View Consistency in Architectures for Cyber-Physical Systems

Ajinkya Bhave, Bruce H. Krogh
Dept. of Electrical & Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15217
{jinx, krogh}@ece.cmu.edu

David Garlan, Bradley Schmerl
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15217
{garlan, schmerl}@cs.cmu.edu

***Abstract*—Current methods for modeling, analysis, and design of cyber-physical systems lack a unifying framework due to the complexity and heterogeneity of the constituent elements and their interactions. Our approach is to define relationships between system models at the architectural level, which captures the structural interdependencies and some semantic interdependencies between representations without attempting to comprehend all of the details of any particular modeling formalism. This paper addresses the issue of defining and evaluating consistency between architectural views imposed by various heterogeneous models and a base architecture (BA) for the complete system. This notion of *structural consistency* ensures that the model elements adhere to the cyber and physical types and the connections between components present in the BA, which serves as the unifying framework for model-based development. Consistency checking between a model and the underlying system architecture is formulated as a typed graph matching problem between the connectivity graphs of the corresponding architectural view and the system's BA. The usefulness of the approach to check system modeling assumptions is illustrated in the context of two heterogeneous views of a quadrotor air vehicle.**

***Index Terms*—system architecture; view consistency; graph morphism; multi-domain modeling; cyber-physical systems;**

I. INTRODUCTION

Current methods for modeling, analysis and design of cyber-physical systems (CPSs) lack a unifying framework due to the complexity and heterogeneity of the constituent elements and their interactions. A wide variety of modeling formalisms are used to capture salient features of these systems that are each amenable to specific types of analysis. The relationships among these various representations and the design implications derived from them are usually managed in an ad hoc manner.

Model-based development (MBD) refers to the use of computer-based, executable models to eliminate errors and reduce uncertainty in the process of translating requirements and specifications into working systems. The goal is to reduce costly testing and costly redesign: catching errors in models is significantly cheaper than finding them in the final system or even in prototype implementations. Successful designs rely on separations of concerns based on time scales, interface protocols, imposition of constraints, and other mechanism to facilitate a decomposition of the design problem into manageable and tractable subproblems. The most rigid and pervasive separation in models of CPSs is the distinction between the cyber and physical aspects of a system.

Ensuring consistent relationships between various system models is an important part of the integrated MBD methodology. Our approach is to define relationships between system models at the architectural level, rather than developing a universal modeling language or a meta-modeling framework for translating between models from different formalisms. We believe that an architectural approach provides the right level of abstraction: one that captures the structure of and interdependencies in a system without attempting to comprehend all of the details of any particular modeling formalism. To enable the representation of system dynamics and physical laws in traditional system architectures, we have introduced the CPS architectural style in [19]. The ability to describe both cyber and physical elements in the same architectural framework allows the architect to create a common base architecture (BA) for a CPS. In [4], we have described how a system's BA provides a unified point of reference for multi-domain models and how each system model can be defined as a view of the BA.

In this paper, we address the issue of defining and evaluating consistency between architectural views derived from various heterogeneous models and the BA for the complete system. Such a notion of consistency ensures that the model elements adhere to the communication constraints and physical laws between components present in the base architecture. This guarantees that the models used for design and evaluation are not based on assumptions about information or signal flow pathways between elements that are inconsistent with the underlying complete system design as reflected in the BA. For structurally consistent models, analysis results based on component connectivity in the model are valid for the underlying system as well.

Consistency can be studied between a single model and the underlying system or between multiple models of the same system. Consistency of a single model with the architecture makes it possible to relate (and subsequently use) verification results derived from the model to the final system implementation. This is possible if the final run-time system is generated in a systematic way to guarantee conformance to the architecture. If multiple models describe the same system, then

the models should be based on consistent assumptions about the system's parts, including the parts that are abstracted away. Only then can the different sub-systems designed using these models be integrated, and the final composed system behavior be the same as the behavior expected using the individual model analysis results.

The next section summarizes the concept of architectural views that relate heterogeneous system models to the BA. Section III describes how we transform architectures into typed graphs of components and connectors. Section IV introduces the concept of architectural view consistency and formulates it as a typed graph matching problem. The applicability of view consistency in system design is discussed in Section V, while the tool framework being developed is detailed in Section VI. In Section VII, we illustrate the structural consistency for the control and software views in the context of the STARMAC quadrotor. Section VIII describes related work in this area, and the concluding section discusses ongoing work to extend our approach to semantic multi-domain model consistency for CPSs.

II. Heterogeneous Models as Architectural Views

The approach proposed here focuses specifically on architectural views that represent the architectures of system models as abstractions and refinements of the underlying shared BA. In this context, well-defined mappings between a view and the BA can be used as the basis for identifying and managing the dependencies among the various models and to evaluate mutually constraining design choices. We define our idea of an architectural view (defined in [4]) as follows:

Definition 1. *An architectural view $\mathcal{V}$ for a design perspective $\mathcal{M}$ is a tuple $< \mathcal{C}_V, \mathcal{R}_V^M, \mathcal{R}_{BA}^V >$ where:*

- *$\mathcal{C}_V$ is the component-connector configuration of the view, with the types, semantics, and constraints defined by the design perspective of the view*
- *$\mathcal{R}_V^M$ is a relation that associates elements in the model with elements in $\mathcal{C}_V$*
- *$\mathcal{R}_{BA}^V$ is a relation that associates elements in $\mathcal{C}_V$ with elements in the BA*

Common design views for the embedded control systems domain are the control, software, hardware, and physical views. For example, a control view of a system would contain a set of components and connectors that are relevant to control analysis and design. The control view consists of controller, estimator, sensor, actuator, and sample and hold components, along with the associated ports. The connectors in a control view represent signal flows between the connected components. Similarly, a software view consists of components that represent the software for the control and estimation algorithms implemented in the final system, along with the embedded software communicating with the system's sensors and actuators. Connectors in the software view represent either (buffered or unbuffered) data, or events (or messages) between communicating software components.

