Architectural Modeling and Analysis of Cyber-Physical Systems

Ajinkya Bhave[†], David Garlan[‡], Bruce H. Krogh[†], Bradley Schmerl[‡], and Akshay Rajhans[†]

[†]Department of Electrical and Computer Engineering
[‡]School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{ajinkya|garlan|krogh|schmerl|rajhans}@cmu.edu

Abstract. Cyber-physical systems are systems in which software and physical elements of a system are equally important. This paper presents an architectural approach to modeling cyber-physical systems that allows physical modeling to be done in conjunction with software modeling, can be specialized to different physical domains, and can be used with existing tools and techniques for modeling software and physical systems. Our approach allows the physical and software modeling to be integrated into an architectural model that facilitates the analysis and comparison of alternative designs of both the software and physical components.

1 Introduction

Cyber-physical systems are systems in which software (cyber) and physical aspects are equally important. For example, the design of building heating systems must take into account both the physical phenomena of heat exchange through windows, walls, and rooms and the behavior of the software that implements the control algorithms. Similarly, the success of vehicle control systems depend on the interactions between the embedded software and the physical vehicle dynamics.

Cyber-physical systems are designed and analyzed using a variety of modeling formalisms and tools. Each representation highlights certain features and occludes others to make analysis tractable and to focus on particular performance attributes. Typically a particular formalism represents either the cyber or the physical elements well, but not both. For example, differential equation models typically represent physical processes well, but do not represent naturally the details of computation or data communication. On the other hand, discrete modeling formalisms such as Petri nets and finite state automata are well suited for representing sequential behavior and control flow, but are not particularly useful for modeling continuous phenomena in the physical world.

Treating the cyber and physical models separately makes it difficult for engineers to evaluate trade-offs between design choices involving alternative configurations of both the software and physical components. What is needed is

a framework that provides a unification of the cyber and physical domains to enable reasoning using multiple modeling formalisms. Such a unified framework should serve as a point of reference for modeling aspects of the systems using notations and tools already familiar to engineers. To enable this, it must be possible to abstract the unified model into views that elide detail not required for particular analyses.

We believe that concepts from software architecture modeling are general enough to be used as the basis for both software and physical modeling. An architecture model provides the scaffolding upon which different kinds of semantic analysis can be built. In [26] we presented elements of a cyber-physical architecture style that can be used to construct comprehensive models of cyber-physical systems. In this paper we elaborate on this style. Specifically, we show how the generalized cyber-physical style can be specialized for specific domains, is amenable to analysis using existing tools and modeling languages, and supports alternative architectural views.

The following section introduces related work in the area of architecture and physical modeling, and discusses why existing approaches are inadequate for modeling cyber-physical systems. Section 3 presents the CPS architectural style and introduces an example system modeled with this style. Section 4 describes two alternative architectural views of the same underlying CPS system. Section 5 explains how these views can be analyzed with existing tools. In Section 6 we discuss the current state of our implementation and how we are applying our approach to real systems. Finally, Section 7 summarizes the contributions of this paper and discusses directions for future work.

2 Related Work

Research on architectures has focused primarily on the cyber aspects of systems, particularly software architectures and the cyber substrates for hosting distributed systems (e.g., processors, memory, buses and networks). In this work, we focus on component and connector (C&C) views of an architecture because we are interested in the dynamic interactions between the cyber and physical parts of a system. A number of architecture description languages (ADLs) have been developed for representing C&C architectures [23]. They all build on the basic concepts outlined in [27, 24, 8]: a software architecture is a graph of components and connectors in which components represent computational elements of a system's runtime structure, and connectors represent pathways of communication between components. Modern ADLs also allow architects to define and use architectural styles to specialize general architectural concepts into specific domains. There are also a range of formalisms and tools for analyzing architectures [23, 15].

Architectural views have long been recognized as a critical aspect of architectural specification [11, 21]. For example, the IEEE 1471 standard for architectural documentation and the Open Group's TOGAF [28] are based on the idea of specifying a collection of architectural views, where different architectural

views represent the concerns of different stakeholders [6]. We can take advantage of this concept to provide different views of a cyber-physical system that are related, but separate out different modeling concerns, and so can be used with existing analyses for cyber and physical systems.

There are two fundamental problems with current architecture modeling that limit its potential to fully address the engineering problems of large-scale, heterogeneous cyber-physical systems: inadequate representation of cyber-physical elements and interactions, and an inability to support tractable analysis based on separation of concerns. There is also a need to handle consistency and mapping between the different views of a CPS architecture, so that we can analyze the coverage of all the views to be satisfied that the entire system is adequately modeled, and to related the analyses done in different views to each other. We need this to infer the correctness of the entire cyber-physical system.

