Multi-View Consistency in Architectures for Cyber-Physical Systems

Submitted in partial fulfillment of the requirements for
the degree of
Doctor of Philosophy
in
Department of Electrical and Computer Engineering

Ajinkya Y. Bhave

B.E., Computer Engineering, Mumbai University
M.S., Robotics, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

December 2011

Keywords: Cyber-Physical Systems, embedded, control, heterogeneous models, consistency

Dedicated to my loving family, who laughed with me through all good times, stood steadfastly by me through all bad times, and who always had unwavering faith in me, even when I sometimes lost that faith in myself.

Abstract

Today's complex cyber-physical systems (CPSs) are created using models throughout the system development life cycle, a process referred to as model-based design (MBD). The heterogeneity of elements in CPSs requires multiple perspectives and formalisms to explore the complete design space. Ensuring the consistency of these various system models is an important part of the integrated MBD approach.

In this thesis, we propose to unify heterogeneous system models through light-weight representations of their structure and semantics using architectural descriptions. Architectures are annotated structural representations that describe systems at a high level of abstraction, allowing designers to determine appropriate assignment of functionality to elements, and make trade-offs between different quality attributes. There are two fundamental shortcomings of current architecture modeling capabilities that limit their potential to fully address the engineering problems of large-scale, heterogeneous CPSs: (i) limited vocabulary to represent physical elements and their interactions; and (ii) inadequate ways to support consistency relations between heterogeneous architecture views of the same system.

This thesis addresses the first shortcoming through the development of the *CPS architectural style* that supports a unified representation of both physical and cyber elements and their interactions in the same framework. This ability allows the architect to create a common base architecture (BA) for a CPS that provides a unified point of reference for multi-domain system models. To address the second shortcoming, the *architectural view* is used as the mechanism to represent the architectures of system models as abstractions of the underlying shared BA. In this context, well-defined mappings between a view and the BA are used to identify and manage semantically equivalent elements (and their relations) between each model and the underlying system.

Structural consistency defines when an architectural view conforms to the structural and semantic constraints imposed by components and connectors in the system's BA. Such a notion of consistency ensures that the model elements adhere to the connectivity constraints and physical laws present between elements in the BA. We define view consistency as the

existence of an appropriate morphism between the typed graphs of a view and the BA. Depending on the type of morphism present, two notions of consistency are defined: *view conformance* and *view completeness*.

Our tool framework is implemented in the AcmeStudio architecture design framework, and consists of a view map language, a graphical view editor, and a set of graph morphism algorithms for consistency checking. We illustrate the application of our architectural approach with two case studies: an autonomous quadrotor with heterogeneous legacy models, and management of model variants in simulation environments for engine control of vehicles.

Acknowledgments

I thank my family for their love and support throughout my Ph.D., and their unwavering belief in me. I express my deep gratitude to my advisor, Bruce Krogh, from whom I learnt the discipline, analytical approach, and clear and precise thinking that now enable me to face new problems and domains with confidence. Bruce strikes the right balance between motivating and helping his students, and allowing them to pursue ideas independently.

I am indebted to David Garlan, who has played the role of a co-advisor, helping me to bridge the gap between the worlds of control engineering and software architecture. It was the detailed discussions I had with David, throughout my proposal and defence phases, that clarified many critical parts of the approach in this thesis. I thank Philip Koopman, who made me understand the practical side of my research, including the hurdles that must be overcome to have acceptance in the industry. Thank you to Dionisio, who graciously accepted to be on my committee at the last minute, and who has played an important part with his valuable advice and feedback on my work. I also thank Bradley Schmerl, who has been the key resource person I went to anytime I needed help with all things related to AcmeStudio, ADLs, iThings, or understanding the Aussie accent.

Raj Reddy and Pradeep Khosla have been a source of knowledge, sagacious advice, and support during my Masters and Ph.D. days at Carnegie Mellon. Because of their presence, this University has always felt more like family and a home away from home.

Thanks to Paulo, Sabina, and Joya for being the last people standing (sitting) for my defence and providing me moral support during the presentation. Thanks also to all the Porter Hall B-ites, especially Luca, Akshay, Rohan, Matthias, Sergio, JY, Jim, and Juhua, who made life interesting, fun, and bearable during the Ph.D. years. Claire Bauerle and Carolyn Patterson make Porter Hall always seem welcoming with their helpful and caring attitudes. Similarly, Elaine and the ECE Grad office make every student's academic life seamless and smooth because of their conscientious approach towards our needs.

A special thanks to my best friend Parth, who stood by me an ocean away in India, reminding me: "Ajinkya, Life is always funny. It's just that, sometimes we forget to laugh with it." Another special thanks to Sujata, who came into my life towards the end of this

long race, but who gave me the strength to complete the final sprint past the finish line.

This work is supported in part by the National Science Foundation (NSF) under grant nos. CNS0834701 and CNS1035800, and by the Air Force Office of Scientific Research (AFOSR) under contract no. FA9550-06-1-0312. I am grateful to these sponsors for their financial support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The term cyber-physical system (CPS) refers to the tight conjoining of and coordination between computational and physical resources [4]. Today's complex CPSs are created using models throughout the system development life cycle, a process referred to as model-based design (MBD) [5]. Models allow designers from different disciplines to share knowledge, facilitate design comprehension, and assess system-level trade-offs. Each representation highlights certain features and occludes others to make analysis tractable and to focus on particular performance attributes. Typically a particular modeling formalism represents either the cyber or the physical elements well, but not both. For example, differential equation models represent physical processes well, but do not represent naturally the details of computation or data communication. On the other hand, discrete formalisms such as process algebras and automata are well suited for representing concurrent behavior and control flow, but are not particularly useful for modeling continuous phenomena in the physical world. Thus, the heterogeneity of elements in CPSs requires multiple perspectives and formalisms to explore the complete design space. Ensuring the consistency of these various system models is an important part of the integrated MBD approach.

There is a need for a representation at some level that encompasses the complete system and is not prejudiced towards the cyber or physical side. Such a representation would serve as a unified point of reference for the variety of more domain-specific models and

also as a framework for exploring design trade-offs across the cyber-physical boundary. We believe that the architectural representation of a system provides the right level of abstraction to define relationships between the various heterogeneous system models. Architectures are annotated structural representations that describe systems at a high level of abstraction, allowing designers to determine appropriate assignment of functionality to elements, evaluate the compatibility of the parts, and make trade-offs between different quality attributes such as performance, reliability, and maintainability. The approach in this thesis is to reason about the consistency of system models at the architectural level, rather than developing a universal modeling language or a meta-modeling framework for translating models between different formalisms.

1.1 Why Architectural Consistency of Models

It is an established fact that detecting defects early in the project life cycle can have huge cost and time savings. A number of studies have shown that current development processes allow 70% of faults to be introduced early in the design, while 80% of them are not caught until integration, testing, or later with a repair cost of 16 times or higher [6, 7]. These findings are summarized in Fig. 1.1 (taken from [1]) which illustrates the percentages for fault introduction, discovery, and cost factors for fault removal at various stages of the design cycle. If we can discover a reasonable percentage of these late system-level faults earlier in the development process, it is reasonable to expect considerable cost savings. A return on investment analysis that focuses on cost avoidance due to early detection of design defects by using the architecture-centric SAVI methodology predicts a nominal cost reduction for a 27 Million Source Lines Of Code (MSLOC) avionics system as $2,391M, while the most conservative estimate is $717M for the same system [8]. Hence, this thesis focuses on checking model consistency at the architectural level because of the potential of architectural analysis to expose design flaws early in the project life cycle [1].

Industrial practice has shown that independently developed models in a typical MBD

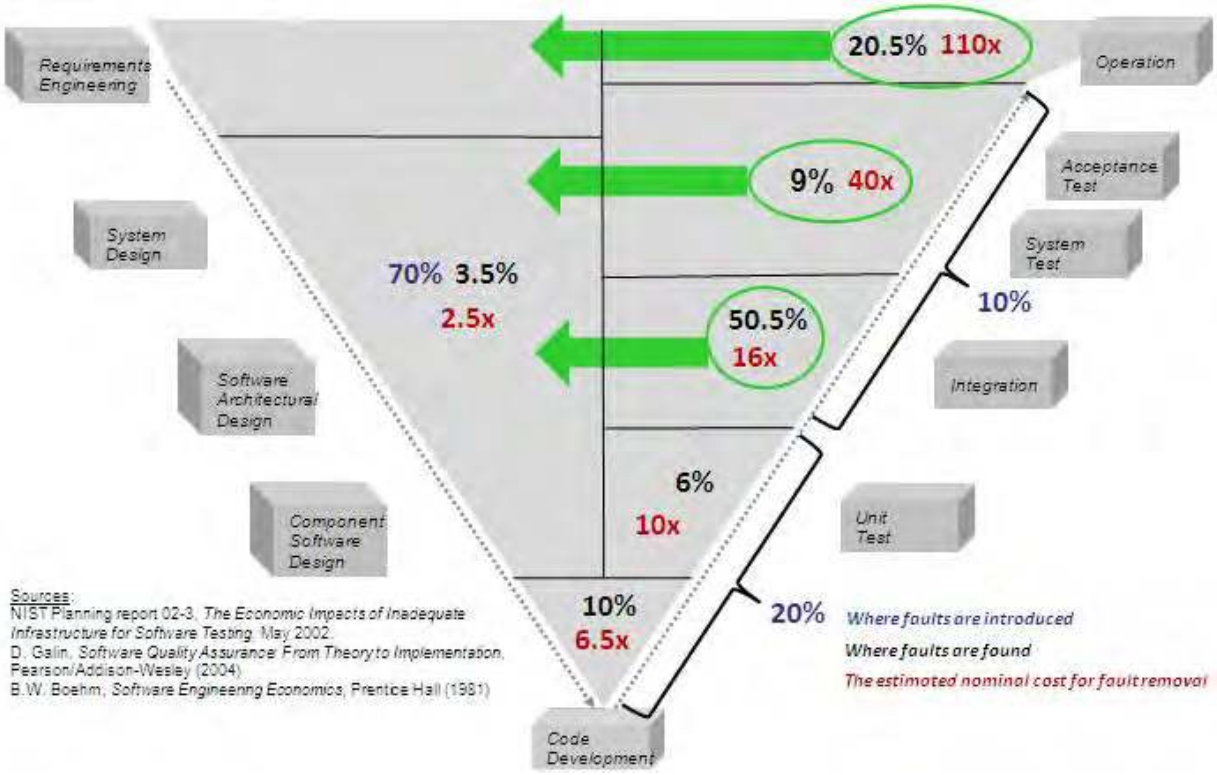

Figure 1.1: Benefits of early discovery of faults in system design cycle [1].

approach tend to result in multiple versions of the same system. We illustrate some types of inconsistencies that can arise in different models of the same system through two case studies. The first one describes how inconsistent assumptions about the system structure can be made in different models for an avionics system (details in Chap. 6). The second one describes the necessity of maintaining consistent architecture between variants of controller and plant models for testing of engine controllers (details in Chap. 7).

Mismatched assumptions in different models due to multiple system versions are the main source of the introduction of faults during the design phase [9]. The lack of a unifying framework to compare models early and throughout the life cycle leads to faults being detected late, causing a large rise in costs for fault removal. Missing mechanisms to enforce consistency of models limits the ability to use the results of model-level analyses to derive system-level properties. Hence, there is a need for an architecture-centric approach that ensures that analytical models are consistent with each other and the evolving architecture throughout the design process.

Based on this discussion, we summarize the areas of system design that are affected by the lack of an architectural approach for CPSs with heterogeneous models:

- The number of system-level faults introduced during the design phase.
- The number of system-level faults detected during the design phase.
- The cost of fault removal during the complete life cycle.
- Limited ability to use analyses from different models to derive system-level properties because of multiple versions of the underlying system.

To address these problems, the approach in this thesis is to unify heterogeneous system models through light-weight representations of their structure and semantics using architectural descriptions.

1.2 Contributions of Thesis

There are two fundamental shortcomings of current architecture modeling capabilities that limit their potential to fully address the engineering problems of large-scale, heterogeneous CPSs: (i) limited vocabulary to represent physical elements and their interactions; and (ii) inadequate ways to support consistency relations between heterogeneous architecture views of the same system. The first limitation prevents the creation of a complete architectural representation of the system that includes the description of the physical plant(s) being controlled by the embedded controller software. The second limitation makes it difficult to share and maintain consistent information between system models that are created and analyzed in different design concerns.

This thesis addresses the first shortcoming through the development of an architectural style that serves as a common representation of the complete system, and the second shortcoming through the abstraction of architectural views to compare the structure and semantics of corresponding heterogeneous models to the CPS architecture. In particular, we make the following contributions:

1. **Extending architecture to represent continuous dynamical physical elements and their coupling.** We have created a CPS architectural style that supports a unified representation of both physical and cyber elements and their interactions in the same architectural framework. This ability allows the architect to create a common base architecture (BA) for a CPS that provides a unified point of reference for multi-domain system models.

2. **Using architecture as the common system representation to relate the structure and semantics of heterogeneous models.** The architectural view is used as the mechanism to 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 are used to identify and manage semantically equivalent

elements (and their relations) between each model and the underlying system.

3. **Defining and evaluating consistency between architectural views and the system's BA.** View consistency defines when an architectural view conforms to the structural and semantic constraints imposed by components and connectors in the system's BA. Such a notion of consistency ensures that the model elements adhere to the connectivity constraints and physical laws present between elements in the BA. This guarantees that the models used for design and evaluation are not based on assumptions about the system's design that are inconsistent with the actual system as reflected in the BA. We define view consistency as the existence of an appropriate morphism between the typed graphs of a view and the BA. Depending on the type of morphism present, two notions of consistency are defined: *view conformance* and *view completeness*.

4. **Tools for automated consistency checking of architectural views.** The first tool is a mechanism (Acme Maps) to define the types of maps possible between each view type and the BA as well as map instances between the elements of a view and the BA . Acme Maps is an extension of the core Acme Architecture Description Language (ADL) that allows the creation and type checking of view relations and element correspondences. The second tool is a graphical editor to compare views visually, define element correspondences (including encapsulations) that must be maintained, and display the results of consistency checking to the user. The third tool is a set of graph morphism algorithms that find the largest set of semantically consistent element mappings between a view and the BA, based on the pre-mapped elements and the type compatibility defined by the user. All tools are implemented as plugins in the AcmeStudio architecture design framework [10], so that they can be extended easily for future enhancements.

5. **Evaluation of multi-view architecture framework.** We illustrate the applica-

tion of our architectural approach with two case studies. The first one demonstrates how heterogeneous models of an avionics system (the STARMAC quadrotor) can be created as views of the baseline architecture. The choice of the modeling domains is motivated by the analysis and verification activities typically found in an embedded control system design process. We apply the consistency check to each view and highlight the mismatches that are detected between the models and the actual implemented system. The second case study illustrates the usefulness of architectural views to manage model variants for *X-In-the-Loop Simulation* (XILS) environments for engine control of vehicles. We show how each simulation scenario can be captured as a view of the system under test, and how view consistency can help the engineer in checking consistency of controller-plant models between various XILS environments.

1.3 Organization of Thesis

The next chapter surveys the related work in the areas of multiple architectural views, physical system modeling, and MBD toolchains. Chapter 3 describes how we extend architecture models to incorporate physical elements and laws, using an approach based on the semantics of acausal interconnection of conjugate physical variables. In Chapter 4, architectural views for heterogeneous models are formally introduced, along with their relations to the model and BA. Chapter 5 defines the notion of view consistency, discusses its application to system design, and formulates consistency checking as a typed graph morphism problem. In Chapter 6, the first of two case studies is presented to illustrate the application of our approach to real-world systems. The study focuses on the consistency of heterogeneous views of a quadrotor vehicle. The second case study, presented in Chapter 7, describes the usefulness of architectural views to manage model variants for XILS environments for engine control of vehicles. We present conclusions of this thesis and discuss future work in Chapter 8.

Chapter 2

Related Work

Multiple efforts have focused on supporting multi-view, model-based system development. In this chapter, we give a brief outline of the development of this work in the areas of architectural approaches, physical modeling tools, and integrated frameworks.

2.1 Architectural Approaches

An architectural description consists of multiple, possibly heterogeneous *architectural views* [11]. In the architecture community, there is a common understanding of what a view is, with several seminal works defining this concept. In [12] the need for views is outlined, while in [13] a view model with a fixed number of commonly used heterogeneous views is introduced. Multiple views and their formal descriptions form the basis for architecture documentation in the book "*Documenting Software Architectures: Views and Beyond*" (DSA) [11], and in [14]. The concepts of architectural description and architectural view have been standardized by the ANSI/IEEE 1471 standard [15], which is identical in content to the ISO/IEC 42010 standard [16]. Unfortunately, existing methods do not formally define the relationships between views. This represents a problem for architectural modeling, since it is critical to understand how design decisions or analyses in one view impact those of another.

The ISO 42010 conceptual model introduces and relates concepts such as architectural

description, concern, viewpoint, view, and model. According to the standard, a viewpoint is a way of looking at an architecture; the view is the result of looking at a specific system's architecture in this way. The standard also incorporates the concept of a viewtype (introduced in DSA) which "defines the element types and relation types used to describe the architecture of a software system from a particular perspective". The ISO 42010 standard defines a single view for each viewpoint, and multiple *architectural models* contained in each view. A view relation in the ISO 42010 standard is called a *correspondence* with the semantics defined by an associated *correspondence type*.

We can relate our architectural approach to the ISO 42010 model as follows. We use a component-connector viewtype to describe the run-time characteristics of the complete system, with multiple viewpoints contained in it. The viewpoints are defined by the different design concerns present in a typical CPS design flow. Each viewpoint has an associated architectural style, which defines the element types, semantics, and constraints that are relevant for the design concern [17]. For each viewpoint, there are one or more views that relate system models created for that design concern to the system's BA. The BA is also considered to be a view, created in a viewpoint with the CPS style as the associated architectural style.

A detailed survey together with a taxonomy and classification of view relation mechanisms is found in [18]. This taxonomy has been used as a baseline for classifying correspondence types in ISO 42010. Based on this taxonomy, relations can be of the *intra-model* or *inter-model* type. Intra-model relations are relations between the same type of architectural models. Since our view relations define correspondences between two component-connector architectures, they are intra-model relations. The level of detail that our relations support is *fine-grained*, since structural correspondences down to the level of component ports and semantic constraints over properties associated with elements can be specified using these view relations. According to the taxonomy, we use *direct references* for individual element

correspondences, *tuples* for element encapsulations, and an *expression language* in the form of first-order predicates to define constraints over element properties. The current usage of our view relations falls under the *consistency checking* category.

Some ADLs have mechanisms for mapping between specific views. For example, Meta-H provides a mechanism for mapping between software and hardware views [19], while SADL allows one to characterize mappings between architecture refinement levels [20]. Each of these capabilities is tied to the semantics of particular kinds of views and to the semantics of a particular ADL. In contrast, our approach considers the general problem of structural mapping between architectural views and introduces a mechanism for creating view relations between sets of elements.

UML is a standardized general-purpose modeling language for object-oriented software engineering [21]. The language is managed by the Object Management Group and has become the industry standard for modeling software-intensive systems. UML includes a set of diagrams to create visual models of the structural and behavioral aspects of software systems. The semantics for UML diagrams is ambiguous and there is no formal notion of relations between different views of a system. UML is used to describe the software aspects of systems only, and has no support for representing physical dynamics.

SysML is a UML 2.0 profile, specialized for systems engineering applications [22]. SysML has additional diagrams (e.g. parametric, requirement) which allow modeling of hardware, software, information, processes, personnel, and facilities. The aim of SysML is to be a general language for systems architectures, but the semantics for consistency between views is not defined formally. Additionally, there is currently no support for describing physical architectures, although work on integrating dynamical equations for elements is ongoing (described in Sec. 2.2).

AADL (Architecture Analysis and Design Language) is an international standard for predictable model-based engineering of real-time and embedded computer systems [23].

AADL offers a set of predefined component categories to represent real-time systems and it is capable of describing functional component interfaces like data and control flows, as well as non-functional aspects of components like timing properties. Preliminary work has been done on integrating aspect-oriented programming constructs into AADL [24]. Architectural descriptions can be extended with *annexes*, such as the *behavior annex* that allows a state machine description of component implementations. However, AADL does not support architectural representation of physical domain entities (except as generic 'device' components), nor does it address how heterogeneous views can reconciled.

Egyed et al. [25] exploit redundancies between different types of UML diagrams to ensure consistency between the views. The approach is limited to UML viewtypes and there is no mechanism for incorporating other system viewtypes. Our approach has support for the same types of view relations between component-connector architecture models, in addition to support for physical element descriptions.

Boucké et al. [26] focus on composing several structural views in the xADL language and define three view relations as part of the approach. The *unification* and *mapping* relations in their approach correspond exactly to our *one-to-one* and *encapsulation* relations. In contrast to our approach, there is no notion of consistency with the underlying system, since composition is done only between views. In addition, depending on which xADL structural views are being composed, the view relations do not deal with any types associated with the architectural elements. This is different from our approach, where element types play a central role in defining element mappings between views.

Radjenovic et al. [27] present an approach called AIM (Architectural Information Modelling) for view consistency. The AIM framework uses a relational database as the core data model for all views and is comprised of three layers - data, rules and views. The relations between different views are defined in the rules layer and can be constraint-based or structural. The structural relations in AIM are similar to our view relations. One limitation

of the approach is the need to translate any modeling language into an AIM-compatible language for view comparison. In addition, AIM focuses on software-centric views only and has no concept of physical architectures or models.

The System Architecture Virtual Integration (SAVI) project of the Aerospace Vehicle Systems Institute is an architecture-centric approach to the analysis of virtually integrated system models with respect to multiple operational quality attributes such as performance, safety, and reliability [1]. SAVI aims to use a multi-aspect reference architecture model as the single source of truth, a model bus for consistent model interchange between repositories and tools, and a component-based framework. Currently SAVI is the second phase of development, hence the representations for the reference architecture and model bus are still not formalized, to the best of our knowledge.

Our approach aligns closely with the SAVI framework. The concept of a base architecture in our approach parallels the reference architecture model, while the concept of an architectural view for a model is similar to the "model bus" and model translators in SAVI. Whereas our approach currently focuses on component-connector architecture descriptions, the long-term goal of SAVI is to be able to incorporate different architectural viewpoints and languages into the same framework.

NAOMI is an experimental platform for enabling multiple heterogeneous models to work together [28]. In NAOMI, a model is defined in terms of the set of input and output attributes that it shares with the system. NAOMI checks consistency for a set of models based on: (1) whether all attributes are updated, (2) if the current values of all attributes do not violate any model-specific constraints, and (3) whether the modification times of all models are earlier than the modification times of their output attributes. In contrast to our approach, there is no concept of an underlying system architecture model in NAOMI. As a result, NAOMI does not define what it means for each model to be individually consistent with the system, nor is there is a mechanism to define physical architectural elements.

2.2 Physical Modeling

There are also a number of tools for modeling and simulating physical systems. 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.

Modelica is a popular object-oriented, open-standard language for constructing component-based models of physical systems [29]. 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 [30]. MathWorks has introduced Simscape, a MATLAB-based modeling language and Simulink blockset that makes it possible to integrate physical models with control-oriented simulations [31]. All these languages are created for detailed simulations of the physical plant, and have minimal support for formal architectural representations of the system. They do not have mechanisms to formally analyze the software aspects of the design, including thread scheduling, timing, and concurrency checking.

The ModelicaML profile is an attempt to integrate UML and Modelica for modeling and simulation of system requirements and design [32]. Similar work has been done for UML and Simulink [33]. There is an ongoing effort to integrate Modelica with SysML for physical domain modeling [34]. 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 (e.g., energy or torque). Defining explicit behavior for flow transmission is up to the modeler. In contrast, the physical entities in our framework are based on the behavioral approach [35], where the semantics of component behavior and interconnections is well defined in terms of conjugate variables for each physical domain.

2.3 Integrated Frameworks

There are tool platforms that offer an integrated solution based on the MBD methodology. These tools have support for hardware/software co-design, model management and model translation between specific domains, and generation of component-level or system-level code for targeted application platforms.

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 [36]. Actors are software or hardware modules that communicate with each other through timed events [37]. 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. In addition, there is no support for acausal, equation-based modeling of physical systems, since the underlying formalism is event-based communication.

The Vanderbilt model-based prototyping toolchain provides an integrated framework for embedded control system design [38]. It provides support for multiple views, such as functional Simulink/Stateflow models, software architecture, and hardware platform modeling along with deployment. The views are created using meta-models of the associated viewpoints and model transformations for translating between specific pairs of views. The toolchain's ESMoL language has a time-triggered semantics, which restricts the functional view to Simulink blocks that can only execute periodically. ESMoL does not support the full semantics of Simulink. In ESMoL the execution of Simulink data flow blocks is restricted to periodic discrete time, consistent with the underlying time-triggered platform. This also restricts the type and configuration of blocks that may be used in a design. There is currently no support for additional views (e.g., physical or verification models), nor a notion of consistency between additional system views. In contrast, our work focuses on architecture-level view comparison, not on meta-modeling or model transformations.

SysWeaver [39] is a model-based development tool that includes a flexible code gen-

eration scheme for distributed real-time systems. The complete toolchain requires a simulation tool (Simulink) and its code generator (MATLAB Embedded Coder), along with SysWeaver. 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 uses the concepts of *semantic dimensions* for the separation of the functional and para-functional aspects of the system, and *couplers* to enable the hierarchical decomposition of these aspects. Since the focus of the tool is on the synthesis of a fully deployable system, there is currently no mechanism for relating models from different formalisms. In particular, there is no support for a physical plant modeling view.

AUTOSAR (AUTomotive Open System ARchitecture) is an open and standardized automotive software architecture framework, jointly developed by automobile manufacturers, suppliers and tool developers. It defines a component-based layered architecture that contains abstractions for the ECU, actuators, and sensors of the vehicle, as well as descriptions of the hardware platform and network topology. The focus of AUTOSAR is on the description and generation of in-vehicle software. Hence, there is limited support for physical modeling of the plant or new architectural viewpoints that are not part of the existing standard.

Chapter 3

Augmenting Architecture with Physical Elements

This chapter describes elements of an architectural style that can serve as a base representation for the CPS domain. The challenge in defining an architectural style for cyber-physical systems is to strike a balance between specificity and generality. Our goal is to provide a representative set of components and connectors that can serve as the foundation for application-specific styles in targeted frameworks. Towards this end, we define three related architectural styles pertaining to the cyber domain, the physical domain, and their interconnection.

3.1 Component-Connector Architectures

Although there is considerable diversity in the capabilities of different ADLs, all of them have support for architectural descriptions based on a *component-connector* representation of systems [40]. In this thesis, the term *architecture* is synonymous with a component-connector architecture, since we focus on analyzing properties and behavior of elements of the run-time system. We use Acme [2] as the ADL to create all our architectural models, because of its built-in support for the following features of component-connector architectures:

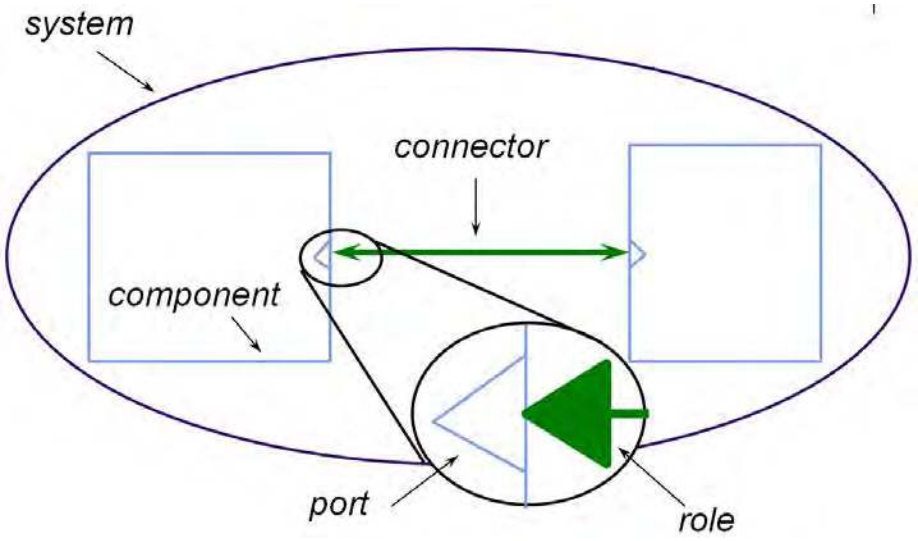

Figure 3.1: Elements of component-connector architectures [2].

- *Components*: These represent the primary computational elements and data stores of a system. Intuitively, they correspond to the boxes in box-and-line descriptions of system architectures. Typical examples of components include clients, servers, filters, and databases. A component can have multiple interfaces, each of which is termed a *port*. A port identifies a point of interaction between the component and its environment, and can represent an interface as simple as a single procedure signature. Alternatively, a port can define a more complex interface, such as a collection of procedure calls that must be invoked in a specific order.

- *Connectors*: These represent interactions among components. Connectors are used to describe the communication and coordination activities among the connected components. Intuitively, they correspond to the lines in box-and-line descriptions. Examples include simple forms of interaction, such as pipes, procedure call, and event broadcast. But connectors may also represent more complex interactions, such as a client-server protocol or a SQL link between a database and an application. Connectors also can have multiple interfaces, each of which is termed a *role*. Each role defines a participant of the interaction represented by the connector. Binary connec-

tors have two roles such as the reading and writing roles of a pipe, or the sender and receiver roles of a message passing connector. Other kinds of connectors may have more than two roles. For example, an event broadcast connector might have a single event-announcer role and an arbitrary number of event-receiver roles

- *Systems*: A system (or configuration) is a graph that defines how a set of components are connected to each other via connectors. The graph is defined by associating component ports with the connector roles in which they participate. For example, ports of filter components are associated with roles of the pipe connectors through which they read and write streams of data. Systems may also be hierarchical: components and connectors may represent subsystems that have internal architectures.

- *Properties*: In addition to defining high-level structure, Acme supports the annotation of architectural elements with an arbitrary list of properties. Properties contain information about a system or its parts that allows analysis about overall behavior (both functional and para-functional). For example, for an architecture whose components represent periodic tasks, properties would define the period, priority, and CPU usage of each component. Properties of connectors would include latency, throughput, reliability, and protocol of interaction.

- *Constraints*: These represent claims about an architectural design that should remain true as it evolves over time. Typical constraints include restrictions on allowable values of properties, topology, and design vocabulary. For example, an architecture might constrain its design so that the number of clients of a particular server is less than some maximum value.

- *Styles*: represent families of architectures for a particular domain. An architectural style typically defines a vocabulary of design element types and rules for composing them [17]. Examples include data architectures based on pipes and filters, and the NASA Mission Data System style to model space rovers for Mars [41, 42]. An

architectural style in Acme is called a *family*.

We use these architectural concepts to define a set of representative styles for the cyber domain, physical domain, and the interfaces between these domains in the following sections. Together, these three styles form the complete CPS architectural style.

3.2 Cyber Style

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

- *Computation*: These components represent the implemented software functionality provided by the system. They include components that perform filtering, state estimation, and control.
- *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.
- *IO Interfaces*: These components perform the computations and timing functions required to interface with hardware devices, including software for user interfaces. This would include, for example, device drivers that process raw sensor data for higher-level components.

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 the following major types of cyber connectors:

- *Send-Receive*: This represents a one-to-one data communication between two computational or IO components. The communication mechanism could be event-based, synchronous or asynchronous, and is specified by the type of role defined for each

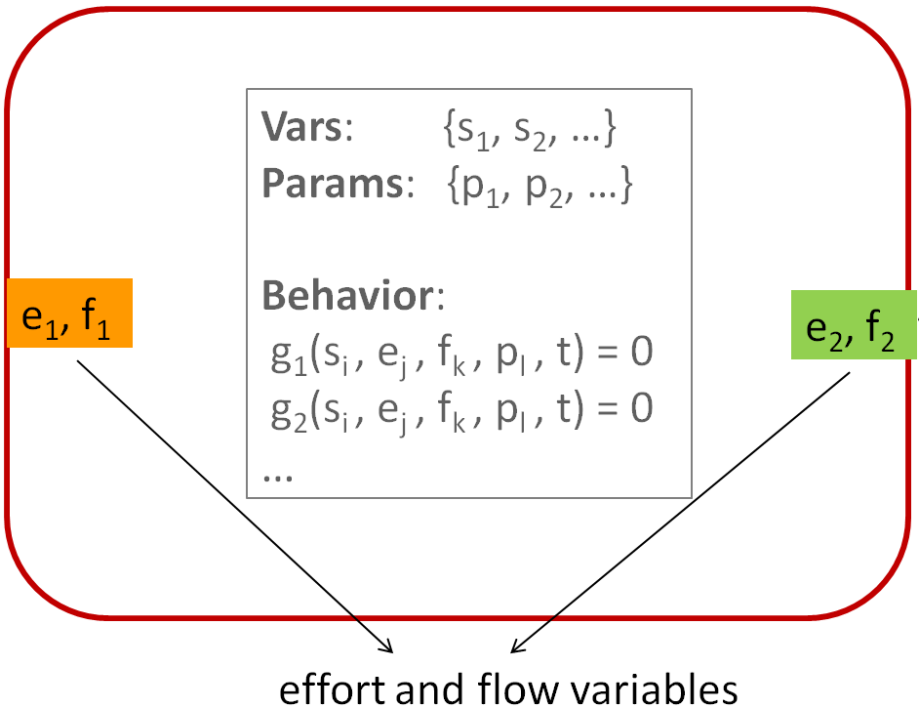

Figure 3.2: Physical component with ports containing effort and flow variables.

connector.