An architectural view for a given design perspective abstracts away from the implementation details of the specific tool that the domain model is created in. $\mathcal{R}_V^M$ defines the translation from entities existing in a specific modeling language to the particular view's components and connectors. The relation is either one-to-one or an encapsulation of model entities, as defined by the modeler's choice of grouping. It effectively creates a "componentized" version of the model and allows grouping of multiple elements in the model to a single element in the view.

$\mathcal{R}_{BA}^V$ is an encapsulation/refinement relation, which enables the system architect to group specific components and connectors in the view and map them to subparts of the BA. Some correspondences are declared explicitly by the architect while other correspondences are inferred, based on the semantics of the underlying view perspective. One-to-many (encapsulation) and many-to-one (refinement) maps are allowed. However, many-to-many maps are not allowed since this can lead to inconsistent connections being hidden inside the encapsulated components.

For each view-to-BA relation, $\mathcal{R}_{BA}^V$, certain element mappings are not allowed. For example, a plant component in the control view is not allowed to be mapped to a cyber component in the BA, and vice-versa, since a plant (and its associated model) is a physical domain concept only. However, a controller component in the view can be mapped to a (set of) cyber or physical components in the BA, since the final controller could be implemented either in software or hardware on the final system. Such mapping rules are defined for each view (and hence each modeling perspective) with the help of a domain expert. These rules prevent arbitrary encapsulations and refinements between view and BA entities. The component-connector structures resulting from carrying out element encapsulations on a view and on a BA are called an *encap-view* and an *encap-BA*, respectively.

III. Typed Graph Morphisms

We use undirected, typed graphs to represent the topology of a system architecture. Mapping architectures into typed graphs allows us to leverage well-studied tools in graph theory that evaluate the topological similarity between two structures.

Definition 2. *Let V_G be the set of vertices and E_G be the set of edges. Let Λ and Σ be posets which define the fixed alphabets for the vertex and edge labels respectively. A typed graph G (over Λ and Σ) is a tuple $< V_G, E_G, s_G, t_G, lv_G, le_G >$ where s_G and $t_G : E_G \to V_G$ are the source and target functions, and $vl_G : V_G \to \Lambda$, and $el_G : E_G \to \Sigma$ are the vertex labeling and edge labeling functions, respectively.*

A Component-Connector Graph (CCG) is a typed graph created from an architecture that retains the connectivity constraints of the architectural elements.

Definition 3. *A CCG of an architecture is a typed graph G such that:*

- *V_G is the set of components, connectors, ports, and roles in the architecture.*

- *E_G is the set of edges, which define which ports(roles) are contained in each component(connector) and the attachments between the ports and roles.*
- *Λ is the set of component, connector, port, and role types as defined by the style of the architecture.*
- *$\Sigma = \{contains, attachment, binding\}$ is the set of edge types in the typed graph.*

The types of edges correspond to the following possible connections between architectural entities:

- *contains*: This edge exists between a component(connector) node and its port(role) nodes. The edge represents the fact that each port(role) belongs to a particular component(connector).
- *attachment*: This edge exists between a port and a role. The edge represents a port-role attachment relation in the architecture.
- *binding*: This edge represents a binding relation between an internal port(role) and a boundary port(role). This situation occurs when a component has (one or more) representations or when a set of components or connectors is encapsulated as a single entity.

Figure 1 shows an example of (a) a simple architecture, and (b) the associated typed graph representation.

A typed graph morphism is a structure-preserving correspondence between two typed graphs that maps adjacent nodes in one typed graph to adjacent nodes with consistent types in the other typed graph.

Definition 4. *Let G and H be two typed graphs. Let $v_G \in V_G$ and $e_G \in E_G$ be a vertex and an edge, respectively, of G. A typed graph morphism (from G to H) $\mathcal{M} : G \rightarrow H$ is a pair $< \mathcal{M}_\mathcal{V} : V_G \rightarrow V_H, \mathcal{M}_\mathcal{E} : E_G \rightarrow E_H >$ such that the following properties hold:*

1) $\mathcal{M}_\mathcal{V}(s_G(e_G)) = s_H(\mathcal{M}_\mathcal{E}(e_G))$
2) $\mathcal{M}_\mathcal{V}(t_G(e_G)) = t_H(\mathcal{M}_\mathcal{E}(e_G))$
3) $vl_G(v_G) <= vl_H(\mathcal{M}_\mathcal{V}(v_G))$
4) $el_G(e_G) <= el_H(\mathcal{M}_\mathcal{E}(e_G))$

If $(\mathcal{M}_\mathcal{E}, \mathcal{M}_\mathcal{V})$ are injective (one-to-one), the morphism is known as a *monomorphism*. If they are bijective (one-to-one and onto), the morphism is an *isomorphism*. The cardinality of a morphism is the number of feasible mappings.

The posets associated with the CCGs of the BA and each view enable the architect to define which element types in the BA are semantically compatible with those in a particular view. This graph labeling lets the morphism algorithm detect incompatible mappings of view components to the BA. These definitions and there application are illustrated for the quadrotor application in VII.

IV. Structural Consistency

In this section we answer the question: What does it mean for an architectural view to be structurally consistent with the BA of a cyber-physical system?

Definition 5. *An architectural view $\mathcal{V}$ is structurally consistent with the BA if there exists a typed graph morphism from the CCG of the encap-view to the CCG of the encap-BA, where the typed graph morphism conforms to the element mappings defined by the architect via the relation $\mathcal{R}_{BA}^{V}$.*

As illustrated in Fig. 2, the consistency check between a view and the BA is transformed into a typed graph matching check between the respective CCGs. An architect can choose whether he wants a *weak* or *strong* structural consistency, based on whether the typed graph matching is a monomorphism or an isomorphism, respectively.

Weak consistency enforces that: (i) every component in the view should be accounted for in the BA, and (ii) every communication pathway and physical connection existing between view elements should be allowed in the BA, by the presence of corresponding connectors. As a result, the view (and hence the model) cannot allow incorrect assumptions about the existence of and connectivity between system elements, if this is not defined in the BA.