In the domain of physical systems, there are also a number of tools for modeling and simulating physical systems. These tools model aspects of the physical dynamics of systems such as thermal dynamics, mechanics, etc. For example, Modelica is a popular object-oriented, open-standard language for constructing component-based models of physical systems [3]. MapleSim is tool for developing models of physical systems and generating efficient code for real-time simulations, particularly for hardware-in-the-loop testing [1]. In contrast to the signal flow semantics used in control system modeling, compositions of physical systems are most naturally modeled using acausal connections, which are symmetric reaction relations for which the directionality of interaction flows is determined by the internal states of both components. Making connections between physical modeling tools and control-oriented modeling frameworks has become an important goal for model-based development. The MathWorks has introduced a collection of physical modeling tools, especially Simscape, a textual MATLAB-based modeling language, along with a Simulink block set, that makes it possible to integrate physical models with control-oriented simulations [4].

These tools for modeling physical systems do not provide representations of many cyber aspects systems, such as timing, deployment, or behavior of the software. More comprehensive modeling tools include Ptolemy II [9], which provides a set of models of computation for modeling both software and physical systems. SysML (Systems Modeling Language) [5] is an extension of the UML2.0 standard for systems engineering applications. SysML reduces UML's software-centric restrictions and adds two new diagram types, requirement and parametric diagrams. However, there currently does not exist an easy way to incorporate physical dynamical models, including feedback control systems using SysML modeling constructs. MARTE [2] is a UML profile that adds capabilities for model-driven development of Real Time and Embedded Systems (RTES). It consists of a set of domain-specific specializations of appropriate general UML concepts providing modelers with first-class language constructs for modeling RTES applications. MARTE has several sub-profiles that address specific modeling dimensions in RTES. Among them are specification of non-functional properties, semantics and values for time, modeling and specifying resource usage,

and time and space related allocation. These modeling approaches typically stop at the interface between the software and the physical environment and do not model the physical environment themselves. Therefore, it is difficult to understand the interactions between physical aspects of the system, for example how heating one room affects the temperature in adjoining rooms, and their effects on the software intended to control them.

3 A CPS Architectural Style

Our work on architectural modeling for cyber-physical systems addresses the deficiencies of the related work in three ways: (a) we develop an architectural style that incorporates physical modeling elements as first class elements in the style, and also provided elements intended to bridge between software and physical elements; (b) we can extend this style to make it amenable to a number of different specific cyber-physical domains; and (c) architectures defined in this style can be used as the basis for analysis using existing modeling tools. In this section, we introduce a CPS architectural style.

The challenge in defining an architectural style is to find a balance between specificity and generality. We focus on cyber-physical systems for embedded monitoring and control. The CPS architectural style proposed in [26] treats cyber and physical elements as equally significant, and that can serve as the foundation for application-specific styles in this broad domain. It comprises three related families pertaining to the cyber domain, the physical domain, and interconnections between these domains. In this section we briefly describe the elements of the CPS architectural style and illustrate its application using a small example.

We define our architectural style in Acme [14], which has support for defining and extending architectural styles, as well as defining *mix-in* styles that can be used to support architectural analysis.

3.1 CPS Components and Connectors

The *cyber family* provides the following elements to represent computation, data and communication:

- *cyber components*: data store, computation, IO interface;
- *cyber connectors*: call-return, publish subscribe.

The elements of the *physical family* correspond to standard constructs in energy-based models of physical dynamic systems using concepts of effort and flow variables [16, 20]:

- *physical components*: energy source, energy storage, energy transducer, energy dissipators;
- *physical connectors*: power flow, equal effort (shared variable)

The elements of the *cyber-physical interface (CPI) family* represent connections between computational and physical systems:

- *CPI components*: cyber-to-physical/physical-to-cyber transducers;
- *CPI connectors*: cyber-to-physical/physical-to-cyber translators.

The difference between CPI components and CPI connectors is a matter of detail and sophistication in the interface. An intelligent sensor that performs signal processing functions might be represented as a CPI component, whereas a simple digital thermometer could be represented as a CPI connector.

These generic CPS components and connectors can be used to define new families with application-specific features and attributes. For example, for CPS applications involving thermal processes, the physical family can be specialized as a *thermal family* by introducing annotations that identify the effort and flow variables as temperature (Kelvin) and heat flow (Watts), respectively. The energy source components become (constant or variable) temperature and heat flow sources, while heat storage elements represent the ability of a material body to store internal energy. Dissipative components reflect phenomena where heat energy is lost over time, such as thermal conduction, convection, and radiation processes. Two types of *thermal connectors* are defined as: *thermal flow*, which introduces dynamic coupling between thermal components through lossless heat flow; and *equal temperature*, which conserves heat energy by enforcing the same temperatures and net-zero heat flow among multiple thermal components.