- *Publish-subscribe*: This represents data communication between a set of computational components or IO interfaces, in which one component generates data that is used by the other connected components.

Each connector type can be refined further, based on the communication semantics of the interacting components, as described in [43]. For example, the send-receive connector could be elaborated to define communication between CORBA components by annotating the roles with the interaction protocol for CORBA.

3.3 Physical Style

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 a *behavioral view of open and interconnected physical systems*, as defined by J.C Willems [35]. 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 energy conservation laws. This is similar to the perspectives taken in bond graphs [44] and Lagrangian mechanics [45], where power-conjugated variables (effort and flow) describe energy flows between sources, storage elements, and dissipative elements. The behavioral perspective is more general than bond graphs however, and imposes more than just energy-based descriptions of components and their interactions. In the behavioral approach, laws that govern physical phenomena impose relations on a component's variables, while interconnection means that variables are shared between the connected components i.e component behaviors are coupled via their common variables.

The physical style is used to model architectures for multi-domain physical systems. A physical component is specified by its *type*, and its *behavior*, as shown in Fig. 3.2. The type identifies the physical domain that the component represents. Each physical port defines the effort and flow variables with which the component interacts with the environment. The component's behavior defines the relationships between its port variables (and any internal state variables, if defined). In other words, it is a specification of what time trajectories are possible for the set of port variables. A behavior can be described, for example, as the solution set of a differential equation or through a transfer function. The behavior is annotated as a component property.

A physical connector (along with its roles) defines the relationship between the effort and flow pairs of all the components that are attached to the connector. When two or more components from the physical style are connected, the implication is that their behaviors

are coupled through their shared port variables. If the ports obey certain properties, then the product of the port's *power conjugate* variable pair equals generalized power in the physical domain of interest [35, p. 68-69]. In such cases, 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. The effort and flow variables for commonly encountered physical domains are given in Table 3.1.

Table 3.1: Effort and flow variables.

Domain	Effort	Flow
Electrical	Voltage [V]	Current[A]
Translational Mechanical	Position [m]	Force [N]
Rotational Mechanical	Angular Position [rad]	Torque [N-m]
Thermal	Temperature [° C]	Heat Flow [watt]
Fluid	Pressure [Pa]	Mass Flow [kg/s]

The physical component types are:

- *Source*: These components deliver constant effort (or flow) to other components, regardless of the load present. This is analogous to an ideal voltage (or current) source. Because of the defined direction of flow variables, such components will have negative power flow as long as they are supplying power to other components.
- *Sink*: These components can consume an arbitrary amount of flow from other components, while maintaining a constant effort. This is analogous to an electrical ground or a large external volume of air that can maintain its temperature, regardless of the heat flowing into it. Because of the defined direction of flow variables, such components will have positive power flow.
- *Energy storage*: These components model dynamic elements or subsystems that store energy, such as components that have capacitive and inductive properties in electri-

cal systems. The set of ports on these components allow power transfer to other subsystems.

- *Dissipative*: These components 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. 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 transducer*: These components 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. Transducers can also represent transformations between system variables of the same physical domain, such as coordinate frame transformations between two rigid bodies in the mechanical domain.
- *Subsystem*: These components model physical elements with complex behavior, that cannot be not modeled by source, storage, dissipative, or transducer components alone. Subsystems can also be used to encapsulate primitive components and connectors to create hierarchies of physical elements.

The physical connector types are:

- *Physical coupling*: The constraints on the effort and flow 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 Kirchhoff's laws in the electrical domain, and force/moment balance laws in the mechanical domain, for coupled components.

- *Physical signal*: These connectors indicate effort and/or flow 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).

3.4 Cyber-Physical Interface Style

We define the Cyber-Physical Interface (CPI) style to bridge between the cyber and the physical worlds. The elements represent connections between computational and physical systems, as well as transformation of information between the two domains:

- *CPI components*: cyber-to-physical (C2P) and physical-to-cyber (P2C) transducers;
- *CPI connectors*: cyber-to-physical (C2P) and physical-to-cyber (P2C) 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.

3.5 Refinement to Specific Physical Domains

Together, the three generic styles described above can be combined and extended to provide a unified representation of a CPS. The component and connector types in the physical style are used to define new styles with features and attributes specific to the physical domain of interest. 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 the electrical style, two components connected by an electrical coupling connector are shown in Fig. 3.3 (a). An electrical physical port has voltage and current as the conjugate variables. The voltage across the component and the current through it (along with either charge or flux as the stored quantity) are defined in terms of the conjugate variables at the ports. The electrical connector enforces Kirchhoff's laws

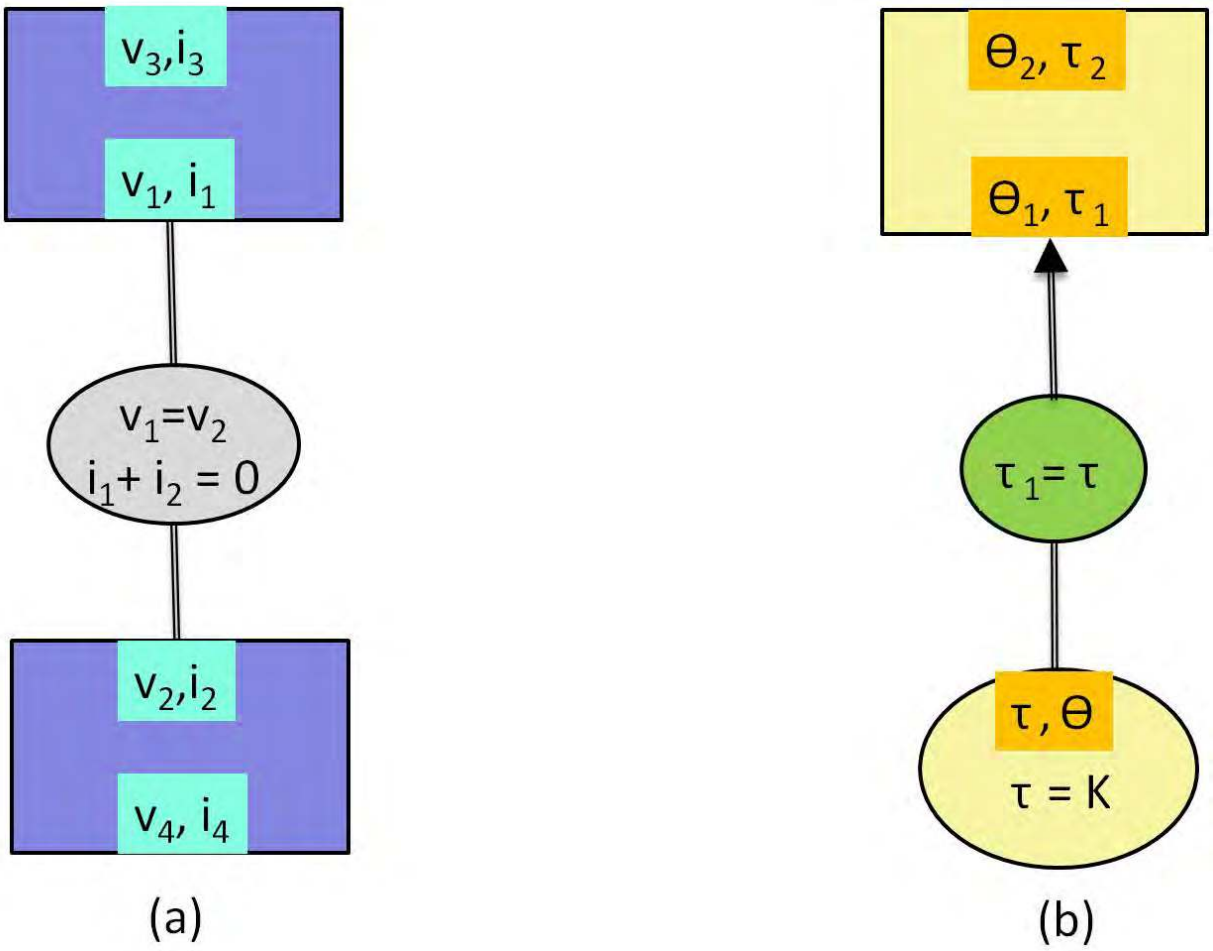

Figure 3.3: Physical connectors for the electrical and mechanical domains.

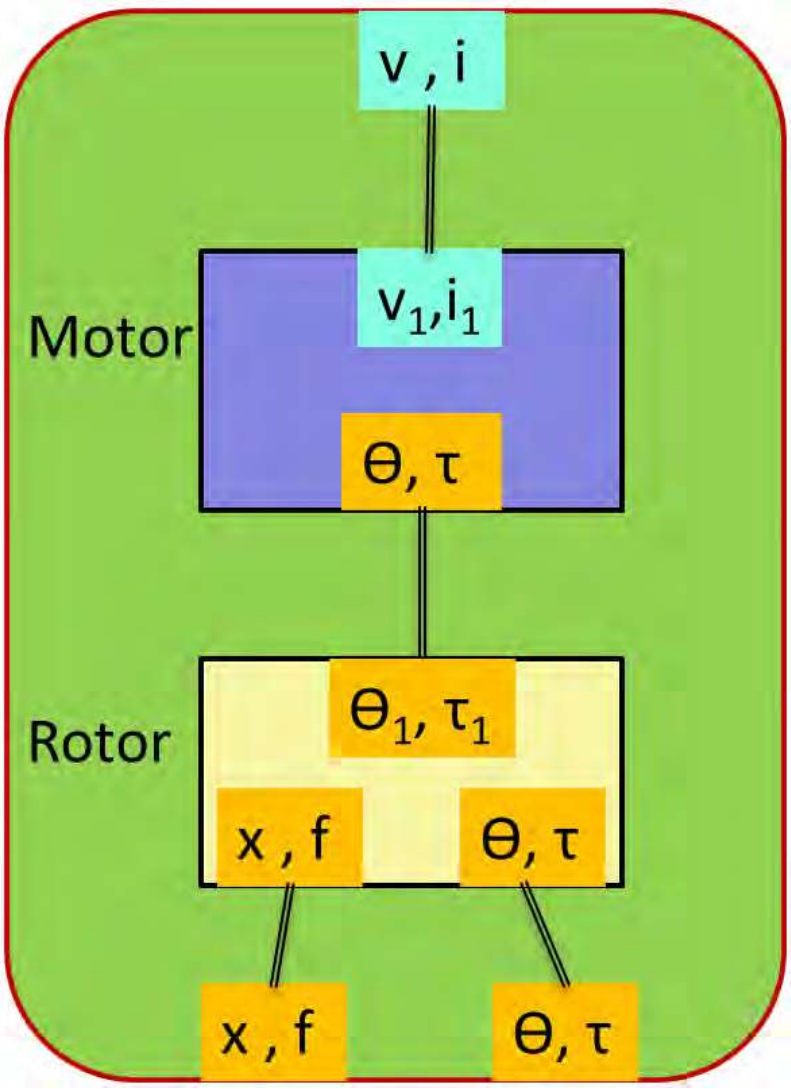

Figure 3.4: Electro-mechanical actuator combining components from electrical and mechanical domains.

between the voltage and current variables on the ports of the connected components.

For the mechanical style, two components connected by a mechanical signal connector are shown in Fig. 3.3 (b). The connector represents the transmission of constant torque of K Newton-meter from a mechanical source component to a mechanical effort storage component. The effort and flow variables on each mechanical component's ports are the torque and angular position being related by the connector. Similarly, a mechanical coupling connector defines a physical coupling between two or more components which constrains them to move together in the given frame of reference, while the forces at the joint sum to zero.

An electro-mechanical actuator is shown in Fig. 3.4. This component is an example of an architectural element that combines multiple physical domains and it is used to model the rotor assembly of the STARMAC quadrotor in Chap. 6. The *Motor* is a physical transducer component that converts the voltage at its electrical input port into a proportional torque at its mechanical output port. The *Rotor* is a mechanical component that converts the torque at its input port into forces and torques acting on the vehicle frame at its two output ports. The dynamics of each component are annotated as part of the element's properties.

For the mechanical domain, mass and moment of inertia (MI) correspond to energy storage elements because 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 using the mechanical subsystem component. A rigid body contains both mass (annotated with the center of gravity (CG) coordinates) and MI as subcomponents, as well as body coordinate frames and frame transformations as behavioral properties. Mechanical dissipative components reflect phenomena where mechanical energy is lost over time, such as static friction and viscous damping.

3.6 Tool Framework for CPS Architectures

We have created the CPS style as an Acme *family*, using Acme's built-in support for flexibly defining and extending architectural types. The basic building block for defining

architectural styles in Acme is a type system that can be used to encapsulate recurring structures and relationships. An architect can define three kinds of types: property types, structural types, and architectural styles. Property types are defined using a set of built-in primitive types (including integer, string, and boolean), and type constructors for records, sets, and lists. Structural types make it possible to define types of components, connectors, ports, and roles. Each such type provides a type name and a list of required substructure, properties, and constraints (called *design rules*). Acme uses a constraint language based on first order predicate logic (similar to UML's Object Constraint Language) to define design rules. Constraints may be specified in one of two ways: as an *invariant* or a *heuristic*. An invariant is a rule that cannot be violated. A heuristic is a rule that should be observed, but may be selectively violated.

```
Port Type PhysicalPortT extends BasePortT with {
        Property effort : string;
        Property flow : string;
        Property effort_units: string;
        Property flow_units: string;
        invariant forall r in self.attachedRoles |
            declaresType(r, PhysicalRoleT);
    }
Component Type PhysicalCompT extends BaseCompT with {
        Property behavior: string;
        invariant forall p in self.PORTS |
            declaresType(p, PhysicalPortT);
}
```

Figure 3.5: Acme specification for physical port and component type

The Acme definition of a physical port type and physical component type (that form part of the physical family) is shown in Fig 3.5. The port type specifies that any port that is an instance of `PhysicalPortT` must contain properties for effort and flow variables and their units. The invariant associated with the type requires that all roles attached to a physical port must be of type `PhysicaRoleT`. To simplify specifications of constraints Acme

also provides a number of built-in functions. For example, `self.attachedRoles` returns the set of roles attached to the port represented by `self`, where the term `self` refers to the entity to which the constraint is associated. The `declaresType(e,T)` function returns true if an element `e` is declared to have type `T`.

`PhysicalCompT` extends the `BaseCompT` base type. The property `behavior` is a generic string and allows the specification of component behavior to be annotated to the element. The specification could be, for example, the filename of a Modelica object that implements the component. The invariant associated with the type requires that all ports of a physical component must be of the type `PhysicalPortT`, where `self.PORTS` returns the ports attached to the component represented by `self`.

There are analogous Acme specifications for physical connectors and physical roles. We have created component and connector types for all the physical elements described in Sec. 3.3, as well as for the complete cyber family and the CPI family. The creation of the three styles as Acme families allows a designer to extend them for each new CPS application as required. We have extended the physical family to create the following Acme families:

- *Mechanical translational*: to incorporate various forces acting on the vehicle frame in the physical architecture of the quadrotor.
- *Mechanical rotational*: to model the torques from the rotors in the physical architecture of the quadrotor.
- *Electrical*: to model the electrical and electro-mechanical aspects of the physical architecture of the quadrotor.
- *Thermo-Fluid*: to model the architecture of the intake and exhaust systems of the engine in the XILS case study.

An example of extending the physical family to the mechanical and electrical domain is shown in Fig. 3.6. The `MechanicalPortT` extends the physical port by specifying position and force as the effort and flow variables for the mechanical translational domain, along

```
Port Type MechanicalPortT extends PhysicalPortT with {
    Property effort = "position";
    Property flow   = "force";
    Property effort_units = "meter";
    Property flow_units = "Newton";
    invariant forall r in self.attachedRoles |
        declaresType(r, MechanicalRoleT);
}
Component Type MechanicalCompT extends PhysicalCompT with{
    invariant forall p in self.PORTS |
        declaresType(p, MechanicalPortT);
}
Port Type ElectricalPortT extends PhysicalPortT with {
    Property effort = "voltage";
    Property flow   = "current";
    Property effort_units = "Volt";
    Property flow_units = "Ampere";
    invariant forall r in self.attachedRoles |
        declaresType(r, ElectricalRoleT);
}
Component Type ElectricalCompT extends PhysicalCompT with{
    invariant forall p in self.PORTS |
        declaresType(p, ElectricalPortT);
}
```

Figure 3.6: Acme specification for the mechanical and electrical element types

with their units. The associated invariant requires that mechanical ports should only be attached to mechanical roles. Similarly, the `MechanicalCompT` component type extends the physical component type, and can only contain mechanical ports. An analogous Acme specification exists for the electrical element types. The specification of such Acme families allows type-checking of elements and their interconnections for an architecture containing entities from multiple physical domains.

AcmeStudio is a customizable graphical editing environment and visualization tool for architectural design based on the Acme ADL, created by Garlan et al [10]. The tool allows the architect to define new Acme families and customize the environment to work with those families by defining visualization styles. AcmeStudio also has a built-in editor

and parser for the Acme ADL. In Fig. 3.7, the BA of the quadrotor has been created as an instance of the CPS family in AcmeStudio. The palette on the right contains all the element types that are present in the CPS family. The designer can choose element instances and connect them together by dragging and dropping items from the palette onto the canvas. The *Properties* pane below the Acme system displays information relevant to the architectural element selected, including its name, type hierarchy, associated rules, and annotated properties. All our Acme families and systems have been created in the AcmeStudio design environment.

3.7 Summary

In this chapter, we describe elements of an architectural style that can serve as a comprehensive representation for the CPS domain. The CPS style allows the unified representation of cyber and physical elements of a system in the same architectural model. We introduce the Acme language and its flexible support for creating styles as Acme families, and illustrate the specification of elements in the physical family. The CPS style is implemented as three separate Acme families, along with mechanical, electrical, and thermo-fluid extensions to the physical family. Extending the physical family to multiple physical domains is done by specifying new element types for that domain, that include the effort and flow variables (and their units) as properties .

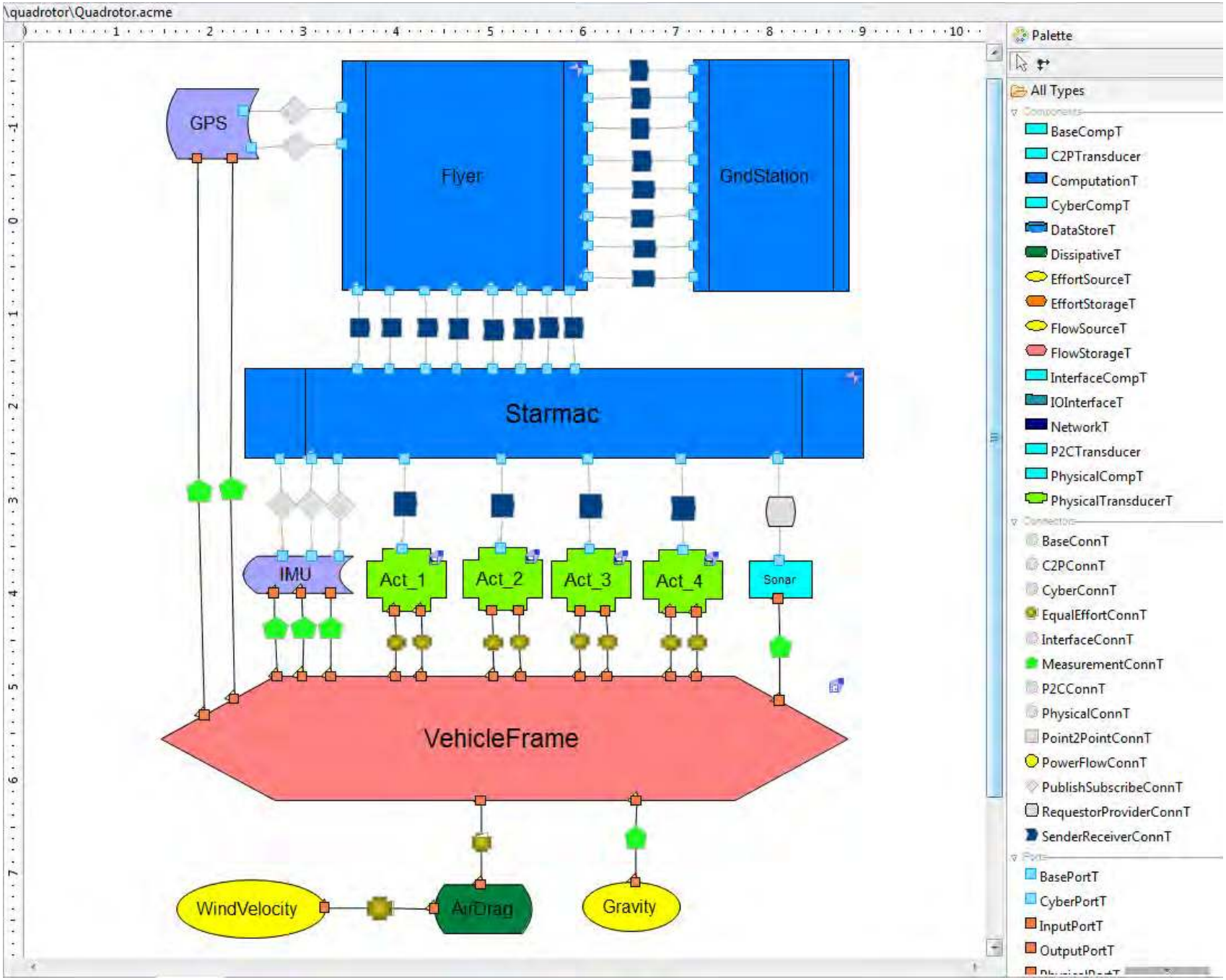

Figure 3.7: Creating the BA of quadrotor in AcmeStudio.

Chapter 4

Architectural Views

In this chapter, we define our notion of an architectural view, the types of structural relations needed between multiple views and the system BA, and the tools we have created to define such view relations.

A view describes the system's architecture from the perspective of a particular viewpoint, also called a *design concern* in this thesis. Hence, architectural views are a mechanism to facilitate the separation of concerns during system design. Although views can be (and usually are) constructed separately, the set of all system views must be related and consistent (in some sense) with the overall architecture, since each view contains a description of the same underlying system.

In contrast to the notion of architecture views, there is a lack of consensus in the architecture community for formally defining and characterizing relations between views [18]. View relations are necessary for establishing consistency of views with the underlying system architecture, and for maintaining that consistency as the system and its constituent models evolve over the design lifecycle. The types of view relations defined in this thesis allows us to check whether the assumptions made in each view about a system's component connectivity and physical structure are correct.

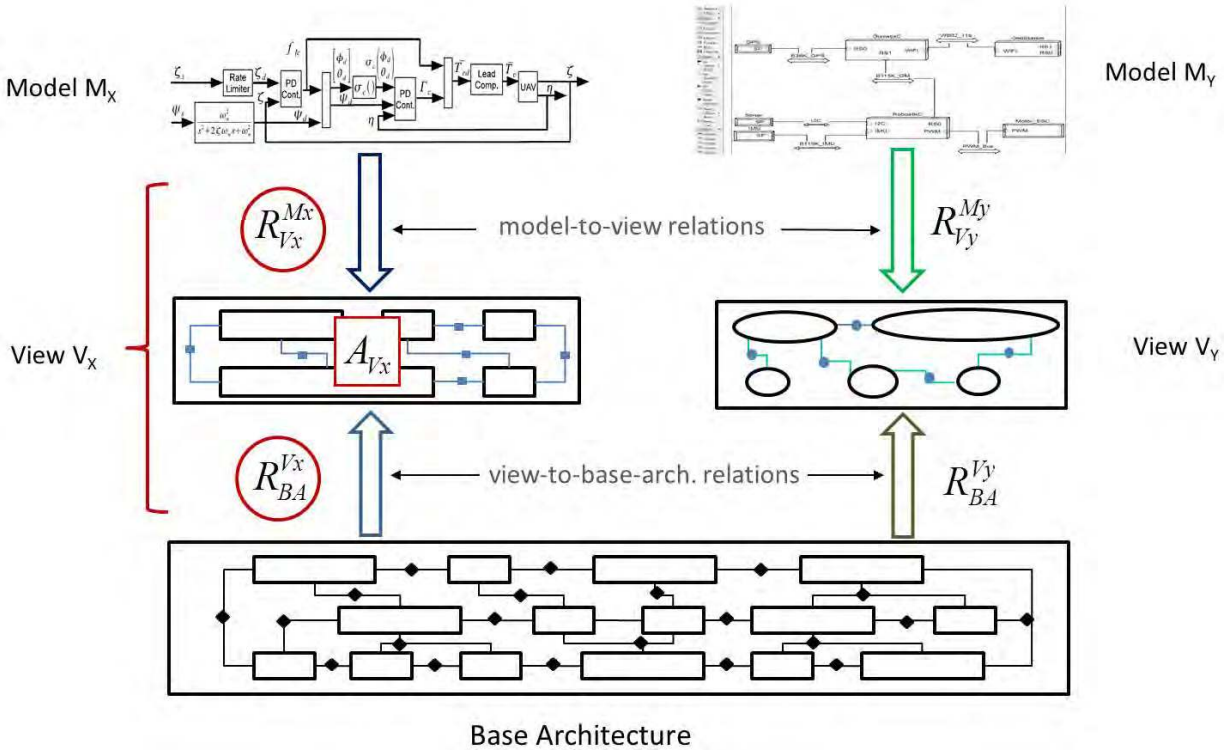

Figure 4.1: Relationship between models and BA through architectural views.

4.1 Formalizing Architecture Views

In this section we relate heterogeneous models to a system's base architecture through architectural views, as shown in Fig. 4.1.

Definition 1. The *base architecture* (BA) of a CPS is an instance of the CPS architecture style that contains all the cyber and physical components and connectors constituting the complete system at runtime that are relevant for system design and analysis.

The BA provides the reference structure for all models used for design and verification. It contains the set of system elements that are related to the analyses carried out in each model, as well as the elements that are common between the models. The BA should contain enough detail to describe the nature of the information exchanged and the physical quantities flowing between components, as well as component connectivity and coupling between physical variables represented by connectors.

Definition 2. An *architectural view* $\mathcal{V}$ for a design concern $\mathcal{D}$ is a tuple $< \mathcal{A}_V, \mathcal{R}_V^M, \mathcal{R}_{BA}^V >$ where:

- $\mathcal{A}_V$ is the component-connector architecture of the view, with the associated architectural style defined by $\mathcal{D}$
- $\mathcal{R}_V^M$ is the *model-to-view relation* that associates elements in the model with components and connectors in $\mathcal{A}_V$
- $\mathcal{R}_{BA}^V$ is the *view-to-BA relation* that associates elements in $\mathcal{A}_V$ with elements in the BA

A system model contains detailed structure and semantics defined by a particular formalism. An architectural view captures the system structure and connectivity assumptions reflected in the model through the view architecture, $\mathcal{A}_V$, and the element encapsulations and one-to-one correspondences defined by the relations $\mathcal{R}_V^M$ and $\mathcal{R}_{BA}^V$. Hence, a view can be thought of as a mechanism to capture the level of abstraction between the model and BA at the structural level. The view exposes certain system structures and their properties that are the focus of analysis in the associated model. In the next two sections, we describe how to create the relations $\mathcal{R}_V^M$ and $\mathcal{R}_{BA}^V$.

4.2 Model-to-View Relations

The various models used in the design of CPSs are created by using multiple domain-specific modeling languages (DSMLs). Each DSML defines the structure and semantics that every model instance should conform to. The model-to-view relation $\mathcal{R}_V^M$ associates elements in the model M with components and connectors in $\mathcal{A}_V$. The correspondence allowed is either one-to-one or an encapsulation of multiple elements in the model to a single element in the view, defined by the modeler's choice of grouping. The rules which define valid correspondences (including encapsulations) are based on the semantic relationship between the model's DSML and the architectural style of the view's design concern.

We observe that most models are composed of interacting components, with the seman-

tics of the interactions defined by (implicit or explicit) connectors. For certain DSMLs, the component-connector structure is obvious. For example, Simulink or LabView models used for control analysis consist of functional components interacting through connectors with data-flow semantics. Modelica or MapleSim models consist of physical components interacting through connectors with acausal interconnection semantics. For other DSMLs, the structure of the model may have to be created using well-defined translation rules. We have created rules for the DSMLs of models used in this thesis that are summarized in Tab. 4.1.

We illustrate the creation of the rules for Finite State Process (FSP), a DSML used to specify and verify concurrent system behavior. FSP models are a specification that contain processes, with their interactions modeled by synchronized events. To create a component-connector architecture for FSP models, we have implemented a translation scheme to create a component for each FSP process, with the ports defining the events that this process can share with other processes in the model. Connectors are created whenever two processes share events, and connectors can also be associated to an FSP process that defines the semantics of the communication protocol. In this way, a structurally and semantically meaningful architecture can be created from the textual specification of an FSP model. Hence, each heterogeneous system model can be thought of as having an underlying architecture, with the architectural style defined by the DSML that the model is created in.

The architecture $\mathcal{A}_V$ of a view V, created for a particular design concern $\mathcal{D}$, contains elements that conform to the architectural style defined by $\mathcal{D}$. We have created the following architectural styles for the views used in this thesis. The architectural elements contained in each style are summarized in Table 4.2.

- *Software style*: The software design concern focuses on the cyber aspects of the system, including concurrent processes and their communication, and analysis of

Table 4.1: Rules for mapping between DSML elements and View elements.

DSML	Components	Connectors	Ports	Allowed Encapsulations
FSP	Primitive process	Primitive process, Shared events	Shared events	Multiple events mapped to single port
Simulink	Blocks, Subsystems	Signal lines	Signals	Multiple signal lines to single connector, same for ports
AADL (Platform)	Processor, Memory, Device	Bus Access	Data	Multiple buses of same type to single connector
Modelica	Physical block	Physical, Signal-flow	Conjugate Variables	Multiple physical lines from same domain to single connector, same for ports

safety and liveness properties. The software style contains component types 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. A *Device* component represents the abstraction of physical processes and dynamics in the system, and is useful for the abstractions made in process algebra models. Connector types represent either (buffered or unbuffered) data, or events (i.e., messages) between communicating software components.

- *Control style*: From a control engineer's perspective, a CPS can be viewed as a signal flow model. The control style contains components to represent controllers, estimators, physical plants, sensors, actuators, and reference/setpoint generators. The connectors represent input-output signal flow between the connected components. The

control view typically ignores controller implementation details such as scheduling of tasks and associated communication jitter and delays.

- *Hardware style*: defines components such as processors, memory, and devices for sensors and actuators that are used to physically implement a system. The connectors represent the various buses (and their communication protocols), such as serial (RS-232), I2C, PCI bus, and wired and wireless networks.
- *Physical style*: is used to model architectures for multi-domain physical systems, and is the same as that defined in the CPS style.

Table 4.2: Architecture Styles for CPS Design Concerns.

Style	Components	Connectors	Ports
Software	Process, Thread, Subprogram, Controller, Estimator, IOAdapter, Device	Data-flow, Event	Data, Event, Data-Event
Control	Controller, Estimator, Reference, Filter, Plant, Sensor, Actuator, Environment	Signal-flow, ControlSig, FeedbackSig, MeasurementSig	Input, Output, Control, Reference, Feedback
Hardware	Processor, Memory, Device, Actuator, Sensor	Bus	InChannel, OutChannel, InOutChannel
Physical	Source, Sink, Storage, Dissipative, Transducer	Coupling, Physical Signal	Physical, Input, Output

In Fig. 4.2, the creation of the control view from the STARMAC Simulink model is illustrated. In this case, the relation $\mathcal{R}_V^M$ maps each top-level Simulink block to a component, and each group of signal lines between them to connectors, resulting in the control view's $\mathcal{A}_V$. The rules for valid encapsulations between the elements of the Simulink model and single elements of the view are based on what group of model entities represents controllers, what group represents plant dynamics, and so on. We allow combining of

multiple Simulink signals into a single view connector, since this encapsulation represents the creation of a vector signal from scalar signals, a semantically valid operation for the control design concern.

The model-to-view relations used in the case studies are created by hand to illustrate the concepts. However, many tool frameworks exist for automatic translation of Simulink (or similar signal-flow) models into a component-connector architecture. One example is the ESMoL language (part of Vanderbilt's GME toolchain) that converts any Simulink model into an ESMoL architecture and annotates it with para-functional properties [38]. Another one is SysWeaver that converts the controller part of a Simulink model into a component-based representation for real-time analysis [39]. There is an ongoing effort to automatically convert Modelica models into SysML component diagrams using triple graph grammars to transform between the meta-models of Modelica and SysML [34]. Similarly, we can envision a tool that automatically creates an architecture from an FSP textual specification, based on the embedded comments for each primitive FSP process and common events defined in the specification.

The focus of this thesis is on mechanisms for defining view-to-BA relations, and the consistency of views at the architectural level, not on the model-to-view relations or their implementation. Thus, tool support to define the relation $\mathcal{R}_V^M$ systematically can be made available as part of the model-based design framework within which our view consistency approach would be used.

