Research Project Report −17-755*

Akshay Rajhans

Spring 2009

*This report documents a research project done for the course 17-755, "Architectures for Software Systems" offered during the Spring semester of 2009 at Carnegie Mellon University.

Contents

1 Introduction **4**

2 Architectures for cyber-physical systems **4**

3 Cyber-physical family **5**

3.1 Cyber family 5

3.2 Physical family 6

3.3 Cyber-physical interface family 7

4 The need for LHA annotations **7**

5 LHA mix-in family **8**

6 LHA plug-in **9**

7 Example **10**

8 Conclusion **18**

9 Appendix **18**

Abstract

This project report presents the Linear Hybrid Automata (LHA) plug-in for AcmeStudio developed as a part of the research project for the Software Architecture course offered during the Spring 2009 semester at Carnegie Mellon University.

As a part of our research effort, we have developed an extension of existing software architecture tools to model physical systems, their interconnections, and the interactions between physical and cyber components. To support the principled design and evaluation of alternative architectures for cyber-physical systems (CPSs), a new CPS architectural style is introduced. Moreover, to perform behavioral analysis on the CPS architectures built using this CPS style, we have developed the capability to annotate the architectures with finite state processes (FSP) or linear hybrid automata (LHA) codes. To facilitate easy editing of FSP and LHA behavioral information codes and to automatically generate analysis codes for Labeled Transition System Analyzer (LTSA) or Polyhedral Hybrid Automata Verifier (PHAVer), respectively, we have developed FSP and LHA plug-ins for AcmeStudio. This report presents the LHA plug-in.

1 Introduction

Today's models and methods for analysis and design of cyber-physical systems (CPSs) are typically fragmented along lines defined by disparate mathematical formalisms and dissimilar methodologies in engineering and computer science. While separation of concerns is needed for tractability, such analytical approaches often impose an early separation between the cyber and physical features of the system design, making it difficult to assess the impacts and tradeoffs of alternatives that cut across the boundaries between these domains.

As a part of the research effort towards this course, we developed extensions to software architectural descriptions to encompass the full range of elements that comprise cyber-physical systems. Architecture description languages (ADLs) support the description of annotated structural representations that facilitate the evaluation of design tradeoffs in terms of important qualities such as performance, reliability, security, and maintainability. It is possible to customize an ADL to a particular domain by defining an architectural style, which describes a vocabulary of structural types, a set of properties and composition rules, and associated analyses to enable such tradeoffs.

Our goal is to create an extensible framework within which a comprehensive set of design tools can be created for CPSs. As first steps in this direction, we have developed a new CPS architectural style, the behavioral annotations for the components and connectors from the style and plug-ins for verification of properties of CPSs based on verification methods for systems with discrete and continuous state variables (hybrid systems).

2 Architectures for cyber-physical systems

Although architectural modeling has been used in specific domains (e.g., avionics) to incorporate physical elements as components, there is currently no adequate way of treating cyber and physical elements equally in a general fashion. This is because currently there is an impoverished vocabulary for the types of physical components found in CPSs, the interconnections that determine the interactions between those components, as well as the interactions between cyber components and physical entities. This section presents extensions that allow both physical and cyber elements to be modeled together as cyber-physical architectures.

We introduce the architectural primitives as the basic building blocks that define the modeling vocabulary for cyber-physical systems. We use the Acme ADL [GMW00] to define these architectural primitives because it has strong support for defining flexible architectural styles, or families. In Acme, an architectural style is represented as a family of element types that follow certain rules and structure in terms of what kind of components and connectors can be

present in the system, and the manner in which they can be connected with each other. This primitive family can be refined into an application-specific family by adding additional elements and rules based on these primitives [DRRS99].

3 Cyber-physical family

Our approach to providing an architectural family for cyber-physical systems is to define an architectural family for the cyber domain and the physical domain, respectively, and then define a bridging family that combines these two families into a cyber-physical family. Thus, in Acme, the cyber-physical interface family inherits its elements and rules from two core families, namely, cyber family and physical family. The cyber family has computational elements such as computers, microcontrollers, and the computational side of intelligent devices such as smart sensors and actuators. The physical family has elements that model the physical environment with which the cyber elements interact. In addition to inheriting the elements of cyber family and physical family, the cyber-physical family also has elements that bridge the boundaries of cyber and physical domain, such as transducers. Taking this approach of using multiple inter-related families allows for the architecture of the cyber aspect and the physical aspect of a system to be modeled separately, if desired, but combined in a principle way using the bridging family.

Each family specifies the allowed set of components, connectors, ports on components that facilitate interfacing of connectors, and the roles played by the components on different ends of connectors. We define component, connector, role and port types, collectively called element types. The families serve as patterns that any system architecture satisfies automatically by including the families in the architecture. In the process, the element types are made available to the modeler as a part of the families. New elements of the system are instantiation of one of these element types. These instantiations of the element types into elements can be thought of as the instantiations of classes into their objects in object-oriented terms. The following sections describe the cyber family, the physical family and the cyber-physical family in detail.

3.1 Cyber family

The challenge in defining an architectural style is to find a balance between specificity and generality. In the CPS domain, the vast majority of software will follow a control software model, and so the cyber family defines a style motivated by these application domains.

Cyber component types

Controller and estimator component types Controller and estimators follow the notion of control and estimation that stems from control theory.

Controllers read the system state and take action by issuing commands. Estimators inspect available information, such as the sensor readings, and based on their possibly partial knowledge of the system update the system states.

State-variable component type We introduce a separate component type to represent the entity in the software that stores and maintains the state variables. This primarily stems from the maintainability requirements of large-scale systems. In simple systems, these could be just passive memory blocks, and in complex systems there could be further details specifying which entity can update a particular state variable, who can read it and so on.