Strong consistency satisfies all the properties of the weak form. In addition, it imposes the constraint that every element in the BA must be represented in the view in some manner. This implies that the view must take into account every part of the complete system, even if some parts are represented in an abstract fashion. In this sense, the strong consistency is a *completeness* check, rather than a sufficiency check.

View consistency involves combining architectural elements in structurally and semantically meaningful ways. For example, to decide the morphism between the view and BA typed graphs, we have to group heterogeneous elements in both typed graphs, and then carry out the typed graph matching algorithm. These groupings involve encapsulation of components, composition of connectors, and possibly combining ports and roles. Except for component encapsulation in a limited sense, none of the operations are currently formally well-defined for architectures in the CPS domain. Hence, to have a formal notion of structural consistency, such abstraction operations on architectural elements have to be meaningfully defined as well.

The main drawback of graph pattern-matching lies in its inherent computational complexity. The subgraph isomorphism and the monomorphism problem is known to be NP-complete [9]. In fact, a naive graph matching algorithm, which generates each possible mapping from the n nodes in the pattern to the m nodes in the target and tests whether these mappings are graph homomorphisms, requires in the worst case O(m^n) tests [24]. The complexity of this problem usually considerably less for typed graphs since the graph labels eliminate many of the alternatives that need to be evaluated. In our application, the complexity is reduced further by leveraging user-defined correspondences of elements in the view and BA, which creates a partial map from where a typed graph morphism algorithm begin its search. We currently use the VF2 algorithm [14], which uses a computationally efficient heuristic, and is significantly faster than the standard Ullman algorithm in many cases [13]. For subgraph isomorphism with n nodes, the best case complexity of the algorithm is O(n^2) and the worst

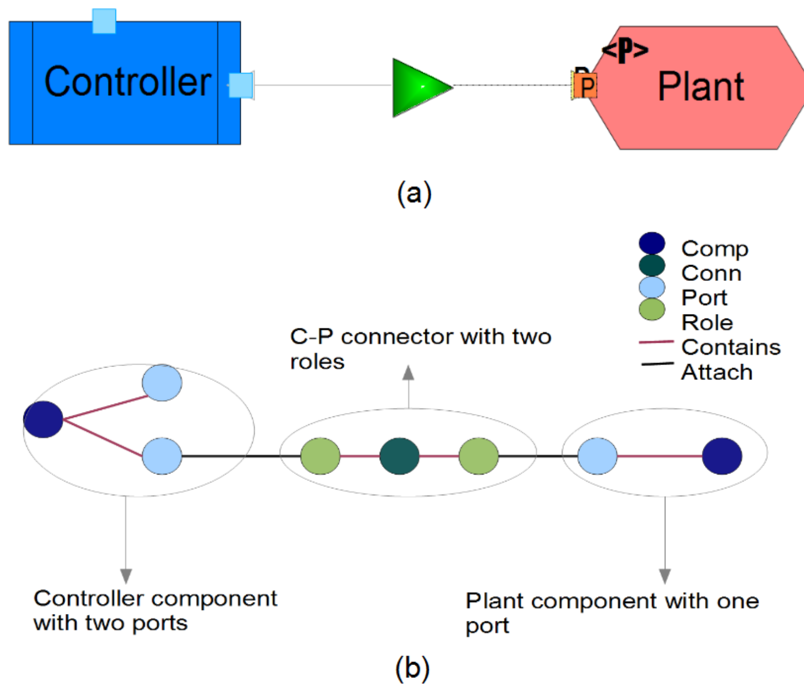

Fig. 1. Example of a Component-Connector Graph.

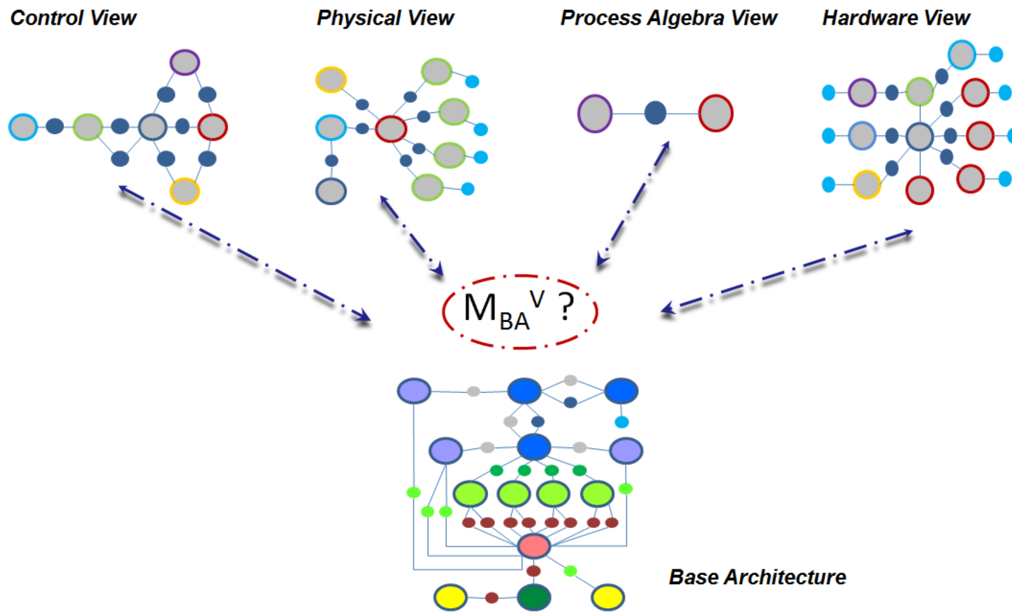

Fig. 2. Defining view consistency as a typed graph morphism check.

case is O($n \times n!$). In addition, VF2 has a memory complexity of O(n), making it useful for working with large graphs.

V. Applicability to System Design

The CPS architecture provides the reference structure for all models used for design and verification. Adherence to the component-port-connector structure of the CPS architecture assures that the structure of each architectural view is consistent with the functional decomposition of the system as represented by the architecture. However, as noted, the architectural views need not have the same structure as the CPS reference architecture. Ports and connectors between components in an architecture view are associated with the ports and connectors in the CPS architecture. These associations can be many-to-one in either direction. The important constraint is that the presence or absence of ports and connectors in either the architectural view or the CPS architecture must be reflected in the other structure. Such a constraint rules out the possibility that a view can introduce a back-door communication channel, not present in the reference structurea property called communication integrity in software architectures [16].