4.3 View-to-BA Relations

In this section we describe our approach for defining relations between different system views and the underlying BA. For the purpose of this discussion, we consider the BA to be a view of the system, albeit a very detailed and comprehensive view, with the CPS style being the associated architectural style. In general, describing arbitrary relations (or "maps") between architectural views is a hard problem, since each view has its own semantics, and

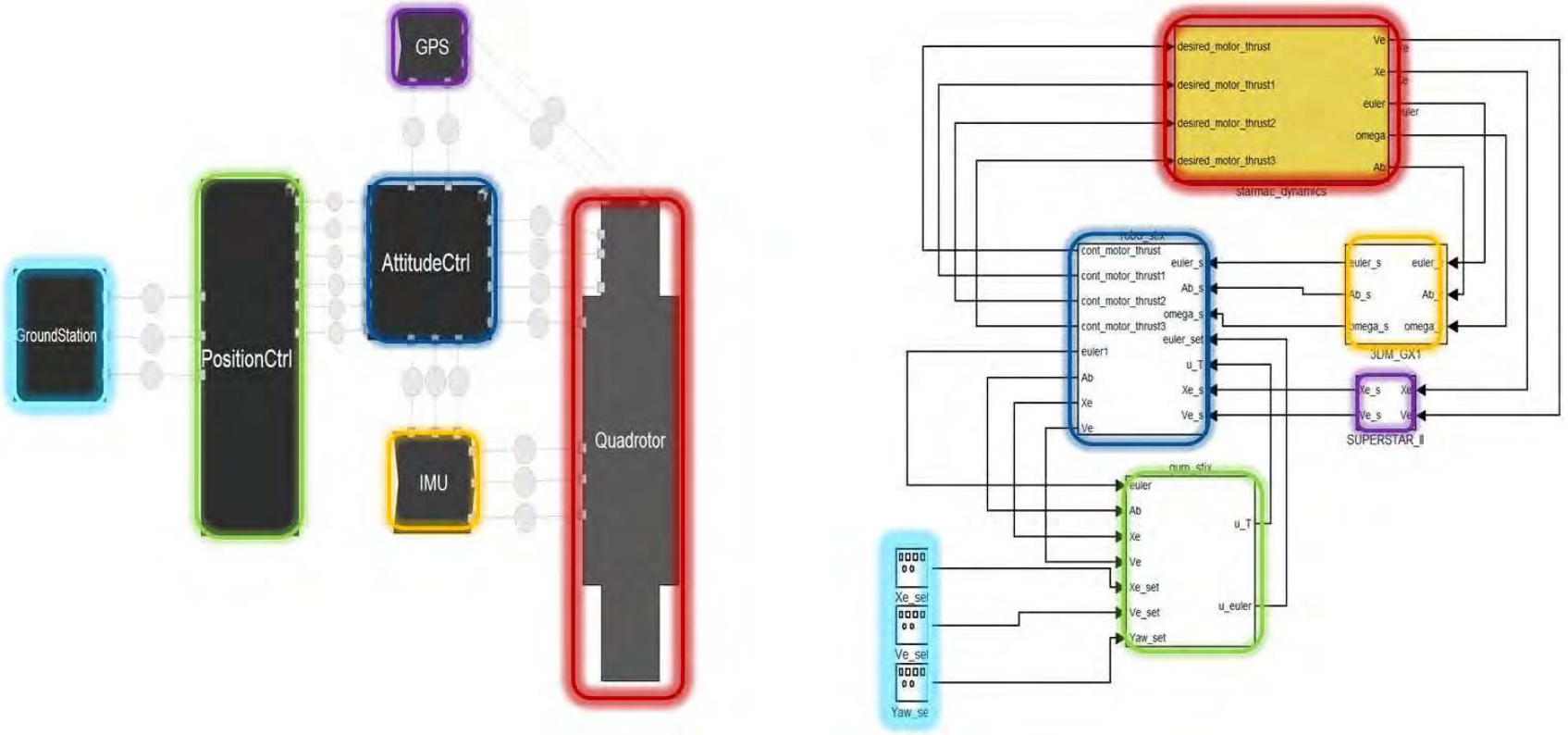

Figure 4.2: Creating the control view from a Simulink model.

there may not be sufficient overlap with the semantics of every other view. Despite the differences in semantics, most views describe their structure using common notations. For example, Medvidovic and Taylor argue that all architectural descriptions must explicitly specify their components, connectors, and architectural configurations [46]. Their survey shows that all ADLs provide means for structural specification of architectural elements. The universal use of structural notation makes it possible to describe maps between the *structural elements* of most views.

All the views considered in this thesis are structurally specified using a component-connector description. Hence, we consider relations that define either one-to-one element correspondences or encapsulations between sets of components and connectors in two different views, where one view is always the BA of the system. The view-to-BA relation, $\mathcal{R}^V_{BA}$, is an n-ary relation that enables the architect to group specific components and connectors in the BA and map them to elements of a view. The allowed correspondences and encapsulations are defined by the semantic relationship between the architectural styles of the two views, and hence $\mathcal{R}^V_{BA}$ also captures semantics of element types between the views.

Many ADLs carry semantic constructs along with structural descriptions. For example, Acme and Armani use properties attached to structural elements to carry semantic information. Armani also uses predicates to impose constraints on structural constructs [2]. Hence, a structural map description can provide a strong basis for describing more sophisticated maps that address semantic properties as well. We have incorporated checking of constraints over element properties (based on the correspondences defined by $\mathcal{R}^V_{BA}$) in our map creation language, Acme Maps, described in Sec. 4.4.

The rules for valid component correspondences under $\mathcal{R}^V_{BA}$ for the control view are summarized in Table 4.3. We have created similar rules for architectural elements in the software, hardware, and physical styles. Since the physical style is directly derived from the CPS physical style, the correspondence rules are one-to-one. These rules give the

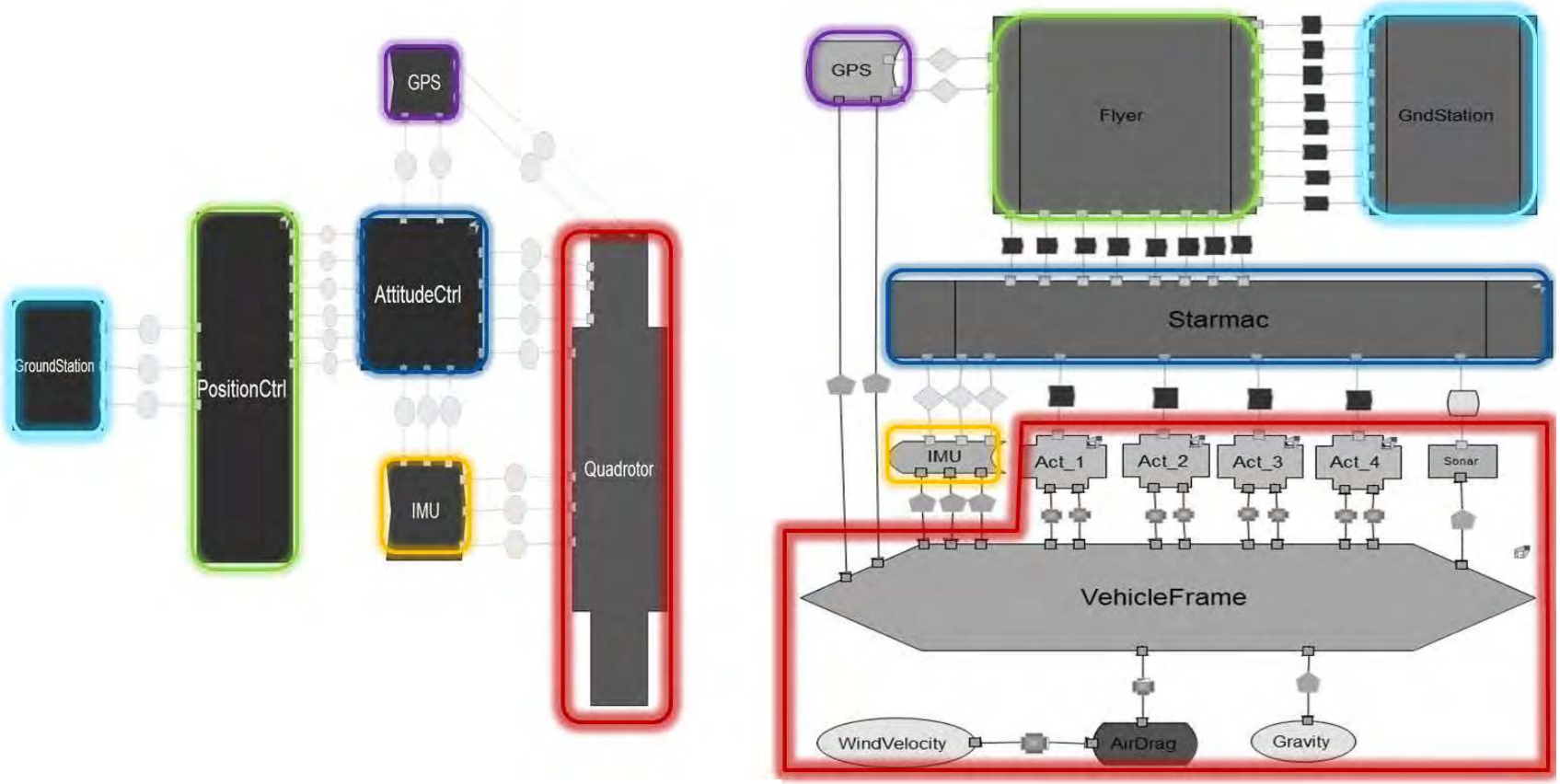

Figure 4.3: Mapping between the control view and BA for the quadrotor.

architect a mechanism to associate semantically meaningful entities between the view and BA structures being compared.

Table 4.3: Rules for mapping between components in the CPS style and different views.

Control	CPS	Software	CPS	Hardware	CPS
Controller	Computation Data Store	Process	Computation IOSoftware	Processor	Computation IOSoftware
Estimator	Computation Data Store	Thread	Computation IOSoftware	Memory	Data Store
Reference	Computation Data Store	Controller	Computation Data Store	Device	CPI Physical
Filter	Computation Physical	Estimator	Computation Data Store	Sensor	CPI Physical
Plant	Physical CPI	IOAdapter	IOSoftware CPI	Actuator	CPI Physical
Sensor	Transducer CPI	Device	CPI Physical		
Actuator	Transducer CPI	Subprogram	Computation IOSoftware		
Environment	Computation Data Store Physical				

Figure 4.3 illustrates the mapping defined between the control view and BA of the quadrotor. The correspondences are created based on the semantic compatibility of the element types defined in Table. 4.3. For example, the controller components in the view can only be mapped to cyber and data components (or their encapsulations) in the BA, and plant components in the view can be mapped to physical or interface components in the BA. Every connector in the control view represents a (cyber or physical) signal. Hence, semantically equivalent connectors between two components in the BA can be mapped to a single connector in the control view.

The encapsulation operation defined by $\mathcal{R}^V_{BA}$ for components is illustrated in Fig. 4.5, where three components have been selected to be encapsulated into a single component,

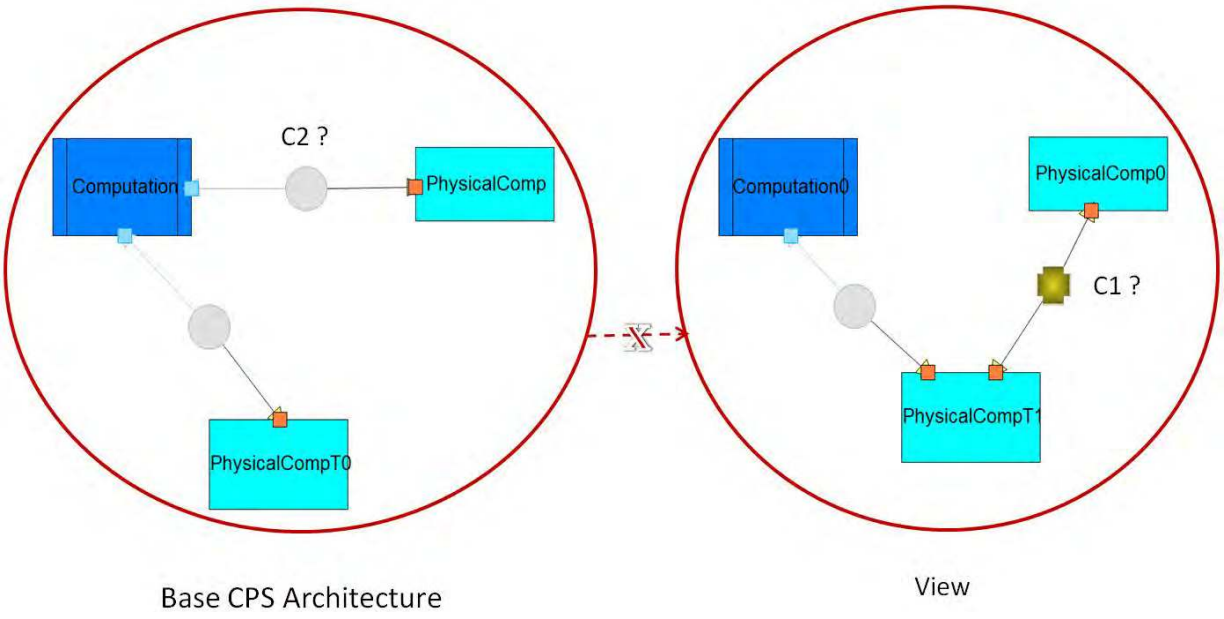

Figure 4.4: Invalid many-to-many map between a view and BA.

and then mapped to a component in the view. Any elements (components or connectors) that are internal to the selected components are hidden in the encapsulated component. Any ports that are connected to elements outside the encapsulation (or are unconnected boundary ports of components) appear as ports of the encapsulated component, as shown for ports $P1$ and $P2$ in Fig. 4.5.

The encapsulation operation for connectors, shown in Fig. 4.6, is analogous to component encapsulation, except that roles $R1$ and $R2$ now appear as roles of the encapsulated connector. However, encapsulations that result in neither a component nor a connector, such as the one shown in Fig. 4.7 (because of the presence of both port and role on the same element), are not permitted. Our tool framework (described in Sec. 4.4) automatically checks for such cases and flags an error when detected.

One-to-one and many-to-one (encapsulation) maps from the BA to a view are allowed. Many-to-many maps are not allowed since this can lead to inconsistent connections being hidden inside the encapsulated components. A potential many-to-many map is shown in Fig. 4.4 between the BA and view of a system. If this map were allowed, the connector C1

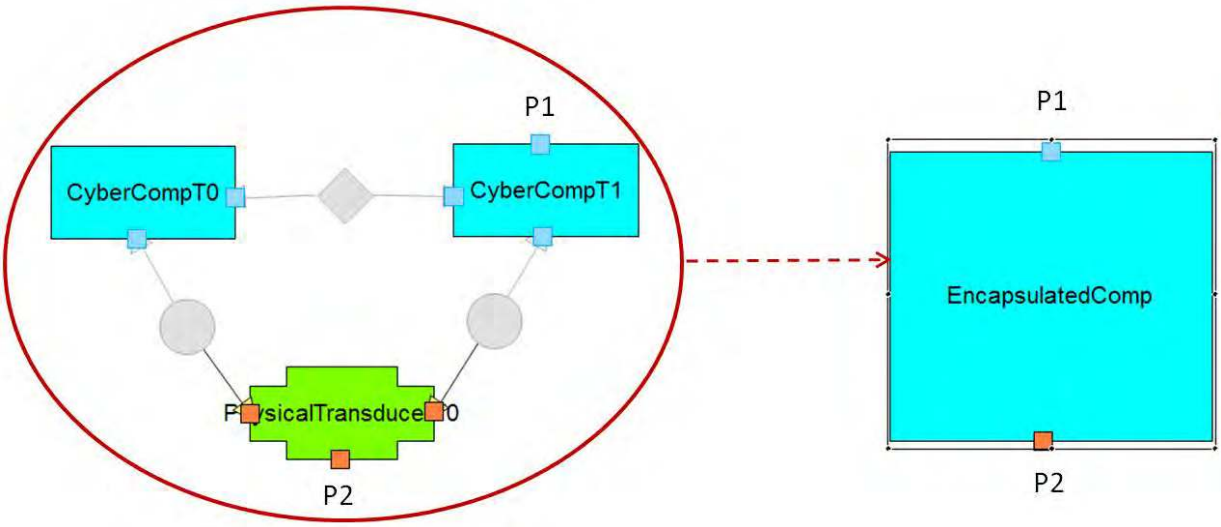

Figure 4.5: Encapsulation of components in a system.

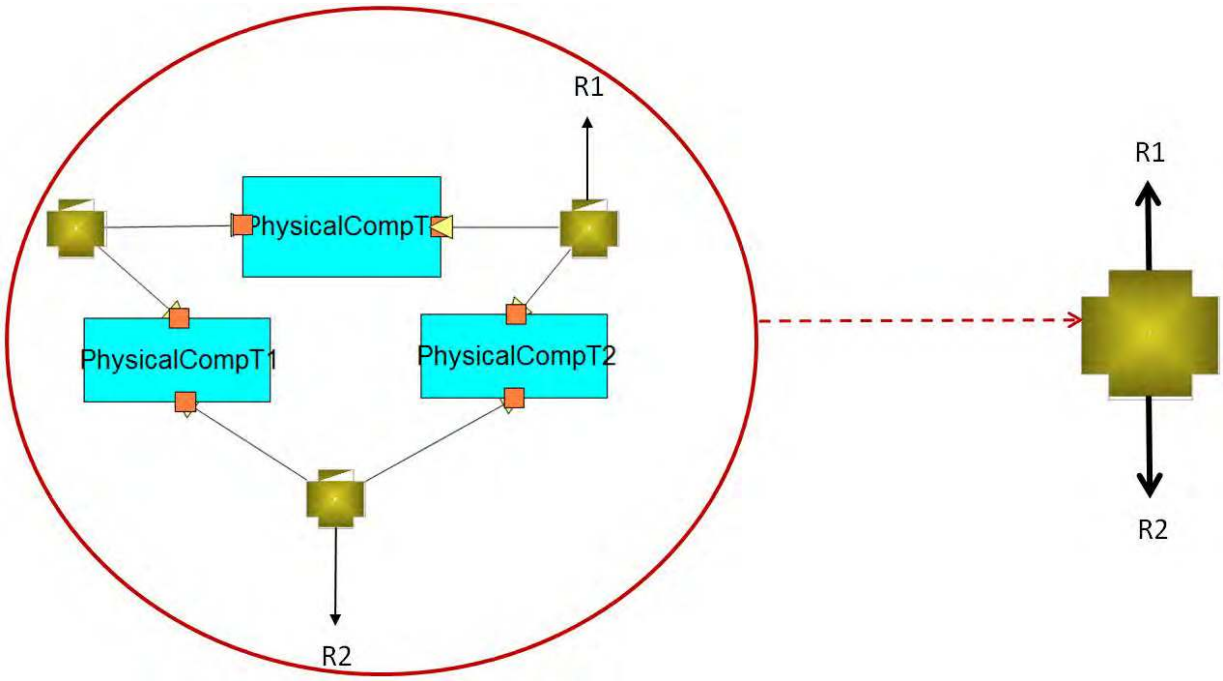

Figure 4.6: Encapsulation of connectors in a system.

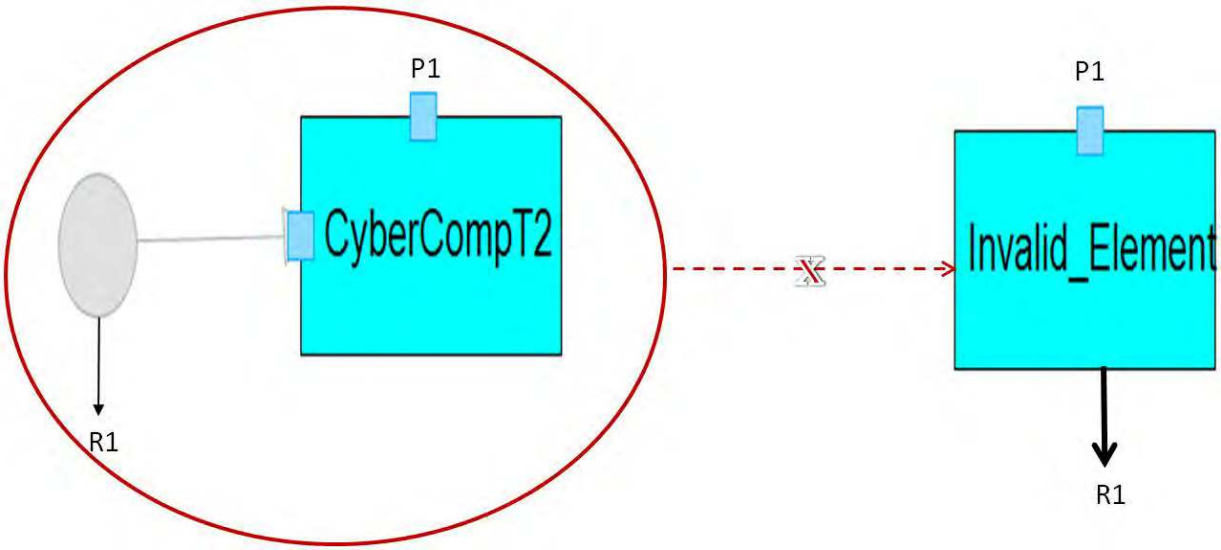

Figure 4.7: Invalid encapsulation of elements in a system.

(present only in the view) and the connector C2 (present only in the BA) would both be hidden inside the encapsulation and would not be detected when the view is checked for consistent structural connectivity with the BA. A similar reasoning holds for disallowing many-to-many maps between connectors, where inconsistent components can be hidden away between encapsulated connectors.

Not allowing many-to-many maps does not restrict the expressiveness of our view relations for structural consistency checking of views. Any many-to-many map between the view and BA elements can be transformed into an encapsulation from BA to view elements. This is done by creating a single view element from the set of view elements in the many-to-many map. Combining view elements in this manner reflects that there is detailed structure in the associated model that is not defined in the BA. The view abstracts these model details into a single element. This operation does not impact the connectivity topology of the view. However, allowing many-to-many maps can be useful in situations where refinements in the view are used to analyze model coverage, as discussed in Chap. 8.

4.4 Tool Framework for Architecture View Maps

In this section, we describe the tool framework to create the relation $\mathcal{R}^V_{BA}$ as *maps* between architectural views and the BA.

An architecture view is created in AcmeStudio as an instance of a specific Acme family, based on the design concern of the view. Figure 4.8 shows the creation of a control view in the AcmeStudio editor. The palette on the right contains all the element types that are present in the control family. Each view is saved as an Acme system in AcmeStudio.

We have created a mechanism for creating view relations between two component-connector views with different architectural styles: the *Acme Maps* language. Acme Maps is an extension to the Acme ADL for expressing maps between the structural aspects of architectural views, and for expressing constraints over those maps using Acme design rules. An important feature of the language is the automatic checking of valid mappings based on constraint rules defined by the user between the architectural styles of the two views.

An Acme Map specification consists of two types of text files. The first is a *map type* file which specifies which types of elements from each of the views are allowed in any map instance between the views. The second is a *map* file that specifies a mapping between a pair of views, based on the map type. Every map is an instance of a map type and has to satisfy the constraints imposed by its map type.

In Fig. 4.9, the general structure of an Acme map type specification is shown. The specification contains the pair of Acme families whose element types are to be related. If we want to create the map for $\mathcal{R}^V_{BA}$, the source is the Acme family of the BA, while the target is the Acme family of the view. Each map type is made up of a number of component (and nested port) and connector (and nested role) map types. Acme design rules can be defined for any element map type and are useful to enforce semantic constraints between the view and BA families. Consider the portion of the control map type specification

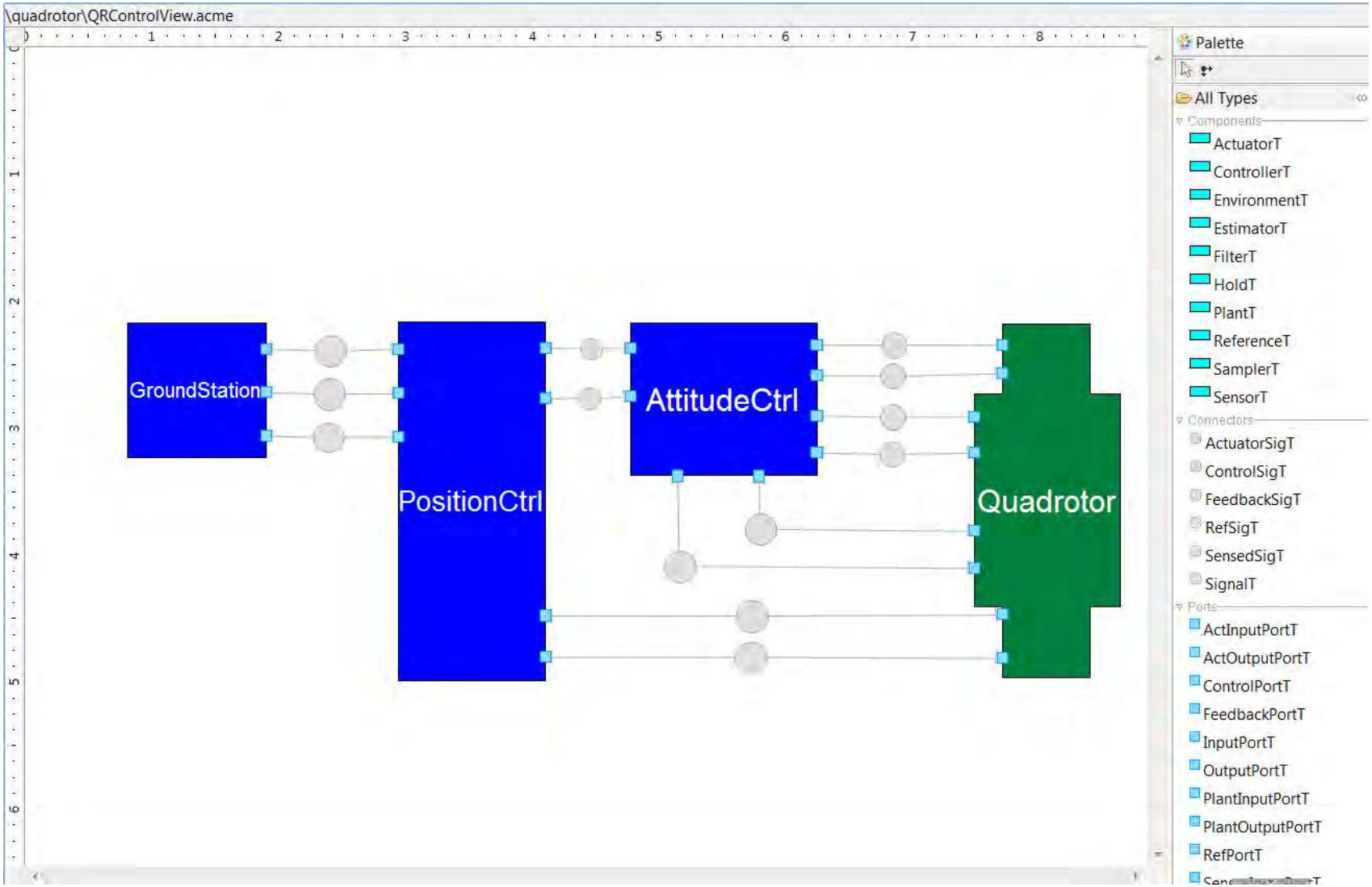

Figure 4.8: Creating a control view in AcmeStudio.

shown in Fig. 4.10. A component map type named *PlantMapT* is defined for components representing plants in the control view. The map type specifies that any plant component in the control family can be mapped to the set of physical components in the physical family and transducer components in the interface family. This is semantically meaningful for a control view because many control models group sensors and actuators along with plant dynamics into a single element and treat it as the plant to be controlled.

The *PlantMapT* also contains nested port map types that specify which types of ports on any plant element can be mapped to ports in the CPS family. The Acme invariant enforces the presence of at least one physical component in any encapsulation of compatible elements from the BA to a plant component. This design rule prevents the architect from accidently mapping only transducer components in the BA to a plant component, since it is not semantically valid for a plant to contain no physical components.

The second component map type (named *ControllerMapT*) contains a (heuristic) design rule between the properties of any controller element in the view and any computational element in the BA. The rule specifies that the sampling period of any controller must be more than the Worst-Case Execution Time (WCET) of the computational element it is mapped to. This is an example of a map type that includes a semantic check over properties of its elements. Any map instance that satisfies this map type will map only those controllers whose sampling periods satisfy this rule to appropriate computation elements in the BA.

In Fig. 4.11, a portion of the map file for the control view map type is shown. The map contains correspondences between the elements of the BA of the quadrotor, represented by the *Quadrotor* system, and those of a control view, represented by the *QRControlView* Acme system. In particular, the component map *pMap1* (of type *PlantMapT*) shows an encapsulation of a set of elements in the BA to the *Quadrotor* plant component in the view. The component map *cMap1* associates the *Starmac* cyber component in the BA to

```
MapType MapTypeName = {
   Source = {AcmeFamilyName};
   Target = AcmeFamilyName;
   [invariant ...]
   ComponentMapType MapTypeName = {
     Source = {ComponentTypeName, ComponentTypeName, ...};
         Target = ComponentTypeName;
         [invariant ...]
         PortMapType MapTypeName = {
           Source = {PortTypeName, PortTypeName, ...};
           Target = PortTypeName;
           [invariant ...] }
         ...
        }   ...
   ConnectorMapType MapTypeName = {
     Source = {ConnectorTypeName, ConnectorTypeName, ...};
         Target = ConnectorTypeName;
         [invariant ...]
         RoleMapType MapTypeName = {
            Source = {RoleTypeName, RoleTypeName, ...};
            Target = RoleTypeName;
            [invariant ...] }
         ...
  }
 ...
}
```

Figure 4.9: Structure of an Acme map type specification.

the *AttitudeCtrl* controller in the view. The sampling period property of *AttitudeCtrl* is automatically checked against the WCET property of *Starmac*, because of the invariant defined in the map type (*ControllerMapT*) of *cMap1*.

Map types play two useful roles in a map specification: (1) they capture the common properties of a class of maps, hence facilitating classification and reuse of map declarations, and (b) they support type checking of map instances. A map type contains structural and semantic constraints. The structural constraints address the structural aspects (e.g., the type of elements associated) that instances of the map type must have. Semantic con-

```
MapType ControlMapT = {
   Source = CPSFam;
   Target = ControlFam;
   ComponentMapType PlantMapT = {
     Source = {PhysicalFam.PhysicalCompT,
                InterfaceFam.P2CTransducer,
                             InterfaceFam.C2PTransducer};
         Target = ControlFam.PlantT;
         invariant exists c : component in sources |
               declaresType (c,PhysicalFam.PhysicalCompT);

     PortMapType ActOutputPortMapT = {
              Source = {PhysicalFam.PhysicalPortT};
              Target = ControlFam.ActOutputPortT;
         }
         ...
   }
   ComponentMapType ControllerMapT = {
         Source = {CyberFam.ComputationT,CyberFam.DataStoreT};
         Target = ControlFam.ControllerT;
         heuristic foreach c : ComputationT in sources |
                           target.samplingPeriod > c.WCET;
        ...
   }
  ...
}
```

Figure 4.10: Portion of a control view map type file with component map and port map types.

```
Map controlMap : ControlMapT = {
   Source = Quadrotor;
   Target = QRControlView;
   ComponentMap cMap1 : ControllerMapT = {
      Source = {Quadrotor.Starmac};
      Target = QRControlView.AttitudeCtrl;
   }
   ComponentMap pMap1 : PlantMapT = {
      Source = {Quadrotor.GPS,Quadrotor.VehicleFrame,
                Quadrotor.MeasurementConnT1,
                Quadrotor.MeasurementConnT6,
                Quadrotor.Act_3,Quadrotor.Act_4,
                Quadrotor.Act_2,Quadrotor.AirDrag,
                Quadrotor.IMU,Quadrotor.Gravity,
                Quadrotor.Sonar,Quadrotor.WindVelocity};
      Target = QRControlView.Quadrotor;
   }
...
}
```

Figure 4.11: Portion of the map file for the quadrotor control view.

straints address relations between element properties and are declared as Acme design rules. We have created Acme map types between the CPS family and the software, hardware, control, and physical families. Every Acme map defined for a particular view uses the map type for the associated family to create the view-to-BA relation.

The process of creating an Acme map in AcmeStudio between the control view and BA of the quadrotor is shown in Fig. 4.12. We have implemented an *Acme View Editor* to facilitate the process of displaying multiple views, and defining, editing, and saving mappings between views. The user first selects the Acme system which represents the BA, and then selects a system representing the view. The view editor displays both systems side-by-side so that mappings between the BA and view can be created easily. The palette on the right contains all the element map types that have been defined in the associated map type specification. To create an element map from BA to view, the user clicks on the appropriate map type in the palette. The editor highlights only those elements in the BA

and view that are compatible as defined in the map type.

In Fig. 4.12, the user has selected the plant map type, and hence only the *Quadrotor* plant component in the view and all physical and interface components in the BA have been highlighted as being selectable for mapping. The current map shows an encapsulation of BA elements mapped to the single plant component in the view. This mapping forms part of the $\mathcal{R}_{BA}^{V}$ relation between the control view and the BA of the quadrotor. In a similar way, the user can define any number of element maps and then save (and reload) the resulting map file for future use. In addition, a map file can be edited on-the-fly by adding new maps or deleting existing maps through the features of the graphical editor. The view editor automatically adds all components and connectors contained in an encapsulation to the corresponding map instance, and also checks for duplicate, erroneous, or incompatible maps between elements. The tool has been useful in quickly and correctly creating maps between multiple views and the BA for both case studies in this thesis.

