Augmenting Software Architectures with Physical Components

Ajinkya Bhave[1], David Garlan[2], Bruce H. Krogh[1], Akshay Rajhans[1], Bradley Schmerl[2]

[1]Dept. of Electrical and Computer Engineering
[2]School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890 USA
email: {ajinkya@ | garlan@cs.| krogh@ece.| arajhans@ece.| schmerl@cs.}cmu.edu

Abstract: This paper presents an extension of existing software architecture tools to model physical systems, their interconnections, and the interactions between physical and cyber components. We introduce a new cyber-physical system (CPS) architectural style to support the construction of architectural descriptions of complete systems and to serve as the reference context for analysis and evaluation of design alternatives using existing model-based tools. The implementation of the CPS architectural style in AcmeStudio includes behavioral annotations on components and connectors using either finite state processes (FSP) or linear hybrid automata (LHA) with plug-ins to perform behavior analysis. The application of the CPS architectural style is illustrated for the STARMAC quadrotor.

Keywords: Software Architecture, Embedded, Control, Physical Modelling.

1. Introduction

Today's models and methods for analysis and design of embedded control systems 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. This paper concerns extensions of software architectures to include the physical elements of embedded control systems.

Architectures typically represent systems at a higher level than simulation models, which represent the details of a particular system implementation. Although architectural modeling has been used in specific domains to incorporate physical elements as components, there is currently no way of treating cyber and physical elements equally in a general fashion. This is because the vocabulary is inadequate for representing physical components and their interactions with computational elements in embedded control systems and other cyber-physical systems (CPS). Our goal is to create a CPS architectural style as a comprehensive framework for using diverse models and tools for analysis and design of cyber-physical systems.

As a system is being designed, engineers typically want to expose some aspects of a system while abstracting away details of other aspects of the system. From an architectural perspective, when the system is modeled for analysis using a particular formalism, the model typically reflects a structural view of the system that differs from the detailed architecture in some respects. In some cases, one may even want to use a different architectural style to represent an analysis or simulation model in terms of components and connectors. This leads to the need to support different *architectural views* of the modeled system. In this paper the relationships of heterogeneous models to the CPS architecture are developed by formalizing the structure of each model as an architecture in its own right, which is then associated to the CPS architecture as a particular view of the system.

The following section describes related work in this area. Section 3 motivates the need for a unifying framework augmented with physical components and multiple system views. Section 4 introduces the CPS architectural style. Section 5 describes an example quadrotor system whose architectural description in the CPS style in presented in Section 6. Section 7 presents two models of the quadrotor in different formalisms as architectural views. Section 8 outlines how analyses can be carried out using appropriate plugins in the architecture design tool AcmeStudio. Section 9 summarizes the contributions of this paper and discusses directions for future work.

2. Related Work

Over the past decade software architecture has emerged as one of the primary techniques for disciplined engineering of large-scale software systems. Software architecture typically models a system as a graph of components and connectors in which the components represent principal computational elements of a system's run-time structure and the connectors represent the pathways of communication between components [2]. These elements are annotated with properties that characterize their abstract behavior and facilitate reasoning about system-level

design tradeoffs. There has been considerable research and development in Architecture Description Languages (ADLs) and tools to support their analysis and realization as code [13]. Standardized notations such as UML 2.0 [11], SysML [1] and AADL [10] provide modeling vocabularies of components, connectors and properties. These notations are supported by tools that provide graphical editing and viewing, hierarchical development [14], checking for component compatibility or substitutability [2], and evaluation of quality attributes such as performance, reliability, and security. Software architectures also support reuse of design expertise and code infrastructure and have been used effectively for a number of embedded control applications [4, 10].

There are also a number of tools for modeling and simulating physical systems. For example, Modelica is a popular object-oriented, open-standard language for constructing component-based models of physical systems [24]. MapleSim is a tool for developing models of physical systems and generating efficient code for real-time simulations, particularly for hardware-in-the-loop testing [25]. 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 state of the interconnected components. Making connections between physical modeling tools and control-oriented modeling frameworks has become an important goal for model-based development (MBD), particularly for avionics systems [19]. MathWorks has introduced Simscape, a textual MATLAB-based modeling language and Simulink block set that makes it possible to integrate physical models with control-oriented simulations [18].

There has been an effort to integrate UML/SysML and Modelica to provide support for modeling and simulating continuous dynamics [23]. The UML profile called ModelicaML enables users to depict a Modelica simulation model graphically alongside UML/SysML information models. The ModelicaML profile reuses several UML and SysML constructs, and also introduces new language constructs to specify equations. Similar work has been done for UML and Simulink [21]. None of these approaches uses physical components and their interconnections, however, to capture the physical interactions between system components and the emergent dynamic behavior.

Rather than developing a universal modeling syntax and semantics or a meta-modeling framework for translating models between tools, our approach is to relate models and verification results at the architectural level, a level of abstraction that captures the structure of the system without attempting to comprehend all of the details of any particular modeling formalism.

