A CSP Model for Hybrid Automata

Peter Amthor

Bremen Institute for Safe Systems, PO 330440, D-28334 Bremen

Fax: +49 421 218-3054

E-Mail: amthor@informatik.uni-bremen.de

<http://www.informatik.uni-bremen.de/~amthor/>

Abstract. This contribution aims to combine hybrid automata with CSP establishing a connection between the elements of hybrid automata and CSP processes using an approach for the decomposition of hybrid systems. As a result we obtain a transformation from hybrid automata into CSP processes, written in Hoare-CSP in the untimed parts and in real-time CSP or HCSP in the timed parts. These processes provide a basis to build up analysis mechanisms like animation and testing, converging in tool support for the specification, transformation, animation, implementation and test of hybrid systems.

1 Introduction

The Bremen Institute for Safe Systems (BISS) aims to develop safe systems using formal methods and provides feasible system development support which is also practicable in industrial applications. One focus in our research is "automated test support for reactive systems" which includes generation, execution, monitoring and evaluation of test cases, as introduced by Peleska and Siegel in [15]. This approach is based on CSP by Hoare [13] and on Hennessy's testing methodology [10]. Currently available tools which deal with CSP specifications only provide a textual representation to describe systems. Here, it would be a useful improvement to have a graphical representation language for the specification process.

Hybrid systems seem to be appropriate to be described in a graphical way. Here, we have continuous components as well as discrete ones. In particular the continuous behaviour can be better specified and investigated with the help of graphical elements, but we won't lose the benefits of the CSP based test automation approach [15]. The solution arises by the combination of test automation for reactive systems based on CSP with the approach of hybrid automata (cf. [1]). The model of automata is a well known and wide spread one, which is in our case very suitable, especially with regard to the presentation of specifications to industrial partners which might not be familiar with other more abstract specification models.

In this paper we first introduce hybrid systems and explain an approach to decompose them into appropriate components. Then hybrid automata are introduced. Following the decompositional approach for hybrid systems, we develop a CSP model for hybrid automata, which is used to realize a work bench which includes existing tools like HyTech [12], FDR [8] and VVT-RT [15] to provide specification, animation, debugging and testing facilities for hybrid systems.

2 Hybrid Systems — A Decompositional Approach

Hybrid Systems consist of discrete and continuous components. Usually, they are modelled as reactive systems, i. e. digital controllers interact with a continuously changing environment. Several variables can be observed in a hybrid system representing e. g. temperature, light intensity or noise level. This leads us to several questions concerning hybrid systems, e. g.:

- Which variables are used and what are their values at a certain time?
- Which functions are chosen to modify the variables' values?
- Which timing constraints appear in this system?
- Which components interact with each other?
- What kind of algorithm is implemented by the considered system?

Fig. 1. Decomposition of a Hybrid System.

One way to find answers to these questions is to analyse the hybrid system and separate the parts relating to certain questions. An appropriate approach to achieve this aim is the decomposition of hybrid systems as shown in figure 1.

Following this, information about variables can be found in a separate module. In order to observe the evolution of variables this particular module has to be investigated. The (external) communication between the components of a hybrid system builds another block. Further, every component of a hybrid system consists of an untimed, timed and hybrid part.

In the hybrid part one finds the functions which change the variables' values continuously over a time interval. Sometimes one would like to distinguish the hybrid behaviour from the so-called timed behaviour. In this case it is useful to locate the discrete change of variables within the so-called timed part. Often, it is more useful to include the timed behaviour into the hybrid behaviour, so one can omit the dashed region of fig. 1. A little example illustrating this aspect is the modelling of a timer. Usually, it is modelled to be discrete in time and value. Initialized with a value, e. g. 5, it is decreased stepwise by one until its value is zero or an interrupt, perhaps a timer reset, occurs. But it is also convenient to consider it as follows: again it is set with a value, then it is continuously changed by the function $\dot{t}=-1$ as long as the condition $t > 0$ holds or it is interrupted.

At last, we get an untimed system which is solely focused on the description of actions within the system and which is connected with the hybrid part (and with the timed part) via internal communication (event synchronization).