4.5 Summary

In this chapter, we introduce an architectural view as a mechanism to capture the level of abstraction between the model and BA at the structural level. Formally, an architectural view captures the system structure and connectivity assumptions reflected in the model through the view architecture, $\mathcal{A}_V$, and the element encapsulations and one-to-one correspondences defined by the relations $\mathcal{R}_{V}^{M}$ and $\mathcal{R}_{BA}^{V}$.

We define the relations $\mathcal{R}_{V}^{M}$ and $\mathcal{R}_{BA}^{V}$ for a view and describe rules to create them for the software, hardware, control, and physical design concerns. To instantiate $\mathcal{R}_{BA}^{V}$ between a view and the BA, we introduce the Acme Maps language for the creation of map types and maps. We illustrate the use of the AcmeStudio multi-view graphical editor that facilitates the creation, editing, and display of view maps.

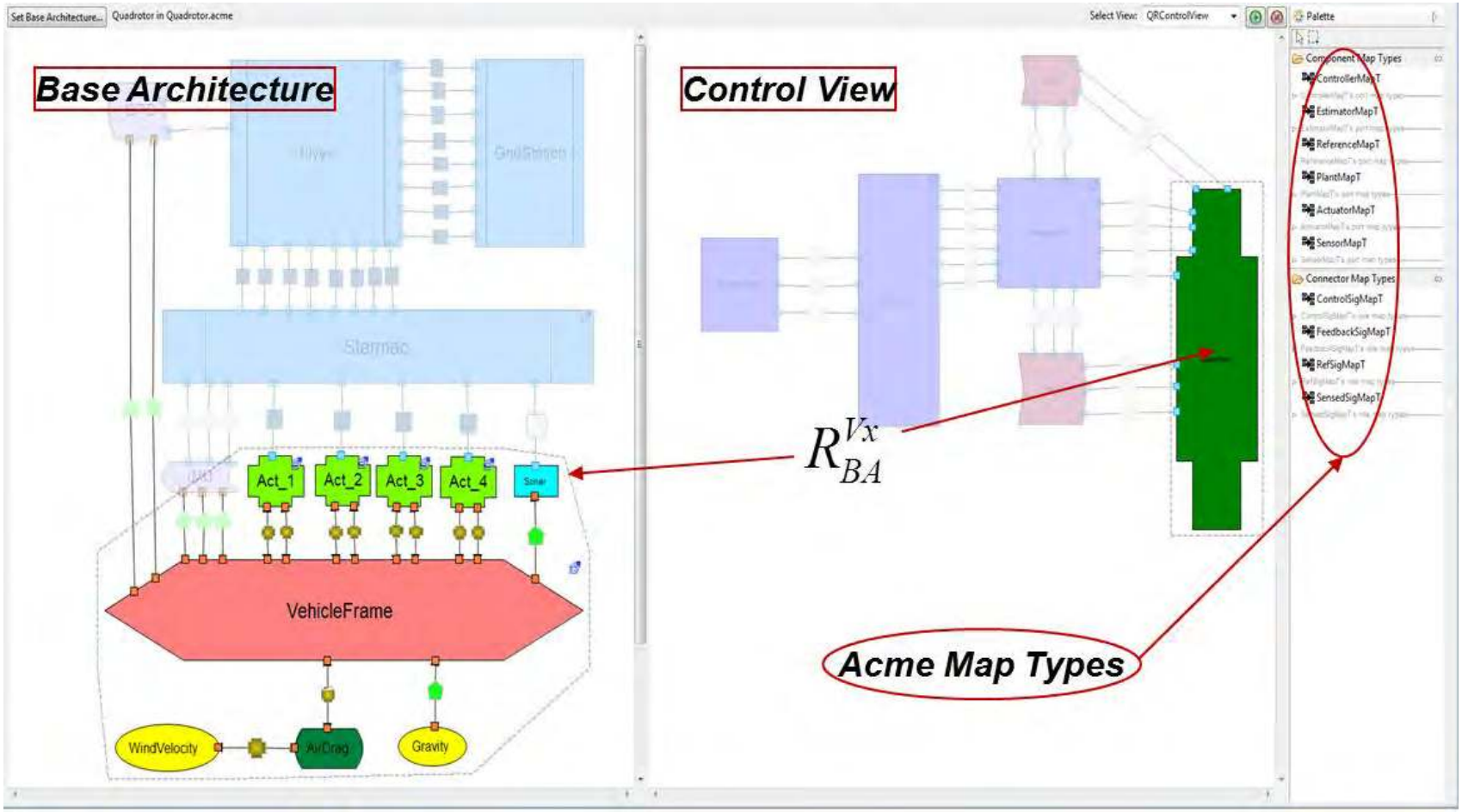

Figure 4.12: Creation of a plant map in AcmeStudio between the control view and BA of the quadrotor.

Chapter 5

Consistency of Architectural Views

In this chapter, we define the notion *structural consistency* between the system's BA and the architectural views derived from various heterogeneous models. Structural consistency ensures that the views adhere to the connectivity constraints and physical laws present between elements in the BA. 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 system 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.

In Sec. 5.1, we introduce *architecture graphs* as typed graphs that retain the interconnection topology of architectural elements. In Sec. 5.2, we formulate structural view consistency as the existence of a morphism relation between the graphs of the view and BA., and define *view conformance* and *view completeness* as two types of consistency conditions. Our graph morphism and maximum common subgraph (MCS) algorithms check for consistency or find the set of maximal consistent elements between the view and BA, respectively, and are described in Sec. 5.4. We study the performance of these algorithms applied to a set of randomly generated graphs in Sec. 5.6, and describe our tool framework for view consistency support in AcmeStudio in Sec. 5.7.

5.1 Architectures as Graphs

Any mathematical formalism that attempts to model a system's architecture has to support two key characteristics. First, an architecture always includes the structure of the system, and structure primarily consists of the topology. Second, typing of components and connectors is necessary to help distinguish between architectures that are topologically identical but represent semantically different systems. For example, one star topology architecture might represent a mainframe computer being used by multiple thin clients, while another might represent the centralized control of several unmanned aerial vehicles by a single ground station.

Due to these requirements, we use undirected, labeled graphs to model a system architecture. Associating types with elements differentiates an architectural model from a simple graph because it defines the correspondence between an element and some real-world entity or physical phenomenon. Mapping architectures into graphs allows us to leverage well-studied tools in graph theory that evaluate the topological similarity between two structures.

Definition 3. A labeled (or typed) graph G is a 6-tuple $< V_G, E_G, \Lambda_G, \Sigma_G, \alpha_G, \beta_G >$, where

- V_G is the set of vertices (or nodes)
- $E_G \subseteq V_G \times V_G$ is the set of edges (or arcs)
- Λ_G is the set of vertex labels
- Σ_G is the set of edge labels
- $\alpha_G : V_G \to \Lambda_G$ is the vertex labeling function
- $\beta_G : E_G \to \Sigma_G$ is the edge labeling function

Definition 4. An architecture graph (AG) is a graph created from a system architecture that retains the interconnection topology of the architectural elements:

- V_{AG} is the set of components, connectors, ports, and roles in the architecture
- E_{AG} 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_{AG} is the set of architectural types for components, connectors, ports, and roles that are defined by the corresponding architectural style
- $\Sigma_{AG} = \{contains, attachment, binding\}$ is the set of edge types in the graph

The set Λ_{AG} is a partially ordered set, or poset, with $\leq$ defined as the *type inheritance* relation. For example, in the control view style, let $ControllerT$ be a type for a generic controller component and let $PIDControllerT$ and $LQRControllerT$ be two subtypes of $ControllerT$ that represent specific kinds of controller components. This is equivalent to defining $PIDControllerT \leq ControllerT$ and $LQRControllerT \leq ControllerT$. Further, suppose $DLQRControllerT$ represents a discrete LQR controller, which is a subtype of $LQRControllerT$. By the transitivity of type inheritance, $DLQRControllerT \leq ControllerT$ also holds.

Σ_{AG} contains the types of edges that correspond to all the 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 node. The edge represents a port-role attachment 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.

The types in Σ_{AG} do not have any type inheritance with each other since each represents

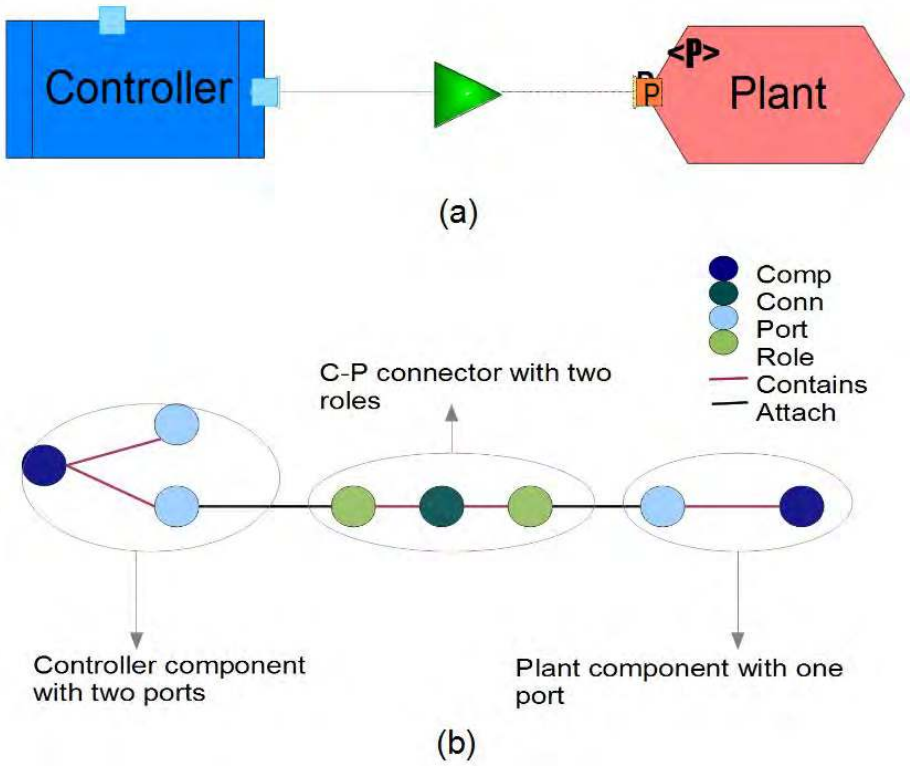

Figure 5.1: Example of an architecture graph.

a distinct class of connection in the architecture.

Figure 5.1 shows an example of a simple architecture and the associated graph representation.

A *graph morphism* is a structure-preserving correspondence between two graphs that maps adjacent nodes in one graph to adjacent nodes in the other, while maintaining the type compatibility of the nodes.

Definition 5. Let G and H be two graphs. Let $\mathcal{T}_H^G \subseteq \Lambda_G \times \Lambda_H$ be a relation defining the compatibility of vertex types between G and H, and let $\mathcal{L}_H^G \subseteq \Sigma_G \times \Sigma_H$ be the corresponding relation between the edge types of G and H. A *graph morphism* (from G to H) is a function $\mathcal{M} : V_G \to V_H$ for which the following properties hold:

1. $e = (u, v) \in E_G \implies e' = (\mathcal{M}(u), \mathcal{M}(v)) \in E_H,\ \forall u \forall v \in G$
2. $(\alpha_G(v), \alpha_H(\mathcal{M}(v))) \in \mathcal{T}_H^G$

3. $(\beta_G(e), \beta_H(e')) \in \mathcal{L}_H^G$

If $\mathcal{M}$ is injective (one-to-one), the morphism is known as a *monomorphism*. If the function is bijective (one-to-one and onto), the morphism is an *isomorphism*. The *cardinality* of a morphism is defined as the total number of possible mappings between the two graphs.

When the morphism definition is applied to architecture graphs, specifically, between a view graph and a BA graph, the relation $\mathcal{T}_{BA}^V$ is defined by the map type of the view. The posets of element types associated with the graphs of the BA and each view are derived from the architectural styles of the BA and view, respectively. The posets enable the architect to define which entities in the BA are semantically compatible with those in a particular view. The relation $\mathcal{L}_{BA}^V$ for architecture graphs is an identity relation between the same types of edges.

We will need the concept of the *maximum common subgraph* (MCS) between a pair of graphs to elaborate on our notion of view consistency. Hence, we introduce the concepts related to the MCS in this section.

Definition 6. Let $G = (V_G, E_G, \Lambda_G, \Sigma_G, \alpha_G, \beta_G)$ and $g = (V_g, E_g, \Lambda_g, \Sigma_g, \alpha_g, \beta_g)$ be two graphs. g is a *node-induced* subgraph of G if

- $V_g \subseteq V_G$
- $E_g = E_G \cap V_g \times V_g$
- $\alpha_g(v) = \alpha_G(v)\ \forall v \in V_g$
- $\beta_g(e) = \beta_G(e)\ \forall e \in E_g$

The subgraph obtained by Definition 6 is called node-induced because the number of vertices from the parent graph is maximized. An alternative way to define a subgraph is in terms of the edges of the parent graph.

Definition 7. Let $G = (V_G, E_G, \Lambda_G, \Sigma_G, \alpha_G, \beta_G)$ and $g = (V_g, E_g, \Lambda_g, \Sigma_g, \alpha_g, \beta_g)$ be two graphs. g is an *edge-induced* subgraph of G if

- $E_g \subseteq E_G$

- $V_g = \{v \in V_G : \exists u \in V_G, (v, u) \in E_g\}$
- $\alpha_g(v) = \alpha_G(v)\ \forall v \in V_g$
- $\beta_g(e) = \beta_G(e)\ \forall e \in E_g$

The edge-induced subgraph contains the maximum number of edges along with all the nodes that are endpoints of the edges. As a result, it can never contain isolated vertices. Since the nodes of an architecture graph represent the elements of interest, and since the mismatch of an edge in such graphs automatically implies that the connected vertex must also be in error, the node-induced subgraph definition is more applicable to our problem.

In the remainder of the thesis, we use the term *subgraph* to mean a node-induced subgraph. If a graph g is a subgraph of another graph G, we henceforth write this relation as $g \subseteq G$.

Definition 8 (Subgraph Isomorphism)**.** If $\mathcal{I} : g \to h$ is a graph isomorphism between graphs g and h, and h is a subgraph of another graph H, then I is called a subgraph isomorphism from g to H.

Definition 9 (Maximum Common Subgraph)**.** Let G and H be two graphs. A common subgraph of G and H is a graph g such that there exist subgraph isomorphisms $\mathcal{I}_G : g \to G$ and $\mathcal{I}_H : g \to H$. We call g a *Maximum Common Subgraph* of G and H, or MCS(G, H), if there exists no other common subgraph that has more nodes than g.

The MCS is not necessarily unique for two given graphs. We call the set of all maximum common subgraphs of a pair of graphs the *MCS set* of the pair.

5.2 Structural View Consistency

Based on the representation of a system architecture as a typed, undirected graph, we can formulate the structural correctness of a view with the BA in terms of the topological similarity between the corresponding architecture graphs.

Definition 10. An architectural view $\mathcal{V}$ is *structurally consistent* with the BA if there

exists a graph morphism from the view graph to the BA graph, where the graph morphism conforms to the element correspondences defined by the relation $\mathcal{R}^V_{BA}$.

Based on whether the mapping is a monomorphism or an isomorphism, the following two notions of view consistency arise.

Definition 11. *View Conformance*: There exists a monomorphism from the view graph to the BA graph that conforms to the element correspondences defined by the relation $\mathcal{R}^V_{BA}$.

Definition 12. *View Completeness*: There exists an isomorphism between the view graph and the BA graph that conforms to the element correspondences defined by the relation $\mathcal{R}^V_{BA}$.

View conformance 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. Checking for view conformance is useful when a view describes a sub-part of the complete system architecture.

The physical view of the quadrotor which contains the elements that define its dynamic equations of motion illustrates view conformance. The view does not contain any cyber elements since the computational aspects of the system are not meaningful in the physical design concern. If we want to ensure that the physical view has been created with valid assumptions about the actual physical system, we need to check that every physical element in the view is allowed in the quadrotor's BA, and that the physical coupling present in the view elements is also present in the BA elements. We should also expect none of the cyber elements in the BA to be present in the view. Hence, the consistency check that should be applied in this case is view conformance, so that every view element is consistent with,

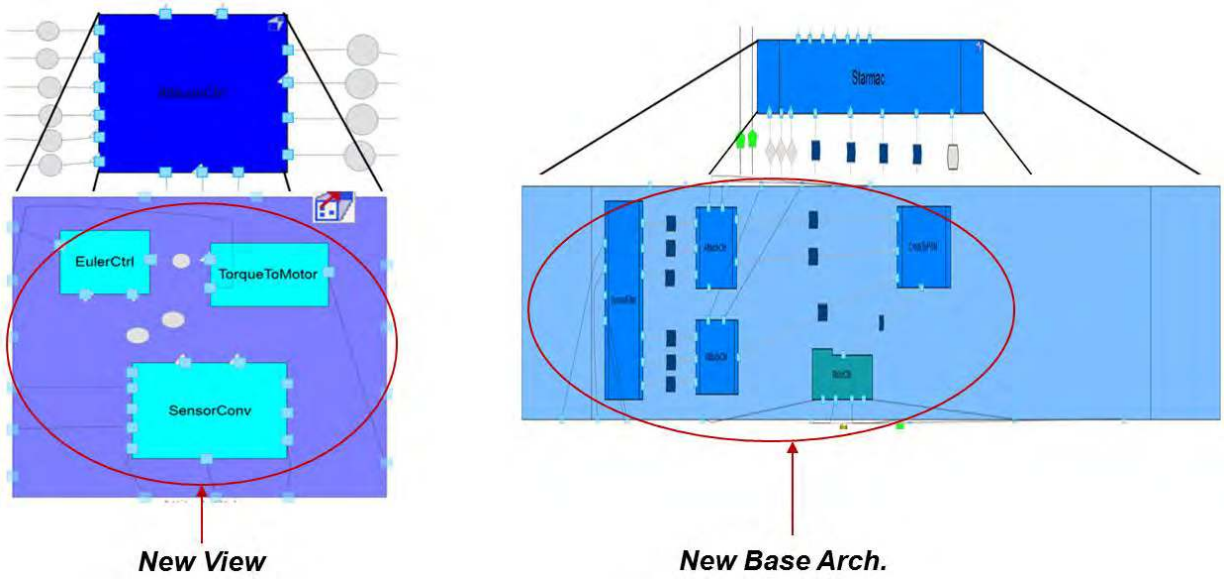

Figure 5.2: Hierarchical consistency checking between view and BA of quadrotor.

and accounted for, by some set of elements in the BA.

View completeness satisfies all the properties of view conformance. 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. For example, the control view models the complete quadrotor system, with the cyber elements represented in the controller elements, and the physical elements contained in the plant component. Hence, every BA element has to be checked in the view, in addition to the view conforming to the BA. This is a case where the view completeness check is applicable.

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 top level of the control view and the BA for the quadrotor. The architect can further check if the internal structure of the *AttitudeCtrl* component in the view is consistent with the internal architecture of the *Starmac* component in the BA. We assume, of course, that both these components do have an internal architecture present. Acme allows the hierarchical

description of the internal details of an element through the concept of one or more *representations* associated with the element. We can now follow the same procedure as for the view-BA case. A new 'view' is created from the internal elements of the *AttitudeCtrl* and the representation of *Starmac* now becomes the new 'base architecture'. A consistency check can be carried out between the graphs of the new view and of the new BA. In this way, the morphism checker can be invoked in a top-down manner to verify consistency between elements (and sub-elements) of the view and the BA. However, the approach does not currently support consistency checks across hierarchies, i.e, the architect cannot compare the internal structure of the *AttitudeCtrl* with the top-level elements of the BA.

Our architectural approach is different from traditional approaches to consistency checking, 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 sequence. Structural 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 its elements.

We have introduced the concept of *parametric consistency* [47] as a first step towards using the architectural framework to evaluate stronger semantic relationships between models, including consistency with respect to behavioral semantics. This is done by allowing each model (and hence the associated view) to import and export component and system level parameters. We use logical constraints over parameters in the architectural views to represent the conditions under which the specifications verified for each model are true and imply the system-level specification. Interdependencies and connections between the constraints in the architectural views are managed in the base architecture using first-order

logic of real arithmetic to ensure consistency and correct reasoning. However, this thesis focuses on the consistency of system models at the structural level, while allowing for component parameters as annotated properties that can be checked during the creation of element mappings.

5.3 Applicability to System Design

The base architecture provides the reference structure for all models used for design and verification. Adherence to the component-port-connector structure of the BA 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 a structure identical to the BA. The important constraint is that the presence or absence of ports and connectors in either the architectural view or the BA must be reflected in the other structure through one-to-one or many-to-one associations from BA to view. Such a constraint rules out the possibility that a view can introduce a "back-door" communication channel not present in the reference structure, a property called *communication integrity* in software architectures [20].

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

If the BA is constructed from validated stakeholder/system requirements, then the BA contains only components and connectors that can be traced back to particular system-level requirements. Under this assumption, 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 analysis and verification, and vice-versa.

5.4 Graph Algorithms for View Consistency

To check for view conformance and view completeness, we have implemented the VF2 algorithm [48]. This algorithm uses a computationally efficient heuristic based on the analysis of the sets of nodes adjacent to the ones already considered in the partial mapping. It is significantly faster than the widely used Ullman algorithm in many cases [49]. For graph isomorphism with N nodes, the best case time complexity of the algorithm is $O(N^2)$ and the worst case is $O(NN!)$. Similarly, for monomorphism from a graph with N_1 nodes to a graph with N_2 nodes, the best case time complexity of the algorithm is $O(N_1N_2)$ and the worst case is $O(N_1N_2!)$. In addition, VF2 has a memory complexity of $O(N_1)$, making it useful for working with large graphs.

If the consistency check for a particular view fails, the architecture framework should inform the user about which elements caused the inconsistency. This is achieved by returning the maximal set of correctly mapped elements between the view and BA. Finding this set is exactly the problem of finding the MCS between the view and BA graphs. Hence, our tool framework currently implements three types of graph matching algorithms: a monomorphism and an isomorphism algorithm (based on VF2) to check view conformance

and view completeness, respectively, and our MCS algorithm (described in the next section) to highlight the set of consistently mapped elements between the view and the BA.

The main drawback of graph pattern-matching lies in its inherent computational complexity. The monomorphism and subgraph isomorphism problems are known to be NP-complete, while the complexity class of the isomorphism problem is not yet known [50]. The problem of finding an MCS of two graphs is NP-complete since it can be reduced to finding the maximum clique (i.e., a fully connected subgraph) in a suitably defined association graph of the two graphs [51]. However, we believe that graph matching algorithms can be practically used for consistency checking because of the following properties of architectural graphs:

- Every vertex and edge of an architectural graph is typed. This allows pruning of a large number of possible but incompatible mappings and eliminates unsuccessful search branches early in the exploration.
- The view-to-base element mappings defined by the architect form a starting point for the algorithms to begin searching. Just like typed elements, the mappings also allow pruning of the search space.
- Architecture graphs can never be completely connected. Hence, they do not represent the worst-case class for graph algorithms. In an architecture, components can connect to each other only through connectors. Ports can only attach to roles and only be contained by components. Similarly for connectors and roles. Their inherent structure can be leveraged to explore the search space more efficiently.
- Architecture graphs are bounded valence graphs, i.e., every node has a number of edges lower than a given threshold, called the graph's valence. This is because every component has to have a maximum number of ports in it, defined by the style of the design concern. Similarly, every connector has a bounded number of roles. Each port and role have a single attachment edge between them. So, the number of edges

for any vertex in the graph is bounded. For this class of graphs, a polynomial-time algorithm exists for checking isomorphism [52].

- The hierarchical organization of a system's architecture is reflected in the graph representation. Each level in the hierarchy contains a small number of elements compared to the total number of elements in the complete nested architecture. This ensures that the level-by-level graph comparison between view and base graphs is practical for large-scale systems.

5.5 MCS Algorithm

In this section, we describe the details our MCS algorithm, which is based on depth-first search with backtracking, to find the maximal set of consistent elements between two graphs.

5.5.1 Data Structure

Our MCS algorithm is based on the VF2 algorithm's search structure, combined with an efficient data structure (introduced in [53]) to store the set of compatible nodes from the second graph G_2 for each node in the first graph G_1, in each state of the MCS search. The data structure is called a *Vertex Matching Matrix* (VMM), and is implemented as an array of dynamic lists, with one list for each node in G_1. Each list contains the nodes from G_2 that are compatible (in both type and structure) to the corresponding node in G_1. The VMM is initialized before the MCS algorithm begins the search phase by creating the set of all type compatible nodes from G_2 for each node in G_1. For the MCS algorithm given in [53], there is no check for structural compatibility of nodes during the VMM initialization phase, since no nodes have been matched by the MCS search phase yet.

In our MCS algorithm, we leverage the pre-mapped components and connectors between the view and BA graphs to restrict the ports and roles in the BA that are structurally compatible with those in the view. For example, if a component is pre-mapped in the view

graph, the ports of that component are structurally compatible only with the ports of the corresponding component in the BA graph. Hence, we add only those ports to the node list in the VMM, because we are dealing with architecture graphs with a well-defined structure. Similar optimizations are carried out for pre-mapped connectors, and their attached roles.

In the worst case, where both the graphs are untyped, the initial size of the VMM is N_1N_2, where N_1 is the size of G_1 and N_2 is the size of G_2. Hence, the memory complexity of the algorithm is $O(N_1^2N_2)$ in the worst case, and $O(N_1)$ in the best case. The best case represents the scenario when every node is pre-mapped between the two graphs, and each row in the array contains a single element. Since we are dealing with typed graphs, and because of the pruning offered due to pre-mapped elements, the worst-case scenario cannot occur. In fact, during all our performance tests, we noticed that the size of the VMM reduces quite rapidly as the search proceeds further down any branch, since incompatible nodes are quickly discarded as the common subgraph grows in size.

We can calculate the effect of pre-mapping by introducing the parameter p, the number of pre-mapped nodes between the two graphs. With p nodes pre-mapped, the new size of the array becomes $(N_1 - p)(N_2 - p)$ and we have to go down $N_1 - p$ levels to match the remaining nodes. So the memory complexity bound reduces to $O[(N_1 - p)^2(N_2 - p)]$. By pre-assigning a large number of nodes between the graphs, the size of the VMM can be reduced substantially.

5.5.2 Search Strategy

A pseudo-code description of the MCS algorithm is shown in Listing 1. The algorithm performs a depth-first search, with a heuristic for pruning unfruitful search paths. Each state s in the MCS algorithm represents a common subgraph of the two graphs. This common subgraph is part of the maximum common subgraph to be eventually formed. The first state is the empty state, in which no nodes have yet been matched. In each state, a pair of nodes ($n1$,$n2$) to be tried next is selected through the function *NextPair(s,n1,n2)*.

The node $n1$ belongs G_1 and $n2$ belongs to G_2. Given an $n1$, the next node in the VMM's list for $n1$ is selected to be $n2$. If all compatible nodes for $n1$ have been tried, a special node called the *null_node* (labeled as ϕ) is matched to $n1$. A *null_node* mapping signifies that $n1$ cannot be mapped to any node in the current search branch, and is a mechanism to allow the search to proceed to the next state in that branch. If all nodes from $G1$ have either been matched or no further compatible matching exists, then the current state is a leaf state of the search tree. In this case, if an MCS has been found in this branch, it is added to the set of all MCSs found so far by the function *AddToSet*. The function adds unique MCSs to the set by checking that no existing MCS in the set is a copy of the one to be added.

Algorithm 1 Calculate all maximum common subgraphs between two graphs

```
procedure FindMCS(State s)
begin
while  NextPair(s, n1, n2) do
  if IsFeasiblePair(s, n1, n2) then
    State s1 = AddPair(n1, n2)
    if  Size(s1) >= currentSize then
      currentSize = Size(s1)
      currentState = s1
    end if
    if IsExpandable(s1) then
      FindMCS(s1)
    end if
    BackTrack(s1)
  end if
end while
AddToSet(currentState)
end
```

The selected node pair is analyzed through the function *IsFeasiblePair(s,n1,n2)* that checks whether it is possible to extend the common subgraph represented by the current state by means of the new pair, so obtaining a larger common subgraph. The function checks if there are any edges from the node $n1$ to the set of nodes $\{n_1^i\}$ of G_1 in the common subgraph. For each such edge, there should be a corresponding edge from node $n2$ to the

node n_2^i of G_2 that is mapped to n_1^i of G_1 in the common subgraph. The same check is carried out for all edges from $n2$ to $\{n_2^i\}$ of G_2 in the common subgraph. If the pair can be added, the function *AddPair(n1,n2)* extends the current partial solution by adding ($n1$,$n2$) to the common subgraph, thereby creating a new state $s1$ of the search space.

The VMM for $s1$ is refined in *AddPair(n1,n2)* by eliminating from each list, all nodes from G_2 that have become structurally incompatible with the corresponding node from G_1 because of the addition of the pair ($n1$,$n2$). The node $n2$ is also eliminated from the lists of all unmapped nodes in G_1, since any node from G_2 can only be mapped once to a node from G_1 in any search branch. Refining the VMM in each new state can drastically reduce the number of node comparisons performed at each level as the search proceeds. The size of the largest common subgraph found so far (stored in *currentSize*) is compared to the current common subgraph, and is updated if the current mapping is larger.

The function *IsExpandable* checks whether the exploration the state $s1$ further will possibly result in a common subgraph of the same size or larger than the ones found so far. This is done by checking how many of the remaining (unmapped) nodes of G_1 in the VMM have non-empty lists, i.e., they have one or more possibly compatible nodes in G_2 in the current branch. If the number of such remaining nodes added to the size of the current common subgraph is less than *currentSize*, the state $s1$ is not explored any further. Otherwise, the $FindMCS$ procedure is recursively called with $s1$ as input. After the state has been analyzed, the *BackTrack* function is invoked to restore the common subgraph of the previous state, and to choose a different new state.

A portion of the MCS search space is given in Fig. 5.3 (b) for the pair of simple graphs in Fig. 5.3 (a). Each oval represents a state in the search space, i.e., a common subgraph between the two graphs. Red ovals represent a maximal common subgraph found during the search. The red crosses denote that the tried mapping was not successful, resulting in a backtracking to the previous state. Using this search strategy, whenever a branch of the

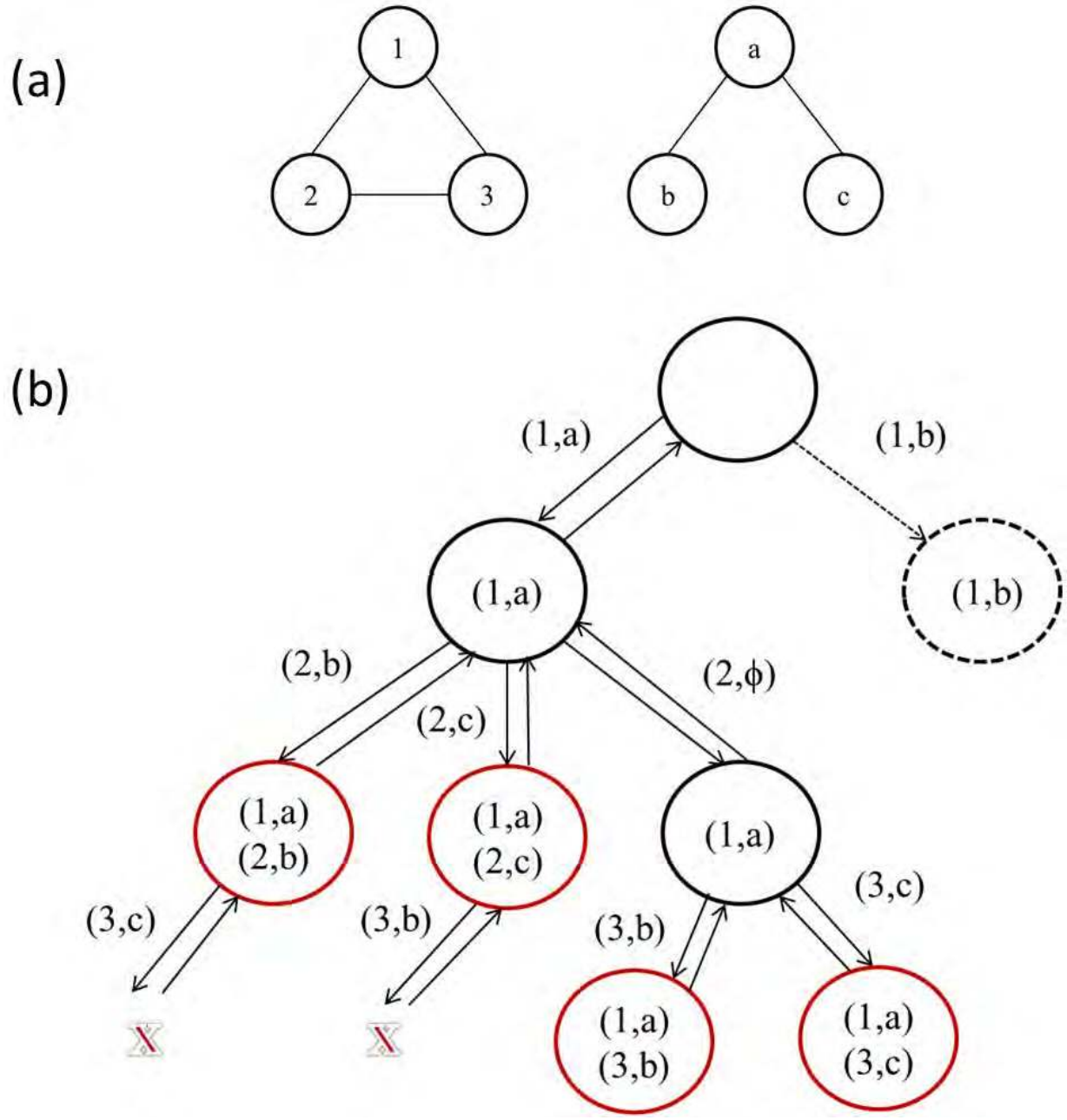

Figure 5.3: Search space of MCS algorithm.

search tree is chosen, it will be followed as deeply as possible until a leaf state is reached, or until a pruning condition is verified.