Correspondence between components, ports and connectors in the architectural views and the CPS architecture would be defined by the engineers who construct the verification models. When inconsistencies are detected, that is, when a morphism between the view and the CPS architecture cannot be established, the designer needs to make modifications to bring the architectural view for the verification model into compliance with the CPS architecture.

We can interpret an architectural view as a way of identifying which parts of the complete system are represented in the model, and which parts are abstracted away. From the modeler's perspective, some parts of the BA are "in focus" and some parts are "blurred". The focused parts are the portions where the modeler insists that there be a fine-grained correspondence between the elements in the view and those in the BA. The blurred parts are the portions where this correspondence is coarser. This translates to a fluid notion of consistency between each view and the BA, and across different views of the same system.

This architectural approach is different from traditional approaches to consistency, which are typically defined within the context of a specific modeling formalism. For example, it is common to use bisimulation relations between labeled transition systems to check that the two systems enter equivalent states at all times, for the same input event pattern. Our definition of consistency enforces that each system model makes valid assumptions about the topology of the underlying system, resulting in equivalent component connectivity and physical signal flows. However, it is a "light weight" notion of consistency in that it does not address whether two components will exhibit the same behavior since system behavior cannot be expressed simply as a topological constraint on components. (The concluding section discusses our current research into using the architectural framework to evaluate stronger semantic relationships between models, including consistency with respect to behavioral semantics.)

The BA is assumed to be constructed from validated stakeholder/system requirements. Hence, the BA contains only components and connectors that can be traced back to particular system-level requirements. By enforcing that each view maintain consistency with the BA, we obtain a way to carry out requirements traceability for the corresponding model as well. Checking consistency guarantees models do not contain extraneous elements, or connections between elements, that are not mandated by some system-level requirement. This gives the design team a mechanism to assure that decisions and future changes made at the system architecture level are reflected correctly in the models used for system design and evaluation.

VI. Tool Framework for Consistency Checking

We are extending the AcmeStudio [21] framework to create prototype tools that implement our approach to multi-view consistency. AcmeStudio is a framework for creating architecture design environments. It is written as a plug-in to the Eclipse framework and permits one to define domain-specific architectural styles and link in analysis tools that may be invoked by the user to analyze systems in those styles.

The design flow for our approach is shown in Fig. 3. A newly created *multi-view editor* in AcmeStudio allows the system architect to study the BA and any system view side-by-side. The view can be created in any of the four perspectives currently being implemented to construct architectural views of models, namely, the control, physical, software, and hardware architectural styles. Additional design perspectives can be added by defining the appropriate architectural style for the perspective, along with the rules for permissible mappings between elements of the view and elements of the BA. The view is assumed to have been created independently from an existing system model.

The architect can define the relation $\mathcal{R}_{BA}^{V}$ between the architectural view for a model and the BA by selecting corresponding sets of elements from windows showing the view and the BA. The element mappings are checked against the correspondence rules for that perspective, and any detected violations are reported to the architect. Once a consistent mapping relation has been created, it is stored in a metadata file specific to each view and BA. Any external plug-in can access this file and use the mapping information to carry out view-specific analyses. We are also implementing a *typed graph morphism checker* plug-in that checks view consistency by comparing the CCGs of the view and the BA. The plug-in traverses the BA and the view architecture instances in AcmeStudio, and extracts element connectivity and type information from each of them. It uses this information to create the respective CCGs. The plug-in also reads the element correspondences contained in the view metadata file to carry out required element encapsulations on each CCG. View consistency is checked by verifying if a morphism exists between the two typed graphs. If it does not, then the set

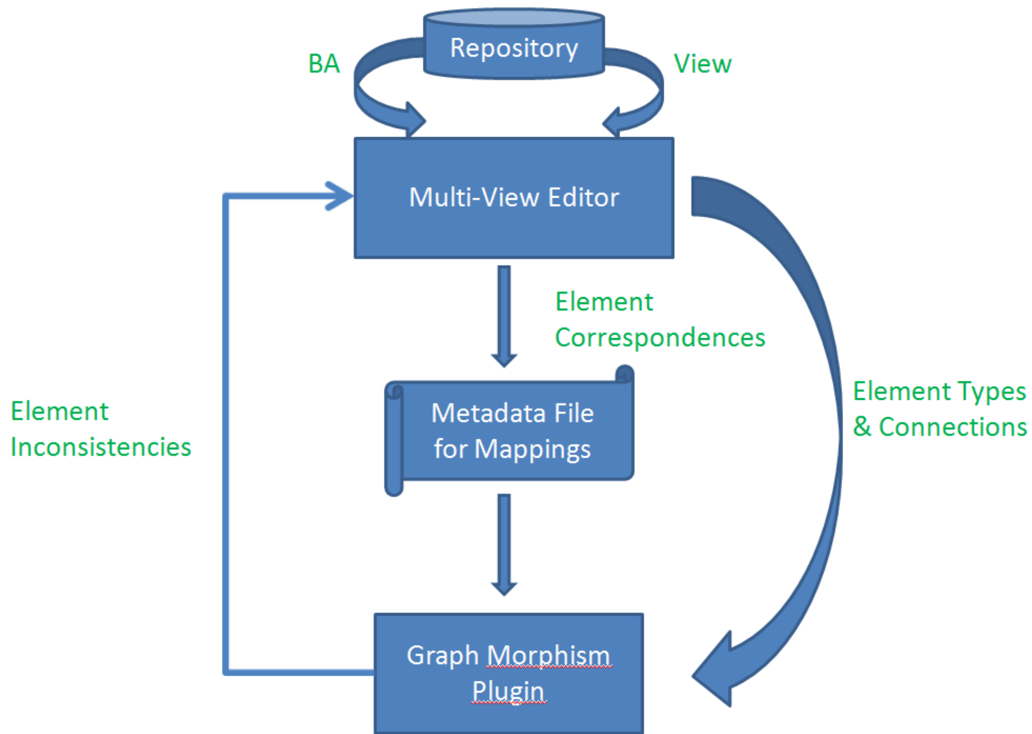

Fig. 3. Design flow using AcmeStudio for view consistency checking.

of maximally matched subgraphs between the view and BA is returned. We have implemented the algorithm for maximally matched subgraphs by extending the VF2 algorithm's local heuristic search criterion.