Sensor and actuator software component types Since most of the cyber-physical systems will have sensing and actuation as a means of inter-communication between cyber and physical domains, we represent their software equivalents in the cyber family. Sensor software is the computational component built on top of sensors that processes sensor readings and makes them available to other computational elements like an estimator. Actuators software converts commands from controllers into an actuating signals in the physical realm.

Cyber connector types In addition to the computational aspects of the software, it is important to represent the communication aspects. This will be important for representing and reasoning about timing between software elements and how this affects the physical aspects of the system. For now, connectors can specify the interaction modes between components, and can be used to check for well formed architectures. We represent two major types of software connectors, a call-return connector representing one-to-one communication and a publish-subscribe connector representing one-to-many. Each of these types can be further specialized to represent particular communication protocols.

3.2 Physical family

Unlike the cyber family, where we build the element types based on the existing knowledge about the possible element types in software architectures, building of the physical family is a fresh start. The challenge here is to determine what the fundamental building blocks of physical systems would be. As a start, we have defined one generic component type to model any physical system and two generic connector types to model directed and undirected interaction. These are further explained below.

Physical component types

Hybrid dynamic system type We define one generic type which is the hybrid dynamic system type. The rationale behind defining just this one component type is that any physical system with continuous dynamics

and possible discontinuities can be generically termed as a hybrid dynamic system. Currently, it is left up to the modeler to further refine it into application-specific component types if deemed necessary

Physical connector types The connectors in the physical domain model the couplings between the dynamics of individual subsystems. There could be directed or undirected couplings between the dynamics of two physical systems. The two connector types to capture these connections are defined as follows.

Input-output connector type This is the connector type models directional coupling between physical components. This is the type of connection normally used in control-oriented models of physical systems, such as the block diagram paradigm used in the Simulink tool from The MathWorks.

Shared-variable connector type This connector type models the undirected (or bidirectional) interaction of coupled dynamics. This is the type of component interactions used in physical modeling tools such as Modelica.

3.3 Cyber-physical interface family

The cyber-physical interface family uses all the types from the cyber family and the physical family. It adds component and connector types that bridge the gap between software and physical systems. To model the interactions between the cyber and the physical worlds, we introduce two directed connector types, P2C (physical-to-cyber) and C2P (cyber-to-physical) connector types are defined. Simple sensors can be modeled as the P2C connectors and simple actuators can be modeled as the C2P connectors.

For more complex interfaces between cyber and physical elements in a CPS, we define the transducer component type, which has ports to cyber elements one side and ports to physical elements on the other side. Devices are modeled as transducer components if they do more than a simple translation between cyber and physical domains, e.g., intelligent sensor nodes. If the devices do just a simple translation between the cyber and the physical domains, they are rather modeled as connectors.

4 The need for LHA annotations

The architectural elements introduced above describe only the structural information about a system, such as the kinds of elements present in the system and the nature of their interconnections. This structural description lacks information about the behaviors of the individual elements and the behavior of the system as a whole. The structural information is sufficient for basic structural analyses to catch modeling mistakes, such as connecting a physical component directly with a purely cyber connector. However, to be able to do meaningful formal analysis on the system behavior, the architecture needs to be annotated

with behavioral information pertaining to the system as a whole as well as to the individual elements.

The physical elements in cyber-physical systems can be modeled most generally as hybrid systems, that is, dynamic systems with both continuous and discrete state variables. Hybrid automata [ACH+93] are an intuitive and expressive framework for modeling hybrid systems. Of the various classes of hybrid automata, LHA [Hen96] are a class of hybrid automata for computational tools are available for analysis, such as algorithms such as the computation of reachable states over infinite horizon [AHH96]. LHA have continuous variables with linear predicates and piecewise constant bounds on the derivatives. The continuous dynamics in LHA makes the behavioral modeling of cyber-physical components more faithful than the purely discrete FSP while still being able to do formal behavioral analysis. LHA also provide a level of approximation to detailed dynamic modeling that is commensurate with the type of analyses that should be performed at the architectural level.

For the analyses of LHA, we make use of a tool called PHAVer [Fre05], which supports compositional verification, as well as the exact reachability computation of LHA. The definition (the syntax) of LHA supported by PHAVer makes a distinction between the continuous state and input variables of an automaton based on whether or not they are inside the scope of the influence of the automaton. The power to distinguish between the two is important because there are directional couplings in the architectures of physical parts of the cyber-physical systems in our CPS style. Moreover, PHAVer allows for the composition of automata based on the input-output relations and synchronization labels. This can capture composition of the behaviors of components into the system behavior, based on how the components are interconnected. This justifies the use of PHAVer as an analysis tool when the behavioral annotations are modeled as LHA. The fact that PHAVer is free from numerical errors and uses an exact arithmetic with infinite precision is an added plus.

For this reason, the LHA model that we use follows the PHAVer syntax.

5 LHA mix-in family

In Acme ADL, the behavioral information can be added onto the architecture by defining properties pertaining to the system and the elements. Properties are user definable records that hold values of the specified type, e.g. name-value pairs represented by records with a string name field of the type string and the value field of the type integer. We have introduced two kinds of behavioral annotations finite state processes (FSP) and linear hybrid automata (LHA). This report focuses on LHA annotations.

To facilitate the LHA annotation, we define a new set of properties that will hold the LHA information pieces. To be able to seamlessly incorporate these LHA-specific properties, we have developed a new *LHA mix-in family*. A new set of properties that hold the LHA models of the components, the composition information (i.e. the input-output relations and synchronization labels) and a set of global constants are defined. The specification of the expected behavior of the system (also modeled as an automaton) is treated as a system-wide property, while the individual automata are component-specific properties.

6 LHA plug-in

We have developed an LHA plug-in to facilitate easy editing of the LHA properties.

To facilitate the LHA annotation, we have introduced a new set of properties that hold the LHA models of the components, the composition information (i.e. the input-output relations and synchronization labels) and a set of global constants. The specification of the expected behavior of the system (also modeled as an automaton) is treated as a system-wide property[1], while the individual automata are component-specific properties.