It is noteworthy that each branch of the search tree has to be followed because, except for trivial examples, it is not possible to foresee if a better solution exists in a branch that has not yet been explored. This full exploration of all the possibilities makes the MCS algorithm computationally expensive as the size of the input graphs grow. In order to use the MCS search practically, our algorithm allows the user to specify whether to return only the first MCS found or the set of all MCSs. Asking the algorithm to return only the first MCS allows it to prune any branches that do not contain a subgraph that is larger than the current one, and results in a much faster search time, as demonstrated in Sec. 5.6.

5.6 Performance Evaluation

To evaluate the performance of our MCS and monomorphism algorithms, we have run each algorithm on a set of pairs of randomly generated typed graphs. Our graph generator allows us to create graphs with the following parameters: number of node types, edge density, size of common subgraph, and number of pre-mapped nodes. The number of node types defines the size of the set of unique labels for the nodes and is defined as a ratio of the size of the parent graph. Every node in a graph is randomly assigned a type from this set based on a uniform distribution. The edge density defines how dense the graph is and is calculated based on the edge probability η (details are given in subsection 5.6.1).

Every graph created by our random graph generator contains a common subgraph embedded in it. The common subgraph is a ring structure whose size can be varied based on the number of nodes of the parent graph. Embedding a known subgraph in all created graph pairs guarantees that our algorithms return at least this graph upon completion. The size of the common subgraph is defined as a ratio of the size of the parent graph. The number of pre-mapped nodes defines how many nodes of the common subgraph in each graph are mapped beforehand, and is defined as a ratio of the size of the common

subgraph. For a given graph size, one hundred pairs of random graphs were generated, and the average time taken over all these pairs was taken as the performance time of the algorithm. All tests were run on an IBM Thinkpad laptop with an Intel dual-core 2.5 Ghz processor and 4 GB of RAM. The software environment was Windows 7 (64-bit) OS, with Eclipse Helios (3.6) as the execution platform.

5.6.1 Calculating Edge Density

Let N_C be the number of components and let each component have P number of ports. Similarly, let N_N be the number of connectors and let each connector have R number of roles. The total number of vertices in the corresponding architecture graph is $N_V = N_C(P+1)+N_N(R+1)$. Each component contributes P edges, since a component is attached to its P ports. Similarly, each connector contributes R edges due to the attached roles. We follow the convention that each port is attached to only one role, and hence contributes a single edge. The edges from each role to its connector are already accounted for in the edge count for each connector. Hence, the total number of edges is $N_E = 2(N_C P) + N_N R$. The binary connector (i.e., a connector with two roles) is the most common in architectures. With the restriction that a port is always connected to a single role, the number of binary connectors is always $N_N = \frac{N_C P}{2}$ and $R = 2$. Hence, for architecture graphs with binary connectors, we have: $N_V = N_C(P+1) + \frac{N_C P}{2}.(2+1)$ with $N_E = 3N_C P$.

An undirected fully connected graph with N vertices has $\frac{N(N-1)}{2}$ edges. We use an *edge probability* parameter, η, to create graphs with N vertices and a random number of edges, given by $E = \eta.\left(\frac{N(N-1)}{2}\right)$. A value of 0 for η results in an unconnected graph (N isolated nodes) while a value of 1 results in a fully connected graph. To arrive at a representative value of η for architecture graphs with binary connectors, we calculate η for various numbers of components and ports. Consider an architecture with 10 components with 5 ports each, which implies 5 binary connectors. Hence, the number of edges N_E is 150 and $\eta = \frac{2N_E}{N(N-1)}$ is 0.0166. However, as the number of components increase, η

reduces rapidly, since architecture graphs are essentially sparse. For an architecture with 25 components with 10 ports each, η is 0.0036, while for 50 components with 10 ports each, η is 0.0018. In [54], a detailed evaluation of a number of MCS algorithms on a large database of random graphs was made. The edge probability parameter for sparse graphs in the database was chosen to be 0.05 to keep the running times of the experiments within practical bounds. Hence, we choose η to be 0.05 for the performance evaluation of our algorithms as well.

5.6.2 MCS Performance

For tests with the MCS algorithm, the number of nodes in the graph was varied from 10 to 100. We chose 40 as the size of the largest graph for the first test to keep the running time of the test (with 100 iterations for each graph size) within practical bounds. Most standard performance benchmarks for MCS algorithms (without pre-mapping of nodes) use between 30 to 40 nodes as the graph size for the same reason [54]. We specified that the algorithm return the first largest MCS found.

The first test evaluates the impact on the performance as the size of the common subgraph between the two graphs is changed while keeping the other parameters constant (no pre-mapped nodes and a fixed size of fifteen for the set of node types). In particular, we increased the size of the common subgraph from 25% to 75% of the parent graph size and plotted the performance in Fig. 5.4. The result shows that as the size of the common subgraph gets larger, the MCS algorithm runs to completion more quickly. This is explained by the observation that once a large common subgraph is found during the search, any branches containing smaller subgraphs are pruned away quickly by the algorithm. This result suggests that views that focus on a very small part of the BA will take longer for the MCS checker to find the set of common elements, owing to the relatively smaller common subgraph between the view and BA graphs. Views that relate to a large portion of the BA are easier for the MCS algorithm to deal with.

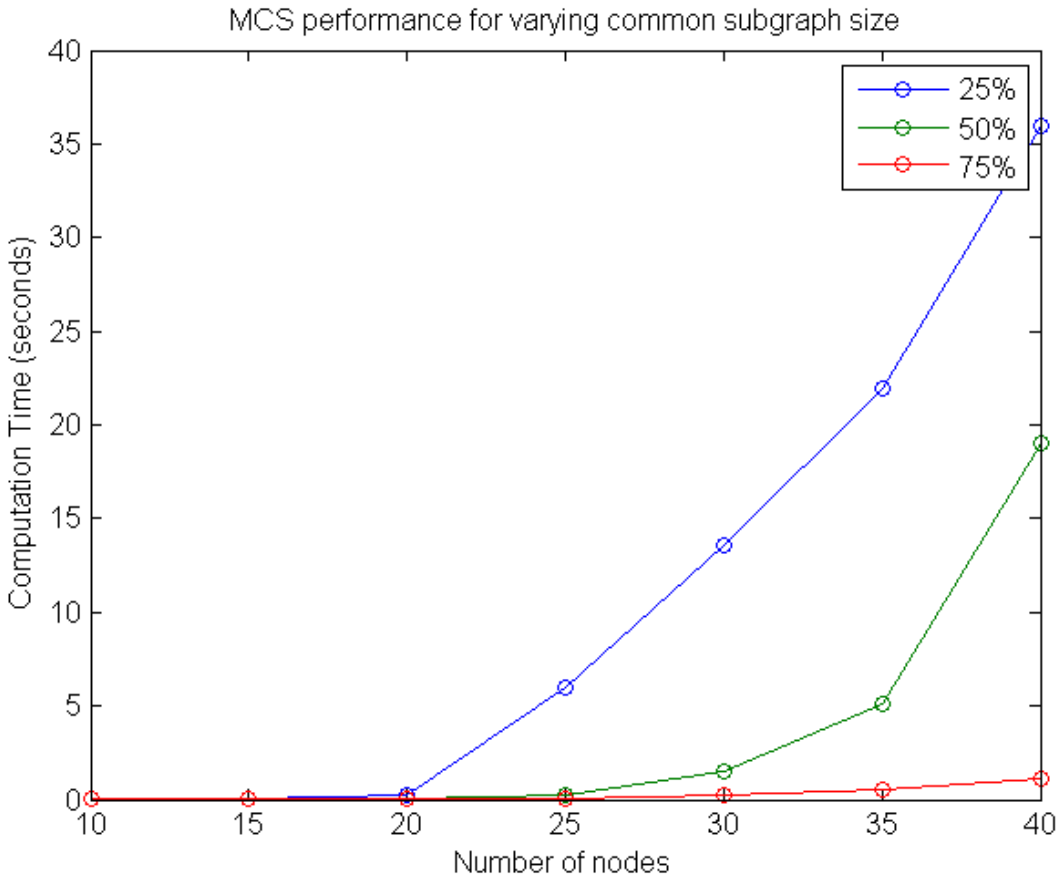

Figure 5.4: MCS performance with varying size of common subgraph.

The second test evaluates the impact of pre-mapping certain nodes in the common subgraph between the two graphs. The size of the common subgraph was kept at 75% of the parent graph size, with fifteen as the size for the set of node types. We varied the number of pre-mapped nodes from 25% to 75% of the size of the common subgraph and plotted the performance in Fig. 5.5. The result shows that the running time of the algorithm reduces substantially as the number of pre-mapped nodes increases. This is explained by the observation that as more nodes are pre-assigned between the two graphs, the initial size of the VMM becomes smaller. As the search progresses, the algorithm is able to rapidly prune search branches much earlier by taking advantage of the extra mapping information for the nodes. We see that the graph for 75% common subgraph without any nodes pre-mapped (in Fig. 5.4) has a much larger running time than any of the times for the same common subgraph size with pre-mapping information included. This result suggests that pre-mapping view elements has a significant benefit on the performance of the MCS algorithm. There is a trade-off between the number of view elements that a designer should pre-map to how long the algorithm will take to find the consistent elements.

5.6.3 Monomorphism Performance

For tests with the monomorphism algorithm, the number of nodes in the graphs were varied from 50 to 500, and the first monomorphism found was returned. The running times for the algorithm are faster compared to those for the MCS search because a large number of branches can be pruned very early in the monomorphism check, since the algorithm is not trying to construct a common subgraph. As for the MCS test, we increased the size of the source graph from 25% to 75% of the target graph size and plotted the performance in Fig. 5.6. The result shows that the running time of the algorithm increases as the size of the source graph increases. This is explained by the observation that it is easier for the algorithm to find a smaller matching, compared to a larger matching. As a best-case example, let the source graph be of size one (one common node of the same type between

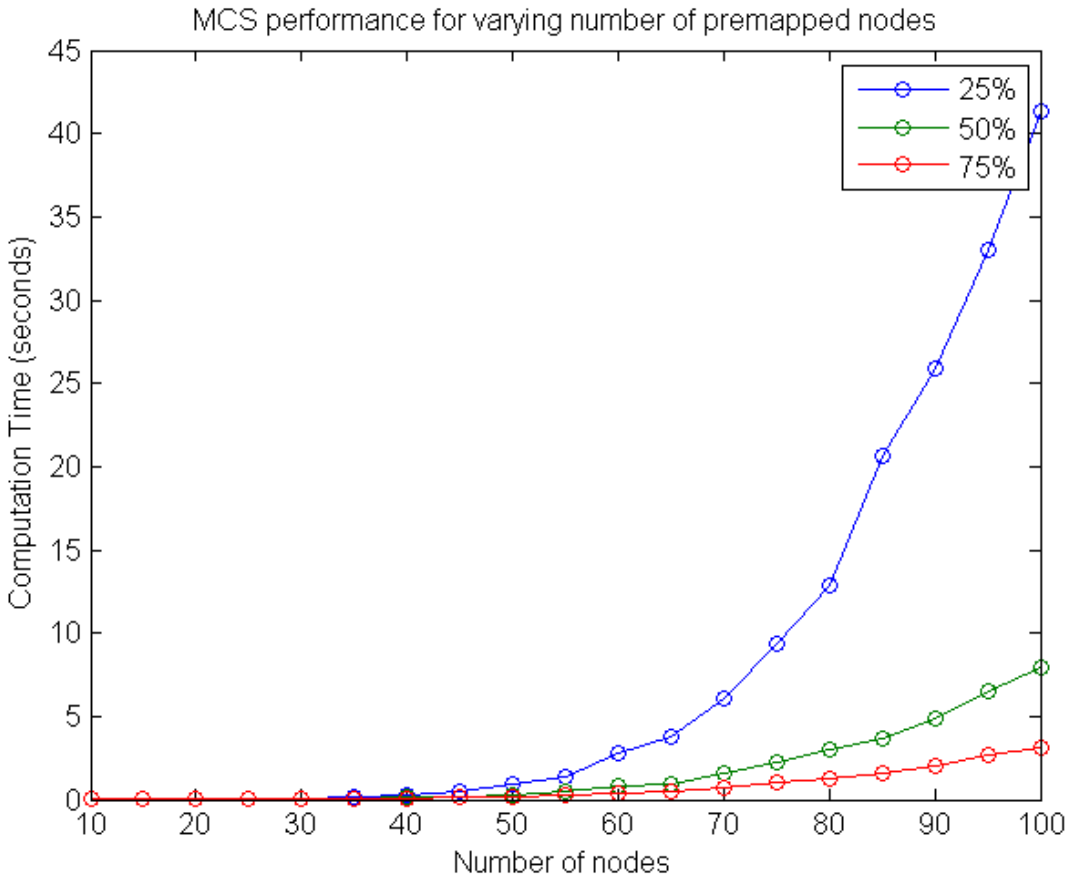

Figure 5.5: MCS performance with varying size of pre-mapped nodes.

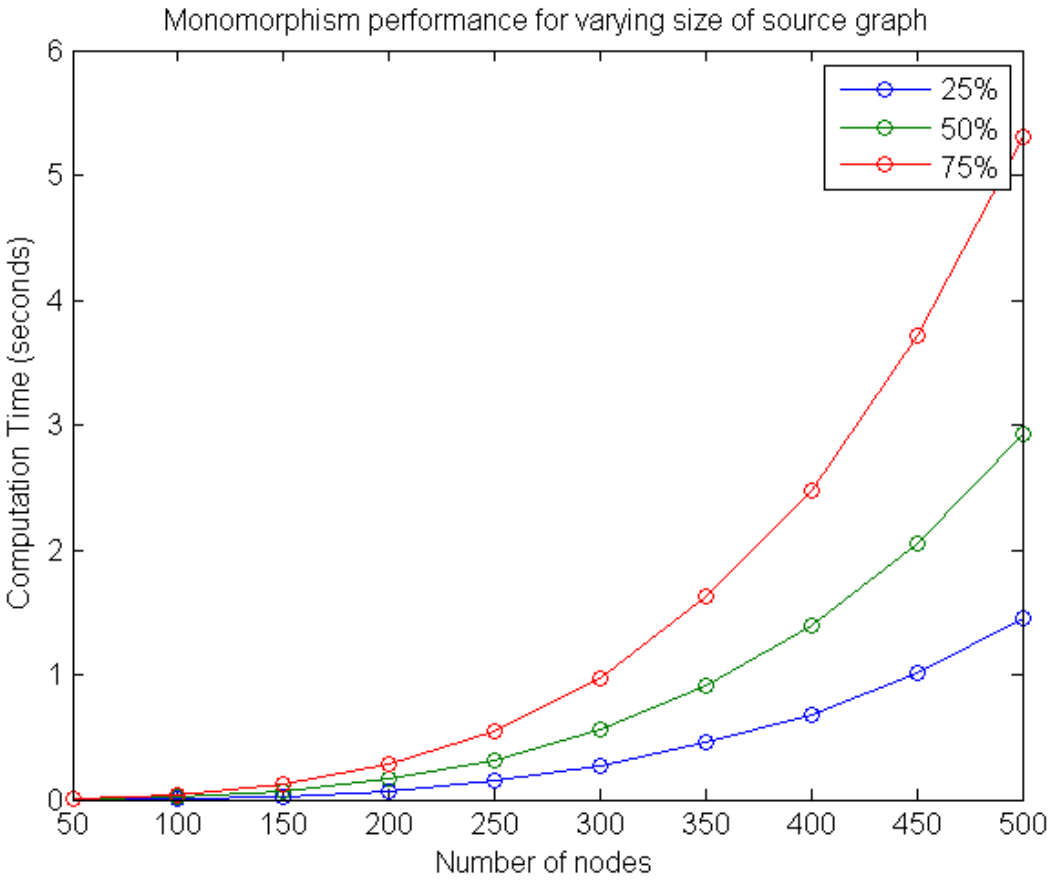

Figure 5.6: Monomorphism performance with varying size of common subgraph.

the two graphs). The algorithm can quickly find the common node and return the result. However, as the source graph grows, a larger number of tests have to be carried out for the structural compatibility of newly added nodes from the target graph. This leads to longer running times, as reflected in the results.

5.7 Tool Framework for View Consistency

The Acme view editor allows the user to check for consistency between the view and BA, once required mappings between them have been defined using the map creation capabilities of the editor. The user brings up a *consistency checker* pane with buttons for the following options:

- Run a view conformance or view completeness check for the current view.
- Run the MCS algorithm to find the set of maximally matched elements, if consistency checking has failed.
- Add the complete system map returned by the consistency checker or MCS algorithm to a map file.
- Add a selected element map (component or connector) to a map file.

In Fig. 5.7, a screenshot of the view editor containing the hardware view and BA is shown, along with the pane containing options for running the graph consistency checkers. The user clicks on the *Find Consistency* button to run the monomorphism check between the view and BA graphs. The graph checker runs as a background process so that the user can still use the editor to perform other tasks. Since the hardware view is structurally consistent with the BA for this example, the graph checker returns a set of successful mappings between the view and BA. In the current example, we have restricted the checker to return a single mapping (*Option 1* in Fig. 5.8).

The map information includes the names of the mapped elements, along with the associated map type of each map instance. Clicking on any map instance highlights the

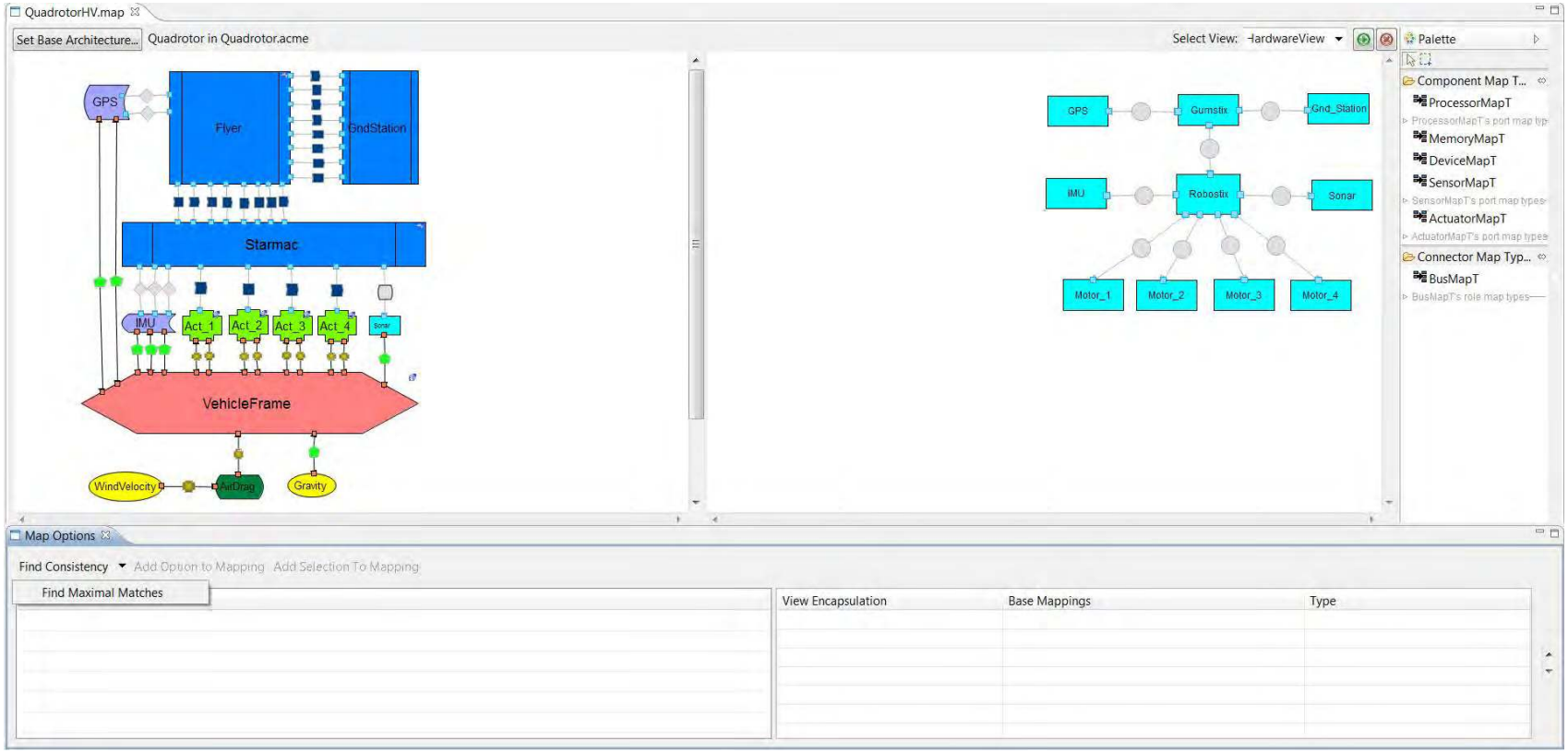

Figure 5.7: Running consistency checks with the AcmeStudio view editor.

corresponding elements visually as is seen in Fig. 5.8. In this case, the highlighted map shows that *Motor_1* in the BA has been mapped to *Act_1* in the hardware view using the *ActuatorMapT* maptype. The user can decide to add this mapping to the current map file or delete the mapping and run another type of consistency check. The user also has the option to define new element mappings at this state. The tool automatically deletes all those mappings from the set that do not contain the newly defined map(s). This feature allows the uses to eliminate undesired mappings from the set of results, based on new information.

5.8 Summary

In this chapter, we define *structural consistency* of architectural views with the BA as the existence of a morphism relation between the graphs of the view and BA. We introduce *view conformance* and *view completeness* as two types of consistency conditions, based on whether a monomorphism or an isomorphism exists between the view and BA graphs, respectively. We describe our graph morphism and MCS algorithms that are used to check for consistency or find the set of maximal consistent elements between the view and BA, respectively, and evaluate their performance on a set of randomly generated graphs. We also describe our tool framework for view consistency support in AcmeStudio that allows the designer to run consistency checkers and visually edit the element maps between a view and the BA.

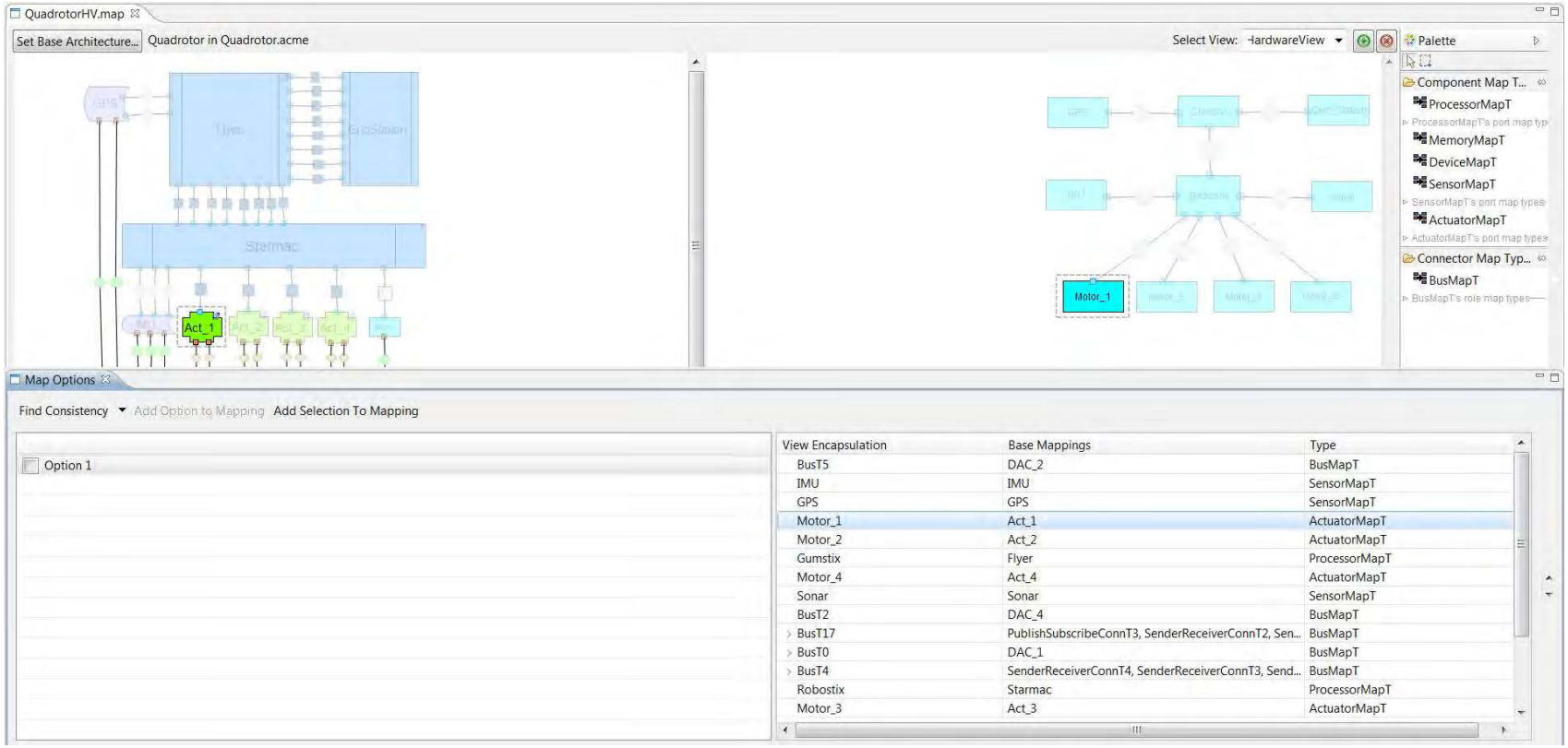

Figure 5.8: Displaying consistent mappings with the AcmeStudio view editor.

Chapter 6

Case Study I : STARMAC

In this chapter we present the first of two case studies that demonstrate the application of our approach to real-world systems. The case study is based on an autonomous avionics system, the STARMAC quadrotor, and is an example of how our architectural approach can be applied to an existing system for which legacy models and implemented code are already present. We first create multiple views of the quadrotor that are derived from the heterogeneous models used for the system's analysis and design. We then check the consistency of each architectural view with the quadrotor's base architecture using our tool framework in AcmeStudio. We elaborate on the mismatches discovered and their impact on the integrity of the implemented system.

6.1 Introduction

The Stanford Testbed of Autonomous Rotorcraft for Multi-Agent Control (STARMAC) [3] is a quadrotor platform developed to test algorithms that enable autonomous operation of aerial vehicles. As shown in Fig. 6.1, the aircraft has four rotors arranged symmetrically about its body frame. The rotors are powered by lightweight, brushers DC motors with a maximum thrust of 8 Newton per rotor. The body frame is a light-weight, custom-made structure to securely contain the controller boards, sensor suite, and batteries.

The STARMAC has a hierarchical control system, with a low-level attitude controller

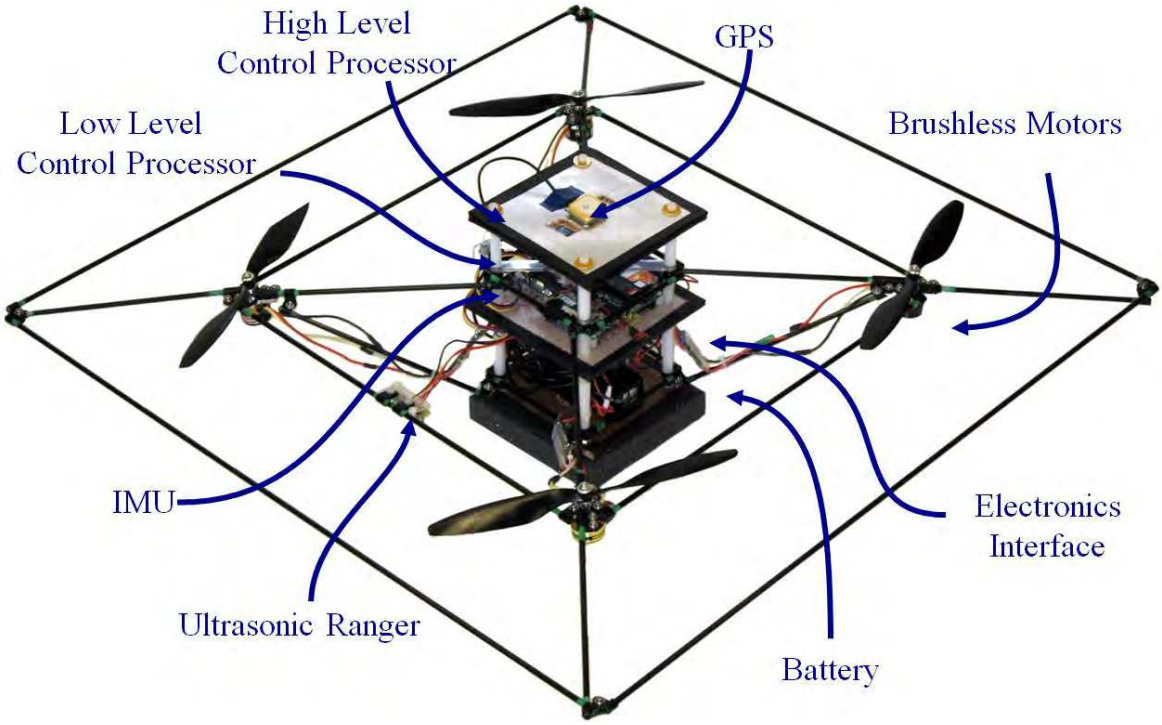

Figure 6.1: The STARMAC quadrotor [3].

(AC) running on a Robostix embedded processor, and a high-level position controller (PC) running on a powerful Intel Atom-based board (or a Gumstix or PC104 board variant). The different hardware variants used for the PC reflect the capabilities needed for different mission scenarios. 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. The two onboard controllers communicate through a serial link connecting the Robostix and Atom boards. Communications between the vehicle and the GSC are managed over a WiFi network, using the UDP protocol.

The STARMAC design team has documented the software subsystems and the hardware architecture of the vehicle. The complete source code for the software running on the Robostix and Atom boards as well as the GSC were made available to us. In addition, an

existing detailed Simulink model (henceforth called SM-1) for the closed-loop quadrotor system containing position and attitude controllers, a six degree-of-freedom (DoF) plant model with nonlinear blade-flapping dynamics, and sensor and actuator models was provided by the quadrotor's control design team. The SM-1 model includes certain platform-level details which are reflected in the model architecture, with subsystems for the Robostix and Atom boards, as well as for the GPS and IMU sensor modules.

Hence, the quadrotor design artifacts include the complete source code, a Simulink model, papers describing the vehicle dynamics with equations of motion, and hardware configuration diagrams. In the following section, we describe the creation of the BA of the STARMAC quadrotor using the available design artifacts.