3.2 An Example Cyber-Physical System

To illustrate the application of the CPS architectural style, we consider a heating system for a two room-building illustrated in Fig. 1. Two rooms are separated by a wall with a specified thermal conductivity. The outside walls of the building provide insulation from the outside temperature with a specified thermal conductivity. We assume a winter heating scenario, that is, we assume the outside temperature is lower than the desired temperature.

The temperature is sensed in Room 1 and the furnace supplies heat directly to Room 1. A thermostat contains a microprocessor that implements an on-off control algorithm by sampling the temperature and the temperature setpoint, which is set manually in Room 1. The thermostat sends on/off commands over a network to the furnace microcontroller, which in turn activates/deactivates the heating coil. In this example we assume air is circulated over the coil constantly, i.e., ventilation is not controlled. As shown in the figure, there is also a manual switch on the furnace that can be used to deactivate the heating coil. When the manual switch is switched from off to on, the heating coil is activated but does not start heating until it receives an ON command from the furnace controller.

The cyber elements in this system are the thermostat microprocessor, the furnace microcontroller, and the network connections from the thermostat to the temperature sensor, setpoint input and furnace. The outdoor temperature, the rooms, and the furnace heating coil comprise the physical elements. The thermometer, the actuator between the furnace microcontroller and the heating coil, the manual furnace switch and setpoint input device connect the cyber and physical worlds.

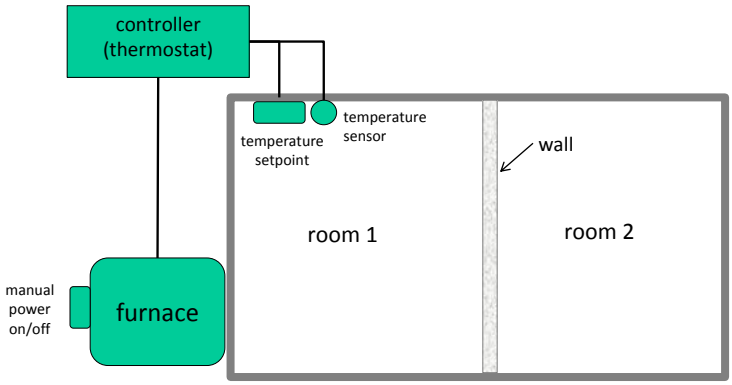

Fig. 1. Two-room building and heating system.

Figure 2 illustrates the use of the general CPS and thermal components and connectors in AcmeStudio to model the two-room heating system. The thermostat microprocessor is modeled as a combination of a computation component that implements the control algorithm and a data store that represents the memory where the setpoint value is stored. The furnace microcontroller is modeled as an IOinterface component, reflecting its function as device-level interface software. The unidirectional communication from the thermostat to the furnace is represented by a send-receive cyber connector. Each room is modeled by a thermal storage component, which represents an enclosed volume with heat storage capacity. The outside environment and the heating coil are both modeled by temperature sources. The wall between the two rooms, and the wall between each room and the outside environment are each represented by a thermal conduction component. The heat exchange between Room 1 and the furnace heating coil is represented by a thermal convection component. The heat flow between all the components is represented by thermal connectors. For example, there is an equal temperature connector that connects the corresponding ports of Room 1 to one side of each wall component and the convection component.

The thermometer is represented by a physical-to-cyber connector from Room 1 to the thermostat controller. The actuator from the furnace microcontroller to the heating coil is a cyber-to-physical connector that converts software commands to physical signals that change the temperature of the coil. The manual switch and the setpoint input device are shown as physical-to-cyber transducer elements, which accept user inputs and convert them into cyber events.

4 Two Alternative Architectural Views

Domain-specific components and connectors built on the CPS architectural style make it possible to create architectural models with all the details needed to completely describe a cyber-physical application. A complete architecture detailing