PHAVer automaton definition consists of the automaton name, a list of synchronizing labels, a list of input variables, a set of output variables, and the automaton body. We define *PHAVerAutomaton property type* that consists of a record that has exactly these fields − a name that is a string; a list of synchronizing labels, a list of input variables and a list of state variables, where every individual label, input variable and state variable is a string; and the automaton body which is a just a big long string. This property applies to every component and lets the user specify the automaton describing the behavior of the component, as well as to the system, which lets the user specify the specification automaton for the system.

These individual automata defined as above, can later be composed in parallel. This parallel composition is treated as a system-wide piece of information.

Given the system which consists of parallel composition of the individual PHAVer automata, the specification on the system is also specified as an automaton, a system-wide property as stated earlier. To check whether the parallely composed system satisfies the required behavior or not, PHAVer checks for the simulation relation between the system and the specification automata.

[1]Note that PHAVer also allows for the specifications to be described on individual components which get composed as a system specification. However, we have not yet considered this case.

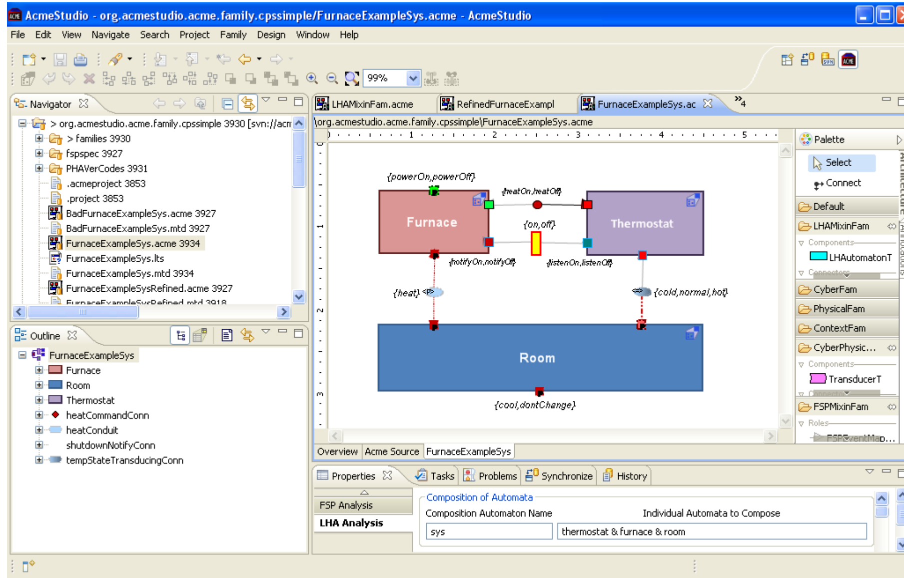

Figure 1: Architectural modeling of a temperature-control system in AcmeStudio

In addition to these automata, PHAVer also allows the user to define a list of constants and their values. In LHA mix-in family, these are treated as name-value pairs, and they belong to the system. The corresponding property type is called *PHAVerConstantList property type*.

Appendix shows a screenshot of the various PHAVer related LHA properties from the LHA mix-in family.

7 Example

We now illustrate the use of this plug-in on an example.

Consider a simple temperature control system comprising a room, a thermostat and a furnace. The thermostat is located inside room and can read the temperature in the room. The furnace can either be turned on or off manually, and while on, the thermostat can dictate whether or not the furnace should heat the room or not. (The way this is controlled in real systems is by turning a blower forcing hot-air ventilation on or off.)

The goal is to maintain the measured temperature of this room close to a specified set point set inside the thermostat. Figure 1 shows the architecture of this system in AcmeStudio.

The architectural model of the system has three high-level components, namely ‘thermostat’, ‘furnace’ and ‘room’. All these components are representations of lower level details in the hierarchy of the architecture. These lower level details are not visible in the figure.

The thermostat consists of a temperature sensor that senses the temperature of the room and broadcasts it on a communication channel. A temperature estimator then reads the temperature reading from the communication channel and updates the temperature state variable. The furnace estimator reads the temperature state keeps updating the furnace state. Temperature controller is the part of the software that keeps track of when the furnace estimator changes the furnace state and communicates these changes to the furnace as commands to turn the heating on or off.

The furnace comprises a furnace actuator software that receives commands sent by the temperature controller in the thermostat and accordingly turns the corresponding heating element called heat coil on or off. Finally, a transducer element called SoftSwitch captures the manual on/off signals and communicates those to the furnace actuator software element to turn the heating element on or off.

The room has been modeled as generic hybrid dynamic systems.