We are currently studying how best to map the information contained in this set back to BA and view in the multi-view editor. We would like the architect to be able to spot the inconsistent elements and take corrective action based on the returned partial matches.

VII. Quadrotor Architecture: Structural View Consistency

The Stanford Testbed of Autonomous Rotorcraft for Multi-Agent Control (STARMAC) [11] is a quadrotor platform developed to test algorithms that enable autonomous operation of aerial vehicles. The aircraft has four rotors for actuation, arranged symmetrically about its body frame. The vehicle has a sensor suite consisting of an inertial measurement unit (IMU), a Global Positioning System (GPS) unit, and sonar. It implements a hierarchical control system, with a low-level attitude controller (AC) and a high-level position controller (PC). A remote ground station controller (GSC) generates reference trajectories for the quadrotor to follow, and has joysticks for control-augmented manual flight. The two onboard controllers communicate through a serial link. Communications between the PC and the GSC are managed over a WiFi network, using the UDP protocol.

The BA of the quadrotor is modeled in the CPS style, which allowed us to represent both the cyber components (control algorithms and real-time software) and the physical dynamics (forces and torques imparted to the vehicle frame from physical sources). A more detailed description of the complete quadrotor CPS architecture is provided in [5]. The STARMAC design team had documented the software subsystems and the hardware architecture of the vehicle. We modeled the quadrotor physical dynamics in MapleSim from first principles, as well as studying the vehicle dynamics from existing control system models in Simulink. The BA of the quadrotor was created from these documents and models.

The control view was then derived from the existing Simulink models for the complete system. The software view was created by studying the source code (written in C) of the real-time programs that implement the position and attitude controllers, and the communication protocol with onboard sensors of the STARMAC. The rest of this section describes how structural consistency with the BA is checked for the control and software views of the quadrotor.

A. Control View

The Simulink model is functionally correct, i.e the control system achieves attitude and position tracking within the performance requirements. However, the model is not *architecturally consistent*, i.e, it does not respect the connectivity constraints imposed by the BA. There are two sets of

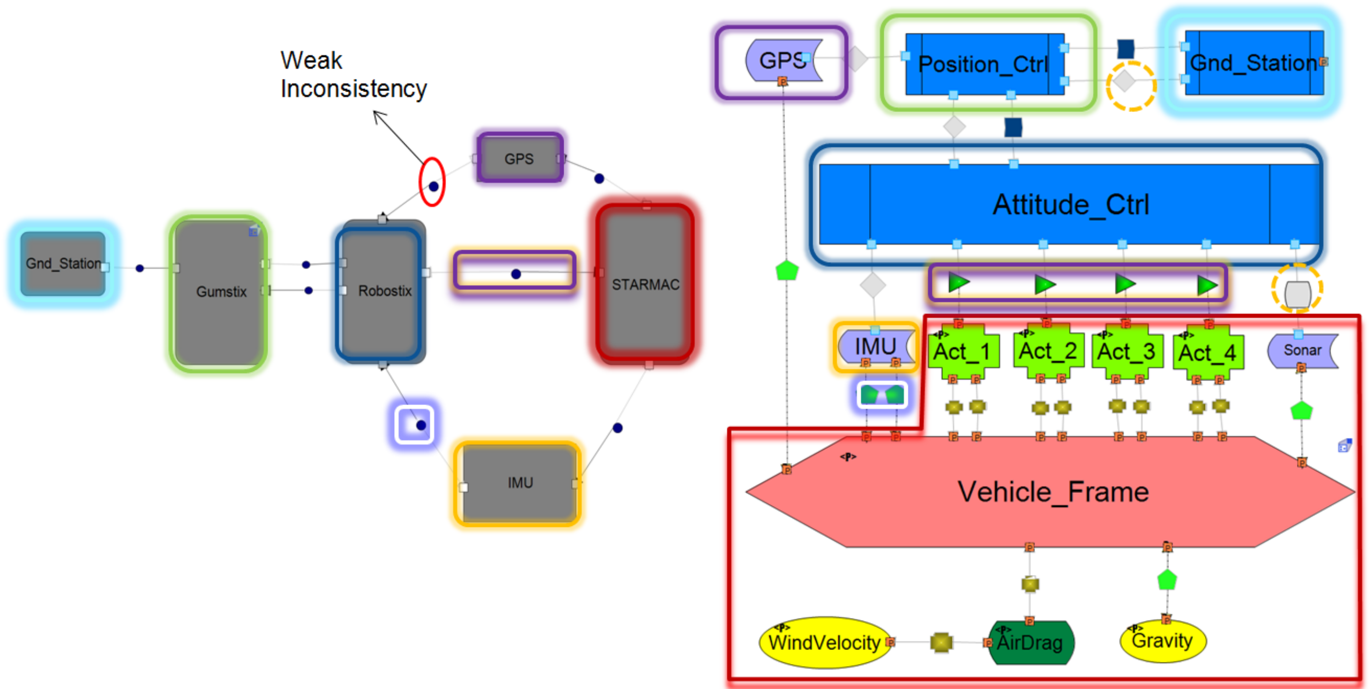

Fig. 4. Consistency between control view and BA.

discrepancies in the Simulink model, which are highlighted in Fig. 4. The first type of discrepancy, marked by a solid circle, is caught by the weak consistency check. It is due to the connection existing in the control view between the sensor output from the GPS and the Robostix. The Robostix then passes these signals to the Gumstix. However, according to the BA topology, the GPS is connected to the position controller directly.