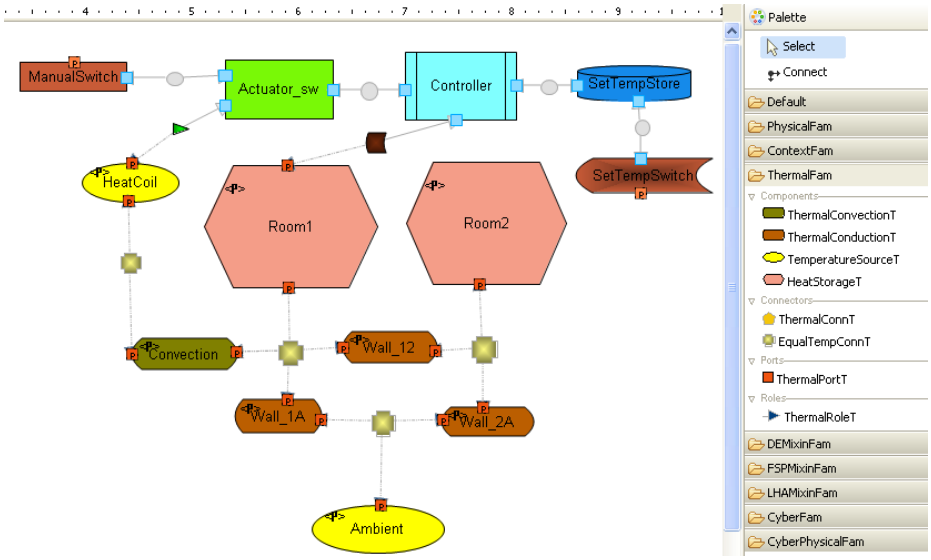

Fig. 2. Complete architectural view of the example system.

all components and connectors for a cyber-physical system is useful to document design alternatives and to guarantee that connections between components are compatible and complete. Such "flat" models are of limited value for formal analysis, however, because the associated behavioral modeling formalisms are not usually so comprehensive. Moreover, analyses of various system properties usually do not require all of the detail in the complete architectural model, and for all but the simplest systems, retaining all of the architectural details would make most analyses intractable. For example, the number of rooms in the physical part of the model and how they are connected to each other (walls, corridors, windows, etc.) may not be necessary for a model of how the software behaves. However, the abstracted characteristics of the rooms (such as how long they take to reach a desired temperature) may be important to the software modeling (e.g., in informing the software how much heat to supply to the room, and how long to turn on the furnace for). To illustrate how a designer can look at a given system from multiple modeling perspectives motivated by the types of behavioral analyses that can be carried out at the system level.

4.1 FSP Architectural View

FSP [22] is a process algebra in the tradition of CSP [19] for modeling behavior of software. FSP can capture the behavior of cyber elements fairly well, while the physical elements are described by abstracting away their continuous dynamics. The components in an FSP architectural view are those entities whose behavior can be described by an FSP *primitive process*. A connector between two FSP

components signifies that the two processes interact with each other through events, and provides mapping between the event alphabet of the components it is connected to. The behavior of the overall system is the parallel composition of the component and connector processes [7].

The FSP architectural view has three separate FSP processes. The *Environment* component encapsulates the thermal dynamics of the two-room building. Its FSP specification contains the temperature states LOW, NORMAL and HIGH, which discretize the continuous temperature range of Room 1. The environment makes transitions between temperature states in response to the HEAT and COOL events, which are communicated through the connector between the environment and the *Furnace* component. The furnace translates, which cyber actions into heating effects on the environment, can be in one of three states: HEATING, NOTHEATING, or SHUTDOWN. Two events, POWERON and POWEROFF, are defined in the furnace specification to model the furnace being switched on and off manually. The *Thermostat* component represents the control actions taken by the cyber world on the environment. The thermostat senses the state of the environment through shared sensing events defined in the connector between the two components. Based on the current temperature value, the thermostat turns the furnace heating coil on or off using events HEATON and HEATOFF defined in the common connector.

The FSP architectural view is generated from the complete CPS architecture by encapsulating the manual switch and furnace microcontroller as a single *Furnace* component, while the thermostat controller, memory and set point input are abstracted into a *Thermostat* component. The rooms, outside environment and heating coil make up the *Environment* component.

4.2 LHA Architectural View

For modeling mixed discrete-continuous dynamical systems, hybrid automata [25] serve as an intuitive and expressive framework. Linear hybrid automata (LHA) [18] are class of hybrid automata in which the dynamics of the continuous variables and conditions for discrete transitions are represented by linear predicates.

In this hybrid system view of the system, each component's behavior is specified by a LHA. The discrete states of the component are modeled by the vertices of a graph (modes), and the discrete dynamics of the component are modeled by the edges of the graph (discrete jumps). Discrete jumps between states are triggered by events that are generated internally based on guard conditions that are affine functions of the continuous state variables or externally by other components because they synchronize on the transitions. Synchronizing on transitions captures the discrete interaction between LHAs. In input-output frameworks, LHAs can also declare continuous variables that belong to other LHAs as their input variables, which they can read but cannot modify. Declaring a state variable from one LHA as an input variable of another captures the continuous interaction between LHAs. A connector in the LHA architectural view defines the set of synchronizing events and input variables shared between the LHAs of

the connected components. The behavior of the complete system is defined as a *composition* of the LHAs defined by the individual components.