6.2 Base Architecture

The BA of the quadrotor is created in the CPS style, which allows for the unified representation of the cyber components (control algorithms and real-time software) and the physical dynamics (forces and torques acting on the vehicle frame). Figure 6.2 illustrates the use of the CPS style to model the quadrotor in AcmeStudio. On the cyber side, each controller in the system (AC, PC, and GSC) is mapped to a separate computation component (*Flyer*, *Starmac*, *GndStation*) that implements the control algorithm. The names of the components are derived from the names of the corresponding main threads in the source code. 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. Each cyber port represents a distinct data unit exchanged between components. The names and types of ports are derived from the names and types of data being passed

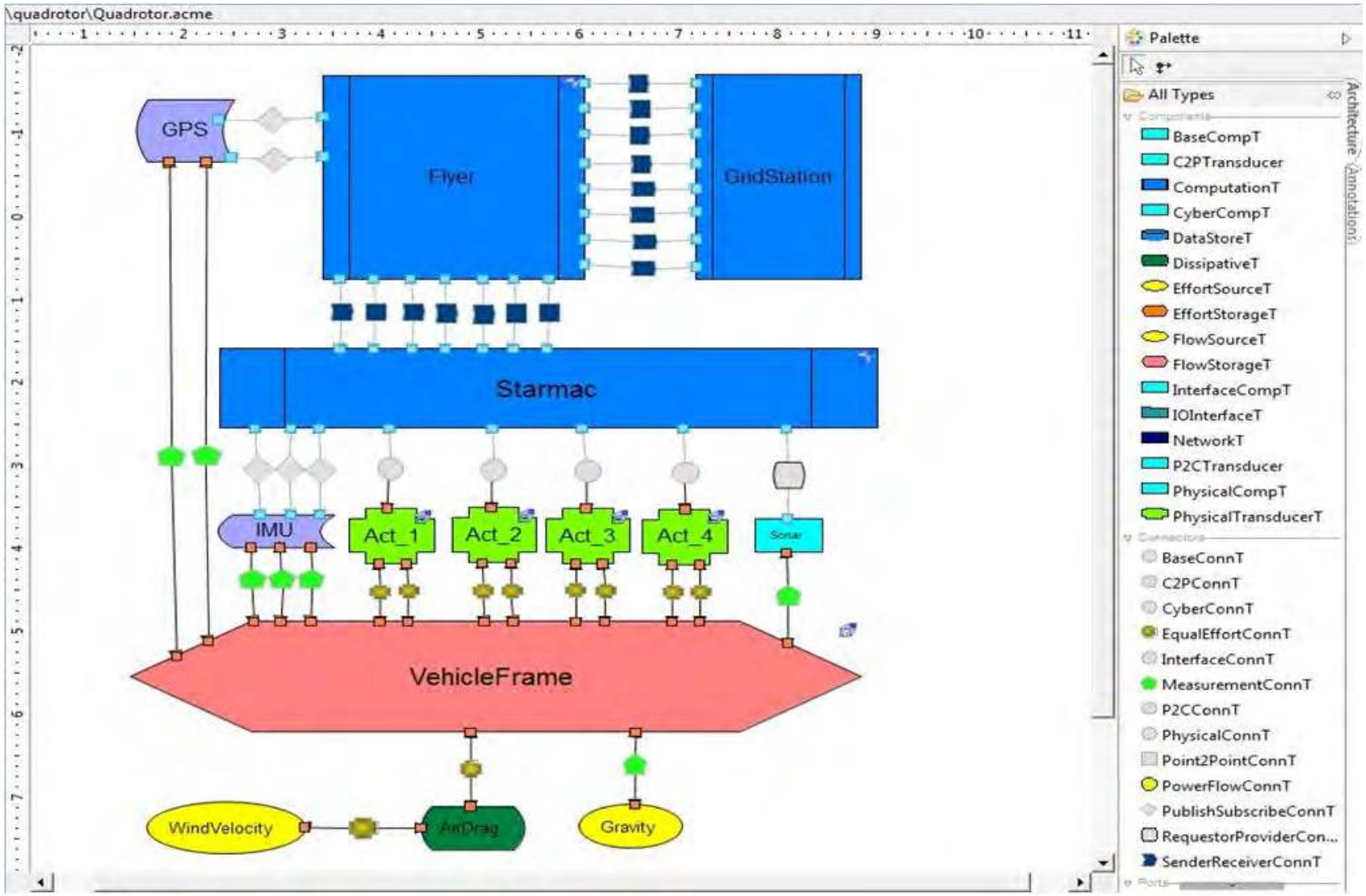

Figure 6.2: Creating the base architecture of the STARMAC.

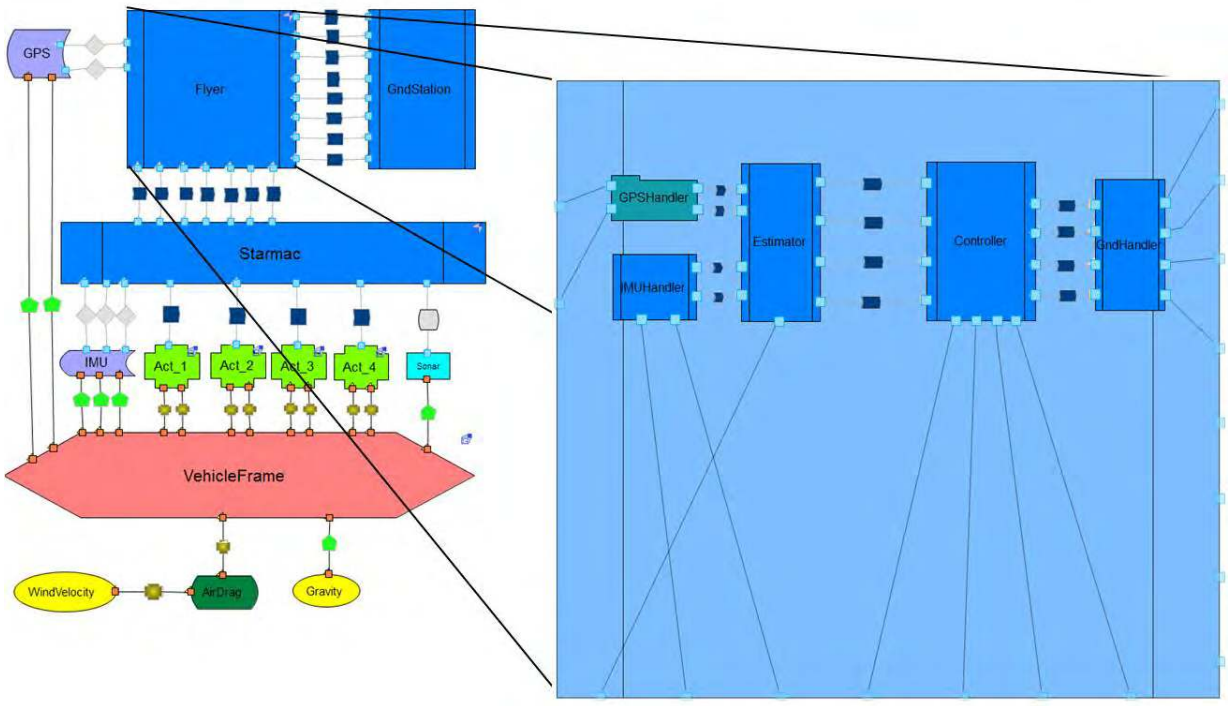

Figure 6.3: Acme representation of the *Flyer* component.

between the corresponding processes in the source code.

We model the quadrotor's physical architecture from first principles, and by studying the six DoF vehicle subsystem in the SM-1 model. The vehicle frame is modeled as a rigid-body mechanical component, whose mass and moment of inertia (MI) are affected by the forces and moments acting at its physical ports, according to the dynamic equations of motion. Each rotor assembly and motor actuator (with its controller) is modeled as a single C2P transducer called *Act*, containing an input cyber port and two mechanical ports, one each for the translational and rotational domains. The *Act* component models the conversion of cyber actuation commands from the attitude controller to an output thrust (force) and torque acting on *VehicleFrame*. Each *Act* is connected to the *VehicleFrame* by two *equal position* physical connectors, one for force balance and one for moment balance. These connectors model the action and reaction mechanical coupling between each rotor assembly and the vehicle frame.

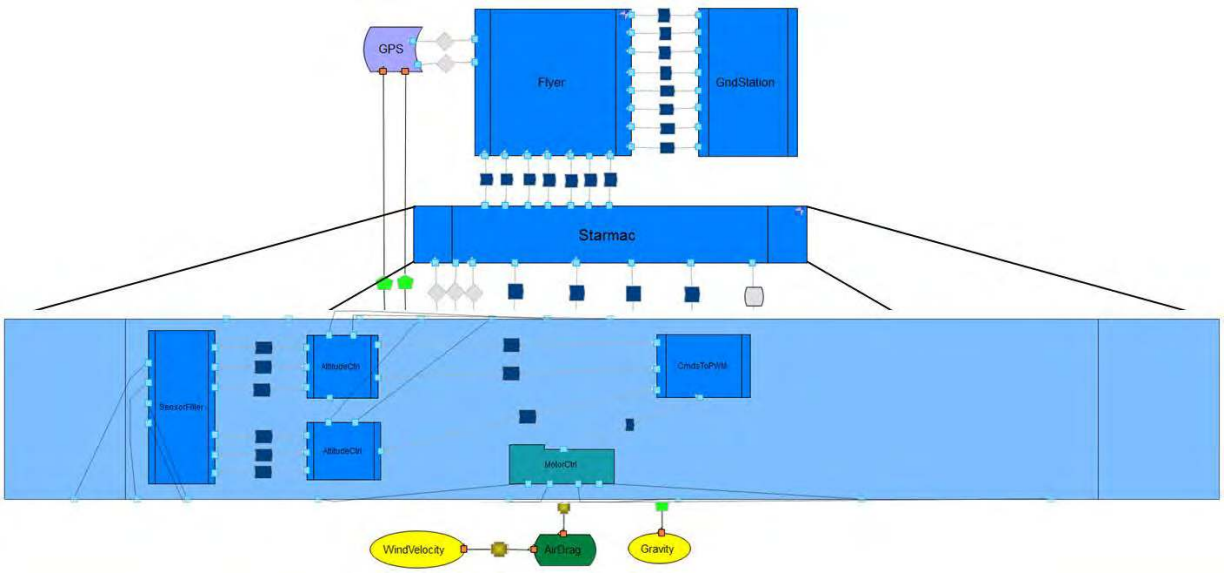

Figure 6.4: Acme representation of the *Starmac* component.

The drag force is described as a dissipative component, whose magnitude depends on the wind velocity (modeled as an effort source) 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 downward force on the quadrotor, and is connected to the vehicle frame by a physical connector. The IMU, sonar, and GPS are all 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 component is annotated with device parameters including detection beam width, effective range, and resolution. The IMU and GPS devices are similarly annotated with their performance parameters.

Several components in the BA are refined further to add structural detail to the architecture. For example, each of the cyber controllers has an Acme representation that contains components for the various threads or functions that make up the code for each

controller. The internal structure of the *Flyer* component is shown in Fig. 6.3. Since the *Flyer* is a multi-threaded process, each sub-component represents a single active thread. The various *Handler* threads (IMU,GPS,Ground) are created for the concurrent communication between the *Flyer* and other devices on the quadrotor. There is also a main *Controller* thread for position control. The *Estimator* thread implements an extended Kalman filter that is used to combine GPS and raw inertial measurements for accurate full-state estimation. The internal structure of the *Starmac* component is shown in Fig. 6.4. The *Starmac* code is implemented as a single thread and each sub-component represents various functions in the thread. The ports on the sub-components represent the parameters passed between these functions.

Similarly, there are Acme representations for each of the *Act* components, and for the *VehicleFrame*. Each *Act* representation (shown in Fig. 6.6) has a sub-structure with a Pulse-Width Modulation (PWM) motor controller, a motor, and a rotor as separate components. The PWM controller is a C2P transducer that converts the controller's actuation command into a physical voltage that drives the motor. The motor component converts an input voltage into a torque output at its shaft. The torque is transmitted through the physical coupling connector to the rotor component's mechanical input port. Based on the torque from the motor and the aerodynamic parameters of the rotor blades, the rotor transmits a resultant force and torque on the quadrotor frame. The *VehicleFrame* representation (shown in Fig. 6.5) currently contains a simple decomposition of the vehicle body into mass and MI components, with all input forces acting on the mass and all input torques acting on the MI.

6.3 Architecture Views

In this section we describe the creation of five views of the quadrotor, based on the models created in the software, control, hardware, and physical design concerns.

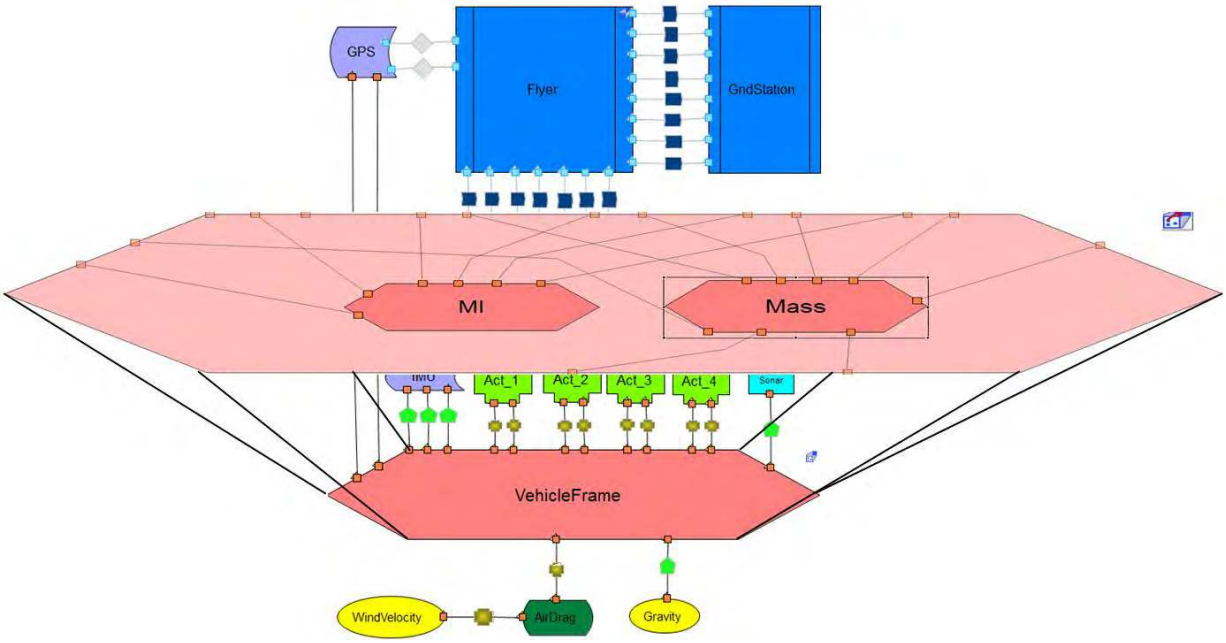

Figure 6.5: Acme representation of the *VehicleFrame* component.

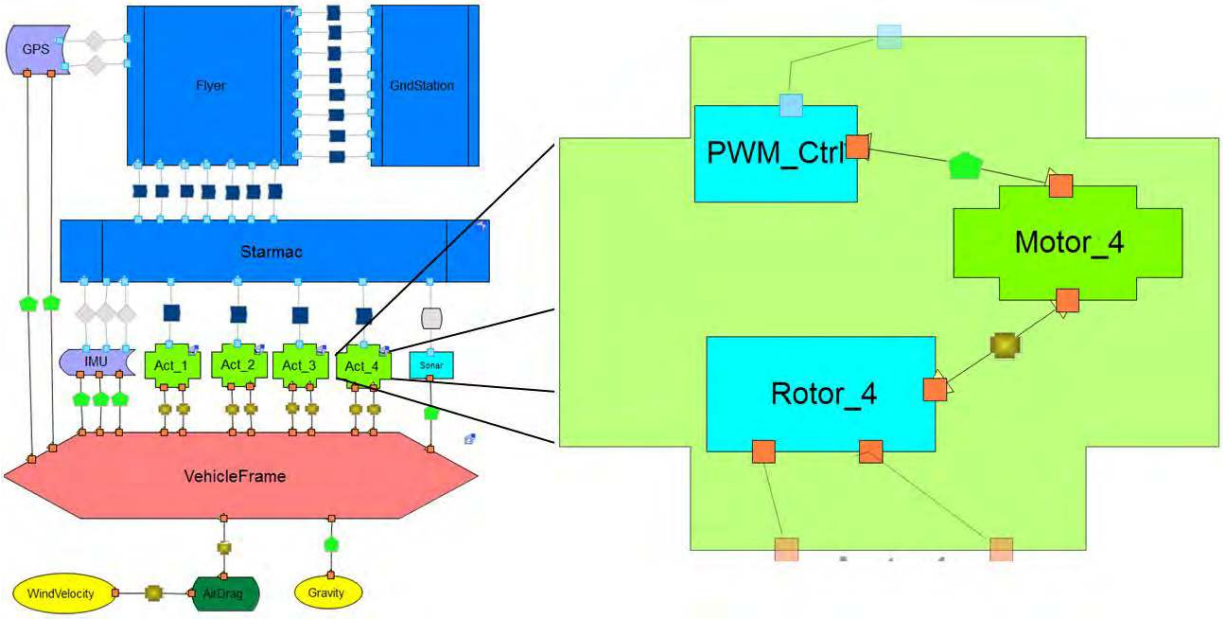

Figure 6.6: Acme representation of the *Act* component.

6.3.1 Software View

To study the safety properties of the control software implemented onboard the quadrotor, we have modeled the behavior of the source code as a Finite State Process (FSP) [55] specification. FSP is a process algebra where behavior is modeled in terms of event patterns, called *processes*, that denote sets of event traces. Each event in a trace represents a discrete transition of a system. 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 Labelled Transition System Analyzer (LTSA) is a verification tool that uses FSP to model the behavior of concurrent systems [55]. We use LTSA's capability to perform compositional reachability analysis to exhaustively search for violations of the following system safety property: the quadrotor should never receive a command to turn off all rotors while the vehicle is flying. The source code of the attitude controller contains logic (shown below) to turn off the rotors based on certain conditions being met, such as intermittent communication with the position controller or ground station.

```
if (refState.loopcount>MAX_LOOPS)
   {
     LED_ON(BLUE); // Indicate time out
     emergencyOff = TRUE;
   }
   else
   {
     LED_OFF(BLUE); // Indicate normal
     emergencyOff = FALSE;
   }
```

```
if ((refState.command.trigger!=1)||emergencyOff){
    for (motor=0; motor<NUM_MOTORS; motor++)
        setpoint[motor] = 0; // TURN OFF MOTORS
 }
```

The FSP model was created by studying the source code (written in C) of the programs that implement the onboard controllers and ground station, and the communication protocol between the controllers and with the sensors of the STARMAC. The behavior of each controller and sensor has been modeled by a primitive FSP processes, and the data exchanged is represented as shared events between the processes. The GSC process sends waypoints to the PC and accepts telemetry data from it. In addition, the GSC periodically sends a *trigger* packet to the AC through the PC as mechanism to check that the communication channel is active. The lossy nature of the wireless medium between the GSC and PC is modeled by allowing the GSC process to drop one or more packets (the event does not occur in the process). The PC relays the trigger packet as well as attitude setpoints to the AC and accepts feedback readings from it. Concurrently, the PC also reads in position data from the GPS. The AC waits for a setpoint from the PC, reads the IMU and sonar data, calculates the motor commands, and sends them to the quadrotor. If the AC does not get any trigger packet from the GSC or a setpoint from the PC within a certain interval of time, it sends a command to turn off all motors (and hence all attached rotors). This behavior is modeled with a *tick* event in the AC process, which leads to a set of *sendOff* events, once a certain number of *ticks* have occurred. Since the PC and AC are connected by a serial link, the communication between them is modeled with a lossless link i.e., the PC sends all the packets to the AC, and the AC never drops any information.

Each sensor is modeled as a simple FSP process which senses the state of the quadrotor process, generates a reading, and sends it to the appropriate controller. The quadrotor's complex dynamics are abstracted by an FSP process ($QUADROTOR$) with three states:

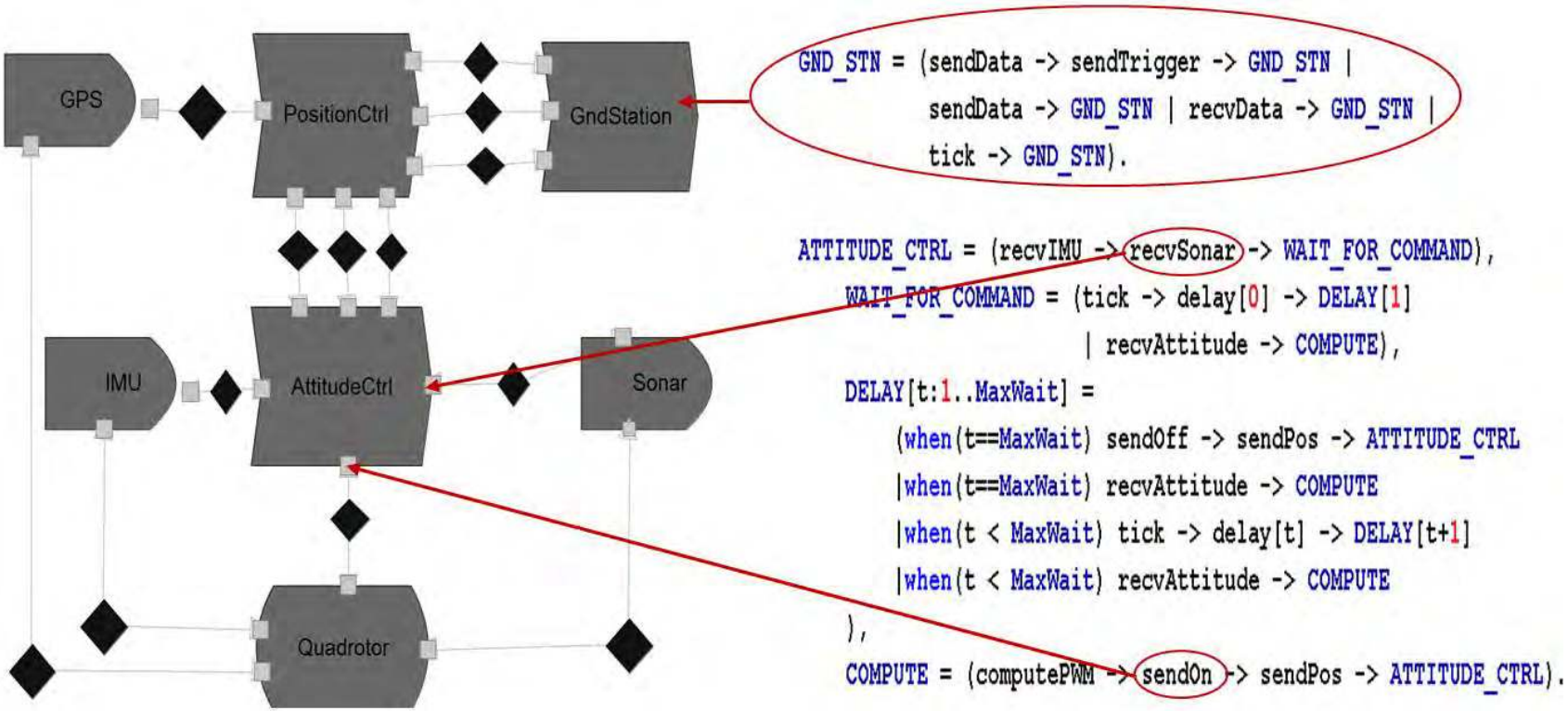

Figure 6.7: Creating software view from quadrotor FSP model.

Flying, *Not Flying*, and *Crash*. The process enters the *Crash* state any time a command to turn off all four rotors is received while the quadrotor is flying. On checking the safety condition of the FSP model with LTSA, we obtain a trace that confirms that quadrotor can receive a "rotors Off" command while it is flying. This is due to the lossy communication medium between the GSC and PC that allows the possibility of multiple trigger packets being dropped, leading to the AC timing out and commanding the rotors to turn off while the vehicle may still be in flight. This analysis shows how assumptions about the behavior of the communication channel (a cyber property) affects the controlled dynamic behavior of the quadrotor (a physical property).

The software view for the quadrotor is created by mapping each process in the FSP specification to a software component. Each port on a component represents the events that the associated FSP process can share with other processes. In Fig. 6.7, a portion of the FSP model is shown, where the *GND_STN* and *ATTITUDE_CTRL* processes are mapped to corresponding components of type *Controller*. Two shared events of the *ATTITUDE_CTRL* process (*recvSonar* and *sendOn*) are mapped to corresponding ports on the *AttitudeCtrl* component. A connector between two components represents the synchronization of shared events on the attached ports. For example, the shared *trigger* events between the *GND_STN* and *POSITION_CTRL* processes, and between the *POSITION_CTRL* and *ATTITUDE_CTRL* processes are represented by explicit connectors between the respective components in the view. In the general case, a connector could also be defined by an FSP process, e.g., a connector that describes the protocol for interaction between the position and attitude controller processes. The *QUADROTOR* process is mapped to the corresponding component of type *Device*, and represents the abstraction of the physical quadrotor behavior in relation to the software functionality.

The mapping between the software view and the BA of the quadrotor is shown in Fig. 6.8. Since the FSP model abstracts the data communication between the controllers

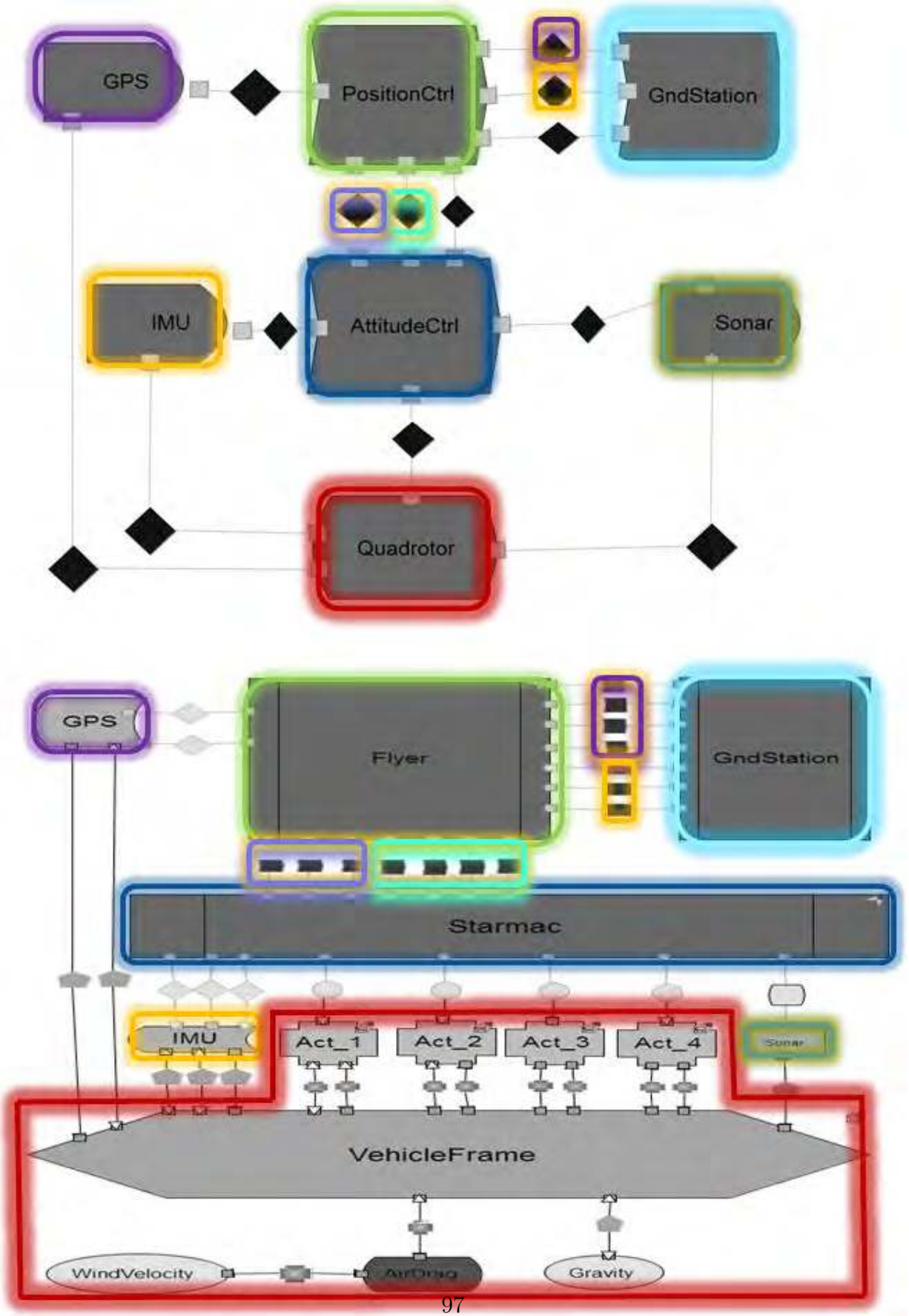

Figure 6.8: Mapping between software view and BA of quadrotor.

through single events, all data connectors between the *GndStation* and *Flyer* components in the BA are mapped to a single connector between the *GndStation* and *PositionCtrl* components in the view. A similar mapping exists for each set of multiple connectors between components in the BA, to a single connector in the software view. The encapsulation of connectors in this way reflects the abstraction made in the view about the details of the data being exchanged between software components. Since the focus of analysis in the FSP model is the communication protocol between controllers and the type of data being exchanged to maintain the safety property of not crashing, the connectors for telemetry and feedback data in the BA are abstracted away as single connectors in the view.

6.3.2 Control View

For the quadrotor, there are two control views, one for each existing Simulink model. The first control view corresponds to the SM-1 model (created by the Stanford team) that contains a detailed nonlinear dynamic model of the quadrotor, the inner and outer loop controllers, the ground station, as well as the dynamics of the IMU and GPS sensors. The creation of the control view from the SM-1 model is shown in Fig. 6.9. The structure of the Simulink model (and hence the view) reflects some of the hardware configuration of the quadrotor, with separate top-level subsystems for the *robo_stix* and *gum_stix* processors, as well as individual blocks for the two sensors. The vehicle dynamics are contained in the *starmac_dynamics* block. Each top-level Simulink block is mapped to a component in the control view, and each group of signal lines between blocks to connectors, resulting in the control view's $\mathcal{A}_V$, as shown in Fig. 6.9.

The mapping between the control view and the BA of the quadrotor is shown in Fig. 6.10. The physical components and connectors representing the dynamics of the vehicle frame in the BA are mapped to the single *Quadrotor* component in the view. This represents the encapsulation of the quadrotor's physical dynamics as a single block defined in the corresponding Simulink model. The *Quadrotor* component can internally contain details

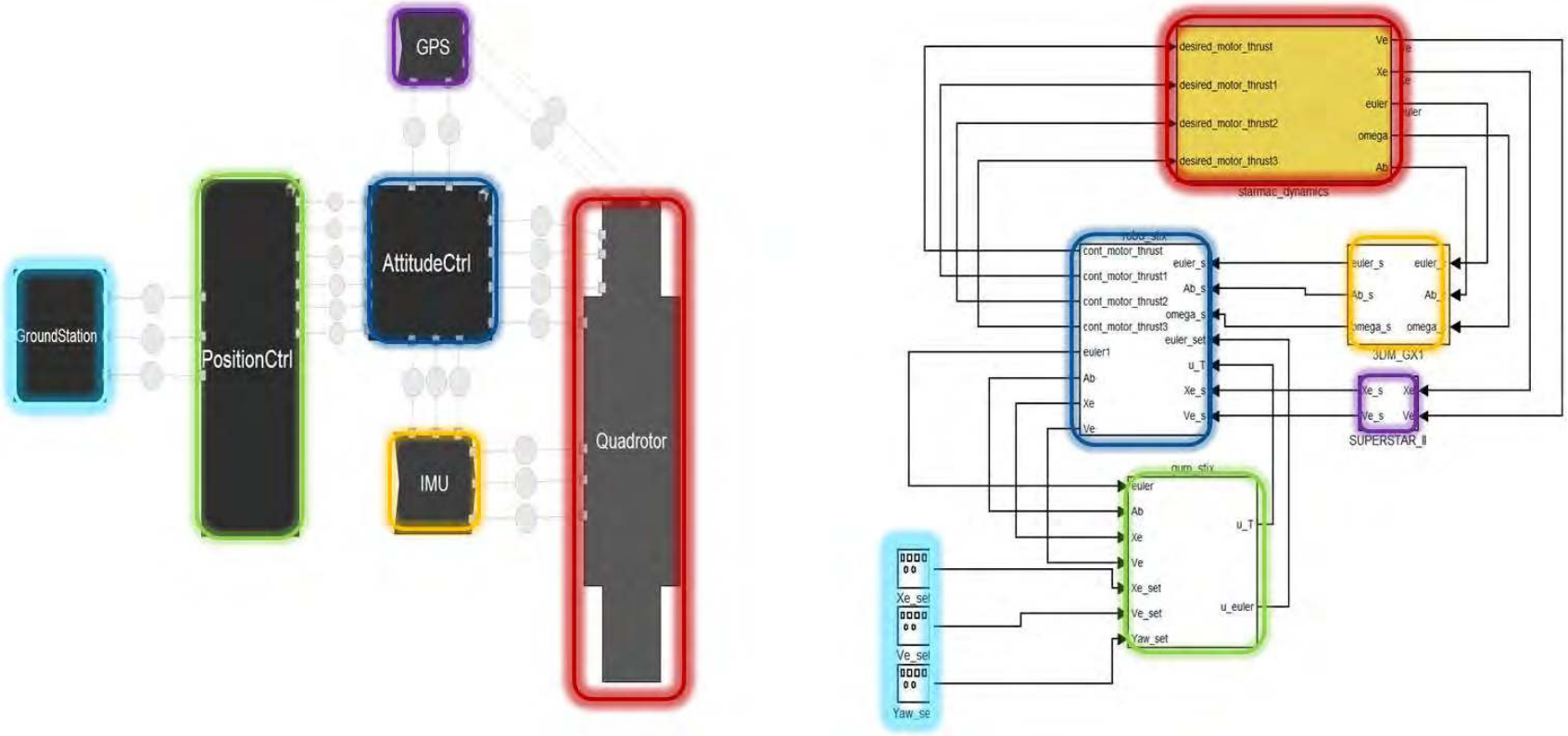

Figure 6.9: Creating control view from Stanford Simulink model.