A sample model of the behavior of the system in PHAVer LHA syntax would be as follows.

```
//------------------------------------------------------
//     Constants
//------------------------------------------------------

t_sample := 2; // sampling time of the controller

deltaH := 5; // delta above t_set for the thermostat to ignore
deltaL := 5; // delta below t_set fot the thermostat to ignore

rc_l := 0.2; // lower bound on the rate of cooling
rc_h := 0.4; // upper bound on the rate of cooling

rh_l := 0.5; // lower bound on the rate of heating
rh_h := 0.7; // upper bound on the rate of heating

t_set := 30; // setpoint for the thermostat

t_0l := 26; // lower bound for the confidence interval of t_0
t_0h := 28; // upper bound for the confidence interval of t_0
```

```
t_hottest := 50; // hottest the room can get, has to be > t_amb
t_amb := 0; // ambient temperature, has to be lower than t_hottest

t_m := 20; // lower bound on the temperature spec
t_M := 40; // upper bound on the temperature spec

//-----------------------------------------------------
automaton furnace
//-----------------------------------------------------
state_var : l;
synclabs: powerOn, powerOff, startHeat, stopHeat, heatOn, heatOff;

loc poweredOff: while True wait {True};
when True sync powerOn  do {True} goto idle;
when True sync startHeat  do {True} goto poweredOff;
when True sync stopHeat  do {True} goto poweredOff;
when True sync powerOff  do {True} goto poweredOff;
loc idle: while True wait {True};
   when True sync startHeat  do {True} goto startingHeat;
   when True sync stopHeat  do {True} goto stoppingHeat;
   when True sync powerOff do {True} goto poweredOff;
when True sync powerOn  do {True} goto idle;
loc startingHeat: while True wait {True};
   when True sync heatOn  do {True} goto idle;
   when True sync powerOff do {True} goto poweredOff;
when True sync powerOn  do {True} goto startingHeat;
loc stoppingHeat: while True wait {True};
   when True sync heatOff  do {True} goto idle;
   when True sync powerOff  do {True} goto poweredOff;
when True sync powerOn  do {True} goto stoppingHeat;

initially: idle & l==0;
end

//-----------------------------------------------------
automaton thermostat
//-----------------------------------------------------

state_var: c; //c = clock variable
input_var: t;
synclabs: tick, startHeat, stopHeat, doNothing;

loc idle: while c <= t_sample wait {c' == 1};
    when c==t_sample sync tick do {c'==0} goto checking;
loc checking: while c <= 1 wait {c' == 1};
```

```
    when (t_set - deltaL) <= t sync startHeat do {c'==0} goto idle;
    when t <= (t_set + deltaH) sync stopHeat do {c'==0} goto idle;
    when (t_set - deltaL) <= t & t <= (t_set + deltaH)
sync doNothing do {c'==0} goto idle;

initially: idle & c==0;
end

//----------------------------------------------------
automaton room
//----------------------------------------------------

state_var: t;
synclabs: heatOn, heatOff, powerOff, error;

loc heating: while t_amb <= t & t <= t_hottest
wait {rh_l <= t' & t' <= rh_h};
    when True sync heatOff do {t'== t} goto cooling;
    when True sync powerOff do {t'== t} goto cooling;
    when True sync heatOn do {t'== t} goto heating;
when t==t_hottest sync error do {t'== t_hottest} goto atHottest;
loc cooling: while t_amb <= t & t <= t_hottest
wait {-rc_h <= t' & t' <= -rc_l};
    when True sync heatOn do {t'== t} goto heating;
    when True sync powerOff do {t'== t} goto cooling;
    when True sync heatOff do {t'== t} goto cooling;
    when t == t_amb sync error do {t'== t_amb}  goto atAmbient;
loc atAmbient: while True wait{True};
    when True sync heatOn do {t'== t} goto heating;
    when True sync heatOff do {t'== t} goto atAmbient;
    when True sync powerOff do {t'== t} goto atAmbient;
loc atHottest: while True wait{True};
    when True sync heatOff do {t'== t} goto cooling;
    when True sync powerOff do {t'== t} goto cooling;
    when True sync heatOn do {t'== t} goto atHottest;
initially: cooling & t_0l <= t & t <= t_0h;
end


//----------------------------------------------------
// Composition
//----------------------------------------------------

sys = furnace & thermostat & room;
```

```
//---------------------------------------------------
automaton spec
//---------------------------------------------------

state_var: t;
synclabs: heatOn, heatOff, error, tick,
  startHeat, stopHeat, doNothing, powerOn, powerOff;

loc always:

while t_m <= t & t <= t_M wait {True};
    when True   sync heatOn         do {True} goto always;
    when True   sync heatOff        do {True} goto always;
    when True   sync startHeat      do {True} goto always;
    when True   sync stopHeat       do {True} goto always;
    when True   sync doNothing      do {True} goto always;
    when True   sync powerOn        do {True} goto always;
    when True   sync powerOff       do {True} goto always;
    when True   sync tick           do {True} goto always;

initially: always & t_m <= t & t <= t_M;
end

//---------------------------------------------------
// Simulation relation checking
//---------------------------------------------------

SIM_PRIME_WITH_REACH  = false;
is_sim(sys,spec);
R = get_sim(sys,spec);
R.print;
```

If we look carefully at the above PHAVer code, we can see that there are six different logical elements in it. They are − (i) constant definitions, (ii) furnace automaton, (iii) thermostat automaton, (iv) room automaton, (v) composition information and (vi) specification automaton. Out of these, (i) constant definitions, (v) composition information and (vi) specification automaton are system-level information pieces. They are captured by properties pertaining to the system. When the system (whitespace in AcmeStudio) is selected, the LHA plug-in displays the system-wide properties editor as shown in Figure 2.

The rest of the parts− (ii) furnace automaton, (iii) thermostat automaton and (iv) room automaton are specific to the components. This information is stored in the properties that are specific to the components 'furnace', 'thermostat' and 'room' respectively. When any of these compoents is selected, the LHA plug-in