To illustrate the LHA architectural view of our example system, the physical dynamics of the rooms, the control action of the thermostat and the switching of the furnace are modeled as three distinct LHAs. The Rooms automaton has four discrete states. When the rooms are in the HEATING mode, the temperature increases with positive bounds on the rate of change. When the rooms are in the COOLING mode, the temperature decreases with negative bounds on the rate of change. The Rooms LHA changes between these two states on the events *heatOn* and *heatOff*, which it receives from the *Furnace* LHA as a discrete interaction via the shared connector. The temperature state variable belonging to the *rooms* LHA is declared as an input variable by the *Thermostat* LHA. The rooms cannot heat beyond temperature *t_Hottest* or cool below *t_Coldest*. This behavior is captured by having the rooms enter two other states atHOTTEST and atCOLDEST when the respective temperature limits are reached.

The LHA architectural view is generated from the complete CPS architecture by defining an Acme representation for each LHA component, which is similar to that for the FSP architectural view. The difference is that the furnace heating coil is now a part of the *Furnace* representation instead of the *Environment*.

5 Analysis

In this section, we illustrate how each of the architectural views can be used to provide analysis of the physical and cyber aspects of the system, using existing tools and techniques. We first show how FSP annotations can be used to analyze the behavior of the software components then showing how LHA can be used to analyze timing aspects of the system.

5.1 FSP Annotations and Analysis

FSP is supported with an analysis tool called the Labeled Transition System Analyzer (LTSA), which can be used to check a system for properties such as absence of deadlock and liveness. To specify liveness properties in the style of linear temporal logic, LTSA uses *fluents* [17]. A named fluent is defined by indicating two sets of events: the first set of events causes the fluent to become true; the second set of events causes the fluent to become false. So, for example, the specification `fluent BEINGCOLD = <cold, {normal, hot}>` defines a fluent `BEINGCOLD`, which becomes true when the COLD event occurs, and becomes false when either the NORMAL or HOT event occurs. Fluents can then be used in specifications, like `assert []<>BEINGCOLD` which states that no matter what state, the system will become cold.

In the FSP view of the architecture, each component and connector can be annotated with an FSP process description defining its behavior. For example, the *Environment* is annotated with the following FSP process:

```
ENV = E[Normal],
  E[temp:TempSetting] =
(be[temp] ->
 C[temp]),
  C[temp:TempSetting] =
(// Low temp cases
 when (temp==Low) heat -> E[Normal]
|when (temp==Low) cool -> E[Low]
|when (temp==Low) dontChange -> E[Low]
// Normal temp cases
|when (temp==Normal) heat -> E[High]
|when (temp==Normal) cool -> E[Low]
|when (temp==Normal) dontChange -> E[Normal]
// High temp cases
|when (temp==High) heat -> E[High]
|when (temp==High) cool -> E[Normal]
|when (temp==High) dontChange -> E[High]
).
```

The rooms can be in three different temperature states as indicated. The *room* event '`be[Low]`', which signifies the room temperature being low and the *thermostat* event `sense[Low]`, which signifies the thermostat sensing the room temperature to be low are renamed as simply `Low`. This event renaming information is annotated to the connector between the environment and the thermostat. In a similar way, the *Thermostat* and *Furnace* are annotated with FSP processes.

The FSP specification is formed by composing the process definitions of each of the components and connectors, along with event mappings, fluents and assertions in the system, and properties defining global states and constants (also defined in the system) and stitches them together. With all this information collected, the system description as the *composite process* is generated, with the LTSA checks inserted. This information is then sent to LTSA to be examined.

Three fluents are defined to characterize the state of the overall system in this example; whether the environment is cold, normal, or hot. A sample LTL specification (indicated by ASSERT) captures a required property of the system: it is always the case that if the environment is cold, then eventually it will be brought to the normal state. When we check this property in LTSA, we get an error message, and a trace indicating that there is a problem.

It turns out that the thermostat does not take into account the fact that the furnace might be manually turned off/on. Hence the thermostat can get into a state where it is believes that the furnace is on, but the furnace has been shutdown and then manually switched on, leaving it in the off (non heating) state. This problem can be viewed as an architectural issue since it indicates the lack of a monitoring connector between the furnace and the thermostat, which would allow the thermostat to know when the furnace has been manually turned off/on. Once we have added in the extra connector, and rewritten the thermostat to handle those new events, the property succeeds.

5.2 LHA Annotations and Analysis

LHA can be analyzed with PHAVer [12], which supports reachability analysis, simulation relation checking and compositional verification of LHA. PHAVer allows for modular descriptions of a system in terms of its component automata.

Each component in the LHA architectural view is annotated by a PHAVer automaton. The 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. These individual automata can later be composed to model the complete system. Each connector is annotated with the lists of synchronizing labels: input-output variables (input of the first automaton/output of the second) and output-input variables (output of the first automaton/input of the second).

