Modeling a Production Cell as a Distributed Real-Time System with Cottbus Timed Automata

Dirk Beyer and Heinrich Rust*

Lehrstuhl für Software Systemtechnik, BTU Cottbus

Abstract. We build on work in designing modeling languages for hybrid systems in the development of CTA, the Cottbus Timed Automata. Our design features a facility to specify a hybrid system modulary and hierarchically, communication through CSP-like synchronizations but with special support to specify explicitly different roles which the interface signals and variables of a module play, and to instanziate recurring elements serveral times from a template. Continuous system components are modeled with analogue variables having piecewise constant derivatives. Discrete system aspects like control modes are modeled with the discrete variables and the states of a finite automaton. Our approach to specifying distributed hybrid systems is illustrated with the specification of a component of a production cell, a transport belt.

1 Introduction

During the last years, several tools have been developed which allow to model hybrid systems as consisting of a discretely structured portion given as a finite automaton, and of a set of continuously varying variables modeling continuous components of the system. Examples are UppAal [BLL+96], Kronos [DOTY96] and HyTech [HHWT95].

These formalisms and their associated tools have been successful in illustrating the usefulness of following concepts:

- Analogue variables with varying range-defined time-derivatives to model the coninuous part of system, i.e. to give time restrictions.
- Discrete variables and states for modelling the discrete part of system.
- Invariants and guards for states and transitions of the finite-automaton portion of the system description.
- The construction of larger systems as parallel composition of subsystems communicating synchronously via CSP-like (cf. [Hoa85]) signals.
- Several possibilities to express nondeterminism which is especially important for modelling environments and hardware components.

We build on this work, and especially on HyTech, in the development of a further refined notation for the modular description of hybrid systems. Our notation 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.

* Reachable at: BTU, Postfach 10 13 44, D-03013 Cottbus, Germany; Tel. +49(355)69-3803, Fax.:-3810; {db|rust}@informatik.tu-cottbus.de

– 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.

We will illustrate our approach to the specification and verification of hybrid systems with the example of a component of a production cell.

The structure of the paper is the following: In Section 2, we will describe the transport belt. This is the production cell component we will use to illustrate our formalism. In Section 3, we introduce the concepts, and in Section 4, we describe the structure of the example specification we consider to be helpful for the specification of distributed systems.

2 A transport belt in a production cell

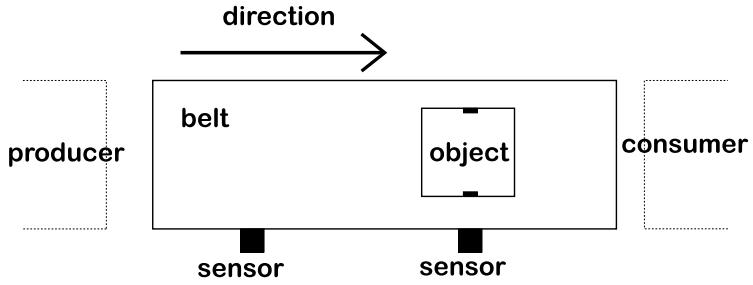

Fig. 1. The belt

We illustrate our approach by modeling a transport belt of a production cell (cf. Fig. 1). This transport belt has the following set of physical features:

– The belt can be either moving or stopped. The direction of movement is fixed. Thus, we can distinguish an end from where objects are fed to the belt and an end to which objects are transfered. We will call them the producer end and the consumer end.
– The belt is equipped with two sensors for objects. One sensor is positioned near the producer end, the other near the consumer end. Each sensor has two sensitivity radii. The smaller radius defines the region in which an object is sensed safely, the larger radius defines the region out of which an object is never sensed. In the region between the larger and the smaller radius, the transitions of the sensor from the off-state to the on-state and vice versa occur.
– The belt has a length and a speed (when switched on).

The control program of the belt fetches objects from the neighbour at the producer end, transports them to the consumer end, and transfers them to the neighbour positioned at that end. It might be described in the following way.

– Wait until the producer wants to transfer an object.
– Tell the producer that the transfer can start.
– Start belt motion.

