Execution Monitor Synthesis for Hybrid Systems — Preliminary Report

Marcus Bjäreland[1]
Dept of Comp and Info Science
Linköping University
S-581 83 Linköping, Sweden
marbj@@ida.liu.se

Abstract

Formal verification methods for hybrid controllers have received a lot of attention during the last decade and so has the problem of synthesizing specifications of hybrid controllers. Such specifications do then fulfill given requirements, such as timing constraints or safety properties. However, the problem of automatically implementing the actual control programs has received much less attention, and one of the reasons for this is that industrial controllers necessarily have to be able to effectively handle every conceivable situation that might occur in reality, which make the construction of the specification very difficult.

In this preliminary report, we take one step towards automatic generation of control systems by proposing that not only the control program should be synthesized from a specification, but also an *Execution Monitor* that is able to detect and classify discrepancies between the model of the closed-loop system and the actual effects of control actions. We show how this can be done for a particular Hybrid Systems example modeled as Hybrid Automata.

1 Introduction

There is a large body of work on the problem of hybrid controller synthesis being done in the Artificial Intelligence (AI) (e.g. [6]), Hybrid Systems (e.g. [9]), and Control Theory (e.g. [10]) communities. Typically, a *specification* of a controller is synthesized from an open-loop specification of the plant and a set of "desired properties". However, the transformation of such a specification to a real control system has not received as much attention ([4, 7] being notable exceptions). One of the reasons for this gap between synthesis and use is that controllers in use necessarily need to be able to handle many more cases than what normally is covered by the verification. According to the folklore of process control, more than 70% of a medium sized PLC program is devoted to fault detection, isolation and recovery. The remaining 30% is taking care of the actual control, and is the part that is suited for verification.

We believe that we are taking one step in closing this gap by proposing not only that controller programs should be synthesized from specifications, but also that *Execution Monitors* (EMs) should be synthesized from the same specifications. We do this by utilizing closed-loop *Hybrid Automata* (HA) specifications [1], assuming that a (reactive) control program has been synthesized, and constructing a rich model where it is possible to reason about the expected effects of control actions. In this preliminary report we show how this can be done for a particular example: A train-and-gate example from [2].

2 Execution Monitoring – Conceptual Framework

Execution Monitoring can be defined as an agent's process of identifying discrepancies between observations of the actual world and the prediction and expectations derived from its representation of the world, classifying such discrepancies, and recover from them. Architecturally, we view the EM as a separate entity connected to the controller of a system, with access to the inputs and outputs of the controller.

From the definition above we can distinguish five *functions* that may constitute an EM:

Situation Assessment: Where the EM computes the current state of the controlled system, based on its representation of it. Answers the question "Where am I?".

Expectation Assessment: Where the EM computes

[1]This research has been supported in parts by the Swedish Research Council for Engineering Sciences (TFR), the Knut and Alice Wallenberg Foundation, and ASEA Brown Bovery (dept ISY/AMC).

the *expected* state of the system, based on its representation. "Where am I supposed to be?"

Discrepancy Detection: The EM compares the results of the situation and expectation assessments, and decides whether a discrepancy between the two has occurred. "Is where I am different from where I am supposed to be?"

Discrepancy Classification: The EM computes a *cause* of the detected discrepancy. "What went wrong?"

Recovery: The EM suggests or performs actions to remedy the faulty situation. "How do I get back on the right track?"

It is unusual to find approaches to EM that implement all five of the functions above in a sophisticated manner. It is also common that some approaches do not explicitly separate two or more of the functions. Despite its simplicity, it is our belief that the framework can be useful to study approaches to EM in a detailed manner.

If we take a simplified analogy to *Fault Detection and Isolation* (FDI) for conventional control systems, situation assessment is not considered a particular problem, while expectation assessment would correspond to *state estimation.* Residual computations then corresponds to discrepancy detection, and fault detection and isolation to discrepancy classification, and, finally, we can view recovery as adaptive, reconfigurable, or learning control (see e.g. [5]). In AI, the focus has been on discrepancy detection, classification and recover, while situation and expectation assessment has, more or less, been taken for granted. For one example on work on EM in AI, see [3].

In this paper we will study the first four functions. Recovery will be postponed to future work.