Because of this strict separation it is possible to use model checking in the untimed part with FDR. This allows also more comfortable testing from the behavioural view, because in the untimed part pure access to the events/actions of the system is possible, which cause the hybrid behaviour of the system by synchronization. Further, the decompositional approach can be used to ensure that an untimed part which has been checked beforehand can be integrated into the overall system without losing the validity of the results obtained, paying attention to side conditions related to the decomposition. This proposal also leads to a decompositional implementation concept which allows easy approaches to animation and monitoring mechanisms. Finally, the components of a decomposed hybrid system could be included into a library for their reuse for another hybrid development. These components should be used like templates that have only to be filled, which means the stepwise extension of a system, starting with the untimed system, modelling the hybrid part, declaring the variables and defining the external communication.

3 Hybrid Automata

Hybrid Systems can be modelled as the parallel composition of hybrid automata representing the hybrid systems' components¹. These automata model the continuous activities of analog variables (such as temperature, time, distance, etc.) as well as discrete events (such as interrupts or output signals). Communication between automata is modelled through event synchronization and shared variables.

A hybrid automaton consists of vertices connected by edges. A vertex is called location representing control modes where variables can be changed continuously. Edges are also called transitions which represent discrete events.

Fig. 2. Thermostat Automaton.

A small example is the one of an thermostat automaton (see fig. 2)². The hybrid system includes only one variable `temp` representing the temperature and one hybrid automaton called `thermostat`. It starts at location `init` and takes the outgoing transition immediately because of its jump condition `asap` assigning 20 to `temp`. This leads the automaton to location `off` where temperature falls according to the flow (rate) condition `dtemp=-1`. Here, the automaton may go on when the temperature falls below 17 degrees and it must go on at the latest when it falls to 15 degrees. In control mode `on` according to the invariant condition

¹ The introduction of (linear) hybrid automata follows [11] and [12].

² According to the example shown in [11].

temp<=25 temperature rises till 25 degrees at the most. Its outgoing transition to off may be taken when temperature has reached more than 23 degrees.

Variables are real-valued and shared. In every state of the automaton there exists a **valuation** for each variable mapping it to its value.

Locations are used to model control modes. A **state** of an automaton consists of a location and a valuation. A **region** is a set of states. Usually, a location has a name, an **invariant condition** and a **flow (rate) condition**. The automaton control may reside in a location only while its invariant condition is true. The rate condition defines the rates of change of the variables' values. In the original model of hybrid automata the initial location is denoted by a small incoming arrow labelled with an initial condition. We omit it by defining the initial location as the vertex with no incoming edges. A state is called **admissible** when its valuation satisfies the location's invariant condition.

Transitions model discrete events and connect locations. Each transition is labelled with an **update set** and with a **jump condition**. It is enabled if its jump condition is true and if it reaches an admissible state after it took place, i. e. the variables in the update set have changed their values. Transitions may be assigned the **urgency flag asap**³. As soon as they are enabled, these transitions must take place. Finally, each transition can be given a synchronization label. These labels are used to define the parallel composition of hybrid automata and must be taken simultaneously by two automata sharing the same synchronization label.

4 A CSP Model for Hybrid Automata

After the introduction of hybrid automata, we would like to present a CSP model that corresponds to the approach of the hybrid system's decomposition. The general structure of such a CSP process which models a hybrid automaton is shown in figure 3. Here, one can recognize the separation of the hybrid automata, the external communication between them and the global variables. The CSP model for a hybrid automaton includes several processes which can be expressed in different kinds of CSP. First, we have the processes **ELEMENTS** and **NEXT** modelled with untimed CSP [13]. They include the vertices and edges with their connections to model the structure of the automaton and to define the causal behaviour of the automaton's components. Further, some processes can be written in hybrid CSP [6]. They are used to access and to change the (continuous) variables. To observe the continuous change of variables a certain granularity is introduced by a timer process, which can also be modelled with real-time CSP [7] instead of hybrid CSP. All components are connected by channels, which are used in CSP to model communication between processes. In the following we like to describe each process and its interaction with the others more detailed.