3. Need for physical architectural concepts

Although architectural modeling has been used in limited ways to support the incorporation of physical elements as components [6], it currently does not have adequate ways of placing cyber and physical elements on the same footing. This is because software architectures have an impoverished vocabulary for the types of physical components found in cyber-physical systems, the interconnections that determine the interactions between physical components and between cyber components, and the behaviors of physical entities.

Figure 1 shows an AADL representation of a flight computer interfacing with a landing gear [20]. Each component represents an AADL thread, with interconnections representing the communication of events or data. The Landing_Gear thread defines the behavior of the physical subsystem in terms of the events generated in response to pilot commands. This architectural description of the landing gear does not go beyond defining the data/signal interface of the controller to the sensors and actuators of the plant. The AADL component view of the gear resembles a black box for the system designer. There is no information about how the gear's subsystems are physically interconnected, or any representation of the controller's assumptions about the landing gear's overall dynamic behavior.

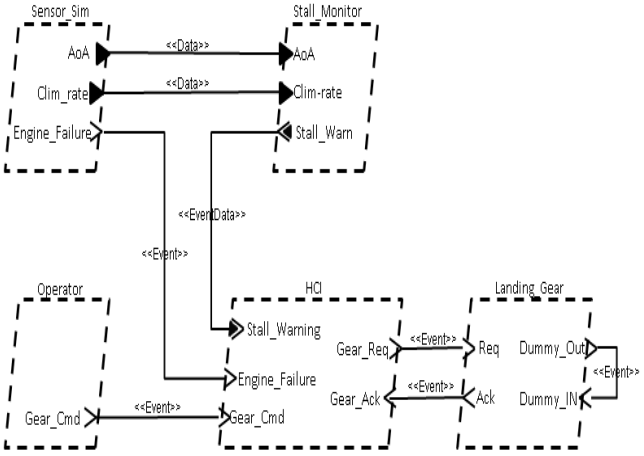

Figure 1. AADL description of aircraft landing gear.

This lack of physical detail is typical in current ADLs that target embedded real-time systems. Another example (from TELECOM ParisTech) is a flight controller for an F14 aircraft [17], which deals with mapping Simulink models into AADL components. A more comprehensive representation of the physical architecture of the aircraft would give designers the

ability to easily map models for the aircraft dynamics (in multiple formalisms) into the corresponding physical components and connectors that describe the aircraft architecture. For example, the detailed Simulink/Modelica models for the aileron, flaps, and rudder could be associated with *actuator* components in the aircraft architecture. A similar mapping to physical domain components could be done for dynamic models of the sensors, the aircraft frame (mechanical), and engine system (fluid, thermal). This approach would help the design team to store the physical modeling information about the system from multiple tools along with component specifications from the cyber domain, in a single, unified framework. It would also give the system engineer the flexibility to explore design alternatives for the various avionic subsystems.

Another important consequence of incorporating the physical subsystems into the architecture is the ability to assess design trade-offs across the cyber-physical boundary. One could envision replacing a particular sensor with another having a better resolution/range, but with greater weight/cost. To study the system-wide effect of this change, one could analyse the performance to weight tradeoffs on the combined software and physical aircraft systems. Several relevant questions could be answered. For example, does the change impact the data rate of the communication channel between the high resolution sensor and the controller? Will the scheduling of the current task set still be valid with more information to process per sensor update cycle? Which would give better control performance: a better sensor with a simpler control algorithm or a less costly sensor with a memory- and processor-intensive controller? The answers to such questions involve choices in both the cyber and physical worlds, including the couplings between the two domains. Having a unified CPS architectural framework can decrease the lead-time to develop several ready-to-use architectures of the aircraft model, with different levels of detail, so that the engineers can investigate many more design alternatives with a high level of accuracy of the analysis, thus minimizing the risks of costly “design-build-test-fix” cycles.

A third benefit concerns the relationships between the multiple heterogeneous models utilized in the design and verification of a complex cyber-physical system. The exchange of information, assumptions, and inferences among the many groups of engineers involved is typically informal at best, and it is particularly difficult when the structure and semantics of the modeling formalisms differ significantly. Consequently, correctness of the design is inferred by a combination of engineering judgment supported by extensive testing of the final system. There is a need to develop an architectural formalism with structural and semantic annotations that can represent both cyber and physical system-level requirements and specifications along with the assumptions and implications associated with the many verification models developed for the system. The proposed CPS architectural style provides the reference context for making meaningful associations across the disparate models used for analysis and verification.

4. CPS Architectural Style

We use the Acme ADL [5], which has strong support for defining extensible architectural styles. In Acme, an architectural style is represented as a family of element types that adhere to rules governing what kind of components and connectors can be present in the system and the manner in which they can be connected. A general style can be refined into an application-specific style by extending existing types and adding additional element types and rules.