3 Example – Railroad Crossing

Our running example is a HA model of a railroad crossing presented in [2]. The model consists of three submodels, a train, a gate, and a controller. In [2] the model was used to verify that the controller guaranteed certain safety properties for the overall system. Our use of this example is to illustrate how EM of the controller can be performed by using information embedded in the HA submodel for the train.

The HA model of the train (Fig. 1) has three locations: far, *near*, and *past*, defined as the distance, x, between the train and the gate. That is, $x \geq 1000$ is the invariant for the location far, $x \geq 0$ for *near*, and $x \leq 100$ for *past*. Furthermore, when the jump condition $x = 1000$ is satisfied in location far, the train HA switches from far to *near*, and sets the boolean vari-

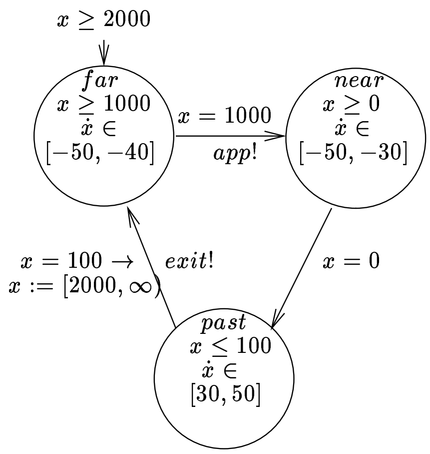

Figure 1: Hybrid automaton modeling the train in the train-and-gate example.

able *app* to true. This variable (and the variable *exit*) occur with exclamation marks in this HA to denote that they are set to true during one sampling interval (and reset afterwards) as a result of an action in this part of the system. In the HA model of the controller we will see the same two variables with question marks to denote that switches are *caused* by changes in their values.[1] When the HA is in *near* and $x = 0$ is satisfied, the HA switches from *near* to *past*. Finally, when the current location is *past* and $x = 100$, then x is reassigned to an arbitrary value in $[2000, \infty)$, and the variable *exit* is set to true. The HA model of

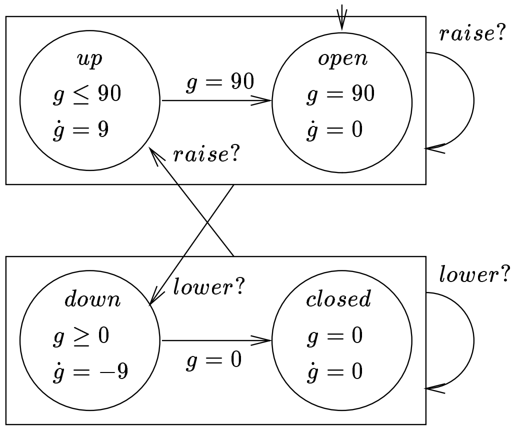

Figure 2: Hybrid automaton modeling the gate in the train-and-gate example.

the gate (Fig. 2) has four location: *up*, *open*, *down*,

[1]The exclamation and question marks are in fact HyTech syntax [8], and are used to denote that the automata are *synchronized* by such variables.

and *closed*. Initially, the gate is open and the angle of the gate to the ground is 90° (the angle is measured by g). Whenever the variable *lower* is set to true (by the controller) the automaton switches to *down* where the gate is lowered a rate of 9° per second. When the angle is 0° a switch to *closed* occurs. If the gate is in location *down* or *closed* and the variable *lower* is set to true (by the controller), the location will be the same. If *raise* is set to true, a switch to *up* will occur, and the gate will start to raise by 9° per second until it reaches 90° where a switch to *open* occurs. The HA

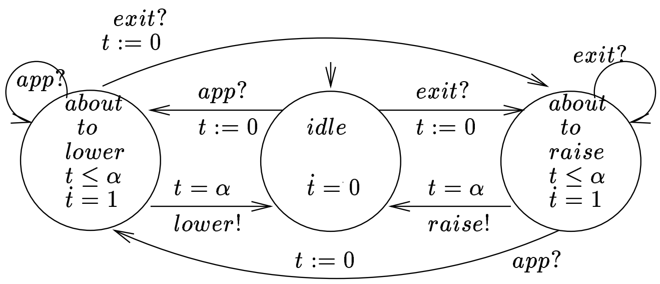

Figure 3: Hybrid automaton modeling the controller in the train-and-gate example.

model of the controller is given in Fig. 3. The idea is that the controller obtains information from the train automaton whether the train is approaching or exiting (i.e. from the variables *app* and *exit*), then waits for a period of time, α, and sets *lower* or *raise* to true, respectively, which is then handled by the gate.
The system with the three automata has been verified in HYTECH [8] to prove that for $\alpha < 49/5$, when the train is within 10 meters from the gate, the gate is always fully closed.