³ as soon as possible

Fig. 3. CSP Model for Hybrid Automata.

According to the hybrid system's decomposition, first, we divide the composition of hybrid automata into several pieces: the global variables, the hybrid automata and the external communication. Then each hybrid automaton will be translated into CSP using the CSP model. The resulting CSP process includes elements from untimed, hybrid and real-time CSP, but only the processes modelled in untimed CSP are taken to generate a transition graph with the FDR tool. All other parts have to be implemented otherwise, to realise animation of the whole hybrid system. Nevertheless the other parts are described with CSP⁴ in the following, too, but we don't try to feed FDR with them because on the one hand to avoid state-explosion and on the other hand the tool cannot cope with specifications including time.

4.1 Variables and External Communication

A suitable way of modelling variables in CSP is the following. The process `VAR` has two parameters `varname` and `value`. A possible initialization might

⁴ For the untimed parts we use the FDR notation, the other parts are written in the notations proposed in [6] and [7].

be `VAR(x,1)`, variable `x` has the initial value one. Further, one can set or get values using the channels `write` and `read`, which are uni-directional. If a message/value arrives for variable `x` over channel `write`, then `x` is set to `newValue`. Otherwise, if another process requests the current value of `x`, then `VAR` passes it over the `read` channel.

```
VAR (varname, value) =
  (write.varname ? newValue -> VAR (varname, newValue))
[]
(read.varname ! value -> VAR (varname, value))
```

Now, one is able to declare several variables initializing for example three variables with zero. The three processes are running in parallel, but they are interleaved (`|||`) and not synchronized with each other.

```
VARIABLES = VAR(x1,0) ||| VAR(x2,0) ||| VAR(x3,0)
```

The external communication can be separated by establishing a communication process which synchronizes two automata for communication. Consider the two processes `P1`, `P2`. Before the separation step they synchronize directly over the event `b`.

```
P1 = a -> b -> P1    P2 = d -> b -> P2    SYNC = P1 [|{b}|] P2
```

The separation step can now be done by introducing a process `COMM` and by splitting and duplicating the synchronization event `b`. So we receive the following specification.

```
COMM = (b.1 -> b.2 -> b.1 -> b.2 -> COMM)
[]
(b.2 -> b.1 -> b.1 -> b.2 -> COMM)

P1 = a -> b.1 -> b.1 -> P1    P2 = d -> b.2 -> b.2 -> P2

SYNC = COMM [|{b}|] (P1 ||| P2)
```

Here, we have the advantage that the two processes `P1`, `P2` are only interleaved, this means, that we can now model them independently from the others. In the end this allows us to get a small transition graph for each hybrid automaton instead of a very large one for all.

Finally, the recomposition of all components is the following parallel composition, where `COMMUNICATION` is the process containing the interleaving of all `COMM` processes appearing in the system and `sync_1, ..., sync_m` are the belonging synchronization events. `HA_1, ..., HA_n` are the hybrid system's components modelled with hybrid automata.

```

HYBRID_SYSTEM = (
    VARIABLES ||| COMMUNICATION
)
[|{read,write,sync_1,...,sync_m}|]
(
    HA_1 ||| ... ||| HA_n
)

```

4.2 The CSP Components for a Hybrid Automaton

The **timed parts** of the CSP model for a hybrid automaton are modelled as auxiliary processes which can be used by the processes in the untimed part via synchronization.