The challenge in defining an architectural style for cyber-physical systems is to strike a balance between specificity and generality. We focus on embedded monitoring and control systems. Our goal is to create a family of general components and connectors that can serve as the foundation for application-specific styles in this broad domain. Towards this end we define the following three related families pertaining to the cyber domain, the physical domain, and their interconnection. The goal of creating the CPS architectural style is to provide a representative set of components and connectors that can be extended to full implementations in targeted application frameworks.

4.1 Cyber family

The cyber side of CPS is the traditional domain for ADLs and provides support for standard real-time monitoring and control applications. The cyber components are:

Data Stores: These components store data as an interface between the computational elements in the system. In simple systems, these could be just passive memory blocks. In complex systems there could be further details specifying what components can read and write to the data store components.

Computation: Computation components operate on and update data posted in data store components. This includes components that perform filtering, state estimation, and control.

IO Interfaces: These components perform the computations and timing functions required to sense and control the physical world. This would include, for example, smart sensor software that processes raw sensor data.

In addition to the computational aspects of the software, it is important to represent the communication elements in the system to reason about timing between software elements and how this affects the physical behavior of the complete system. We represent two major types of cyber connectors, a *call-return connector,* representing one-to-one communication and a *publish-subscribe connector* for one-to-many communication.

The communication mechanism could be event-based, synchronous or asynchronous, and is specified in the role defined for each connector. Furthermore, each connector type can be refined further, based on the communication semantics of the interacting components. For example, the publish-subscribe connector could be elaborated to define communication between CORBA components.

4.2 Physical family

There are several challenges in developing a suitable architectural representation of the physical side of cyber-physical systems. Architectural models should not have all the details required for a full simulation of the physical dynamics. At the same time, the architectural components and connectors should correspond to intuitive notions of physical dynamics in the same way cyber components and connectors correspond to elements of computational systems. To achieve this balance, we introduce components and connectors based on an energy view of physical systems. This provides a domain-independent perspective, including the ability to represent interactions between different physical domains and the possibility to specify system properties such as power flow and causality. This is similar to the perspectives taken in bond graphs [8] and Lagrangian mechanics [12], where power-conjugated variables (effort and flow) describe signal flows between sources, energy storage elements, and dissipative elements.

The physical family is an architectural style to model multi-domain physical systems. When two or more components from the physical family are connected, the implication is that they exchange energy in some way. The ports of each physical component define its effort and flow variables. The product of this "power conjugate" pair equals generalized power in the physical domain of interest. Power flowing *into* a component is defined as positive. This is analogous to positive mechanical work being defined as work done by the applied force *on* the component. A component's constitutive equations (attached as behavior annotations) define the relationships between the port variables and the internal state variables. Each physical domain extends the basic port type by defining effort and flow variable types relevant to that domain, along with their units and ranges of acceptable values. For example, an electrical physical port would define voltage and current as the conjugate variables for electrical components. There would be two (or more) ports for each generic electrical component. The voltage across the component and the current through it (along with either charge or flux as the stored quantity) would be defined in terms of the conjugate variables at the electrical ports.

A connector (along with its roles) defines the relationship between the effort and flow pairs of all the components that are attached to the connector. The connectors constrain the form of energy exchange between the components by enforcing the laws of conservation of the respective physical quantities at all times. For example, an electrical connector's semantics would enforce Kirchhoff's current and voltage laws between the conjugate variables of its connected components.

The physical component types are:

Energy sources: A source component delivers constant effort (or flow) to other components, regardless of the load presented to it. This is analogous to an ideal voltage (or current) source. Because of the defined direction of power, such components will have negative flow as long as they are supplying power to other components. A source behaves like an energy sink when it consumes power from its surroundings. An example is a rechargeable battery component in charging mode.

Energy *storage*: An energy storage component models dynamic elements or subsystems that store energy, such as components that have capacitive and inductive properties in electrical systems. Ports on these components allow power transfer to other subsystems.

Dissipative components: These model physical elements that lose energy over time. They correspond to resistors in electrical circuits and dampers or friction losses in mechanical systems. Power losses can take place in complex ways within physical components. Hence, the semantics of a dissipative component is completely defined by its *energy loss function*, which describes the relationship between the effort and flow variables of all its connected ports.

Physical transducers: Transducers represent power transfer or energy conversion between different types of physical domains. These components are particularly useful in modeling multi-domain systems with, for example, electromechanical devices that transform energy between the electrical and mechanical domains. Transducers contain at least one port from each of the physical domains they interconnect.

The physical connector types are:

Equal effort: The constraints on the power conjugate variables of the component ports coupled with this

connector are: (1) the effort variables of all connected ports are equal; and (2) the sum of the flow variables of all connected ports is zero. This connector represents the application of Kirchoff's laws in the electrical domain and force/moment balance laws in the mechanical domain to coupled components.