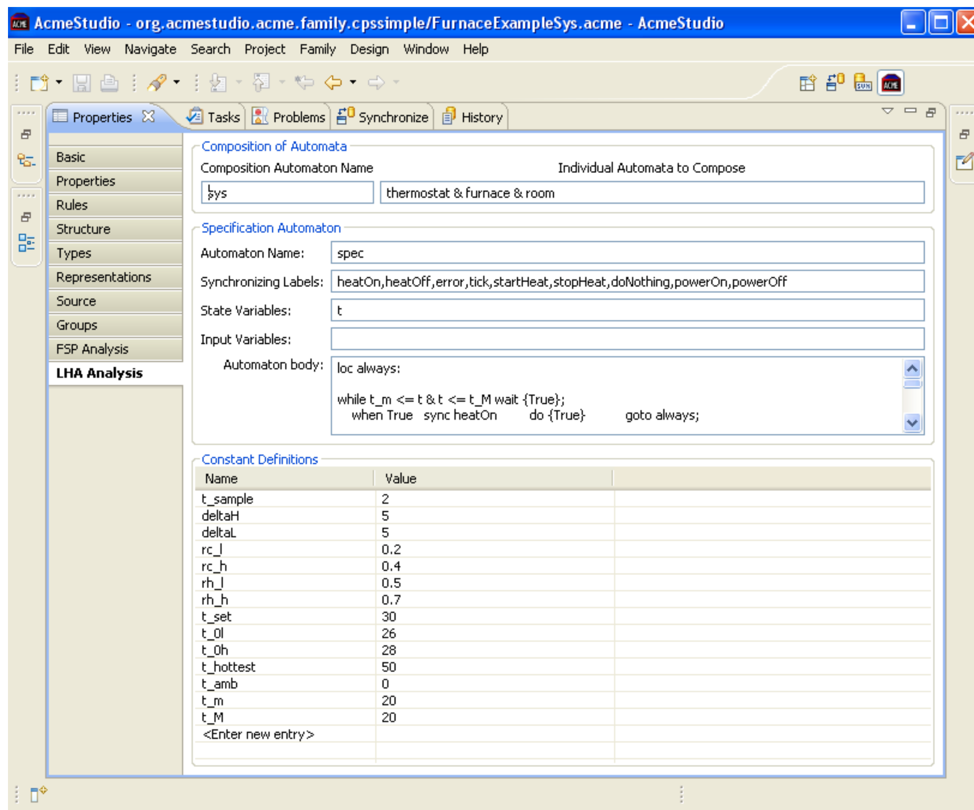

Figure 2: LHA plug-in: System-level properties editor

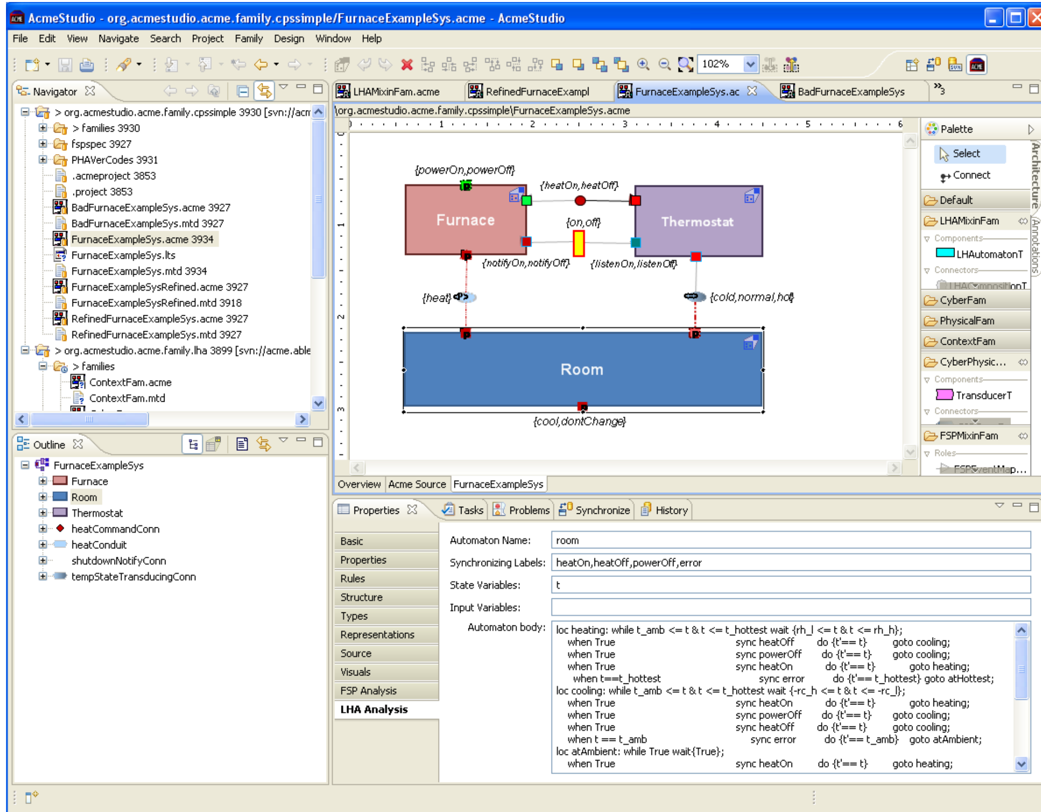

Figure 3: The LHA plug-in screen when the component 'room' is selected

displays the component-specific editor screen, one sample of which is shown in Figure 3. Figure 4 shows the same screen of the LHA plug-in in detail.

This screen shows the automaton room stored in pieces as per the properties defined in the *LHA mix-in family*. There are similar screens when the components 'furnace' and 'thermostat' are selected. Those screens can be found in the appendix, in figures 6 and 7.

There is no PHAVer data associated with the roles. Therefore, when any role is selected, the plug-in displays a blank screen showing that there is no PHAVer data to display.

Lastly, when any connector is selected, the plug-in is programmed to show a mapper screen. This has been reserved for future work. We wish to incorporate the renaming capability for the synchronizing labels, and the input and state variables of the individual automata when they are composed. This will be done