– Wait until the object has been transfered to belt.
– Tell the producer that the transfer has been accomplished.
– Transport object to other end of belt.
– Stop belt motion.
– Tell the comsumer that an object is to be transfered.
– Wait for the consumer to tell that the transfer can start.
– Start belt motion.
– Wait till the consumer tells that transfer has been accomplished.
– Stop belt motion.
– Start program at the beginning.

The context of the belt consists of two neighbours:

– There is a producing neighbour. Its functionality is modeled as repeated transfer of objects to the belt.
– There is a consumer neighbour. Its functionality is modeled as repeated receipt of objects from the belt.

3 Cottbus Timed Automata: Informal description

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 defi ne 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 identifi ers to name them.
– **Signals.** Signals are used for communication between modules running in parallel. Signals are modeled after CSP-like events. This means the following:
  - Each module has an alphabet of signals.
  - A transition in a module can be labeled with a signal. If a transition is labeled with a signal, this transition can only be taken if in the same moment, all other modules which have this signal in their alphabet also take a transition which is labeled with this signal.

  It will become clear later that a module may contain as control components both instances of other modules and a local automaton. The parallel composition of all these control components can again be described as an automaton. It is this automaton which is considered when synchronization with interface signals becomes important.
  The basic communication model only allows synchronous communication. This communication model abstracts from the differences between inputs and outputs. We will discriminate between these types of signals. For this, we use the notion of a **signal restriction**. We say that a signal is restricted in a module if the module does not in every confi guration admit the signal.
  Signals can have one of four access types:
  - Signals declared as 'local' are not visible outside the current module. They are used for communication between submodules and the automaton of the current module.

- The other access types for signals are 'input', 'output' and 'multiply restricted'. Signals declared with these access types belong to the interface of the module. This means that when the module is instantiated, these signals can be identified with signals which are declared in the instantiating module.
  In the sequel, we will call the module in which a signal is declared the "current module".
  The different access types for interface signals make different usage types of these signals in the system explicit:
  * Input signals may not be restricted in the current module. This means that the current module must in every configuration be able to react on such signal.
  * Output signals may be restricted in the current module, but they may not be restricted in any other module. This means that the decision when this signal appears is only taken in the current module, not in any other one.
  * Multiply restricted signals may be restricted both in the current and in other modules.

– **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. Thus, the value of a variable is influenced via two mechanisms:
  - For each discrete state the system may be in and for each variable, the CTA determines a range of values for the time derivative of the variable. This allows to model continuous changes of an analogue system component.
  - When a transition is taken, a new value can be assigned to a variable. This models discrete changes.

  The values of variables are used to model how long the system may stay in a given discrete control mode, and to model if a given transition from one control mode to another can be taken. This is done by formulating invariants and guards for states and transitions. These invariants and guards are predicates over the values of the analogue variables.
  A **variable restriction** is a notion similar to that of a signal restriction. We say that a variable is restricted in a module if its derivative is defined in a state of the module, or if the variable is assigned to in a transition of the module.
  For variables like for signals, we have four different access modes:
  - Local variables are not accessed outside the current module.
  - Input variables may not be restricted in the current module.
  - Output variables may be restricted in the current module, but not outside of the module.
  - Multiply restricted variables may be restricted both in the current module and outside of it.

  All but local variables belong to the interface of a module. Like for interface signals, this means that when the current module is instantiated, these variables may be identified with variables declared in the containing module.
– **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.
  Instances of previously defined modules and the explicitly defined automaton run in parallel. As described earlier, they may communicate through the use of common signals. The signal alphabet of the automaton defines the set of signals with which the automaton must synchronize when they occur.
  With each state, we may 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.

This does not mean that the automaton can never be in a state in which the invariant is false. When an invariant is not fulfilled, this only means that time may not pass while the automaton is in this state. Thus, this forces a discrete transition to be taken.
- 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 may 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 assignments to analogue variables. When the transition is taken, the variables 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.

– **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 component (a signal or a variable) of the containing module may not be associated with more than one interface component. This ensures that inside one module, there are no aliases for one signal or variable.

  In determining if a signal or variable is restricted, restrictions in instances are considered as restrictions in the current module.
  When a previously defined module is instantiated in the current module, not all of its interface components must be identified with signals and variables in the containing module. Typically output signals and variables which are not needed in the environment are left out.
  An instance represents an automaton. The alphabet of the automaton corresponding to the instance is the set of signals which are identified with interface signals of the instantiated module.