Locations are used to model the continuous change of a variable. The process is started by synchronization over `doLoc.i`, then it gets the values from the variables which are used in the functions `f_1, ..., f_n`. With the synchronization event `startTimer` the timer process is started. Further, the continuous change of the variables goes on as long as the timer is not elapsed. The timer induces a certain rate (granularity) to the continuous change process. Having received the `timerElapsed` event the new variable values are sent to the `VARIABLES` process. Then the end of the location process is denoted by `doneLoc.i`.

```

LOCATION_i = doLoc.i ; readVariables ; startTimer ;
              ⎧  $\dot{x}_1 = f_1$ 
              ⎪ ...
              ⎩  $\dot{x}_n = f_n$ 
              ⌋  $\mathbf{t}$  timerElapsed ; writeVariables ;
doneLoc.i ; LOCATION_i

```

The timer process used in the `LOCATION` process is a nice example where one can take either hybrid CSP or real-time CSP to model it. In this case, the real-time modelling seems to be more compact.

```

TIMER = startTimer  $\xrightarrow{1}$  timerElapsed -> TIMER

```

In the hybrid modelling the local timer variable `t` is set to zero at the beginning. Then the value is continuously increased till the condition `t=1` becomes true and the timer elapses.

```

TIMER = startTimer ; t:=0 ; (<  $\dot{t} = 1$  >  $\rightarrow$  t=1) ;
timerElapsed ; TIMER

```

Transitions are used to model discrete actions, like the discrete assignment of values to variables or like commands modelled through synchronization. The `TRANSITION` process starts with the `doTrans.i` event. Then it also reads the variables' values which are necessary for the following computation. Further, a synchronization event may precede or succeed the assignment part, but it is optional. Finally, the `doneTrans.i` event occurs.


```

TRANSITION_i = doTrans.i ; readVariables ; synchronization-event ;
                x1 := f1; ...; xn := fn;
                synchronization-event ; writeVariables ; doneTrans.i ;
TRANSITION_i

```

Additionally, we need processes to evaluate the conditions in locations and transitions. This is done in the `CONDITION` process. It is activated by the event `evalCond.i`. Then all variable values are read which are needed in the condition `cond_i`. According to the evaluation of the condition the events `evalTrue.i` or `evalFalse.i` take place.

```

CONDITION_i = evalCond.i ; readVariables ;
              (cond_i → evalTrue.i ; CONDITION_i) ;
              (not(cond_i) → evalFalse.i ; CONDITION_i)

```

The untimed parts model the structure of the automaton and the causal behaviour of the components in general. One part of the idea is to build a process `E` for every element of the automaton identified by an identification number (`i`) and classified as edge (`e`) or as vertex (`v`). The process `E` can be started by synchronization over `start.i`. In case of a vertex (`c==v`) the continuous change of variables takes place by executing the corresponding `LOCATION` process by synchronization over `doLoc.i` and `doneLoc.i`; in the case of an edge the discrete assignments of an edge are done using the `TRANSITION` process by synchronization over `doTrans.i` and `doneTrans.i`.

```

E (i, c) = start.i ->
(
  if (c == v) then
    (doLoc.i -> doneLoc.i -> SKIP)
  else
    (doTrans.i -> doneTrans.i -> SKIP)
)
; startNext.i -> E (i, c)

```

During the execution of the automaton it has to be decided how long the automaton stays at a vertex and which outgoing edge shall be taken leaving the vertex or having executed an edge which vertex is the destination where the edge points at and is it allowed to enter the vertex. These decisions are made in process `N` which is started by `E` synchronizing over `startNext.i`. Then `N` uses the `CHOOSE` process to get a list of the automaton elements which can be chosen in the next step to continue the automaton's execution. For every automaton element there exists a process `N` containing the elements which are reachable from it considering the automaton structure. Usually, for an edge this is the destination vertex and for a vertex it is the vertex itself plus all its outgoing edges.

```

N (i, ol) = startNext.i -> CHOOSE (i, ol, <>) ; N (i, ol)

CHOOSE (i, ol, cl) =
  if (null(ol)) then
  (
    if (null(cl)) then (stopAutomaton.i -> STOP)
    else
      (|~| x : set(cl) @ start.x -> SKIP)
  ) else (
    evalCond.head(ol) ->
    (
      (evalTrue.head(ol) -> CHOOSE (i, tail(ol), cl^<head(ol)>))
      []
      (evalFalse.head(ol) -> CHOOSE (i, tail(ol), cl))
    )
  )