Equal variable: This connector indicates that the variables at the ports of the components are identical. There is no directionality associated with this connector.

Measurement: Measurement connectors indicate conjugate variables that are determined by one physical component and used as an input in another physical component. Thus, these connectors are directional and correspond to connections in traditional block diagrams (e.g., signal lines in Simulink).

4.3 Cyber-physical interface family

We define the cyber-physical family to bridge between the cyber and the physical worlds. The following elements of the cyber-physical interface (CPI) family represent connections between computational and physical systems:

CPI components: cyber-to-physical (C2P)/physical-to-cyber (P2C) 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.

Together, the three generic families can be combined to and extended to provide a unified model of a cyber-physical system. The generic CPS component and connector types can be used to define new families with application-specific features and attributes. For example, the physical family can be specialized to the translational/rotational mechanical domain by creating physical ports that define velocity/ angular velocity as effort variables and force/torque as flow variables. The source components become force/torque and velocity/angular velocity sources, respectively. Mass and moment of inertia (MI) correspond to energy storage elements. They represent the ability of a material body to store kinetic and potential energy. The energy storage concept can be generalized to a rigid body component, which contains both mass (annotated with the centre of gravity (CG) coordinates) and MI as subcomponents. The rigid body also contains body coordinate frames and transformations between them as annotated properties. Dissipative components reflect phenomena where mechanical energy is lost over time, such as static friction and viscous damping.

We can define three mechanical connectors, corresponding to the three physical connector base types. For example, an *equal velocity* connector defines a mechanical coupling between two or more components which constraints them to move with the same velocity in the given frame of reference, while the joint forces sum to zero. Equal variable connectors equate the force and velocities of the connected components, while measurement mechanical connectors can be used to interface force (torque) and velocity sensors to rigid body components.

5. Example: STARMAC Quadrotor

The Stanford Testbed of Autonomous Rotorcraft for Multi-Agent Control (STARMAC) [22], a fleet of quadrotor helicopters, has been developed as a test bed for novel algorithms that enable autonomous operation of aerial vehicles. As shown in Figure 2, the vehicle has four rotors, arranged symmetrically about its body frame. The rotors are powered by lightweight, brushless DC motors, which result in a thrust of 8 N per rotor.

Figure 2. The STARMAC quadrotor [22].

The hardware configuration of the STARMAC is shown in Figure 3. The vehicle is equipped with three separate sensors for full state estimation. A Microstrain 3DMG-X1 inertial measurement unit (IMU) provides three-axis attitude, attitude rate and acceleration, through a built-in estimation algorithm that relies on three gyroscopes, three accelerometers and three magnetometers. Height above the ground is determined using a sonic ranging sensor, either the Devantech SRF08 or the Senscomp Mini-AE with 3 to 5 cm accuracy. Three-dimensional position and velocity measurements are made using differential GPS relying on the Novatel Superstar II GPS unit. This unit outputs raw measurements at 10 Hz and the resulting position accuracy is 1 to 2 cm relative to a stationary base station. 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, which performs real-time control loop execution and outputs PWM motor commands, occurs on a Robostix microcontroller based on the Atmega 128 processor. The high level planning, estimation and control

occurs on either a lightweight Gumstix running embedded Linux on a PXA270 microprocessor, or on an Advanced Digital Logic ADL855 PC104 running Kubuntu Linux.

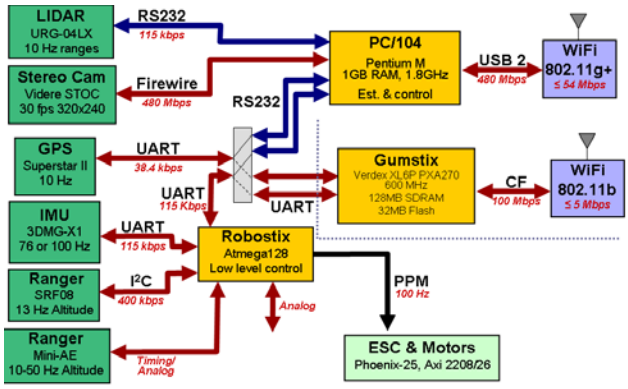

Figure 3. Quadrotor hardware architecture [22].

The Gumstix (0.05 kg) provides sufficient computational resources for carrier-phase differential GPS calculations and can be used to perform autonomous position control without the ability to add additional coordination algorithms between vehicles. As an alternative, the PC104 (0.48 kg) provides a wealth of computation power, at the cost of additional weight and hence, shortened flight times. The Robostix and Gumstix/PC104 communicate via a 115 kbaud RS-232 (serial) link. Communications between the high-level computers and the ground station are managed through UDP over a WiFi network. The Gumstix uses 802.11b, and the PC104 uses 802.11g.