4 Representation

To be able to do EM monitoring at all, we need to be able to have a representation of the controller and the controlled system that enables us to figure out what the current controller state is, and what the expected state is, after an execution of an action. We suggest that the representation is a state transition diagram where every node either is a state in which the system is idle, or state in which a control action is executed. The management of transitions between idle states and non-idle states will be the task of the situation assessment, while transitions from non-idle to idle states are managed by the expectation assessment. It is natural to construct such a diagram by connecting *condition-action pairs* with each other, that is, every state is a

$$\begin{aligned}
&\langle idle, true\rangle \wedge exit \Rightarrow start_clock\\
&\langle idle, true\rangle \wedge app \Rightarrow start_clock\\
&\langle about_to_lower, t \leq \alpha\rangle \wedge exit \Rightarrow start_clock\\
&\langle about_to_lower, t \leq \alpha\rangle \wedge t = \alpha \Rightarrow\\
&\qquad lower \wedge stop_clock \qquad (1)\\
&\langle about_to_raise, t \leq \alpha\rangle \wedge app \Rightarrow start_clock\\
&\langle about_to_raise, t \leq \alpha\rangle \wedge t = \alpha \Rightarrow\\
&\qquad raise \wedge stop_clock \qquad (2)
\end{aligned}$$

Table 1: The control program.

condition and every action is an edge in the diagram. Moreover, one of the tasks of EM is to compare the expected effects of action with the actual effects. This mean that the states in the diagram have to be formulated in terms of inputs to the controller, that is, every state corresponds to some particular sensor reading.

4.1 The Controller as Condition-Action Pairs

As mentioned in the introduction, there exists a body of research in synthesizing controllers for hybrid systems. In this section we will adopt a simple strategy inspired by [9], where we view every distinct mode switch as caused by the invocation of a control action. In this way we will construct the required condition-action pairs.
We define an *HA-state* of a HA to be a tuple $\langle l, inv\rangle$, where l is a location (a symbol) and inv is the invariant of l.
A *control rule* is an expression $S \wedge j \Rightarrow a$, where S is an HA- state, j a jump condition, and a a control action. The conjunction $S \wedge j$ will often be called a *precondition* of a control rule. A *control program* is a set of control rules.
To synthesize a controller from a particular HA we need to make some assumptions: First, every mode switch has to be due to a control action, i.e. *discrete change is always controlled change.* In our example (Fig. 3) we need to view every resetting of the clock, t, to be a control action, *start_clock*, which is assumed to reset and start the clock. Moreover, we introduce the control action *stop_clock* that is assumed to stop the clock. The action *start_clock* replaces $t := 0$ as an action on edges, and *stop_clock* is introduces for every edge where the resulting location includes $\dot{t} = 0$. [2]
The control program for the controller in Fig. 3 is shown in Table 1. It is possible to generate a state transition diagram from the program in Table 1, but

[2]To be able to verify a system with this change, we would have to model the clock too with HA. However, this fairly simple modeling task is left out of this paper.

it is not useful for EM, since the conditions does not explicitly correspond to inputs to the controller, but on the locations.

4.2 Replacing the Locations by Sensor Readings

It is clear that the control program above is valid if it is started in the *idle* location, the train is started in far, and the gate in *open*, i.e. the control program will behave well under the exact circumstances the automata are verified for. However, in engineering practice it is unrealistic to assume anything about the initial state. Thus, it is necessary to base the EM on *external information* (such as sensor or actuator information), rather than on assumption on the current state of the controller. Also, we can see that the preconditions of control rules (1) and (2) cannot be distinguished *logically* based on their invariants. Thus, we require the control rules to be "physically" mutually exclusive, i.e. that it is physically impossible for two precondition to be satisfied simultaneously.[3]

Now, we will take advantage of the fact that we know whether the clock is running or not (the truth-value of the variable *clock_on*), and that we can sense the current location of the train (that the controller has access to the truth-values of the variables far, $near$ and *past*).