3.1 Notation

Later, we present some example specifications which illustrate some of our notational choices (cf. Fig. 2, 3, 4):

– Keywords are written with capital letters. A module definition is started with the keyword `MODULE`.
– At the beginning of the module, signals and variables are declared. The keywords `INPUT`, `OUTPUT`, `MULTREST` and `LOCAL` introduce sections for the different types of signals and variables.

– The INITIALIZATION-keyword introduces the initial condition for the automaton. The initial conditions for variables are given as arithmetical predicates, initial conditions for states are given with equality expressions having the form STATE=s for automaton states s.
– The keyword AUTOMATON defines an automaton. In braces, the set of states of the automaton is described.
  Each state is introduced with the keyword STATE. It has a name. In braces, the components of the state are described:
  - INV introduces the invariant. If it is missing, it is assumed that the invariant is identically true.
  - DERIV introduces the restrictions for variables. If it is missing, it is assumed that there are no restrictions for the derivatives in this state. For a variable v, the expression DER(v) represents the time derivative of v.
  - TRANS introduces transitions. First, a set of target states is given. If there is more than one target state it means that there is a nondeterministic choice of target states. Afterwards, the components are given in braces:
    * GUARD introduces the guard. If this clause is missing, the guard is assumed to be identically true.
    * A SYNC-clause gives a synchronization signal. The signal name is prefixed by a `?' for input signals of the automaton, by a `!' for output signals and by a `#' for multiply restricted signals. A missing SYNC-clause indicates that the transition is not labeled with a signal.
      If the SYNC-signal is followed by a THEN and another signal, this denotes a notational shortcut. It is explained in Section 3.2.
    * A DO-clause introduces information about discrete value changes of the analogue variables.
– INST is the keyword used for instantiations. First the name of the instance is given, then a FROM-clause gives the name of the module which is instantiated, and after the keyword WITH, a set of identifications of module interface components with signals and variables of the context module are given. Each such identification has the form v1 AS v2. v1 must be a signal or variable of the interface of the instantiated module, and v2 must be the signal or variable of the context module with which v1 is identified.

3.2 Notational conveniences

To make usage the formalism easier, we introduce some conventions:

– Each automaton must have an alphabet of signals with which it must synchronize. This is not given explicitly, but derived implicitly from the set of signals occuring in any transition of the automaton. The signs ?, ! and # are used to mark signals as input, output or multiply restricted.
– We allow a disjunction of follower states in a transition. Such a transition is equivalent to several transitions which only differ in the follower state.
– In the SYNC-clause of a transition, we can give a further signal after a THEN. This is translated into a two-step transition. The first step leads to an internal state with an identically false invariant (thus it is urgent) and with one transition which is labeled with the signal given behind the THEN. This makes it easy to specify signal translations.

– By their definition, input signals may not be restricted in a module, but often, we know from the context that in some configurations, a given input signal should never occur. It would be inconvenient if we were forced to declare transitions for the input signals in these configurations. We complete the automaton automatically in the following way:
  - Generate a further state for the automaton. We call it the error state.
  - For each state s in the automaton and for each input signal i, do the following:
    * Let T be the set of transition outgoing from s which are labeled with i.
    * Let g be the disjunction of all guards of the transitions in T.
    * If g is identically true, state s does not restrict the input signal i. No completion is necessary.
    * If g is not identically true, insert a new transition from s to the error state labeled with i and guarded with the negation of g. After this completion, the automaton can react on i in every configuration for state s.

    This algorithm also generates transitions from the error state into the error state for all input signals.
  - If all transitions leading into the error state come from the error state, it is superfluous, because the automaton was complete before. We remove it and all transitions involving it.

4 The example specification

4.1 Global structure

For systematic modular specification of distributed hybrid systems, we chose to restrict ourselves to a specific style to structure our specification. It contains the following components, some of which are further hierarchically subdivided:

– One module describes the belt hardware. This includes the module describing belt and piece movement and the two sensors.
– One module describes the control program.
– One module describes the belt context. The two neighbours (a piece producer and a piece consumer) are modeled independently and then combined, via instantiation, to a module encompassing the whole context.
– One module contains specialized test automata which make verifications easier.
– The module `MBeltSystem` represents the total system. There are no interface signals or variables: This module thus describes a closed system. It consists of the composition of all other modules.