The second set of discrepancies are caught by the strong consistency check, and are highlighted by dotted circles. One discrepancy is because of a connection between the STARMAC block and the Robostix which represents height readings from the sonar sensor. This connection is present in the BA between the Attitude_Ctrl and the encapsulated element containing the VehicleFrame and Sonar components but is missing in the control view. The second discrepancy is because of a data connector from the position controller to the ground station that is present in the BA and missing in the control view. This connector represents telemetry data that the vehicle periodically sends to the ground station for mission status updates.

As a result, there is a mismatch between how the hardware and software components are connected on the physical vehicle and the topology assumed by the Simulink model. Such architecture-level mismatches are caught by the view consistency definition, since the misplaced/missing connector will prevent a CCG morphism between the view and the BA. The reason that the Simulink model works correctly in this instance is because of two reasons. The first is that the Robostix block passes the GPS signals untouched to the Gumstix. The second is that the height readings are obtained from an alternative sensor (the GPS), without using the sonar sensors.

The ramifications of these mismatches could be serious. Suppose, for example, that the stability of the attitude controller (Robostix block) was verified based on the assumption that the GPS signal would be directly available to it. When the final quadrotor system is implemented based on the actual hardware architecture, the attitude controller has no access to the GPS sensor. Hence, the stability results obtained in the control view would not be applicable to the actual system. This is an unintended and potentially dangerous consequence of view inconsistency.

The missing connection from the sonar sensor to the attitude controller can also have serious consequences. Since there is an assumption in the current Simulink model that height readings are only obtained from the GPS, the Robostix block contains an LQG controller based on a linearized version of the quadrotor. However, the Robostix control code implemented manually on the actual vehicle is much more complex, based on an LQG controller for attitude, augmented with a nonlinear sliding mode algorithm for height control [23]. This approach was necessitated because the low-cost sonar sensor suffers from non-Gaussian noise in the form of frequent false echoes and dropouts. The inconsistency between the sensor characteristics assumed in the control view and the sensors being used on the quadrotor leads to an inconsistency between the control algorithm in the design and the implemented controller on the

vehicle.

The missing data connector from the position controller to ground station is an example of a communication between components that is neglected in the control view, since it does not contribute to the functional correctness of the control algorithm. However, it has an impact of the specification of the bandwidth and the quality of the wireless link between the vehicle and the ground station, as well as on the execution time of the position controller component.

Structural consistency can be applied in a hierarchical manner between a view and the BA. For example, suppose that the consistency check succeeds between the control view and the BA. The architect can further check if the internal structure of the Robostix controller in the Simulink model is consistent with the internal architecture (called *representation* in AcmeStudio) of the Attitude_Ctrl in the BA. We assume, of course, that both these components do have an internal structure present. We can follow the same procedure as for the view-BA case. A new 'view' is created from the internal elements of the Robostix and the 'baseline architecture' now becomes the representation of Attitude_Ctrl. A consistency check can be carried out between the CCG of the 'view' and the CCG of the 'baseline architecture'. In this way, the consistency check can be invoked in a top-down manner to check consistency between elements (and sub-elements) of the view and the BA.

B. Software View

The FSP model of the quadrotor studies the concurrent behavior of the position and attitude controller and the sensors they access. The position controller and ground station are together modeled as the POS_CTRL FSP process which sends setpoint events (with some jitter in the sending time) to the attitude controller, while concurrently reading data from the GPS_SENSOR process. The ATT_CTRL process receives a setpoint event, reads data from the IMU_SENSOR process, then computes and sends the actuator command for each motor, in a single loop. Each sensor is modeled as a primitive FSP process that sends readings at a fixed rate, in terms of the number of *tick* events of a local clock. The current attitude controller software on the quadrotor has a safety check that stops all the motors if either (1) the IMU stops sending data or (2) no packet is received from the position controller within a fixed interval of time. To keep the attitude controller's timer from being reset, the position controller periodically sends a 'heartbeat' packet to it. This protocol is modeled by forcing POS_CTRL to send one *heartbeat* event after a fixed number of tick events have elapsed. If ATT_CTRL does not receive this event within a defined interval of time, it will enter into the reset state and generate a *stopMotors* event.

The FSP model is constructed to check properties such as: for a position controller that does and does not implement the heartbeat mechanism, is there any sequence of events that lead to the *stopMotors* command being issued by the attitude controller. This FSP specification is analyzed by the Labeled Transition System Analyser [15] tool. The analysis tells us that the property is violated if a POS_CTRL is used, which does not implement the heartbeat protocol.

To create the software view for the FSP model, the POS_CTRL and ATT_CTRL processes are each represented by a *Controller* software component. The SENSOR processes are each mapped to a *SensorAdapter* component, which represents the embedded software/firmware running on each sensor device. The events being shared between processes are modeled as either event or data connectors, depending on the semantics of the data communication. The communication of the heartbeat packet is modeled as a data connector between the position and attitude controllers. This data communication assumption is critical to the safe operation of the actual, implemented system. However, it does not appear in the control view (or the hardware or physical view either), since the control algorithm assumes ideal communication between the two controllers.

As an illustration of the graph labeling posets, consider the software view of the quadrotor (Fig. 5), where a particular connector specifies a communication protocol between control components, and hence has a *data* or *event* type associated with it. The architect can define that such a connector be mapped only to a *send-receive* or *publish-subscribe* connector type in the BA, which are both mechanisms for components to communicate. However, the software connector should not be mapped to a mechanical connector (or any physical connector) in the BA. This gives the architect a mechanism to associate semantically meaningful entities between the two system structures being compared.

Figure 5 also shows the mapping between the software view and the BA. A weak consistency check fails since the heartbeat data connector has not been currently modeled in the BA. Once the BA is modified to include this connector between the position and attitude controller, the consistency check with the control view will fail, since the heartbeat connector does not match any of the existing connectors in the control model. This discrepancy can make the system designer aware of this critical communication assumption implemented in the software of the quadrotor.