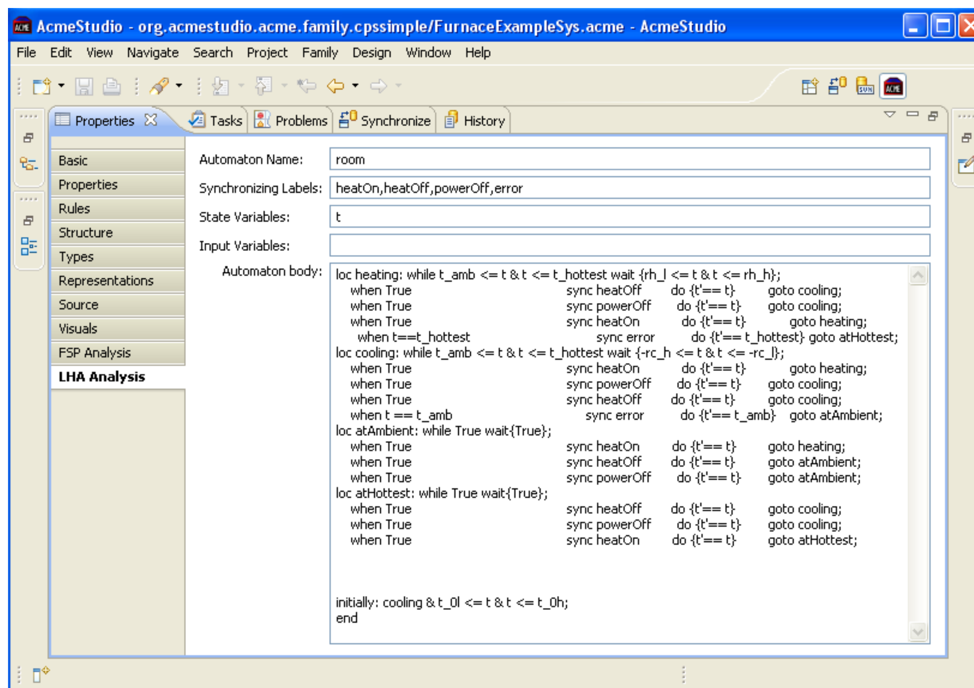

Figure 4: LHA plug-in: Room automaton in detail

when the connectors will connect the two components corresponding to the two automata. Currently this is not supported in PHAVer. Therefore, this screen is currently just two blank columns.

These additional screenshots of the plug-in can be found in the appendix.

8 Conclusion

In this report we presented a plug-in that facilitates easy editing of Linear Hybrid Automata (LHA) annotations of the components and connectors of cyber-physical system architectures. As a part of future work, we are working on dynamic renaming of input-output variables and synchronizing labels of different automata under composition. The possible options to implement this are either the addition of this renaming feature in PHAVer or some text pre-processing using some scripting language such as Perl. We have not implemented this feature yet.

9 Appendix

This section has a few screenshots that clarify the operation of the LHA plug-in.

First is the screenshot of the properties defined as a part of LHA mix-in family. It was tricky to fit this figure on one page because of its dimensions. It may not print on letter paper very well. However, in the PDF version of this document, it will serve the purpose of capturing all the records of the properties defined under the LHA mix-in family. The reader will be able to see the contents of the figure by zooming in and rotating the page while viewing the PDF copy of this report. Figure 5 shows this screenshot.