In the next section we describe each component separately.

In the context of distributed systems, the distinction between a context model and the system model proper is especially important. The context model typically is a simplifying abstraction of the true context of the system. If we can prove critical system properties for the system in this simplified context, and if we can prove that the true context models implement these abstractions, we can deduct that the critical system properties are fulfilled for the system model in its true context. We plan to apply the ideas of Abadi and Lamport (cf. [AL93]) to our formalism: They investigate for which classes of properties we can deduct properties of the total system from proved properties of the system components.

```
--This module computes the output of a sensor on the belt.
MODULE MSensor
{
    INPUT
        in: SYNC;
        pos: ANALOG;
        sens_pos: CONST;
    OUTPUT
        sens_on: SYNC;
        sens_off: SYNC;
    LOCAL
        --Some local constants to model the range of sensor.
        sensor_radius_min = 3: CONST;
        sensor_radius_max = 5: CONST;
    --Initially,we assume to be in the off-state after the sensor
    --  and the plate is not on the belt.
    INITIALIZATION
    {
         STATE = off_after;
    }
    AUTOMATON
    {
        --In this state,the sensor is off because the plate is too far
        --  before the sensor.
        STATE off_before
        {
            --We stay here as long as the plate has not
            --  reached the minimal range of the sensor.
            INV pos <= sens_pos -sensor_radius_min;

            --After the plate has reached the maximal sensor range,we
            --  can generate an on-signal and go to the on-state.
            TRANS on
            {
                GUARD
                {
                       pos >= sens_pos -sensor_radius_max;
                }
                SYNC !sens_on;
            }
        }
        --We can stay here as long as we are in the on-state,that is,
        --  as long as the plate is in the maximal sensor range.
        STATE on
        {
            INV pos <= sens_pos + sensor_radius_max;
            --We can leave this state whenever the plate leaves the
            --  minimal sensor range.
            TRANS off_after
            {
                GUARD
                {
                       pos >= sens_pos + sensor_radius_min;
                }
                SYNC !sens_off;
            }
        }
        --This location represents the situations where the plate
        --  is far after the sensor.
        STATE off_after
        {
            --We leave this state when another plate is put onto
            --  the belt.
            TRANS off_before
            {
                SYNC ?in;
            }
        }
    }
}

```

Fig. 2. The sensor model

4.2 The model of the belt hardware

The belt hardware is modeled hierarchically with the modules `MBeltMovement`, `MSensor` and `MBeltModel`. We will describe `MSensor` in some detail.

`MBeltMovement` describes what happens when the belt moves. It determines the position of a plate.

The module `MSensor` (cf. Fig. 2) describes the behaviour of a sensor.

- **Inputs.** The signal `in` tells the sensor if a new plate has entered the belt. `pos` is read by the sensor model to determine the position of the plate, and the constant `sens_pos` represents the sensor position on the belt, measured in the same way as the plate position as distance from feeding end of the belt.
- **Outputs.** The sensor module generates two signals which represent the sensor switches: `sens_on` is sent when the sensor switches on, and `sens_off` is sent when the sensor switches off.
- **Locals.** Two radii are used in the sensor model to express indeterminism about the distance the center of the plate is allowed to have from the sensor to be sensed successfully. `sensor_radius_min` is the radius around the sensor position such that a plate with a center in this radius is sensed surely. `sensor_radius_max` is the radius out of which a plate is never sensed. While the plate travels between these radii, the sensor switches on respectively off.
- **Initial condition**. We assume in the beginning that the sensor is off and the plate is behind the sensor, not before it.
- **Automaton.** We model the sensor with three states: Two represent off-states of the sensor, one for situations in which a plate is before the sensor, the other for situations in which a plate is behind the sensor. One represents the state when the sensor is on. The invariants and guards of these states express what has been described previously about the use of the two radii.

The module `MBeltModel` combines the previously defined modules into a single module representing the transport belt without its control program. The sensor module is instantiated twice, for two different sensor positions, because the belt has two sensors.

4.3 The environment model

The environment model consists of three modules, two to model each of the neighbours of the belt, and one which instantiates these two and represents the whole of the context.