```

Now, it can be detected by the use of the `CONDITION` process, started by `evalCond.i`, which of the elements are enabled at the current state. The process `CONDITION` evaluates the condition of a certain automaton element and returns the result by synchronization over `evalTrue.i` or `evalFalse.i`. In the case of true the element is added to the list of the next possible events (`cl`) otherwise not. Finally, it is decided by non-deterministic choice (`|~|`) which element is taken from `cl` to continue the automaton's execution and it is started by `start.i`. When `cl` is empty the automaton is not able to continue and the event `stopAutomaton.i` appears.

```

INIT (i) = start.i -> SKIP

ELEMENTS = E (v0,v) ||| E (v1,v) |||
           E (e0,e) ||| E (e1,e) ||| E (e2,e)

NEXT = INIT (v0) ||| N (v0,<v0,e0,e1>) ||| N (v1,<v1,e2>) |||
       N (e0,<v1>) ||| N (e1,<v1>) ||| N (e2,<v0>)

AUTOMATON = ELEMENTS [|{start,startNext}|] NEXT

```

The composition of the `E` and `N` processes shall be explained by the above example. The automaton consists of two vertices `v0,v1` and three edges `e0,e1,e2` declared in five `E` processes which are interleaved in a process `ELEMENTS`. Vertex `v0` has two outgoing edges `e0,e1`, vertex `v1` has `e2`. The edges `e0,e1` lead to `v1`, `e2` leads back to `v0`. These relations are defined in the `N` processes which are interleaved in a process `NEXT`. Additionally, an `INIT` process is included in the interleaving of `NEXT` to ensure that the automaton begins at vertex `v0`. Finally, the processes `ELEMENTS` and `NEXT` are composed in parallel in `AUTOMATON` and they are synchronized over `start` and `startNext`.

5 A Development Environment for Hybrid Systems

The decompositional approach for hybrid systems and the CSP model for hybrid automata have been used for several parts of HySC - a development environment for hybrid systems, which allows us to specify, transform, animate, implement and test hybrid systems (cf. [3]). The implementation combines tools like HyTech (cf. [12]) — a symbolic model checker for hybrid systems — FDR 1.42 (cf. [8]) for the untimed CSP parts and MATLAB (cf. [14]) with its mathematical support. Further, it includes also a graphical user interface with a graph editor both written in Tcl/Tk. HySC tries to include the ideas of well known programming work benches and debuggers, still used in the programmer's life. In this section we focus on the specification and animation part of HySC, showing the usage of the tool with little examples.

Fig. 4. HySC — The Graph Editor.

The specification process starts by creating a hybrid system with the graph editor (fig. 4). Objects like vertices, edges, etc. can be drawn and annotated with labels. In the main window one can define the parallel composition of several hybrid automata. Here, also the variables of the system have to be declared. Then, every automaton has to be modelled in its own window.

One benefit of HySC is the graphical representation of hybrid systems, additionally, one gains another output - a textual representation in a so-called ha-file

(see fig. 5). This ha-file is needed as input for halib, a C-library which contains functions for the analysis of hybrid automata. First, the text of ha-file is parsed, to extract informations like: vertex and edge parameters, occuring variables, conditions and functions.

Fig. 5. HySC — From Specification to Animation.

Secondly, these informations are used to produce a so-called csp-file, which includes the untimed CSP representation of the specified system according to the above introduced CSP model. The FDR tool and another C-library, vvtamlib, produce a transition graph (TG) from the csp-file and offer functions for graph navigation, which is essential for the animation part. In the end, the libraries halib and vvtamlib allow us to:

1. load a hybrid system specified as the parallel composition of hybrid automata,
2. translate it into a CSP process,
3. generate the transition graph of this process and
4. execute the hybrid system by traversing the transition graph of the underlying CSP model.

Different kinds of animation can be done in HySC to execute the specified hybrid system:

- animation for all or for only several components and
- graphical or event based animation.

The amount of hybrid system's components can increase very quickly. So, the animation of the whole system might not be observable in an appropriate way. Therefore, the animation of only one or little more components should be also possible. In this case the components which are not animated have to be compensated by additional interaction of the user.