VIII. Related Work

Multiple efforts have focused on supporting multi-view, model-based system development. The field of computer automated multiparadigm modeling (CAMPaM) is introduced in [17], and the current issues that need to be addressed are outlined, along with a survey of promising approaches.

The Vanderbilt model-based prototyping toolchain provides an integrated framework for embedded control system design [18]. It provides support for multiple views, such as functional Simulink/Stateflow models, software architecture, and hardware platform modeling along with deployment. The toolchain's ESMoL language has a time-triggered semantics, which restricts the functional view to Simulink blocks that can only execute periodically. There is currently no support for additional views (e.g., physical or verification models), nor a notion of consistency between additional system views.

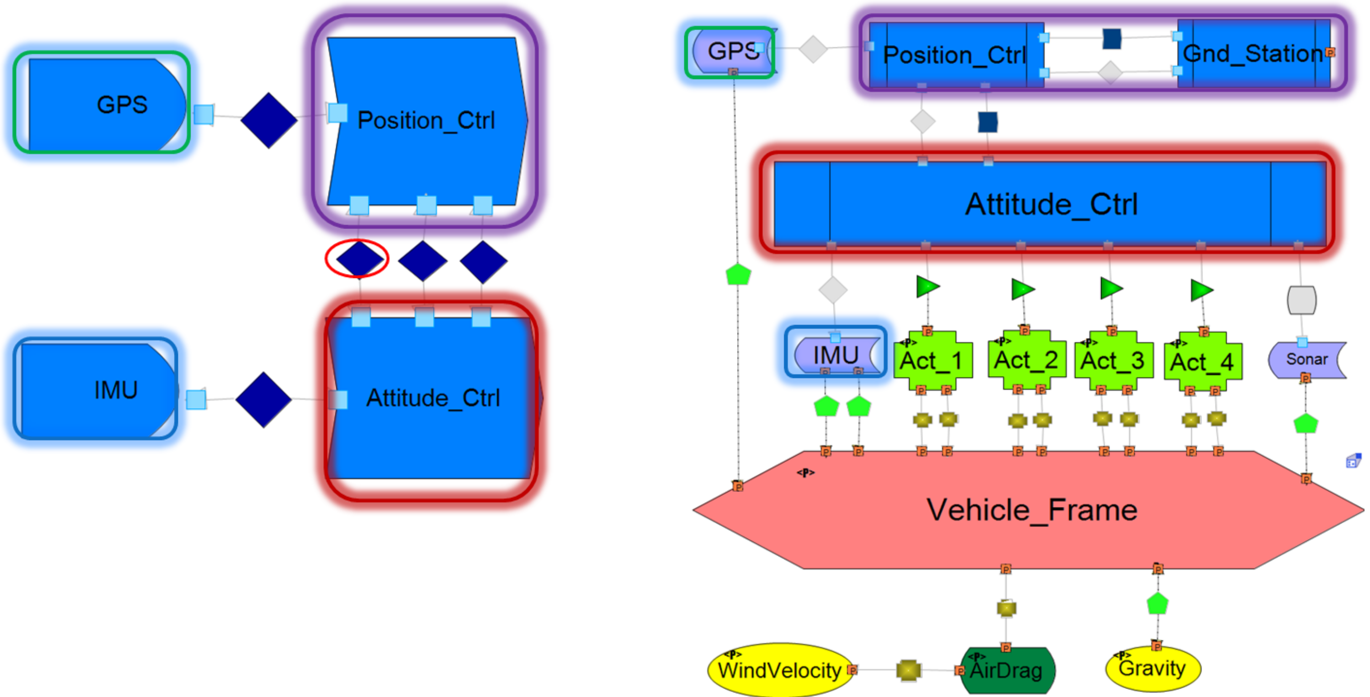

Fig. 5. Consistency between software view and BA.

The approach in semantic anchoring [7] to transform between system models concentrates on the specification of the dynamic semantics of domain-specific modeling languages. The method is based on the observation that a broad category of component behaviors can be represented by a small set of basic behavioral abstractions such as Finite State Machines (FSMs), Timed Automata and others. The underlying assumption is that the behavior of these abstractions is well understood and precisely defined. The methodology and toolchain are described in [6]. In contrast, our work focuses on architecture-level view comparison, not on meta-modeling or model transformations.

The ModelicaML profile is an attempt to integrate UML and Modelica for modeling and simulation of system requirements and design [20]. Similar work has been done for UML and Simulink [22]. SysML is a UML 2.0 profile, specialized for systems engineering applications [1]. There is an ongoing effort to integrate Modelica with SysML for physical domain modeling [12]. However, in all these approaches, there does not exist an easy way to incorporate physical dynamical models into the overall framework. For example, SysML flow ports do not have a well-defined semantics to model flows of physical quantities (energy or torque). In addition, there are no consistent rules or guidelines on how to define relationships between multiple views in any of these frameworks.

The EAST-ADD [4] is a UML2.0/SysML-based ADD for automotive embedded software and systems. Its goal is to integrate system information on different abstraction levels and for different automotive application domains. EAST-ADD is an information structure that can be used within a tool and for model exchange but is not a tool itself. The focus of the language is on software design, with support for hardware modeling. Current aspects covered include system structure, behavior, requirements, and to some extent environment modeling. However, it has no support for the types of view consistency that are addressed in our approach.

In [10], the approach is to create a multi-view integration layer that consists of a set of support tools to facilitate the coupling between views, a data management layer for the storage of inter-view relations and shared information and support tools that facilitate combined code-generation. However, currently the only two formalisms supported seem to be 20-Sim and gCSP.

Ptolemy II is a tool that enables the hierarchical integration of multiple “models of computation” in a single system, based on an actor-oriented design [3]. Actors are software or hardware modules that communicate with each other through timed events [2]. Even though Ptolemy II supports hierarchy and incorporation of multiple formalisms at the detailed simulation level, it is not possible to define architectural styles or high-level design tradeoffs. Ptolemy is primarily a simulation tool, rather than an architecture description framework.