To characterize the controller locations we begin in the verified initial location *idle* and note that the clock is off, and that the corresponding initial location in the train HA is far. For *about_to_lower* we can see that the previous train location may have been far, and that a switch in the train HA, from far to $near$ must have occurred, that is, when the controller is in *about_to_lower* it is possible that the train is in $near$. It is also possible that the train switches to *past* before $t = \alpha$ is satisfied. Thus, for *about_to_lower* we know that the clock is on, and that the train is in either $near$ or *past*. If the condition $t = \alpha$ eventually is satisfied the controller switches to *idle* which implies that the train is in far (from the initialization), $near$ or *past* when the controller is in *idle*. Similarly, we note that *about_to_raise* can be characterized by $clock\_on \wedge far$. To summarize, we can characterize the location *idle* with the formula $\neg clock\_on \wedge (far \vee near \vee past)$ which we may translate to $\neg clock\_on$, *about_to_lower* with $clock\_on \wedge (near \vee past)$. But these characterizations only consider the activations of control actions, and to do EM we also need to represent the states in which the controller is waiting in a location. For example, when the train is approaching, and the clock is running, there is a time interval when no control action is invoked by the controller. By replacing the HA-states in the control program above by the characterizations we obtain the set of rules in Table 2. Note that the last three rules model the cases where the controller is waiting in a location *idle*, *about_to_lower*, and *about_to_raise*, respectively.

[3] One way of constructing physically mutually exclusive preconditions is to ensure that they are logically mutually exclusive. However, this may be very inconvenient for the control designer.

$(S1)$	$\neg clock\_on \wedge exit \Rightarrow start\_clock$
$(S2)$	$\neg clock\_on \wedge app \Rightarrow start\_clock$
$(S3)$	$clock\_on \wedge (near \vee past) \wedge exit \Rightarrow start\_clock$
$(S4)$	$clock\_on \wedge (near \vee past) \wedge t = \alpha \Rightarrow lower \wedge stop\_clock$
$(S5)$	$clock\_on \wedge far \wedge app \Rightarrow start\_clock$
$(S6)$	$clock\_on \wedge far \wedge t = \alpha \Rightarrow raise \wedge stop\_clock$
$(IS1)$	$\neg clock\_on \Rightarrow nop$
$(IS2)$	$clock\_on \wedge (near \vee past) \wedge t \leq \alpha \Rightarrow nop$
$(IS3)$	$clock\_on \wedge far \wedge t \leq \alpha \Rightarrow nop$

Table 2: The condition-action pairs.

4.3 Constructing the State Transition Diagram

By recalling the original controller HA it is possible to depict the control program above, as a *state transition diagram* as in Fig. 4, according to the following principles:

1. For every switch in the controller HA construct a rule as above.

2. For every location in the controller HA construct a *idle state*, which basically is the state corresponding to a location not including any jump conditions, but with the invariant of the location. Above we can see that the rule $IS2$ corresponds to the location *about_to_lower*, and that it differs only from $S3$ and $S4$ in the jump conditions and the invariant.

3. Connect idle states to all corresponding states with unlabeled edges. In Fig. 4 we can see unlabeled edges from $IS2$ to both $S3$ and $S4$.

4. Connect states to idle states according to the original HA. For example, since there is a switch from *idle* to *about_to_lower* with the jump condition *app*, we connect $S1$ with the idle state corresponding to *about_to_lower*, i.e. $IS2$.

In general, if the control program has n rules, and the original HA has m locations, the state transition diagram will have $n + m$ states (m is the number of idle states).