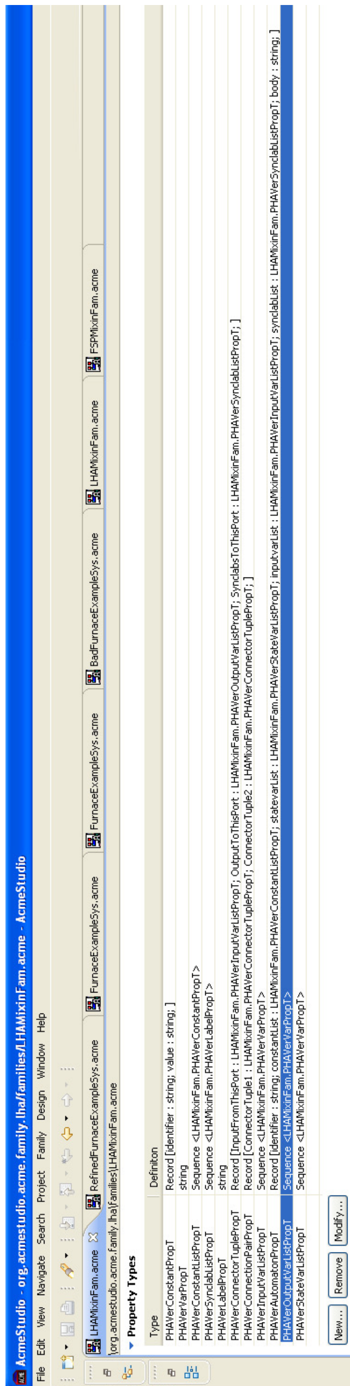

Figure 5: Properties in LHA mix-in family

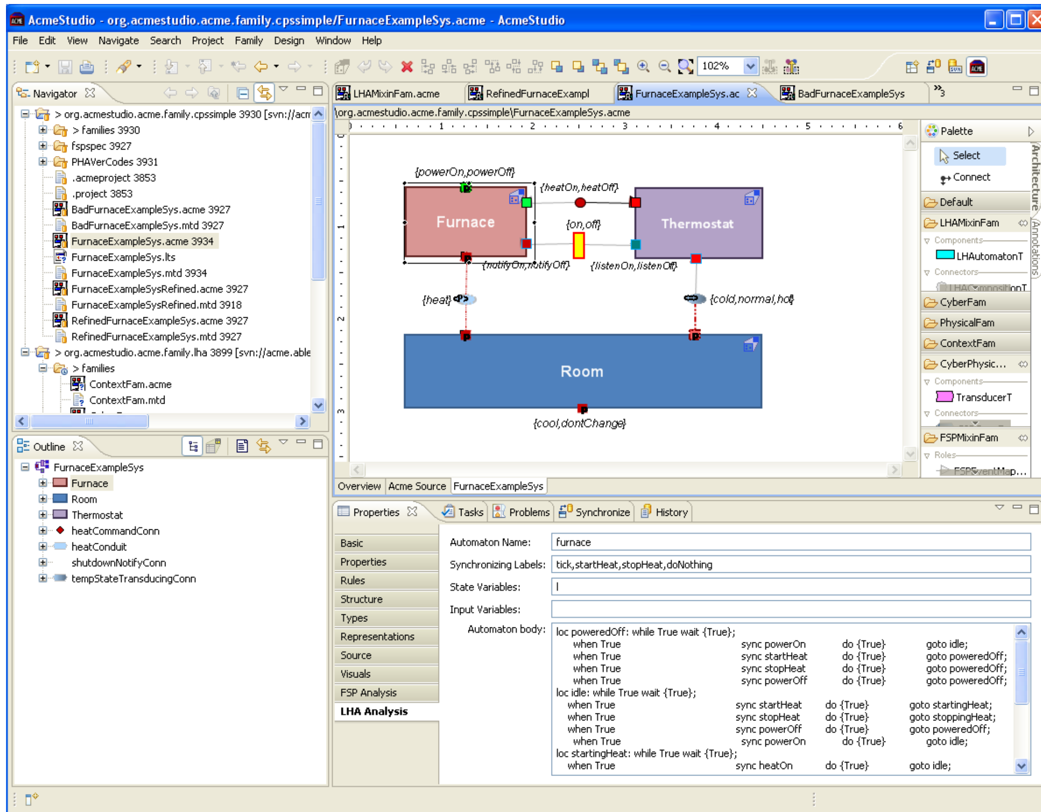

Figure 6: The LHA plug-in screen when the component 'furnace' is selected

The next two screenshots shown in Figures 6 and 7 are of the LHA plug-in when the compoenents 'furnace' and 'thermostat' are selected. These are similar to the one shown in Figure 3 and have been included here just for reference.