To illustrate for our example system, the *Thermostat* is annotated with the following PHAVer automaton:

```
automaton thermostat
state_var: c; //c = clock variable
input_var: t;
synclabs: tick, startHeat, stopHeat, exp;
loc idle: while c <= tau_sample wait {c'==1};
when c==tau_sample sync tick do {c'==0} goto checking;
loc checking: while c<=0 wait {c'==1};
when t<=tset_L sync startHeat do {c'==0} goto idle;
when t>=tset_H sync stopHeat do {c'==0} goto idle;
when tset_L<=t & t<=tset_H sync tick do {c'==0} goto idle;
initially: idle & c==0;
end
```

In a similar way, there exist LHAs for the *Furnace* and *Environment* components, with the relevant synchronization labels. The complete system is the composition of the individual component LHAs, along with the specification that the system will be checked against.

The FSP analysis of our example system described above exposed a problem: the thermostat's knowledge of the furnace state (on or off) could be erroneous, if the furnace was manually switched off. One solution was to add another connector that notifies the manual shutdown of the furnace to the thermostat. Another simple solution would be to have the thermostat time-out after a certain period and resend a command to the furnace. For this solution to work, the time-out of the thermostat has to be adequately fast. LHA can model this sampling rate, along with the room temperature dynamics using continuous variables.

This sampling strategy is modeled in the thermostat's LHA description as follows. The thermostat is in *idle* mode until its clock, a continuous state variable `c`, hits the sampling period `tau_sample`. Once that happens, the thermostat enters *checking* mode where it checks the continuous temperature variable `t` taken as input from the *rooms* automaton, and depending on where it lies with respect to the hysteresis band around the setpoint $[tset_L, tset_H]$, commands the *furnace* through the synchronizing labels `startHeat, stopHeat` or the externally

unobservable `tick`. The variable `tau_sample` is defined as a design parameter, i.e. a system constant.

With the specification modeled as an LHA, we can check whether the room temperatures stay in the desired range when the furnace is turned on and off manually. In particular, we validate the specification that (i) the temperature t remains in the acceptable zone $[t_m, t_M]$ whenever the furnace has not been turned off, (ii) whenever the furnace turns off, it stops checking what happens to the temperature and (iii) whenever the furnace turns back on, it waits for a little while, to allow the furnace to warm up, and then goes to (i) and starts checking again. This waiting period is modeled using an automaton called *recover* that resets the temperature test after a specified time given as a system constant.

The range of values for the thermostat sampling time and the recover time that obey the system specifications are found using the PHAVer analysis query

```
sys = thermostat & room & furnace & recover;
is_sim(sys,spec);
```

A correct set of values is one for which PHAVer returns '`Yes`' as an output for the '`is_sim`' analysis question. By analyzing the continuous-time behavior of the physical system (the room), the designer has an alternative solution to the furnace switch-off problem, using a validated thermostat sampling time.

6 Implementation and Validation

We have implemented the FSP and LHA analyses as plugins to AcmeStudio. Each analysis defines a *mixin* style that defines the properties required to annotate the architecture to make it suitable for the analysis. The plugins use these mixin families to provide custom user interface views in AcmeStudio for viewing and entering information pertinent for each analysis. Each plugin goes over the architectural model and extracts the relevant properties for analysis, producing a file for analysis by the respective tool, LTSA or PHAVer.

To validate our approach on a real-world example, we are modeling and analyzing the Stanford Testbed of Autonomous Rotorcraft for Multi-Agent Control (STARMAC) [13]. We report on this in detail in [10], and summarize it here. STARMAC is a fleet of quadrotor helicopters, developed as a test bed for novel algorithms that enable autonomous operation of aerial vehicles. Figure 3 shows the vehicle with four rotors arranged symmetrically about its body frame, each powered by lightweight, brushless DC motors, for a thrust of 8 N per rotor.

The vehicle is equipped with three separate sensors for full state estimation. An inertial measurement unit (IMU) provides three-axis attitude, attitude rate and acceleration. Height above the ground is determined using a sonic ranging sensor. Three-dimensional position and velocity measurements are made using differential GPS. An onboard extended Kalman filter is used to combine GPS and raw inertial measurements for accurate full-state estimation. Computation and control are managed at two separate levels. The low-level attitude control,