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, we see that 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 $\rho \in R^3$ 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,

$$F = -D_b e_v + mge_D + \sum_{j=1}^{4}(-T_j R_{R_j,I} z_{R_j})$$

$$M = \sum(M_j + M_{bf,j} + r_j \times (-T_j R_{R_j B} z_{R_j}))$$

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.

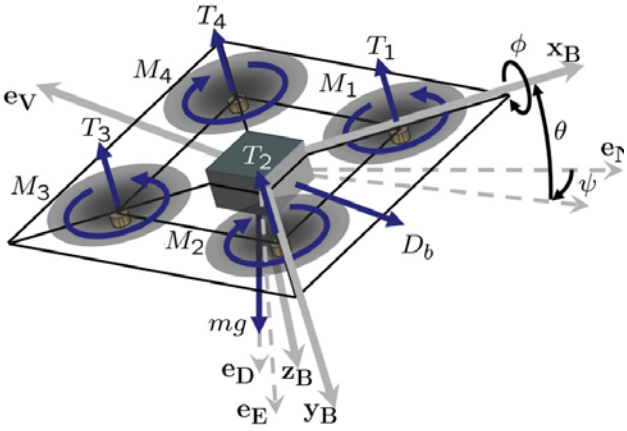

Figure 4. Quadrotor free body diagram [22].

6. Quadrotor architecture in the CPS style

Figure 5 illustrates the use of the CPS style to model the quadrotor in AcmeStudio. On the cyber side, each 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. Since there is no direct communication channel between the attitude controller and the ground station, no connector exists between them.

The vehicle frame is modelled 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 refine the architecture, this composite 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.

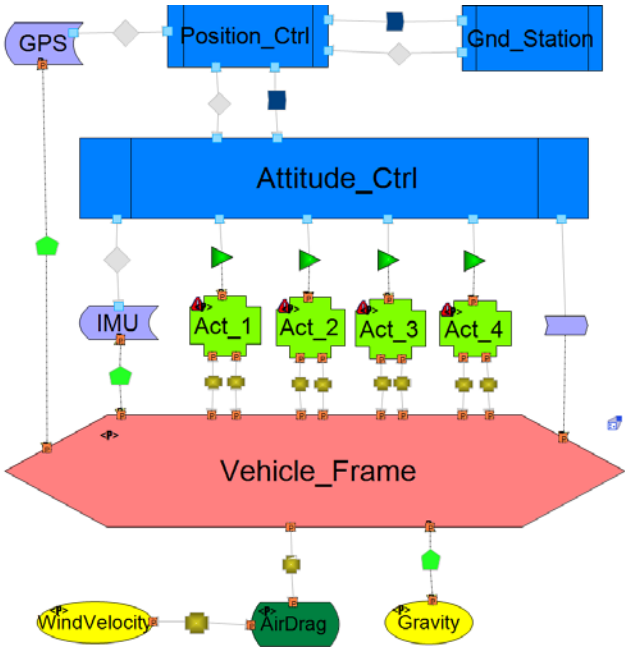

Figure 5. Quadrotor CPS architecture.

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 (PWM signals) for each motor.

7. Models as architectural views

In virtually all verification tools, models are constructed as collections of interacting components or modules. Thus, each verification model has a structure that can be viewed as an architecture with syntax and semantics defined by the particular formalism underlying the design of the tool. From a structural perspective, an architectural view supports the description of a derived architectural model to abstract over details that are irrelevant for a particular analysis. The relationships between components in the structure of a verification model and components in the CPS architecture will not generally be one-to-one, however. Current tools do not provide insights into the relationships between architectural views. This represents a problem for architectural modeling, since it is generally impossible to understand how design decisions or analyses in one view impact those of another.

The approach proposed here focuses specifically on component-and-connector views representing the architectures of verification models as abstractions of a shared more-detailed baseline architecture. In this context, well-defined mappings between a view and the full architectural description can be used as the basis for identifying and managing the dependencies among the architectural views and to evaluate mutually constraining design choices. The full baseline architecture thus becomes the repository for retaining results from various analyses and designs so that the interdependencies are explicit. The rest of this section describes how an architect can represent two different models of the quadrotor system as distinct architectural views.

7.1 Data flow view

From a control engineer's perspective, the quadrotor system can be viewed as a data flow (Simulink) model, as shown in Figure 6. The position and attitude controller components in this architectural view are represented by the *robostix* and *gumstix* subsystems in the Simulink model. The vehicle dynamics are represented by the *starmac_dynamics* block, and the GPS sensor is defined by the *Superstar_II block*. If each of these Simulink blocks could be mapped to a component type in a new 'data flow' architectural style, then the Simulink model could be thought of as an architecture instance in its own right. The component and connector semantics for this architectural style come from the underlying data flow semantics of the Simulink metamodel. With such a mapping defined, one can ask how the data flow architecture is related structurally and semantically to the underlying quadrotor CPS architecture.