`MBeltInContext` (cf. Fig. 3) specifies the behaviour of the plate producing neighbour. It has a cyclical behaviour: First it allows a transfer to be started via the signal `#start_in_transfer`. After some indeterminate time, the signal `!in` is emitted. This represents the moment in which the plate reaches the producing neighbour's end. After this, the automaton waits for a message that the plate has been transferred successfully (`?stop_in_transfer`), and starts from the begin.

`MBeltOutContext` is similar to `MBeltInContext`: It specifies the behaviour of a plate consumer: It also waits for a start signal, then it waits for a signal to occur which tells us that the plate has reached the producer's end, and then, after some time, a stop-message is produced.

`MBeltContext` (cf. Fig. 4) instantiates the two previously discussed modules. It combines their interfaces.

```
--This module models the neighbour producing plates.
MODULE MBeltInContext
{
    INPUT
        stop_in_transfer: SYNC;

    OUTPUT
        in: SYNC;

    MULTREST
        start_in_transfer: SYNC;

    INITIALIZATION
    {
        STATE = start;
    }
    AUTOMATON
    {
        --Wait some time,then try to start a transfer from
        --   the input belt to modeled belt.
        STATE start
        {
            --After some time,send a start-transfer-signal
            --  to input belt.
            TRANS in_transfer
            {
                SYNC #start_in_transfer;
            }
        }

        --Accomplish a transfer from input belt to belt.
        STATE in_transfer
        {
            --After some indeterminate time,output a load-message.
            TRANS wait_for_stop
            {
                SYNC !in;
            }
        }

        --Wait for message about the accomplished transfer.
        STATE wait_for_stop
        {
            TRANS start
            {
                 SYNC ?stop_in_transfer;
            }
        }
    }
}

```

Fig. 3. The input context model

5 Discussion

Theoretical results for existing hybrid automata models (cf. [Hen96]) carry over to our model. This makes automatized verification methods possible, also using symbolic methods for the representation of configurations. Like in the HyTech tool, one important analysis technique is forward- and backward-reachability analysis for given regions. Currently, we are implementing a tool allowing automatized analysis of systems modeled as CTA.

In our example, we presented an approach for the modular specification of larger hybrid systems. The proofs which are possible for the transport belt are only valid if the neighbouring production cell components implement the abstractions given in the automata which model the environment of the transport belt.

```
1  --This module describes the context of belt.
2  MODULE MBeltContext
3  {
4      INPUT
5          --This signal is received by output belt from belt1
6          --  when the plate leaves belt.
7          out: SYNC;
8          --This signal is received from input belt when the transfer
9          --  has been accomplished.
10         stop_in_transfer: SYNC;
11
12     OUTPUT
13         --This signal is generated by input belt when the plate
14         --  leaves it.
15         in: SYNC;
16         --This signal is generated by output belt when the transfer
17         --  has been accomplished.
18         stop_out_transfer: SYNC;
19
20     MULTREST
21         --On this signal synchronize the input belt and
22         --  the modeled belt when a transfer may start.
23         start_in_transfer: SYNC;
24         --On this signal synchronize the modeled belt and
25         --  the output belt when a transfer may start.
26         start_out_transfer: SYNC;
27
28
29     --Here we instantiate the two context modules.
30     INST iBeltInContext
31     FROM MBeltInContext
32     WITH
33     {
34         in AS in;
35         start_in_transfer AS start_in_transfer;
36         stop_in_transfer AS stop_in_transfer;
37     }
38
39     INST iBeltOutContext
40     FROM MBeltOutContext
41     WITH
42     {
43         out AS out;
44         start_out_transfer AS start_out_transfer;
45         stop_out_transfer AS stop_out_transfer;
46     }
47 }
48
```

Fig. 4. The context model

A detailed model of the total system might be too large for automatic analyses. Thus, an analysis method taking into account the modular structure of the system would be helpful. We investigate how we can apply the results of Abadi and Lamport [AL93] in our context.

Acknowledgements

We thank the HyTech-team for their work. It was a source of inspiration and ideas to us.

References

[AL92] Martin Abadi and Leslie Lamport. An old-fashioned recipe for real time. In J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*, LNCS 600, pages 1–27, 1992.

[AL93] Martin Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.

[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.

[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.

[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.

[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.

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