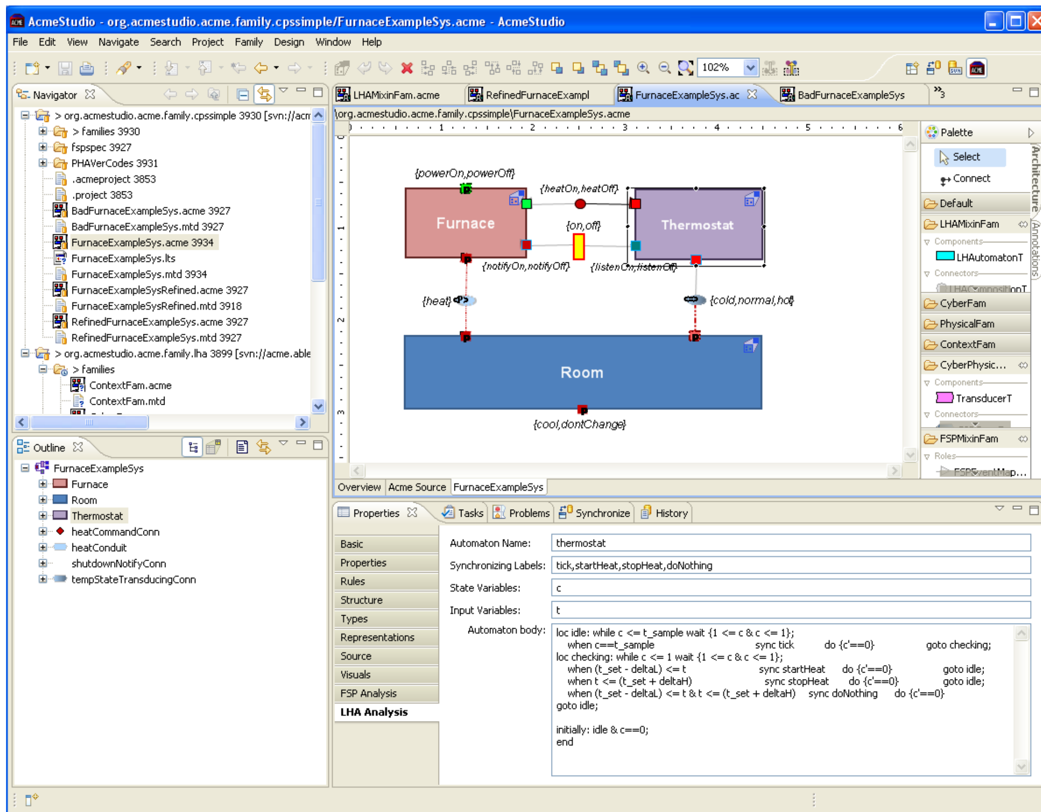

Figure 7: The LHA plug-in screen when the component ‘thermostat’ is selected

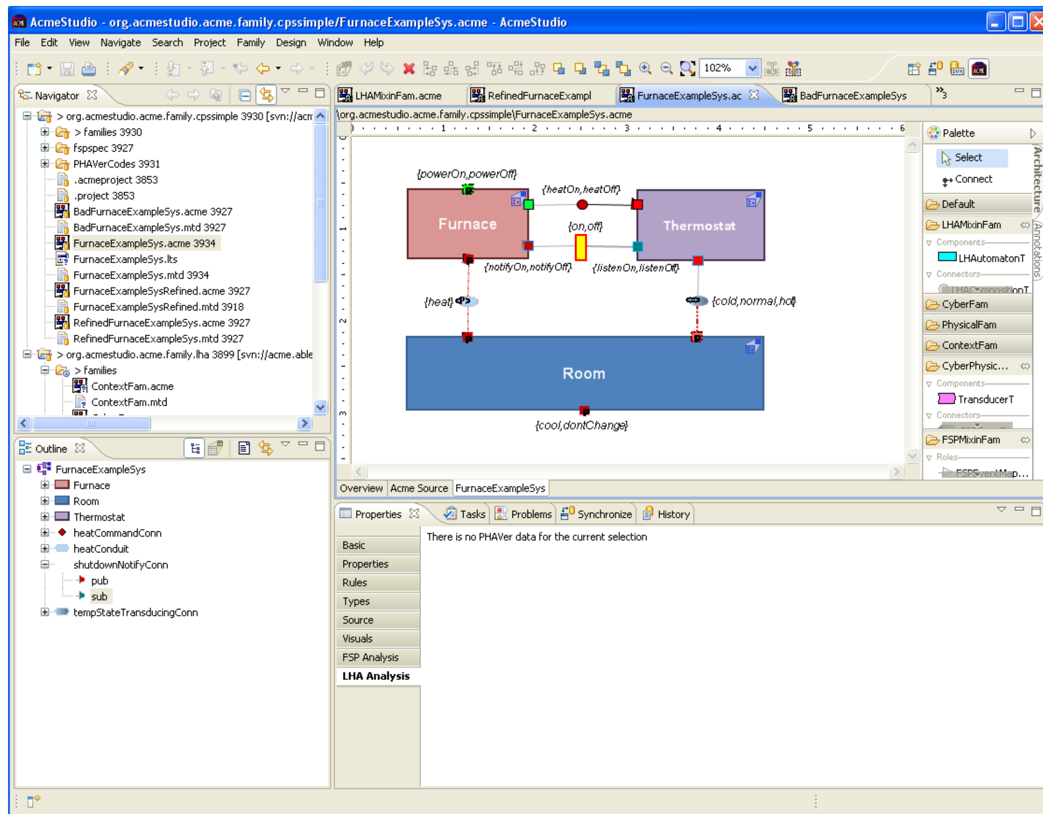

Figure 8: The LHA plug-in screen when a role that has no PHAVer data is selected

Figure 8 shows the LHA plug-in when an element (specifically, any role) is selected that has no corresponding PHAVer data. Note that this is different from having a blank data. In case of blank data, a blank skeleton will be shown.

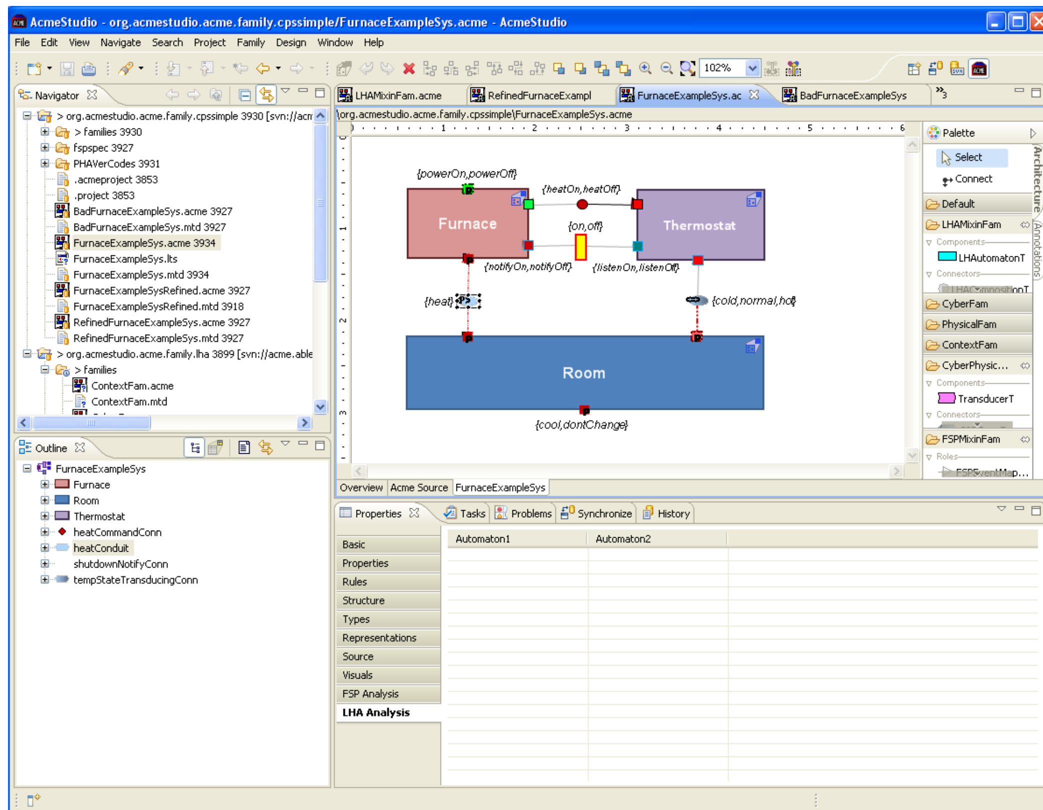

Figure 9: The LHA plug-in screen when the connector between ‘furnace’ and ‘room’ is selected

Lastly, in Figure 9, we have a screen capture that shows the mapping page that has been left for future work. PHAVer does not support renaming of synchronizing lablels and input and state variables yet. If and when it supports this kind of renaming, we will be developing this screen further to show the mapping of input-output variables and renamed (new name and old name of) synchronizing lables.

References

[ACH+93] 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 Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors, Hybrid Systems, volume 736 of LNCS, pages 209-229. Springer, 1993.

[AHH96] R. Alur, T.A. Henzinger, P.-H. Ho. Automatic symbolic verification of embedded systems. IEEE Trans. on Software Engineering 22(3):181-201, 1996.

[DRRS99] Dvorak D, Rasmussen R., Reeves G., Sacks A., Software Architecture Themes in JPL's Mission Data System, AIAA Space Technology Conference and Expo, Albuquerque, NM, 1999.

[Fre05] G. Frehse. PHAVer: Algorithmic Verification of Hybrid Systems past HyTech. Proceedings of the Fifth International Workshop on Hybrid Systems: Computation and Control (HSCC), Lecture Notes in Computer Science 3414, Springer-Verlag, 2005, pp. 258-273.

[GMW00] Acme: Architectural Description of Component-Based Systems. David Garlan, Robert T. Monroe, and David Wile. In Gary T. Leavens and Murali Sitaraman editors, Foundations of Component-Based Systems, Pages 47-68, Cambridge University Press, 2000.

[Hen96] Thomas A. Henzinger. The theory of hybrid automata. In Proc. 11th Annual IEEE Symposium on Logic in Computer Science, LICS'96, New Brunswick, New Jersey, 27-30 July 1996, pages 278-292. IEEE Computer Society Press, 1996. An extended version appeared in Verification of Digital and Hybrid Systems (M.K. Inan, R.P. Kurshan, eds.), NATO ASI Series F: Computer and Systems Sciences, Vol. 170, Springer-Verlag, 2000, pp. 265-292.