The data flow view focuses on the control performance of the quadrotor and typically ignores controller

implementation details such as scheduling of tasks and associated communication jitter and delays. In particular, it assumes lossless communication between the GSC and the position controller onboard the quadrotor. However, in the implemented system, the two controllers communicate via a lossy wireless network. To analyze how the quality of the communication channel affects the temporal assumptions of the controllers, the designer would have to create a detailed simulation of the network in Simulink model. However, this class of behaviors could be analyzed fairly easily in a process algebra perspective of the system, as shown in the following section.

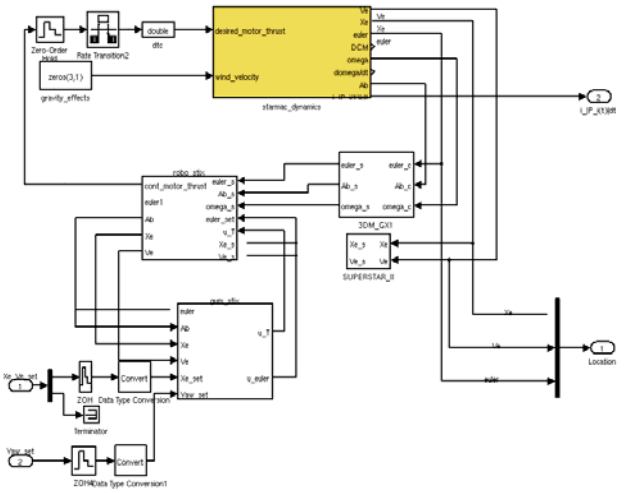

Figure 6. Quadrotor Simulink model[22].

7.2 Process algebra view

Finite State Processes (FSP) [13] is a process algebra in the tradition of CSP [27], where behavior is modeled in terms of event patterns, called processes that denote sets of traces. Each event in a trace represents a discrete transition of a system. Patterns are built up out of operators that indicate the occurrences of events, sequencing of events, choice, and parallel composition. Parallel processes synchronize on shared events, which can also be used to pass values between processes. In general, FSP captures the behavior of cyber elements fairly well, while 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 describes the protocol for that interaction.

The FSP architecture view of the quadrotor currently abstracts over the dynamics of the quadrotor and focuses on the communication between the ground station and position controller components. The quadrotor system is a composition of the two FSP components in which the GSC sends a single position command to the PC after it gets a command from the user to move the Rotor to a location. The PC continuously provides position status information back to the GSC. It is assumed that the PC controls the rotor assembly reliably so that if it is told to move the rotor to a given position, it will do so. This is an assumption that would be verified with other more-detailed models of that part of the system, probably in a different modeling formalism.

```
// Ground Station
GroundStation = GS[0][0][False],
  GS[actual:POS][desired:POS][sent:BOOL] = (
       getNewPos[newPos:POS] -> GS[actual][newPos][False]
     | at[newPos:POS] -> GS[newPos][desired][sent]
     | when (actual!=desired && !sent) sendCmd[desired]
       -> GS[actual][desired][True]
     ).
// Position Controller
PositionController = PS[0][0],
  PS[actual:POS][desired:POS] = (
       goTo[newPos:POS] -> PS[actual][newPos]
     | when (actual != desired) controlRotors
       -> PS[desired][desired]
     | curPos[actual] -> PS[actual][desired]
  ).
// Connectors
//Lossy connector
MsgConnLossy = (getMsg[val:VAL] -> DeliverMsg[val]),
  DeliverMsg[val:VAL] = (
       try -> putMsg[val] -> MsgConnLossy
     | try -> MsgConnLossy).

//Lossy connector with retry
MsgConnRetry = (getMsg[val:VAL] -> DeliverMsg[val]),
  DeliverMsg[val:VAL] = (
       try -> putMsg[val] -> MsgConnRetry
     | try -> DeliverMsg[val]).

// System with lossy connector
||QuadRotorL = (GroundStation || PositionController
          || cmd:MsgConnLossy || status:MsgConnLossy)
          /{sendCmd/cmd.getMsg, goTo/cmd.putMsg,
            curPos/status.getMsg, at/status.putMsg}.
// System with retry connector
||QuadRotorR = (GroundStation || PositionController
          || cmd:MsgConnRetry || status:MsgConnRetry)
          /{sendCmd/cmd.getMsg, goTo/cmd.putMsg,
            curPos/status.getMsg, at/status.putMsg}.

// Condition to check
assert CorrectControl =
  [](forall[p:POS] (getNewPos[p] -> <> at[p]))
```

Figure 7. FSP specifications for quadrotor.

Each component of interest in the architectural view is annotated with a process of the FSP specification in Figure 7 (e.g., the Gnd_Station component of the CPS architecture is annoted with the GroundStation process). The connectors between Gnd_Station and Position_Ctrl are modeled with one or the other of the connector behaviors. Having alternative connector protocols allows us to compare the behavior of the overall system depending on the protocol of the connection: a lossy connector might represent a

wireless UDP link while a lossy connector with retry might model a wireless TCP. The architecture can then be used to yield the complete FSP specification shown in Figure 7.