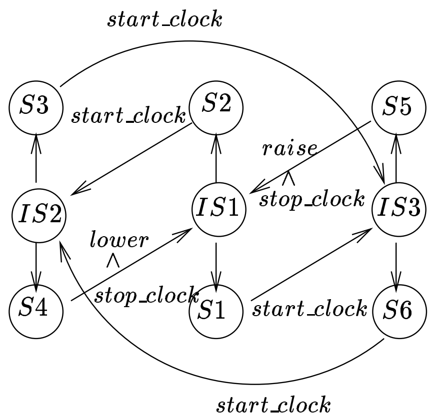

Figure 4: A state transition diagram representing the controller.

State	γ	δ
$S1$	$\neg clock\_on$	$exit$
$S2$	$\neg clock\_on$	app
$S3$	$clock\_on$	$(near \vee past) \wedge exit$
$S4$	$clock\_on \wedge t = \alpha$	$near \vee past$
$S5$	$clock\_on$	$far \wedge app$
$S6$	$clock\_on \wedge t = \alpha$	far
$IS1$	$\neg clock\_on$	$true$
$IS2$	$clock\_on \wedge t \leq \alpha$	$near \vee past$
$IS3$	$clock\_on \wedge t \leq \alpha$	far

Table 3: The division of the states.

5 Execution Monitoring

We begin by defining the execution semantics of our control program. If $V = \{v_1, \ldots, v_n\}$ is the set of variables of (or inputs to) the controller (in our example we have $V = \{clock\_on, exit, app, near, past, far, t = \alpha, t \leq \alpha\}$.[4]) a vector $\sigma = \langle b_1, \ldots, b_n \rangle$, with $b_i \in \{0, 1\}$, is called a *sample*. We determine the truth-value of a state in a given sample as usual in propositional logic. During the execution of a controller, it will receive a stream of samples.

Now, we define how the four first functions of EM are implemented in this work.

Situation Assessment: When a sample is received by the controller, there are only three distinct possibilities:

1. Either exactly one idle state, is, and no other state is satisfied.

2. One idle state, is, and a state, s, where there exists an unlabeled edge from is to s, is satisfied.

3 No state is satisfied.

In the first case, the current state is said to be is, and in the second it is s (since the controller will execute the action corresponding to the edge going from s). In the third case the current state can be set to be a dummy state s_{fault}.

Expectation Assessment: When the initial sample is received by the controller the expectation assessment mechanism is idle.

We define a (partial) function $Expect : S \times A \to 2^S$, where S is the set of states, and A is the set of control actions. The set of states in our example is the set of $S1 - S6$ and $IS1 - IS3$, the set of control actions is $A = \{start\_clock, raise \wedge stop\_clock, lower \wedge stop\_clock\}$. From Fig. 4 we can obtain the definition of $Expect$. If the controller executed an action a in the state s, we have two possibilities:

1 $a = nop$ which means that s is an idle state. Let $M = \{s'_1, \ldots, s'_m\}$ be the set of all states such that there is an unlabeled edge from s to s'_i, for $1 \leq i \leq m$. Then $Expect(s, a) = M \cup \{s\}$, that is, we expect that any of the states connected to s, and s can be satisfied next.

2. $a \neq nop$ which means that a "real" control action is executed, and that s is a non-idle state. If $M = \{s'_1, \ldots, s'_m\}$ is the set of all states such that there is an edge labeled with a from s to s'_i, then $Expect(s, a) = M$.

Discrepancy Detection: This is not particularly problematic: If the controller performs action a in state s, and the controller reads a sample σ which does not satisfy s, we say that we have a discrepancy iff σ does not satisfy any state $s' \in Expect(s, a)$.

Discrepancy Classification: For this function we have a number of possibilities. We need to consider the purpose of the EM. In our case we are interested in distinguishing between whether the discrepancy is caused by the controller or by the train, that is, we seek the explanation of the discrepancy in the model of the train, or in the model of the controller. To do this we can in our particular case utilize the structure of the states. We can see that every state can be written as a conjunction $\gamma \wedge \delta$ where γ only concerns variables $clock\_on$ and $t = \alpha$, and δ only concerns app, $exit$, far, $near$, and $past$. In Table 3 the γ and δ parts of our six states are presented. For a state s we write s_γ and s_δ to denote the respective parts.