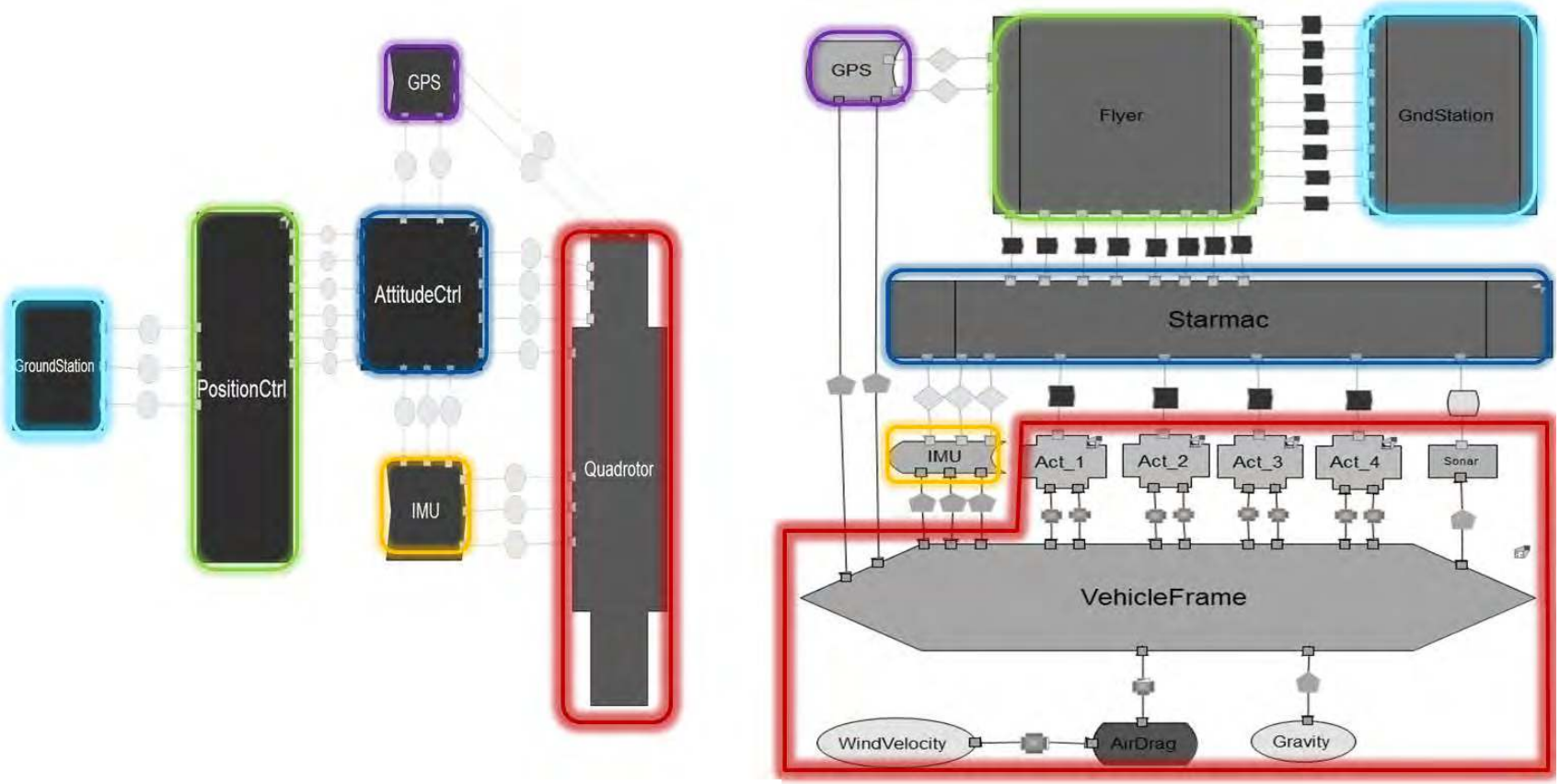

Figure 6.10: Mapping between control view for SM-1 and BA of quadrotor.

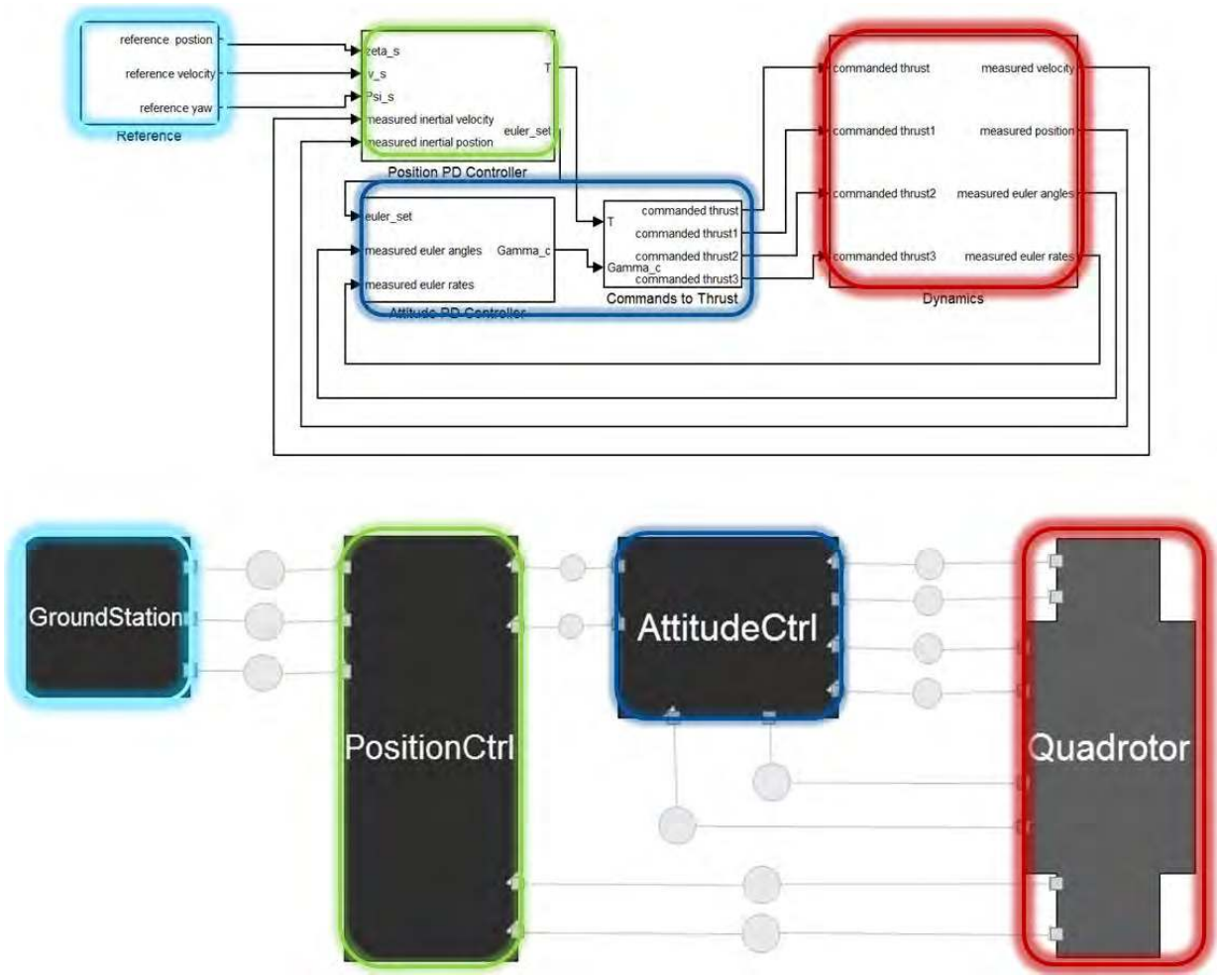

Figure 6.11: Creating control view from CMU Simulink model.

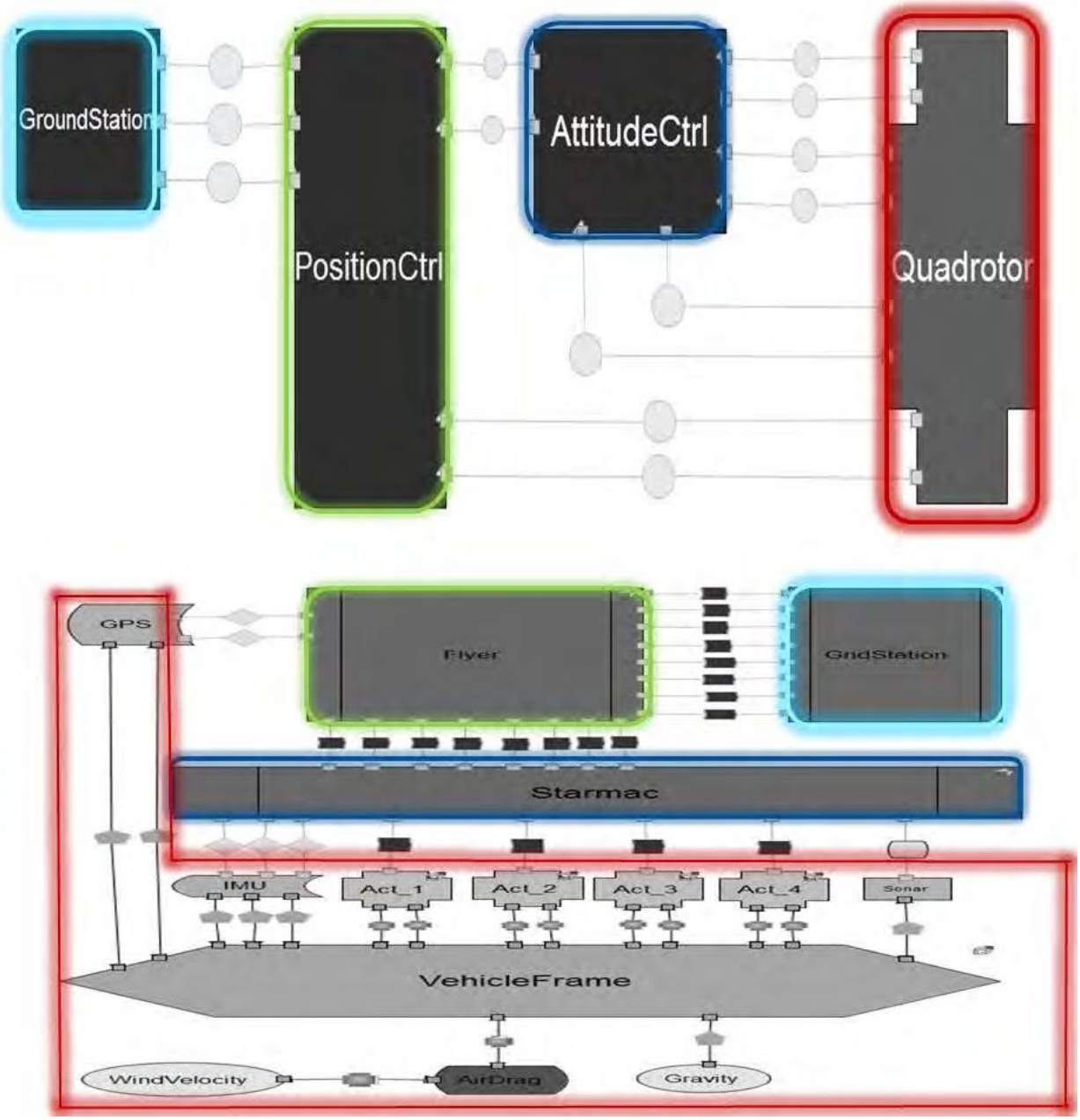

Figure 6.12: Mapping between control view for SM-2 and BA of quadrotor.

about the physical structure and this internal representation can be mapped to and compared with the internal structure of the corresponding Simulink block (*starmac_dynamics*) in a hierarchical manner. The two controllers and sensors in the view are mapped to appropriate elements in the BA as well.

Our team at Carnegie Mellon created another Simulink model (henceforth called SM-2) that focuses on the control design of the quadrotor only, with the model's architecture based on the standard controller-plant feedback configuration. The creation of the control view from the SM-2 model is shown in Fig. 6.11. The architecture does not contain any elements representing sensors or actuators, since these are typically encapsulated into the plant dynamics in a traditional controller-plant feedback configuration. The mapping between the control view's $\mathcal{A}_V$ and the quadrotor's BA is shown in Fig. 6.12. The physical plant dynamics are encapsulated as a single view component, which now includes all the sensors in the system. The controllers in the BA are mapped to individual components in the view. The number and type of connectors in this view are different from those in the SM-1 control view. This reflects the different assumptions made in each view about the types and sources of the control and feedback data available to the controllers, and indicates the possibility of different control algorithms being used in the corresponding Simulink models.

6.3.3 Hardware View

The hardware view describes the hardware architecture of the complete quadrotor system, which includes the aerial vehicle and the remote ground station. The hardware architecture documentation of the STARMAC that is used by the Stanford team is shown in Fig. 6.13. Based on this design specification and the layout of the physical hardware on the vehicle, the hardware model was created in AADL using the open-source OSATE framework [23].

The vehicle is equipped with three sensors for full state estimation. A Microstrain 3DMG-X1 IMU provides three-axis attitude, attitude rate, and acceleration measurements.

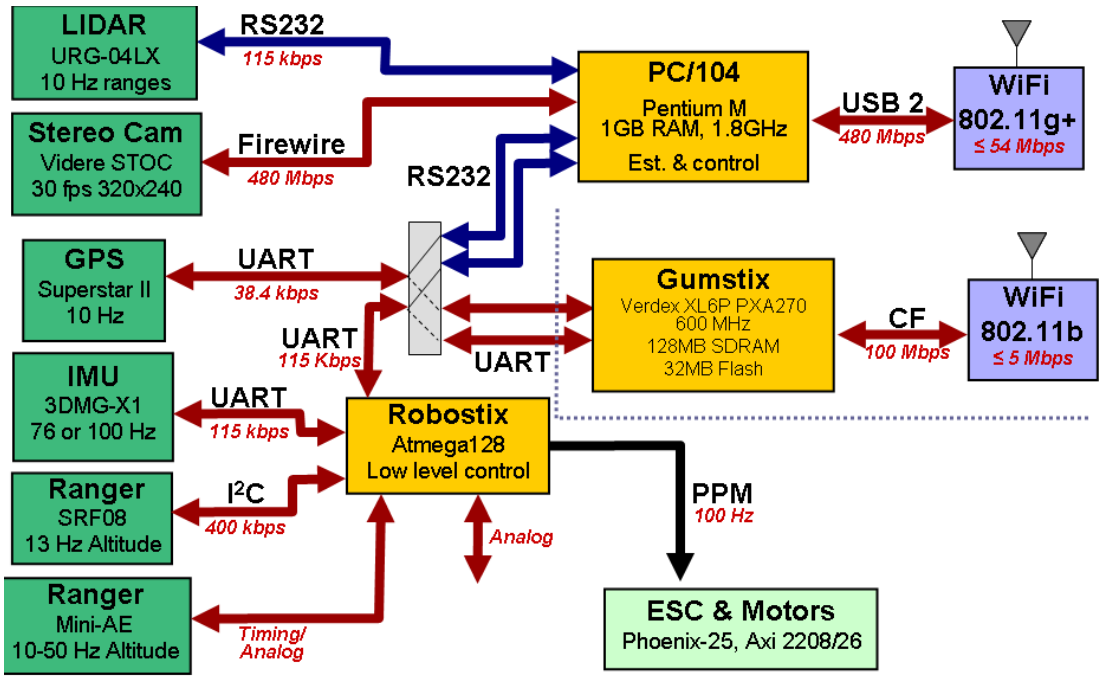

Figure 6.13: Quadrotor hardware architecture [3].

Height above the ground is determined using a sonic ranging sensor, either the Devantech SRF08 or the Senscomp Mini-AE. Three-dimensional position and velocity measurements are made using differential GPS relying on the Novatel Superstar II GPS unit. The Robostix and Gumstix (or Atom/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 sonar sensor and IMU are connected to the Robostix through I2C and serial links, respectively. The GPS is connected to the Gumstix board through a serial link as well.

The hardware view is created from the AADL model, as shown in Fig. 6.14. Each AADL processor and device is mapped to a component with the corresponding type, and each communication bus is mapped to a corresponding connector. The mappings were created manually for this example. One could envision a plugin that automatically creates the view by translating the textual specification file of the AADL model into the corresponding view

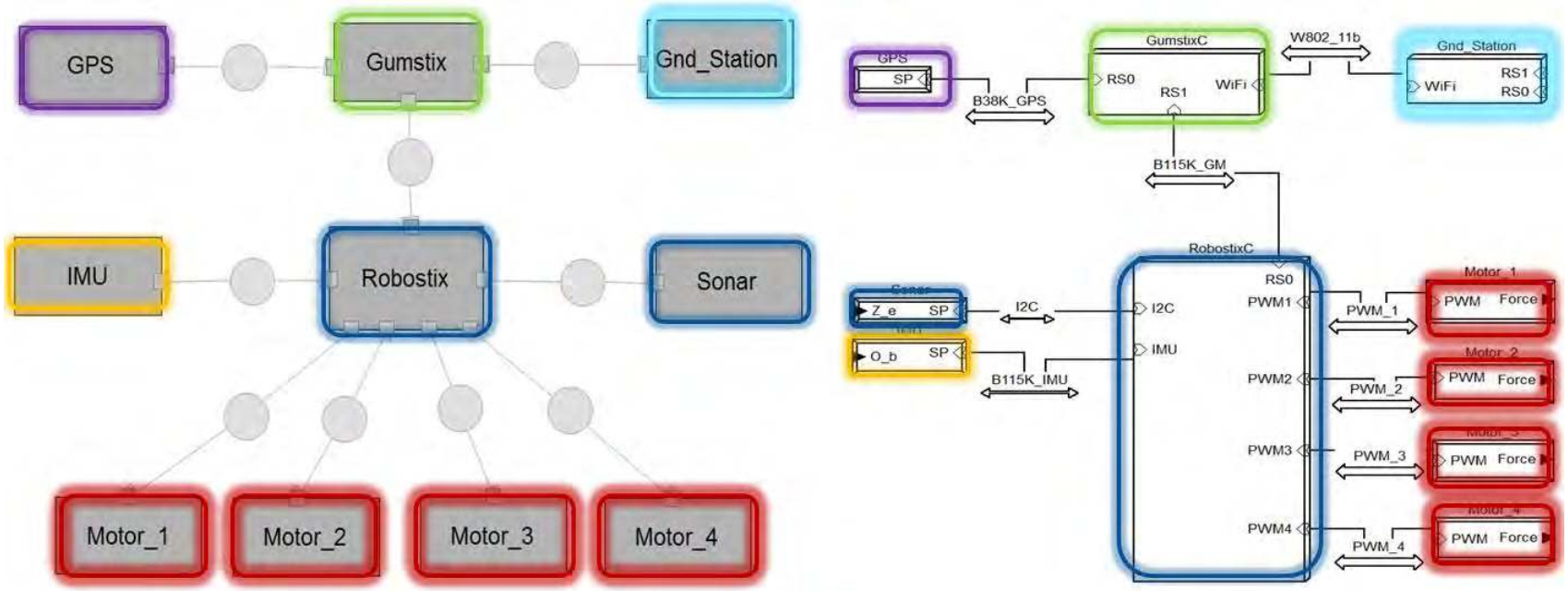

Figure 6.14: Creating the hardware view from an AADL model.

elements, based on the association between the AADL hardware element types and the view's architectural style.

The mapping between the hardware view and the BA of the quadrotor is shown in Fig. 6.15. Since a connector between two hardware components represents a communication channel between them, all data connectors between two components in the BA are mapped to a single connector in the view. This is a semantically meaningful encapsulation for hardware connectors because multiple types of data between system elements can go over the same physical communication channel. The software components are mapped to the hardware processors on which they run in the actual system.

6.3.4 Physical View

The physical view models the dynamics of the quadrotor in terms of the forces and torques applied by the rotors to the vehicle frame. With reference to Fig. 6.16, the roll, pitch and yaw angles (ϕ, θ, and ψ, respectively) are controlled by providing differential thrust to the vehicle frame via the motors. Differential thrust between opposite motors provides roll and pitch torques while differential thrust between the two pairs of counter-rotating motors provides yaw torque. Position control with respect to the inertial (North-East-Down) coordinate frame is accomplished by controlling the magnitude and direction of the total thrust. A drag force, D_B, also acts on the vehicle in the direction opposite to that of the vehicle's velocity, e_V.

The nonlinear dynamics of the quadrotor can be modeled by a point mass m with moment of inertia $I_b \in R^{3\times3}$, location $\rho \in R^3$ in the inertial frame, and angular velocity $\omega \in R^3$ in the body frame. The vehicle undergoes forces $F \in R^3$ in the inertial frame and

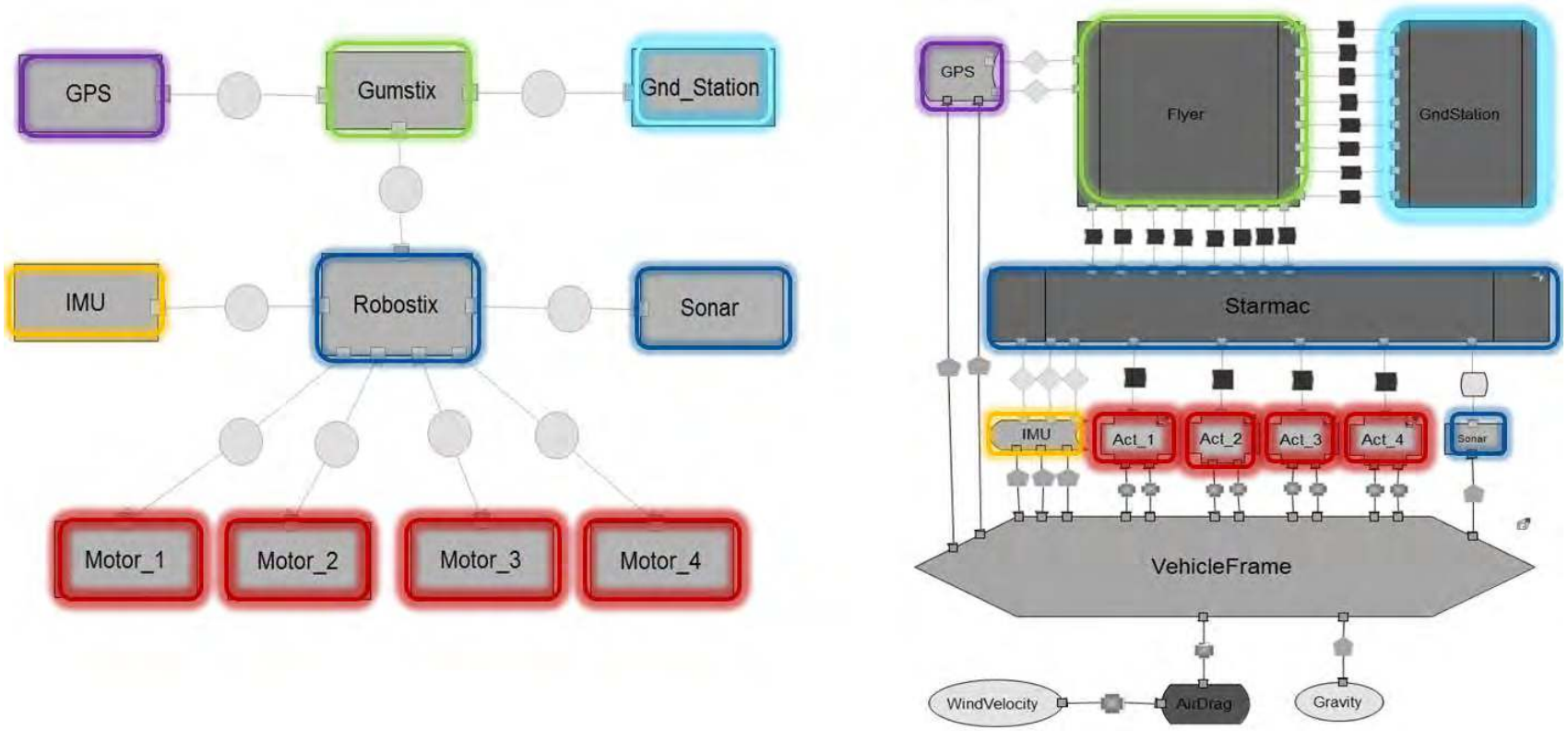

Figure 6.15: Mapping between hardware view and BA of quadrotor.

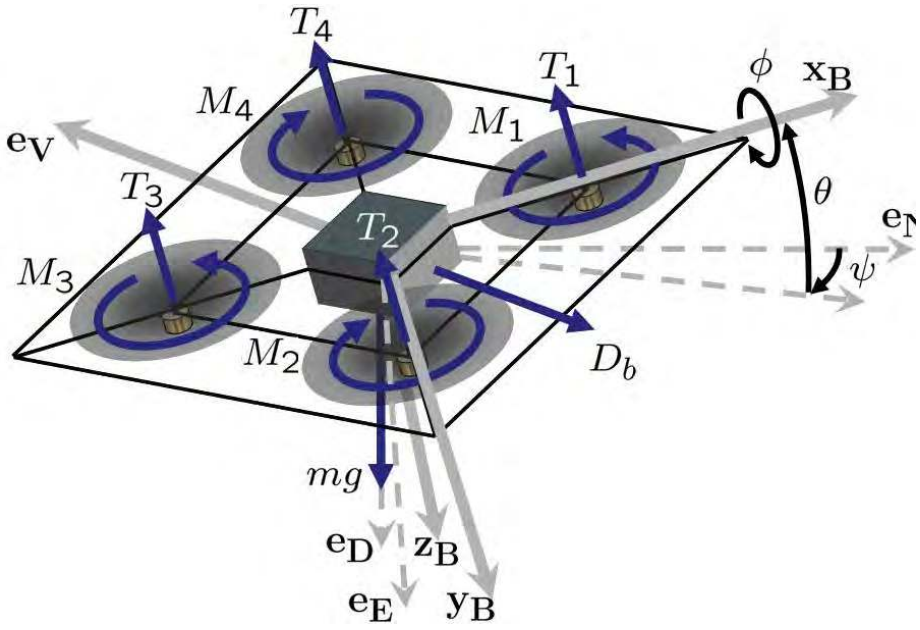

Figure 6.16: Free-body diagram of quadrotor dynamics.

moments $M \in R^3$ in the body frame, yielding the equations of motion,

$$\vec{F} = -D_B \vec{e}_V + mg\vec{e}_D + R_I^B \sum_{i=1}^{4} T_i \vec{z}_B$$

$$\vec{M} = \sum_{i=1}^{4} T_i (\vec{r}_i \times \vec{z}_B)$$

where g is the acceleration due to gravity, and R_I^B is the rotation matrix from body coordinates to inertial coordinates.

We have implemented the dynamics model of the quadrotor in the Modelica language using the OpenModelica tool [56]. The creation of the physical view from the Modelica model is shown in Fig. 6.17. The physical architectural style enables the formal representation of such dynamic behavior in the overall system architecture. The semantics of the physical view are defined in terms of components representing physical phenomena, interacting through the non-causal interconnections between the effort and flow variables of each attached component's ports. Since the Modelica language also has equation-based

Table 6.1: Models and Views created for STARMAC quadrotor.

Model	View	Analysis
FSP	Software	Safety properties of the control software
Simulink SM-1	Control	Stability and control performance with platform and device details
Simulink SM-2	Control	Stability and control performance with idealized control design
AADL	Hardware	Processor and bus requirements for implementation platform
Modelica	Physical	Quadrotor dynamics with rotor forces and torques

semantics, there is an almost direct mapping between the elements in the model and in the view.

The mapping between the physical view and the BA of the quadrotor is shown in Fig. 6.18. Since the architectural style of the physical view is derived completely from the physical style of the CPS family, there is again an almost direct mapping between the view elements and the physical elements in the BA. However, force and torque connectors between the *Act* components and the *VehicleFrame* in the BA are mapped to single connectors in the view. This is due to the fact that Modelica allows multiple conjugate variables to be defined on a single port, while our CPS style currently makes each pair of conjugate variables explicit by defining separate ports (and hence individual connectors) for each pair.

This encapsulation of multiple, semantically equivalent connectors in the BA to a single connector in the view has also occurred in the other views, and seems to be a common map pattern for the models that we have studied so far. The multiple views for the quadrotor that are created based on the existing analysis models are summarized in Table. 6.1.

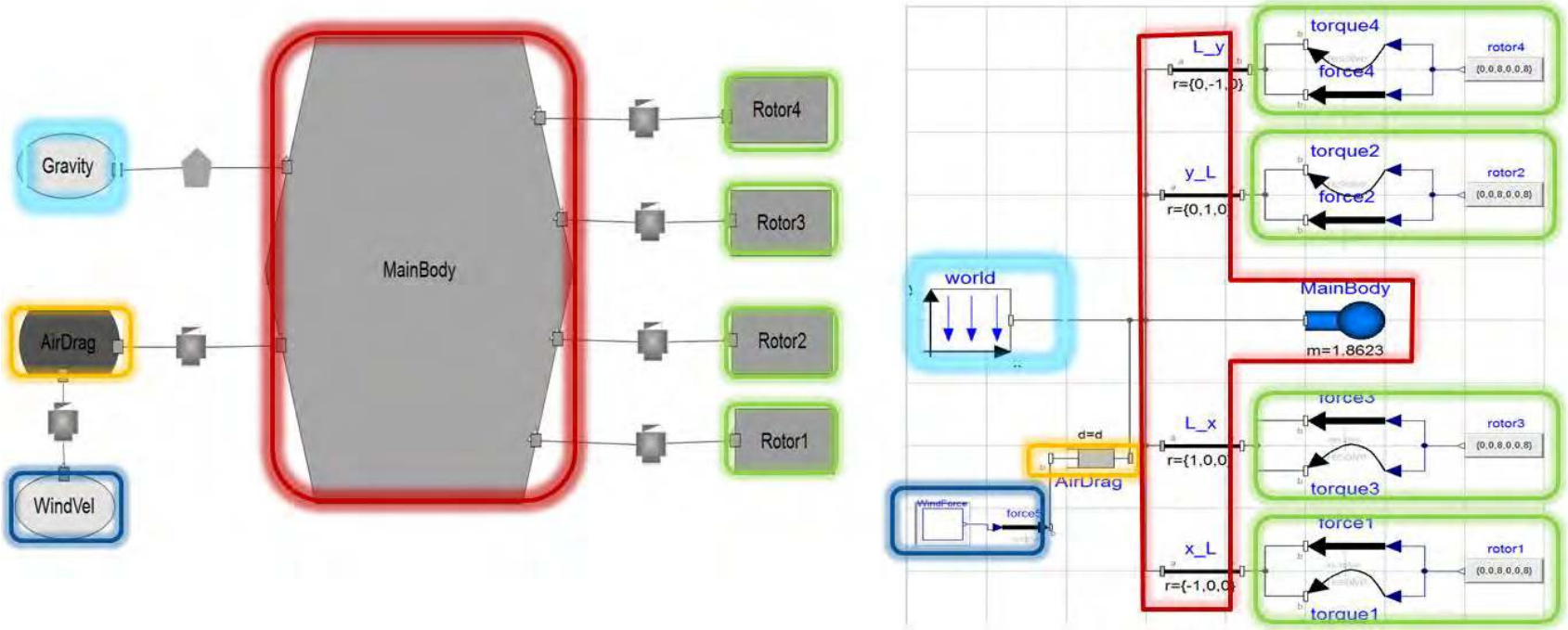

Figure 6.17: Creation of physical view from Modelica model of quadrotor dynamics.

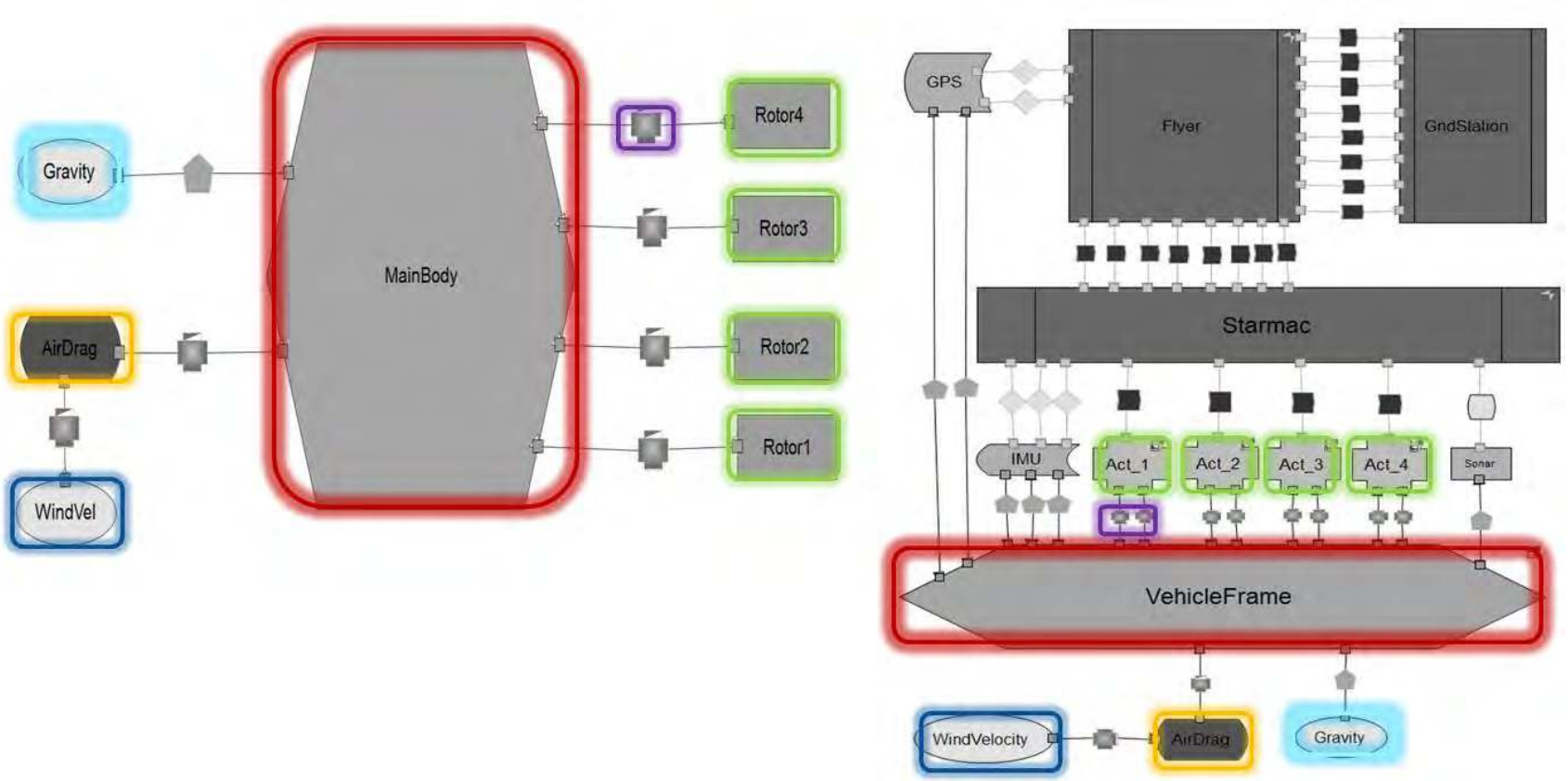

Figure 6.18: Mapping between physical view and BA of quadrotor.