The property to be checked is the following: if the user tells the GSC to move the rotor to a particular position (abstracted as either position 0 or position 1), the GSC will eventually receive a status message from the PC that it is at that location. This FSP specification is analyzed by the Labelled Transition System Analyser [13] tool. The analysis tells us that the property holds when retrying connector is used, but fails when the lossy connector is used. This illustrates how a system designer might compare design tradeoffs in the cyber world between different protocols of interaction between GSC and PC.

8. Behavioural annotations and analyses

The architectural elements describe only the structural information about a system. To be able to do meaningful formal analysis on the system behavior, one must annotate the architecture with behavioral information. In Acme ADL, the behavioral annotation can be implemented via *properties* to capture the behavioral information. We have implemented the annotations for two types of behavioral modeling frameworks: FSP and Linear Hybrid Automata (LHA) [9]. We have developed plug-ins as extension points of AcmeStudio, which will display only the relevant information pertaining to the element or system selected by the user. We have also built into the plugins the ability to generate analyzable text files from these properties. The plug-ins traverse the architecture, gather the relevant information distributed through out the structure and generate a text file that is analyzed by the relevant tool. Figure 8 shows a schematic of this analysis flow. Currently, there exists one plug-in for FSP that generates a file that can be analyzed by LTSA and another plug-in for LHA that generates a file that can be analyzed by Polyhedral Hybrid Automaton Verifier (PHAVer) [7].

Because of the flexible development environment in AcmeStudio, a system designer can create a custom plugin for each of the formalisms used to model the system. Architectural elements are then annotated with properties relevant to each formalism. Figure 9 shows an LHA plugin being used to display annotations for each architectural element in an LHA view of a heating control system. The plugin framework is leveraged to generate analyses results from the heterogeneous behavioral annotations, and the results can be combined together or studied separately. Thus, the CPS architectural style along with these analysis tools provides a unifying framework to develop new methods for optimizing designs with respect to performance measures that characterize important features of the system behaviors.

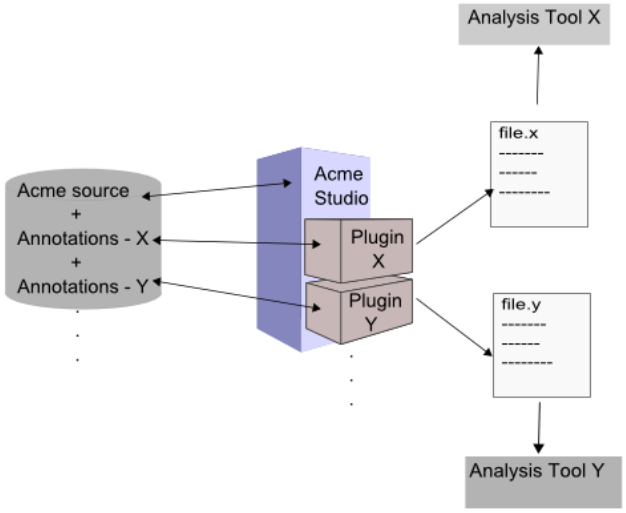

Figure 8. Behavioral analysis using plugins.

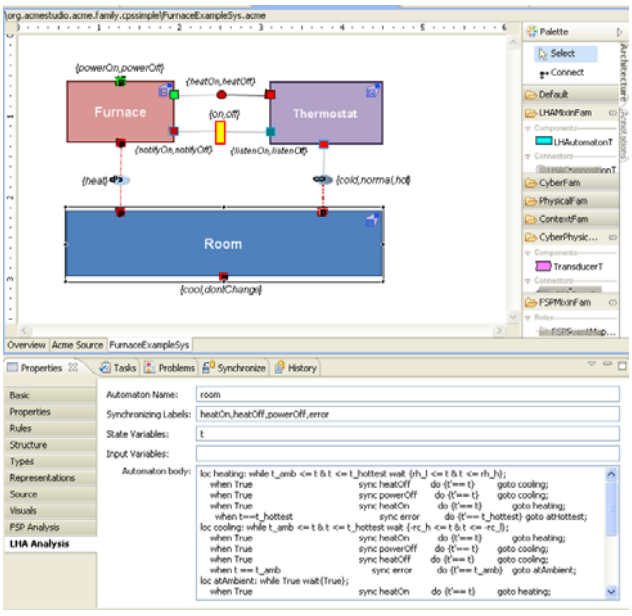

Figure 9. LHA annotations in AcmeStudio.

9. Conclusion

This paper presents a presents a way to augment architectural descriptions with physical elements. 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.