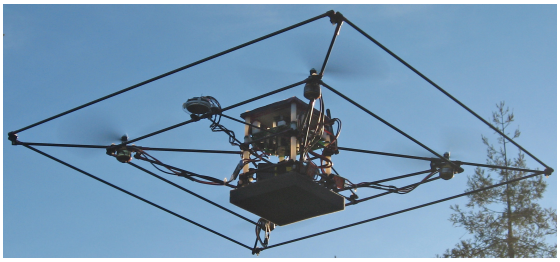

Fig. 3. The STARMAC quadrotor.[13]

which performs real-time control loop execution and outputs PWM motor commands, occurs on an onboard low-power microcontroller. The high-level planning, estimation and control occurs on a more powerful single-board computer. The ground station controller (GSC) is a high-level motion planner and coordinator for the quadrotor. It generates reference trajectories for the quadrotor to follow, displays telemetry data received from the vehicle, and manages coordination among multiple aircraft. The ground station also has joysticks for control-augmented manual flight, when desired. With reference to Figure 4(a), the nonlinear dynamics of the quadrotor helicopter are those of a point mass m with moment of inertia $I_b \in R^{3\times3}$, location $p \in R^3$ in inertial space, and angular velocity $\omega \in R^3$ in the body frame. The vehicle undergoes forces $F \in R^3$ in the inertial frame and moments $M \in R^3$ in the body frame, yielding the equations of motion, where D_b is the aerodynamic drag force, and g is the acceleration due to gravity. $R_{R_j,I}$ and $R_{R_j,B}$ are the rotation matrices from the plane of rotor j to the inertial coordinates and the body coordinates, respectively. One of our objectives is to formally represent such dynamic behavior in the CPS architecture for the quadrotor system.

Figure 4(b) illustrates the use of the CPS style to model the quadrotor in Acme. Each cyber controller (attitude, position, and GSC) is mapped to a separate computation component that implements the control algorithm. The communication of setpoints from a higher-layer controller to a lower-layer controller is modeled as a send-receive connector. The periodic relaying of vehicle state from the lower control layer to the higher layer is modeled as a publish-subscribe connector. This illustrates the use of distinct connector types to represent different communication patterns between the same components. The vehicle frame is modeled as a rigid-body component, whose mass and MI are affected by the forces and moments acting at its ports, according to the dynamic equations of the quadrotor. The vehicle frame is annotated with the body and inertial reference frames along with the $R_{R_j,I}$ and $R_{R_j,B}$ transformations. Each rotor and motor actuator is modeled as a single electromechanical transducer called Act, containing an electrical port and two mechanical ports, one each for the translational and rotational domains. The component models the conversion of input motor voltage to an output thrust (force) and torque acting on VehicleFrame. As we

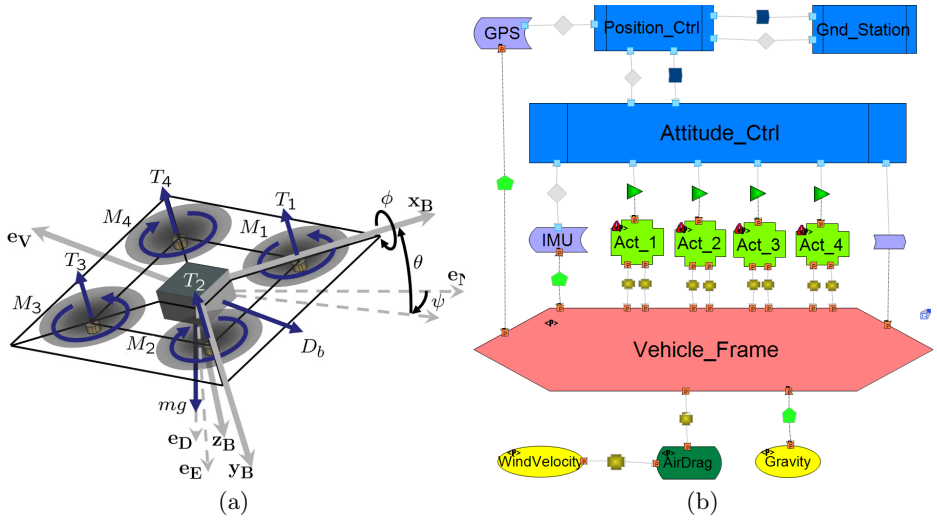

Fig. 4. Quadrotor (a) free body diagram [13] and (b) CPS architecture.

refine the architecture, this component can have sub-structure, where the motor and rotor are separate components, with a torque connector between them. Each Act is connected to the VehicleFrame by two equal velocity connectors, one for force balance and one for moment balance. This models the action and reaction phenomenon between each rotor assembly and the vehicle frame.

The drag force is described as a dissipative component, whose magnitude depends on the wind velocity and the aircraft velocity, among other parameters. The complex empirical relationship of drag force to the velocities at its ports is annotated as a behavior property of the component. Gravitational force is modeled as a flow source component, since it exerts a constant force on the airframe. It is connected to the vehicle frame by a measurement connector. The IMU and GPS are both modeled as P2C transducers, since they perform filtering on their raw sensor readings. On the cyber side, they are connected to their respective controllers by publish-subscribe cyber connectors, since these sensors send periodic streams of data to the controllers. On the physical side, they are connected by measurement connectors to the vehicle frame. The sonar sensor is modeled as a simple P2C connector, going from the vehicle frame to the attitude controller. The connector is annotated with sonar parameters including detection beam width, effective range, and resolution. Each Act component is sent actuation commands from the attitude controller through C2P connectors, representing the conversion of cyber commands to voltage for each motor.

