set of signals, V set of variables, I input, O output):

- $m_1.GI \subseteq m_2.GI$. The occurrence of a signal g in a module $m.GI$ 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 in a module $m.GO$ 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 trace set S_2 of the transition system generated by m_2 is a subset of the trace set S_1 of the transition system generated by m_1. We use the set theoretical conceptualization 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.

3 Tool support

We are developing a verification tool for CTA models. In the current version we support the analysis of a hierachical model and the transformation to the normal form. From this we can build a model which can be analysed with the HyTech tool. But the CTA models which we can translate to HyTech are a restricted subset of all CTA models. Thus we are developing our own implementation of the techniques.

We developed a library to support the representation of regions as polyhedra and all the algorithms we need. For this we use the ddm method which is described in [Che68].

The complete tool will work in the following way (c.f. Fig. 2): Our tool needs two different inputs: The model description of the system consists of a set of modules. These modules are defined by the CTA model notation. The other input is a set of analysis commands which we use to verify the required properties of the system specification. The main verification techniques are reachability (forward and backward) analysis and proving implementation relations between modules.

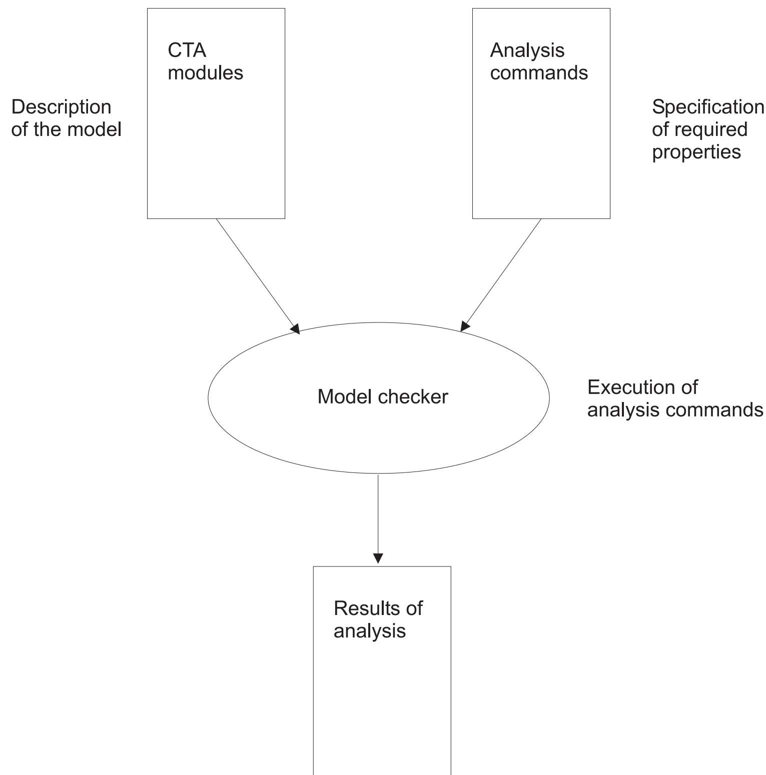


Fig. 2. CTA Modelchecker

4 Discussion

In our formalism we solved some important problems of the existing automata based methods. In CTA it is possible to develop system models with modular structure and abstraction layers for controlling the complexity of real systems. We have concepts to support an assumption/guarantee principle. The tool we developed supports the most important analyses.

In our further work we will investigate the maturity of our tool in applying our formalism to more complex problems.

References

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

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

[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 formal definition for a modular hybrid modelling notation. Technical Report I-3/1999, BTU Cottbus, 1999.

[Che68] N. V. Chernikova. Algorithm for discovering the set of all solutions of a linear programming problem. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 8(6):282–293, 1968.

[Hen96] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS 1996)*, pages 278–292, 1996.

[Rus99] Heinrich Rust. Modeling a production cell component as a hybrid automaton. Technical Report I-2/1999, BTU Cottbus, 1999.