There are several directions for further research and development. In the current implementation, the

architectural views are connected to the detailed CPS architecture through hierarchical Acme representations that identify the relationships between components and connectors in each view with the components and connectors in the CPS architecture. Acme does not currently support resolution of inter-view correspondences, however, so further work is needed both in theory and tool support to formalize and analyze issues of consistency and completeness of various architectural views. Currently, the analysis plugins rely on external analysis tools to present results. A key usability issue is to provide the results in the context of the architectural view from which they originated, and this is ongoing implementation work. Such tools would provide a unified context to explore design alternatives that cut across the boundaries that currently separate the methodologies that focus on either the cyber or physical elements of cyber-physical systems.

10. Acknowledgments

The authors thank F. Murray Fishbeck for his help with the development of the STARMAC example. This work was supported in part by National Science Foundation (NSF) under grant no. CNS0834701 and by U.S. Air Force Office of Scientific Research (AFOSR) under contract no. FA9550-06-1-0312.

11. References

[1] SysML Home Page. http://www.sysml.org/

[2] R. Allen and D. Garlan. A formal basis for architectural connection. ACM Trans. On Software Engineering Methodology (TOSEM), July 1997.

[3] Clements P. Bass, L. and R. Kazman. Software Architecture in Practice. Addison-Wesley, second edition, 2003.

[4] P. Binns and S. Vestal. Formal real-time architecture specification and analysis. In 10th IEEE Workshop on Real-Time Operating Systems and Software, May 1993.

[5] R. T. Monroe, D. Garlan, and D. Wile. Foundations of Component-Based Systems, chapter ACME: Architectural Description of Component-Based Systems, pages 47-68. Cambridge University Press, 2000.

[6] Reeves G., Sacks A., Dvorak D., and Rasmussen R. Software architecture themes in JPL's mission data system. In AIAA Space Technology Conference and Expo, Albuquerque, NM, 1999.

[7] G. Frehse. Proceedings of the fifth International workshop on hybrid systems: Computation and control (HSCC), 2005.

[8] P. Gawthorp. Bond Graphs and Dynamic Systems. Prentice Hall, 1996.

[9] T. A. Henzinger. The theory of hybrid automata. In 11th Annual IEEE Symposium on Logic in Computer Science (LICS'96), pages 278-292, New Brunswick, New Jersey, 1996.

[10] J. Hudak and P. Feiler. Developing AADL models for control systems: A practitioner's guide. Technical Report Technical Report CMU/SEI-2007-TR-014, Software Engineering Institute, Carnegie Mellon University, 2006.

[11] G. Booch, J. Rumbaugh, and I. Jacobson. The Unified Modeling Language Reference Manual, Second Edition. Addison Wesley, 2004.

[12] D. Jeltsema and J.M.A. Scherpen. Multidomain modeling of nonlinear networks and systems. Control Systems Magazine, Aug. 2009.

[13] J. Magee and J. Kramer. Concurrency: State Models and Java Programming, Second Edition. Wiley, 2006.

[14] M. Moriconi and R.A. Reimenschneider. Introduction to SADL 1.0: a language for specifying software architecture hierarchies. Technical Report SRI-CSL-97-01, SRI International, 1997.

[15] N. Medvidovic and R.N Taylor. A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering, 26(1):70-93, 2000.

[16] D. P. Gluch P. H. Feiler and J. J. Hudak. The Architecture Analysis and Design language (AADL): An introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, February 2006.

[17]http://csse.usc.edu/csse/event2009/AADL/presentation/09_02_02-SAE_AADL-Simulink.pdf

[18] Simscape from Mathworks. http://www.mathworks.com/products/simscape/

[19] G. Verzichelli. Development of an Aircraft and Landing Gears Model with Steering System in Modelica-Dymola. Modelica 2008.

[20] M. Chkouri, A. Robert, M. Bozga, and J. Sifakis. Translating AADL into BIP - Application to the Verification of Real-time Systems. In Proceeding of Model Based Architecting and Construction of Embedded Systems, 2008.

[21] J. Shi. Combined usage of UML and Simulink in the design of embedded systems: Investigating Scenarios and Structural and Behavioral Mapping. 4th workshop of Object-oriented Modeling of Embedded Real-time Systems, Paderborn, Germany, Oct. 2007.

[22] G. Hoffman, S. Waslander, and C. Tomlin, Quadrotor Helicopter Trajectory Tracking Control. Proc. of the AIAA Guidance, Navigation, and Control Conference, 2008.

[23] T. Johnson. Integrating models and simulations of continuous dynamics into SysML. Proc. of the 6th International Modelica Conference, 2008.

[24] Modelica Association. http://www.modelica.org/

[25] MapleSim from Maplesoft. http://www.maplesoft.com/products/maplesim/

[26] http://eve.enst.fr/aadl/wiki/CaseStudySimulink

[27] C. Hoare. Communicating Sequential Processes. Prentice Hall, 1985.