In the event based animation session, events from the transition graph (generated from the system) can be chosen from the user via a dialog window (fig. 6), where the left list box contains the next possible events and a trace of all chosen events is listed in the right list box.

The graphical animation is also based on the transition graph. But here, every event is related to an element of an automaton. So, instead of offering an event to the user, the related automaton element is highlighted in green color.

The graphical trace is realized by highlighting the executed elements in blue. Additionally, a separate textual trace log might be helpful, too.

Fig. 6. HySC — Event-based Animation.

During the animation/execution of the hybrid system, one is able to observe the values of selected variables in the variable display. When a variable is continuously changed, MATLAB is started and the value change is drawn in a coordinate system. Here, MATLAB is very useful, because curves can be drawn without difficulty (fig. 7) and the conditions of vertices and edges can be evaluated with this tool, too.

Fig. 7. Variable Display and MATLAB.

6 Discussion

In this paper we presented a combination of a compositional approach for hybrid systems with hybrid automata and CSP to provide CSP-based animation for hybrid systems modelled with hybrid automata. The main contribution of this proposal is the development of a CSP model for hybrid automata. We have shown a way how to transform a system described by the parallel composition of hybrid automata into a CSP process which can then be used for the system's animation, traversing the transition graph of the process. These ideas finally led to the implementation of HySC - a development environment for hybrid systems - with specification and animation facilities.

The next goal is to support test automation for hybrid systems. In this case again the CSP model can be used as basis for deriving test cases from the CSP process which represents a hybrid system. These test cases might then be applied to an implementation which has been derived from the specified hybrid system.

Another topic of future work is the modelling of hybrid systems in general. Usually, the controller/environment (C/E-) model described in fig. 8 is often used.

Fig. 8. The Controller/Environment Model.

Considering the example of a heat control system the C/E-model can be applied as follows. Temperature is the critical variable which is measured by the sensor. According to the measured value the controller uses the actuator, e. g. a heating, to increase the temperature. The aim is now to identify more components which are also that important to be modelled in a separate part. One point might be to separate the critical variables strictly from the endangered plants. This could be an advantage when many variables and/or many plants are involved in the system, because then the change of the variables' values can be better modelled and observed.

Finally, the translation of hybrid automata into CSP can be used to integrate specifications described with hybrid automata into a written CSP specification.

References

1. Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems. In [9], pages 209–229, 1993.
2. Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sonntag, editors. *Hybrid Systems III - Verification and Control*, volume 1066 of *Lecture Notes in Computer Science*. Springer, 1996.
3. Peter Amthor. HySC - A Development Environment for Hybrid Systems' Components. In [4], pages 46–50, 1998.
4. Rudolf Berghammer and Yassine Lakhnech, editors. *Tool Support for System Specification, Development and Verification*, number 9803 in Technical Reports of the Institute of Computer Science. Kiel University, 1998.
5. E. Brinksma, W. R. Cleaveland, K. G. Larsen, T. Margaria, and B. Steffen, editors. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95)*, volume 1019 of *Lecture Notes in Computer Science*. Springer, 1995.
6. Zhou Chaochen, Wang Ji, and Anders P. Ravn. A Formal Description of Hybrid Systems. In [2], pages 511–530, 1996.
7. Jim Davies. *Specification and Proof in Real-Time CSP*. Cambridge University Press, 1993.
8. Formal Systems (Europe) Ltd. *Failures Divergence Refinement, User Manual and Tutorial, Version 1.42 for Sun Sparc with Solaris 2*, 1995.
9. Robert L. Grossmann, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors. *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*. Springer, 1993.
10. M.C. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.
11. Thomas A. Henzinger. The Theory of Hybrid Automata. In *11th Annual IEEE Symposium on Logic in Computer Science (LICS 1996)*, pages 278–292, 1996.
12. Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. A User Guide to HyTech. In [5], pages 41–71, 1995.
13. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, 1985.
14. The Math Works Inc. *MATLAB - User's Guide*, 1996.
15. Jan Peleska and Michael Siegel. Test Automation of Safety-Critical Reactive Systems. Technical Report 19, South African Computer Journal, 1997.