SysWeaver [8] is a model-based development tool that includes a flexible code generation scheme for distributed real-time systems. The functional aspects of the system are specified in Simulink and translated into a SysWeaver model

to be enhanced with timing information, the target hardware model and its communication dependencies. The translation from Simulink is not completely automated if closed-loop controllers are present. Sysweaver's computational framework semantics is restricted to tasks that exchange information via message-passing (time or event-based). There is also no support in SysWeaver for a physical plant modeling view.

IX. Discussion

This paper presents a new tool for evaluating the consistency between the base architecture for a cyber-physical system and the heterogeneous models used for the design of such complex systems. Consistency is defined in terms of the relationships between the structure of the models represented as architectural views of the base architecture constructed in a CPS architectural style that includes both the cyber and physical elements of the complete system. This notion of structural consistency assures that models are topologically consistent with the system-level design, and also semantically consistent in terms of the types of components and connectors used in the model architecture. Extensions to the Acmestudio tool for architectural description languages support this multi-view support with a plug-in to evaluate inter-view consistency through typed graph morphisms.

Structural consistency is a lightweight notion of consistency to assures designers are not making inappropriate assumptions about the system structure and basic component and connector semantics. We are currently extending this architectural approach to address a richer set of consistency issues that arise in multi-model development. One direction of research concerns the introduction of architectural annotations that represent the assumptions made in the construction of models that reflect system-level assumptions incorporated into annotations the base architecture. This will provide a means of verifying the consistency between assumptions made in different models, and a framework for evaluating the coverage of analysis models relative to system-level assumptions. We are also investigating annotations to capture the verification results generated from different model formalisms and a logical framework for integrating these results to verify system-level properties.

Acknowledgment

This work is supported in part by National Science Foundation (NSF) under grant nos. CNS0834701 and CNS1035800, and by Air Force Office of Scientific Research (AFOSR) under contract no. FA9550-06-1-0312.

References

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

[2] G. Agha. Concurrent object-oriented programming. *Commun. ACM*, 33(9):125–141, 1990.

[3] S. S. Bhattacharyya, E. Cheong, and I. Davis. Ptolemy II heterogeneous concurrent modeling and design in java. Technical report, 2003.

[4] A. Bhave, D. Garlan, B. Krogh, and B.Schmerl. Multi-domain modeling of cyber-physical systems using architectural views. In *Proc. of the Embedded Real Time Software and Systems Conf. (ERTS^2 2010)*, 30 Nov. 2010.

[5] A. Bhave, D. Garlan, B. Krogh, A. Rajhans, and B.Schmerl. Augmenting software architectures with physical components. In *First Analytic Virtual Integration of Cyber-Physical Systems Workshop (RTSS 2010)*, 19-21 May 2010.

[6] K. Chen. *A Semantic Anchoring Infrastructure for Model-Integrated Computing*. PhD thesis, Vanderbilt University, 2006.

[7] K. Chen, J. Sztipanovits, and S. Abdelwahed. Toward a semantic anchoring infrastructure for domain-specific modeling languages. In *5th ACM InternationalConference on Embedded Software*, volume September, 2005.

[8] D. de Niz, G. Bhatia, and R. Rajkumar. Model-based development of embedded systems: The Sysweaver approach. *IEEE Real Time Technology and Applications Symposium*, pages 231–242, 2006.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to NP-Completeness.* Freeman, 1979.

[10] M. Groothuis and J. Broenink. Multi-view methodology for the design of embedded mechatronic control systems. In *IEEE International Symposium on Computer Aided Control Systems Conference*, pages 416–421, 2006.

[11] G. Hoffman, S. Waslander, and C. Tomlin. Quadrotor helicopter trajectory tracking control. In *Proc. of the AIAA Guidance, Navigation, and Control Conference*, 2008.

[12] T. Johnson, C. J. J. Paredis, and R. M. Burkhart. Integrating models and simulations of continuous dynamics into sysml. In *6th International Modelica Conference*, pages 135–145. Modelica Association, 2008.

[13] C. S. L. P. Cordella, P. Foggia and M. Vento. Performance evaluation of the VF graph matching algorithm. In *Int. Conf. Image Analysis and Processing*, pages 1172–1177, 1999.

[14] C. S. L. P. Cordella, P. Foggia and M. Vento. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 Workshop Graph-Based Representations in Pattern Recognition*, pages 149–159, 2001.

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

[16] M. Moriconi, X. Qian, and R. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):356372, 1995.

[17] P. J. Mosterman and H. Vangheluwe. Computer automated multi-paradigm modeling in control system design. *IEEE Transactions on Control System Technology*, 12:65–70, 2000.

[18] J. Porter, P. Volgyesi, N.Kottenstette, H.Nine, G.Karsai, and J. Sztipanovits. An experimental model-based rapid prototyping environment for high-confidence embedded software. In *RSP '09: Proceedings of the 2009 IEEE/IFIP International Symposium on Rapid System Prototyping*, pages 3–10, Washington, DC, USA, 2009. IEEE Computer Society.

[19] 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.

[20] W. Schamai, P. Fritzson, C. Paredis, and A. Pop. Towards a unified system modeling and simulation with ModelicaML: Modeling of executable behavior using graphical notations. In *7th International Modelica Conference*, 2009.

[21] B. Schmerl and D. Garlan. Acmestudio: Supporting style-centered architecture development. In *In Proceedings of the 26th International Conference on Software Engineering*, Edinburgh, May 2004.

[22] J. Shi. Combined usage of UML and Simulink in the design of embedded systems: Investigating scenarios and structural and behavioral mapping. In *4th workshop of Object-oriented Modeling of Embedded Realtime Systems*, 2007.

[23] S. Waslander, G. Hoffmann, J. Jang, and C. Tomlin. Multi-agent quadrotor testbed control design: Integral sliding mode vs. reinforcement learning. In *In Proceedings of the IEEE/RSJ International Conference on Intelligent Robotics and Systems 2005*, pages 468–473, 2005.

[24] A. Zündorf. Graph pattern-matching in progres. In *5th Int. Workshop on Graph Grammars and their Application to Computer Science*, Lecture Notes in Computer Science, pages 454–468. Springer-Verlag, 1996.