7 Discussion and Future Work

This paper presents an approach to modeling cyber-physical systems using an architectural style that can be specialized for particular domains. The CPS architecture style provides a set of components and connectors for developing a complete architectural description of systems involving both cyber and physical elements. The CPS architecture becomes a frame of reference for multiple architectural views of a system corresponding to different modeling formalisms. Architectural views for two common modeling formalisms and associated analyses are illustrated for a temperature control system. The style is amenable for use with existing tools for analyzing software and physical systems.

The are several directions for further research and development. Further work is need, both theoretically and in implementation, for maintaining consistency among different views. We are currently exploring the issue of consistency from three perspectives: how to specify projections so that views can be automatically constructed from the detailed model, rather than done by hand as they are now; how to manage structural consistency so that changes in one view are reflected in the other views; and how to manage semantic consistency, so that the analysis and properties defined in one view can inform the analyses in other views.

Acknowledgments

This work was supported in part by National Science Foundation (NSF) under grant no. CNS0834701 and by Air Force Office of Scientific Research (AFOSR) under contract no. FA9550-06-1-0312.

References

1. MapleSim. `http://www.maplesoft.com/products/maplesim/index.aspx`.
2. MARTE. `http://www.omgmarte.org/`.
3. Modelica. `http://www.modelica.org/`.
4. Simscape. `http://www.mathworks.com/products/simscape/`.
5. SysML. `http://www.sysml.org/`.
6. ISO/IEC Standard for systems and software engineering - Recommended practice for architectural description of software-intensive systems. *ISO/IEC 42010 IEEE Std 1471-2000 First edition 2007-07-15*, 2007.
7. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.
8. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, second edition edition, 2003.
9. S. S. Bhattacharyya, E. Cheong, and I. Davis. Ptolemy II heterogeneous concurrent modeling and design in java. Technical report, 2003.
10. A. Bhave, D. Garlan, B. Krogh, A. Rajhans, and B. Schmerl. Augmenting software architectures with physical components. In *Proceedings of the Embedded Real Time Software and Systems Conference (ERTS^2 2010)*, 19-21 May 2010.

11. P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.
12. G. Frehse. Phaver: Algorithmic verification of hybrid systems past hytech. In *Fifth International Workshop on Hybrid Systems: Computation and Control (HSCC), Lecture Notes in Computer Science 3414*, pages 258–273. Springer-Verlag, 2005.
13. S. W. G. Hoffman and C. Tomlin. Quadrotor helicopter trajectory tracking control. In *Proc. of the AIAA Guidance, Navigation, and Control Conference*, 2008.
14. D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.
15. D. Garlan and B. Schmerl. Architecture-driven modelling and analysis. In *11th Australian Workshop on Safety Related Programmable Systems, SCS'06*, volume 69. Conferences in Research and Practice in Information Technology, 2006.
16. P. Gawthorp. *Bond Graphs and Dynamic Systems*. Prentice Hall, 1996.
17. D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2003.
18. T. A. Henzinger. The theory of hybrid automata. In *11th Annual IEEE Symposium on Logic in Computer Science, LICS'96*, pages 278–292. IEEE Computer Society Press, 1996.
19. C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
20. D. Jeltsema and J. M. A. Scherpen. Multidomain modeling of nonlinear networks and systems. *Control Sytems Magazine*, Aug 2009.
21. P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 2(6):42–50, 1995.
22. J. Magee and J. Kramer. *Concurrency: State Models and Java Programming, Second Edition*. Wiley, 2006.
23. Medvidovic, Nenad, Taylor, and R. N. A classification and comparison framework for software architecture description languages. *tse*, 26(1), 2000.
24. D. Perry and A. Wolf. Foundations for the study of software architecture. In *ACM SIGSOFT Software Engineering Notes*, volume 17(4), 1992.
25. T. A. H. Rajeev Alur, Costas Courcoubetis and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. *Hybrid Systems LNCS*, 736(4):209–229, 1993.
26. A. Rajhans, S.-W. Cheng, B. Schmerl, D. Garlan, B. H. Krogh, C. Agbi, and A. Bhave. An architectural approach to the design and analysis of cyber-physical systems. In *Third International Workshop on Multi-Paradigm Modeling*, Denver, Oct 2009.
27. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
28. TOGAF. Developing architectural views. *http://www.opengroup.org/architecture/togaf8-doc/arch/chap31.html*.