Let s be a state, and a a control action such that $Expect(s, a)$ is defined. Assume the action a is executed in state s, and that the controller is receiv-

[4] Since t only is used in two particular comparisons to α we choose to view the comparisons as a boolean variable. We could easily change the semantics to handle real-valued variables, as well.

ing a sample σ such that it does not satisfy any $s' \in Expect(s,a)$. Thus, we have detected a discrepancy. We can now classify the discrepancies for every $s' \in Expect(s,a)$ into three distinct cases:

1. σ satisfies s'_δ, but not s'_γ.

2. σ satisfies s'_γ, but not s'_δ.

3. σ does not satisfy s'_γ, nor s'_δ.

For case **1** we know that the δ part of the system (in our case the train) performed according to our expectations, but that the γ part did not. *Since there is no reason to blame δ, we blame γ for the discrepancy.* For case **2** we now that the γ part of the system (in our case the controller) performed according to our expectations, but that the δ part did not. *Since there is no reason to blame γ, we blame δ.* For case **3** we choose to blame both parts.

5.1 Example

Consider the following scenario: Initially, there are two trains close to each other on the track, and they are more than 1000 meters from the gate. The clock in the controller is off. Thus, our situation assessment function will note that the idle state $IS1$ is the only satisfied state. According to *Expect* the set of expected next states is $\{S1, S2\}$. Now the first of the two trains reach the 1000 meter limit, and *app* is set to true. Therefore, $S2$ is satisfied, and the clock is started. We then expect $IS2$ to be satisfied next, but that does not happen because the second train reaches the 1000 meter limit, which sets *app* to true again. We are in a situation where *clock_on*, *near*, and *app* all are true, which does not correspond to the expected state, $IS2$, and therefore a discrepancy has occurred. Since the received sample assigns 1 to *app*, and *app* is not in $IS2$, we can (by looking at Table 3) blame the discrepancy on δ, that is, on the model of the train.

6 Conclusion and Future Work

In this preliminary report we have introduced a conceptual framework of Execution Monitoring, and shown how a representation suitable for EM can be automatically generated from a HA model of a controlled system.

We are currently formally analyzing the range of applicability of the techniques introduced in this paper, and we will, in the future, investigate the possibilities of introducing recovery techniques.

References

[1] R. Alur, C. Courcoubetis, T. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and and verification of hybrid systems. In *Workshop on Theory of Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer Verlag, Lyngby, Denmark, October 1992.

[2] R. Alur, T. Henzinger, and P. Ho. Automatic symbolic verification of embedded systems. In *Proceedings of IEEE Real-Time Systems Symposium*, Raleigh-Durham, NC, December 1993.

[3] J. Ambros-Ingerson and S. Steel. Intergrating planning, execution and monitoring. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI'88)*, Saint Paul, Minnesota, August 1988. American Association for Artificial Intelligence, AAAI Press/The MIT Press.

[4] P. Amthor. HySC – A development environment for hybrid systems' components. In R. Berghammer and Y. Lakhnech, editors, *Tool Support for System Specification, Development and Verification*, volume 9803, pages 46 – 50. Institute of Computer Science, Kiel University, 1998.

[5] P. Antsaklis and K. Passino, editors. *An Introduction to intelligent and autonomous control.* Kluwer Academic Publishers, 1993.

[6] C. Baral and S. Tran. Relating theories of actions and reactive control. *Linköping Electronic Articles in Computer and Informations Science*, 3(009), 1998. URL: `http://www.ep.liu.se/ea/cis/1998/009/`.

[7] M. Bjäreland and D. Driankov. Synthesizing discrete controllers from hybrid automata - preliminary report. In *Working Papers of the AAAI Spring Symposium on Hybrid Systems and AI*, Stanford, CA, USA, March 1999.

[8] T. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. In *Proceedings of the Ninth International Conference on Computer-Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 460–463. Springer-Verlag, 1997.

[9] T. Henzinger and P. Kopke. Discrete-time control for rectangular hybrid automata. In *Proceedings of the Twentyfourth International Colloquium on Automata, Languages, and Programming (ICALP'97)*, volume 1256 of *Lecture Notes in Computer Science*, pages 582–593. Springer-Verlag, 1997.

[10] B. Lennartsson, M. Tittus, B. Egardt, and S. Pettersson. Hybrid systems in process control. *IEEE Control Magazine*, October 1996.