6.4 View Consistency

In this section, we apply the view conformance and view completeness checks to the views of the quadrotor, and highlight what inconsistencies were uncovered and their impact on the consistency of the complete system.

6.4.1 Software View

We first run a conformance check between the software view and the BA of the quadrotor, using the AcmeStudio view editor tool. The check fails and the view editor gives us the option to find the maximal set of mapped elements. In general, the MCS algorithm returns a number of feasible mappings between the view and the BA. For the case study, we have limited the number of maximal mappings returned to the first one found, so that there is a practical time bound for the algorithm to return the search result. The returned mapping from the MCS algorithm is shown in Fig. 6.19. Using the option to display only unmapped elements in the view, the editor has highlighted the connector between *GndStation* and *PositionCtrl* and the connector between *PositionCtrl* and *AttitudeCtrl* components, along with their associated roles and ports. These elements represent the sending of trigger packets from the GSC to the AC via the PC. Since the BA did not have any corresponding connectors, the consistency check failed and the MCS algorithm returned all view elements as mapped, except this set of missing elements.

We realise the fact that since the software view for this case study is comparatively small in size, the missing connectors could have been noticed by visual inspection as well. However, one could envision the view-to-BA mapping for larger systems being done automatically, either by having consistent names for corresponding architectural elements, or by having a set of structural patterns in the view being mapped to a corresponding set of patterns in the BA, using graph grammar specifications. In such cases, manually finding inconsistent elements in the view can become a non-trivial task, and the MCS algorithm

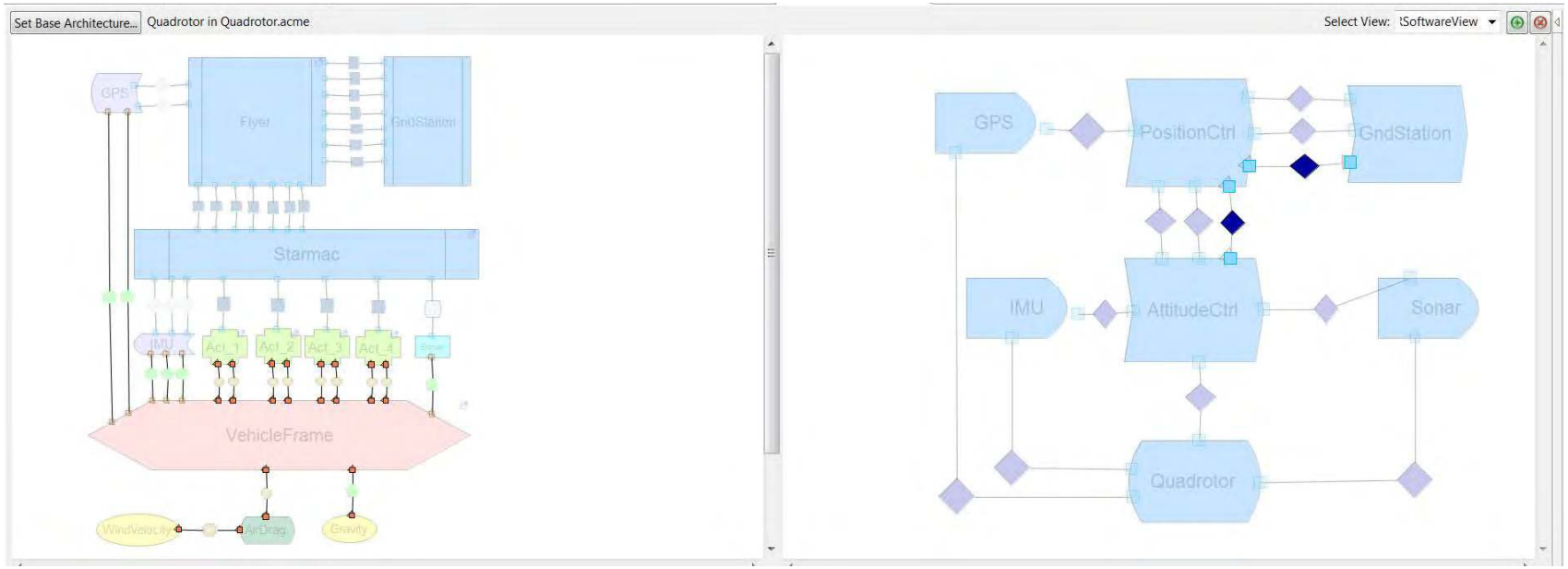

Figure 6.19: Inconsistent elements between software view and BA of quadrotor.

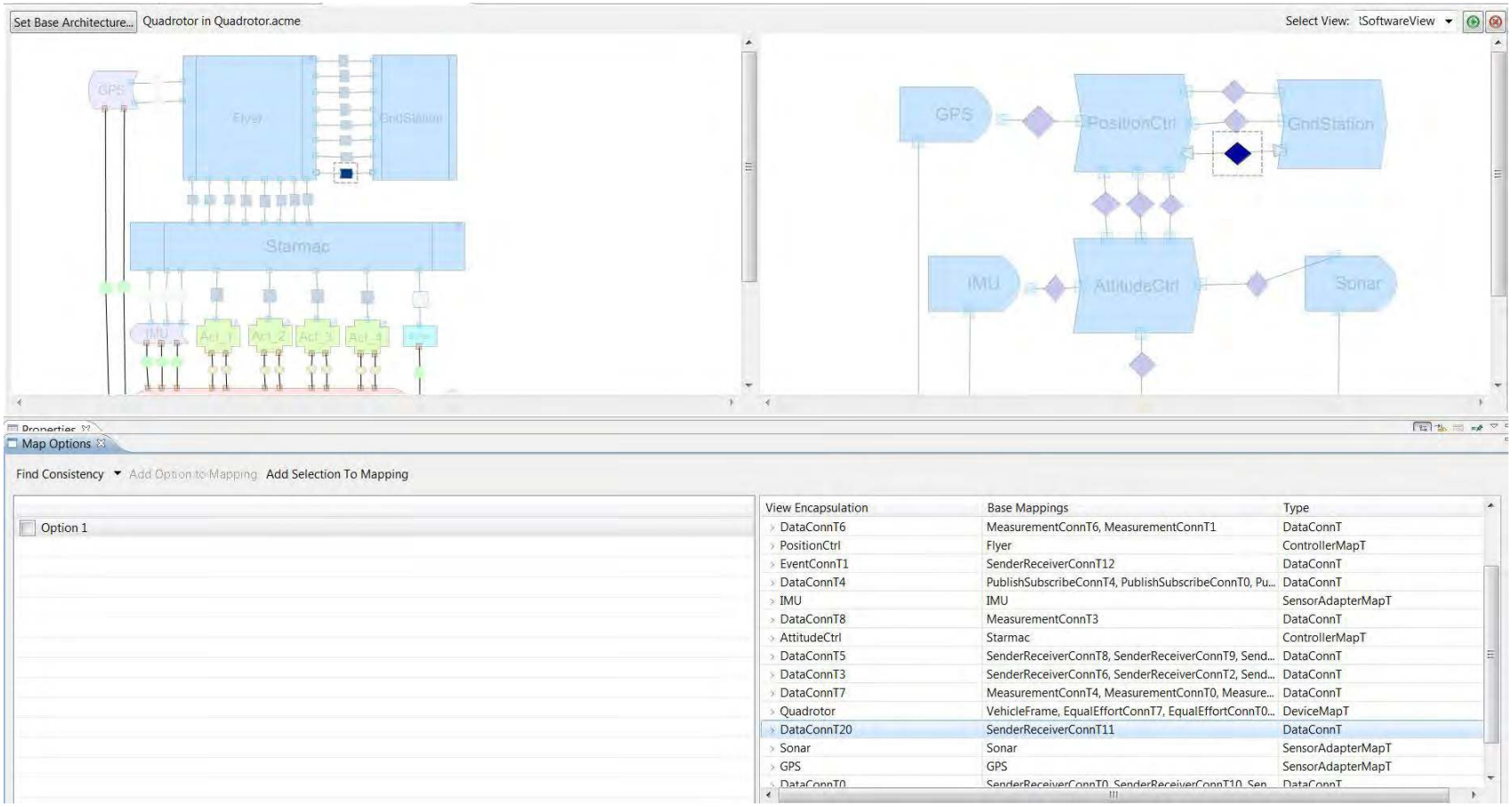

Figure 6.20: Successful conformance check between software view and BA of quadrotor.

could prove useful in discovering the set of such missing view elements.

Once the inconsistent elements are discovered, we choose to modify the BA by adding the missing connectors between the respective components, so that the significance of the *trigger* protocol for the safety of the quadrotor is highlighted in every other view of the system. This is an example of how the BA of the system can be modified during the design flow, so that important details about system structure or function that are discovered in a particular view can be made available to every other view in the future. The BA and each view of the system evolve throughout the system design process, as new information about the system is discovered through the analysis of the associated models.

Once the missing connectors are added in the BA, we run the conformance check again for the software view. In Fig. 6.20, the mappings returned by the checker are displayed. The *trigger* connector between *GndStation* and *PositionCtrl* in the view is now correctly mapped to the newly added corresponding connector in the modified BA. A similar mapping is returned for the view connector between the *PositionCtrl* and the *AttitudeCtrl* components. We use the modified BA of the quadrotor as the new BA for subsequent consistency checks with the other system views.

6.4.2 Control View

For the first control view, the conformance check with the BA fails. Hence, we run the MCS algorithm to find the set of maximal common elements. As before, from the mapping returned by the MCS algorithm, we determine all the view elements that remain unmapped. Our tool post-processes the result, and highlights all the architectural elements attached to the unmapped elements as well. In this example, the MCS algorithm returns a set of unmapped ports in the view. Hence, all connectors (and roles) attached to those ports are also inconsistent. The set of inconsistent elements in the first control view are highlighted in Fig. 6.21.

There are two causes of inconsistency between the control view and BA. The first

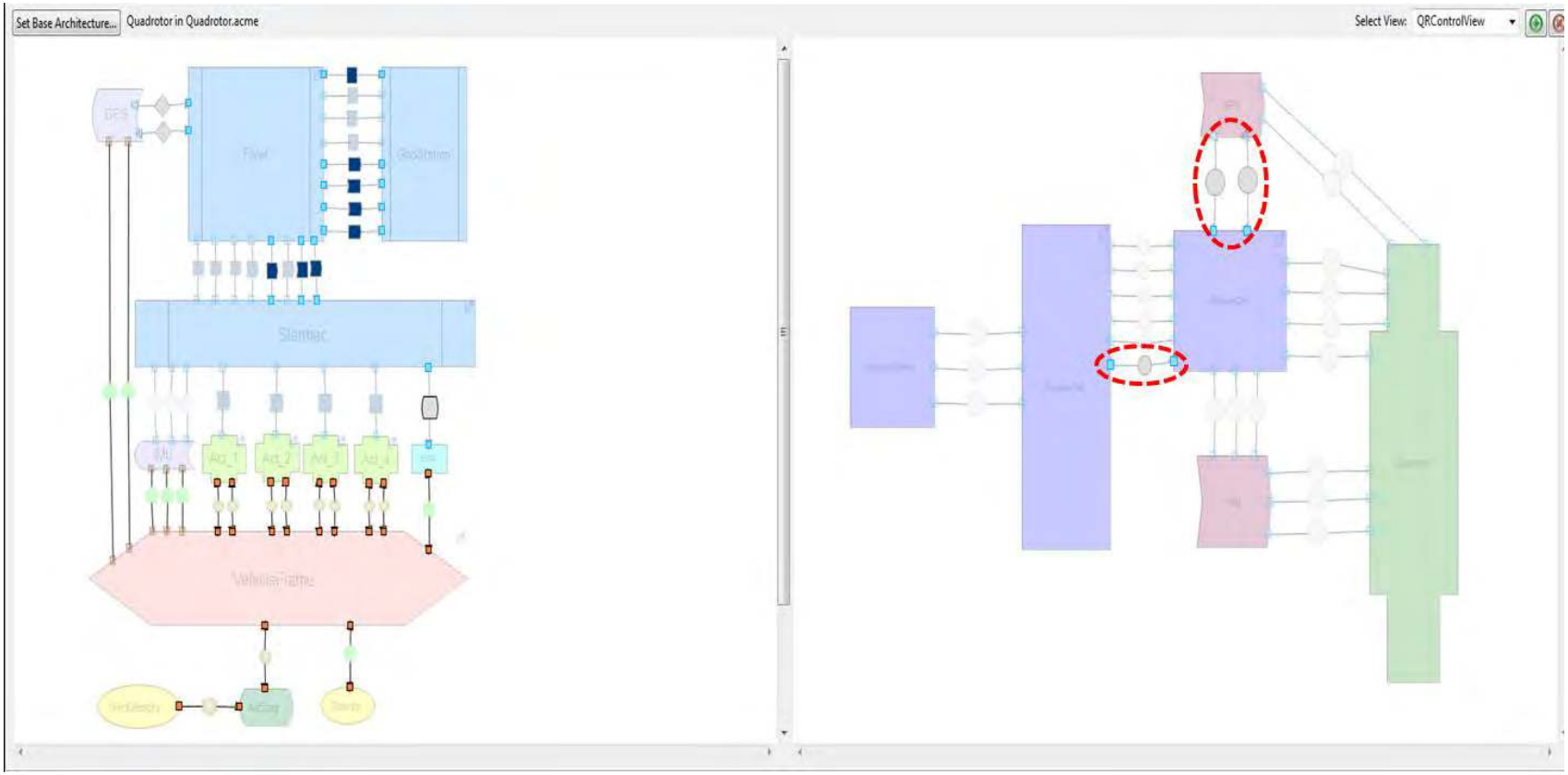

Figure 6.21: Inconsistent elements between control view for SM-1 and BA of quadrotor.

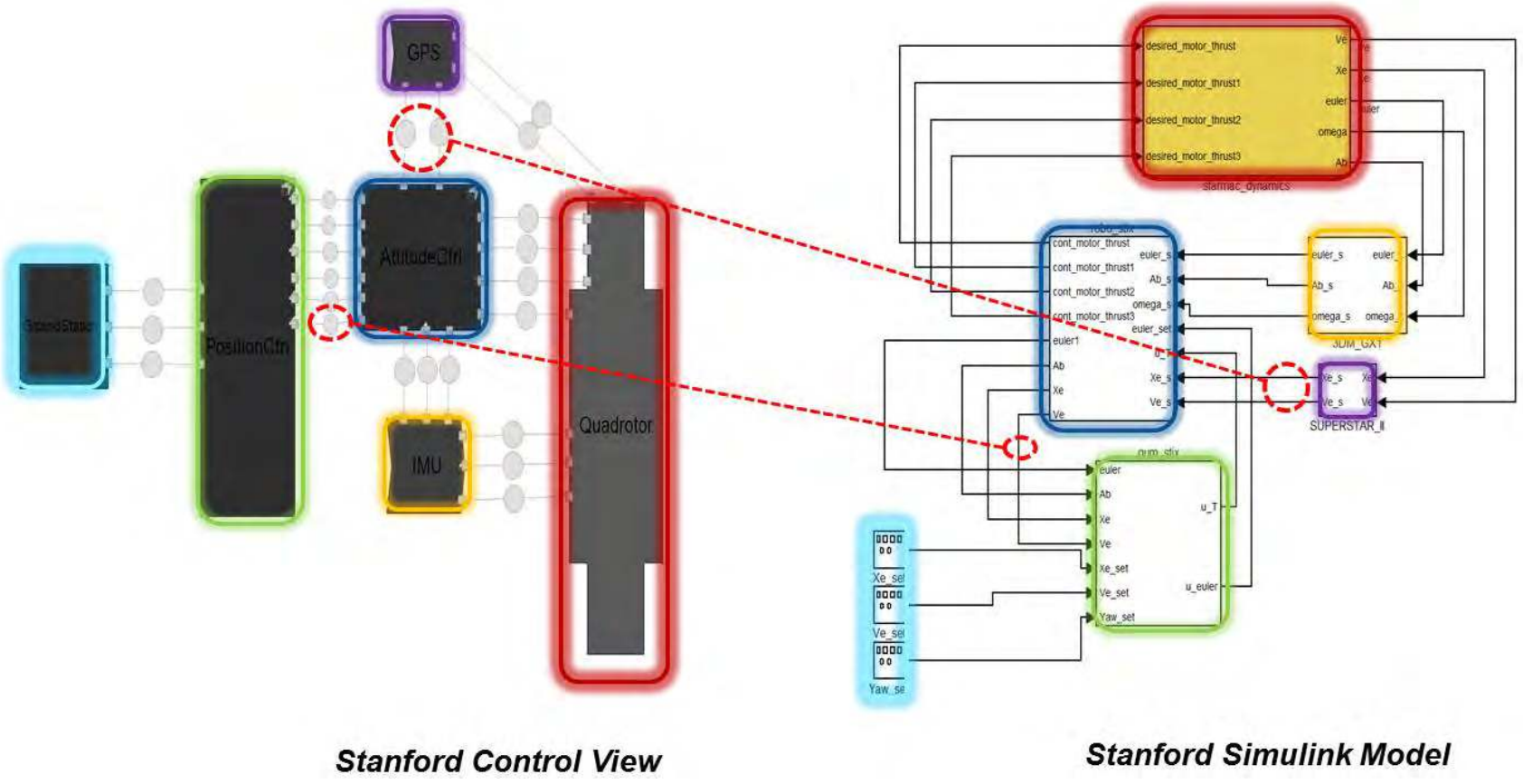

Figure 6.22: Inconsistency in control view for SM-1 traced back to Simulink model.

inconsistency arises due to the connectors existing between the sensor output ports of the *GPS* and the input ports of the *AttitudeCtrl* component in the control view. However, according to the BA topology, the *GPS* sensor is connected to the position controller (*Flyer* component) directly. The second inconsistency is caused by an extra connector between the AttitudeCtrl and *PositionCtrl* components. This connector represents velocity readings from the *GPS* sensor, passed through by the AttitudeCtrl to the PositionCtrl. No such corresponding connector (with compatible type and direction of data flow) exists in the BA. Both of these inconsistencies can be traced back to the SM-1 model, as shown in Fig. 6.22.

This is an example where the system model is functionally correct, i.e., the control system designed in the SM-1 model 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. 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 SM-1 model. Such architecture-level mismatches are caught by the view consistency definition, since the misplaced/missing connectors prevent a successful morphism between the view and the BA graphs.

The reason that the SM-1 model works correctly in this instance is because the *robo_stix* block passes the GPS signals untouched to the *gum_stix* block, where they are actually used. However, the ramifications of these mismatches could be serious. Suppose, for example, that the stability of the attitude controller is verified based on the assumption that the GPS signal is 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 are not applicable to the actual system. This is an unintended and potentially dangerous consequence of having inconsistent views (and models).

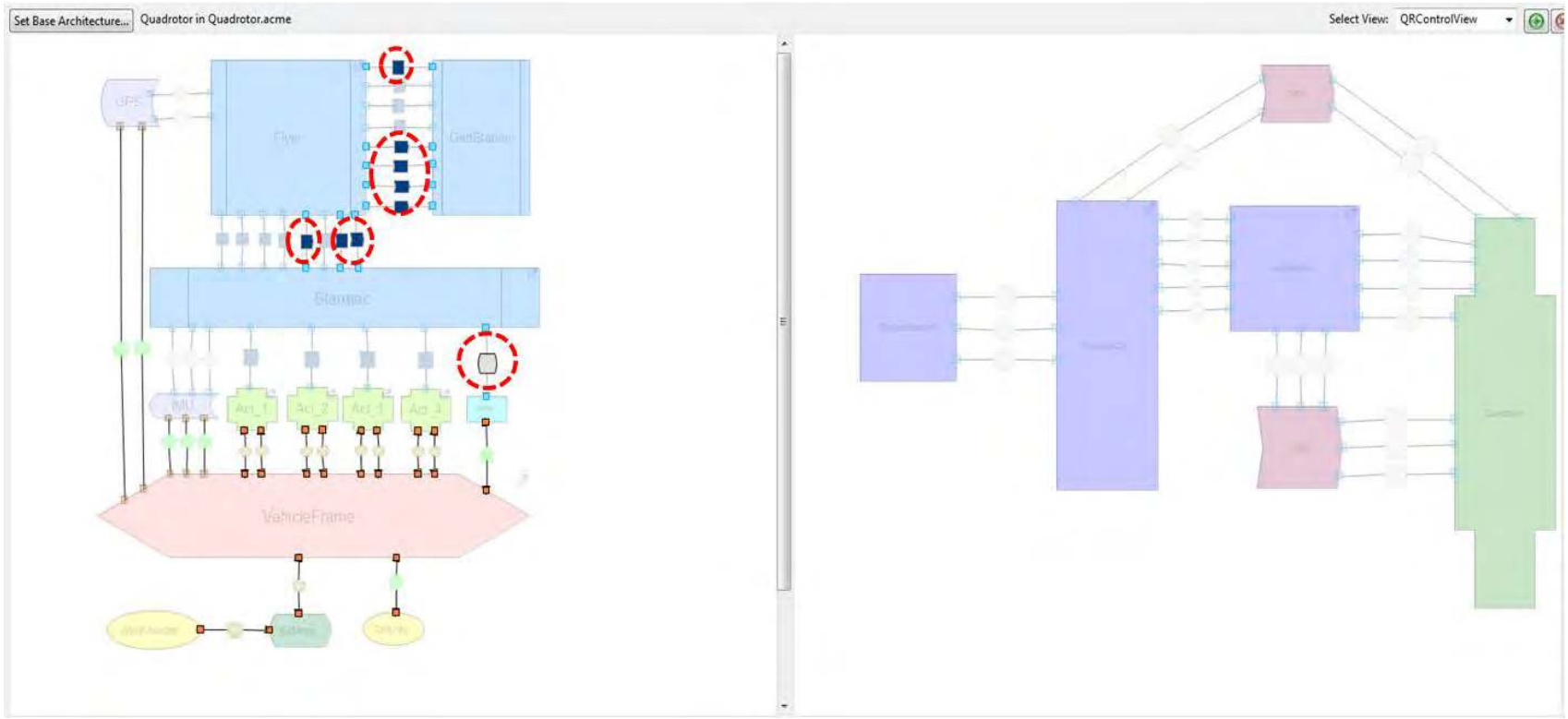

Figure 6.23: Completeness check between control view for SM-1 and BA of quadrotor.

We correct the inconsistency in the view (and the SM-1 model) by connecting the GPS sensor directly to the position controller and removing the extra connector from the attitude controller to the position controller. The conformance check for the corrected control view succeeds. Since the focus of a control view is on the complete system (both cyber and physical concerns), we also check for view completeness, which fails. We run the MCS algorithm to find out which elements in the BA are not being mapped to any elements in the view. The resulting set of elements, displayed in the AcmeStudio view editor, is shown in Fig. 6.23.

There are three sets of connectors (and their associated roles and attached ports) that cause view incompleteness. The first set represents telemetry data that the position controller periodically sends to the ground station for mission status updates, and the trigger connectors added to the BA due to the software view analysis. This type of mismatch is an example of communication between components that is typically neglected in the control view, since the data does not contribute to the functional correctness of the control algorithm. However, it has an impact on the specification of the channel 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. The absence of the trigger connectors in the control view shows an important software protocol that is not accounted for in the control design.

The second set of connectors represents aerodynamic correction factors and the altitude setpoint sent by the position to the attitude controller. This type of mismatch indicates that the types of data used for control actions in the SM-1 model and in the implemented software are different. In fact, the control software on the actual vehicle uses estimation components to filter readings inside the position and attitude controllers. These components are present in the detailed representations of the components in the BA of the quadrotor. The controllers used in the SM-1 model do not have the same implementa-

tion as those in the source code. Even though structural view consistency cannot detect behavioral inconsistency between components, it is possible that indications of behavior mismatch can be detected based on the differences in the types of data that used by the components to implement their functional requirements.

The third set of connectors consists of a single connector between the *Sonar* sensor and the *Starmac* component (representing the attitude controller) in the BA. This connector represents height readings from the sonar which are used in maintaining a set altitude in the attitude controller. This connection is missing in the view because an incorrect assumption is made in the SM-1 model that height readings are given to the attitude controller by the GPS sensor. The missing connection from the sonar sensor can have serious consequences. Since there is an assumption that height readings are only obtained from the GPS, the *robo_stix* block contains an LQG controller based on a linearized version of the quadrotor. However, the control code implemented on the actual vehicle is more complex, based on two cascaded PID controllers, one for attitude and another for height control. This approach is 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 SM-1 model and the sensors being used on the quadrotor leads to an inconsistency between the designed control algorithm and the implemented controller on the vehicle.

For the second control view (created from the SM-2 model), the view conformance check succeeds. However, the view completeness check fails and the resulting unmapped elements in the BA are shown in Fig. 6.24. There are four sets of connectors (and their associated roles and attached ports) that cause view incompleteness. Three of the sets are the same as those for the first control view. The fourth set (highlighted with a dotted circle in Fig. 6.23) represents feedback data (attitude, attitude rates, and altitude) that the attitude controller sends back to the position controller onboard the actual vehicle.

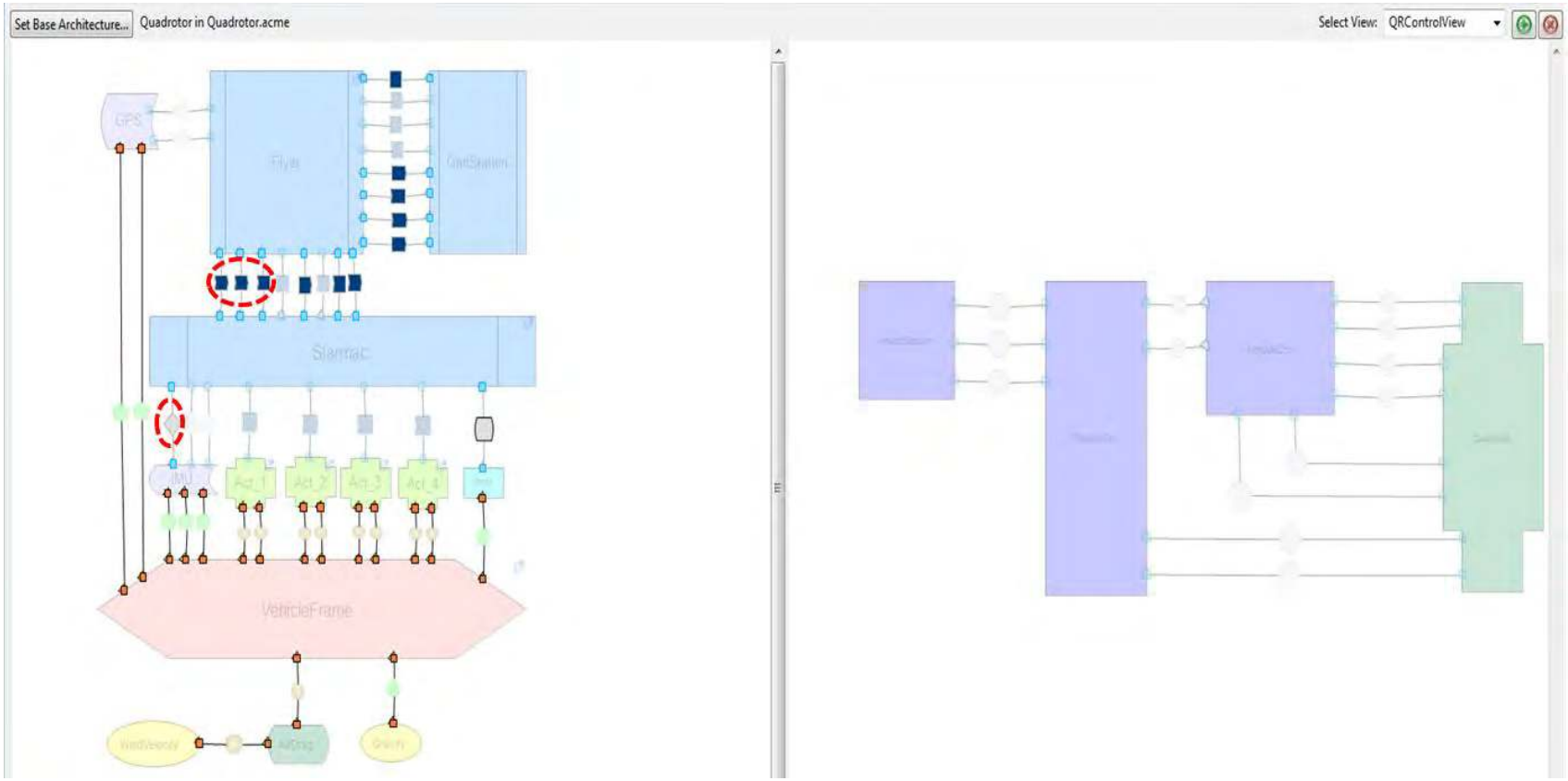

Figure 6.24: Completeness check between control view for SM-2 and BA of quadrotor.

The set also includes a connector from the IMU to the attitude controller that represents measured acceleration data. The implemented position controller uses the altitude data to obtain a more accurate vehicle position by comparing the height with the GPS readings. The implemented attitude controller uses the acceleration to compensate for the tilt of the vehicle for height measurement.

This mismatch once again highlights the possibility of different controllers being used in the SM-2 model and the actual system. This can be verified by comparing the internal structure of the controller blocks in the model with the actual code. The SM-2 model contains two PID controllers but no estimation or filtering logic since the assumption is that all sensor readings are noise-free and completely accurate. In addition, there is no logic to incorporate acceleration for height calculations. This assumption could lead to potential problems during system integration, if the controllers in the SM-2 model are directly used for code generation without verifying the system sensor behavior against this assumption.

6.4.3 Hardware View

The hardware view is concerned with the hardware implementation of the embedded system onboard the quadrotor, and not with the vehicle dynamics. Hence, the view is checked for conformance with the BA, and this check is successful. The resulting map returned by the graph morphism checker, and displayed in the multi-view editor, is shown in Fig. 6.25. If the architect instead checks for view completeness, then the algorithm returns with a failed result. In order to have completeness with the BA, the hardware view would have to include a device component that represents the encapsulation of the vehicle frame, gravity and wind drag components, as well as connectors from this new component to the sensor components in the view. For example, the architect can choose to represent the missing quadrotor dynamics as a new processor component that runs a detailed simulation model of the quadrotor. This could be useful for HILS testing of the controller code, similar to the

approach followed in the Vanderbilt toolchain [57], and in our second case study (described in Chap. 7).

6.4.4 Physical View

The physical view is concerned with a subset of the CPS architecture, namely the dynamics of the vehicle frame, and the forces imparted by the rotors connected to it. This is another illustration of a view that focuses on a portion of the entire system. We run a view conformance check that is successful, as shown in Fig. 6.26. However, the view completeness check fails since all the cyber elements in the BA are absent in the physical view, because of the nature of the associated design concern that the model is created in. If view conformance is needed, then the view (and associated Modelica model) has to include components representing the hierarchical controllers and the ground-station, along with the associated sensors.

We summarize the types of inconsistencies discovered by applying our view consistency analysis to the STARMAC quadrotor system in Table 6.2.

6.5 Summary

In this chapter we present a case study based on an avionics system, the STARMAC quadrotor, to check for view consistency. The case study is an example of how our architectural approach can be applied to an existing system, for which legacy models and implemented code are already present. We create the BA of the quadrotor, and multiple views that are derived from the heterogeneous models used for the system's analysis and design. We check the structural consistency of each architectural view with the quadrotor's BA using our view conformance and completeness tools implemented in AcmeStudio. We elaborate on the mismatches discovered and their impact on the integrity of the implemented system.

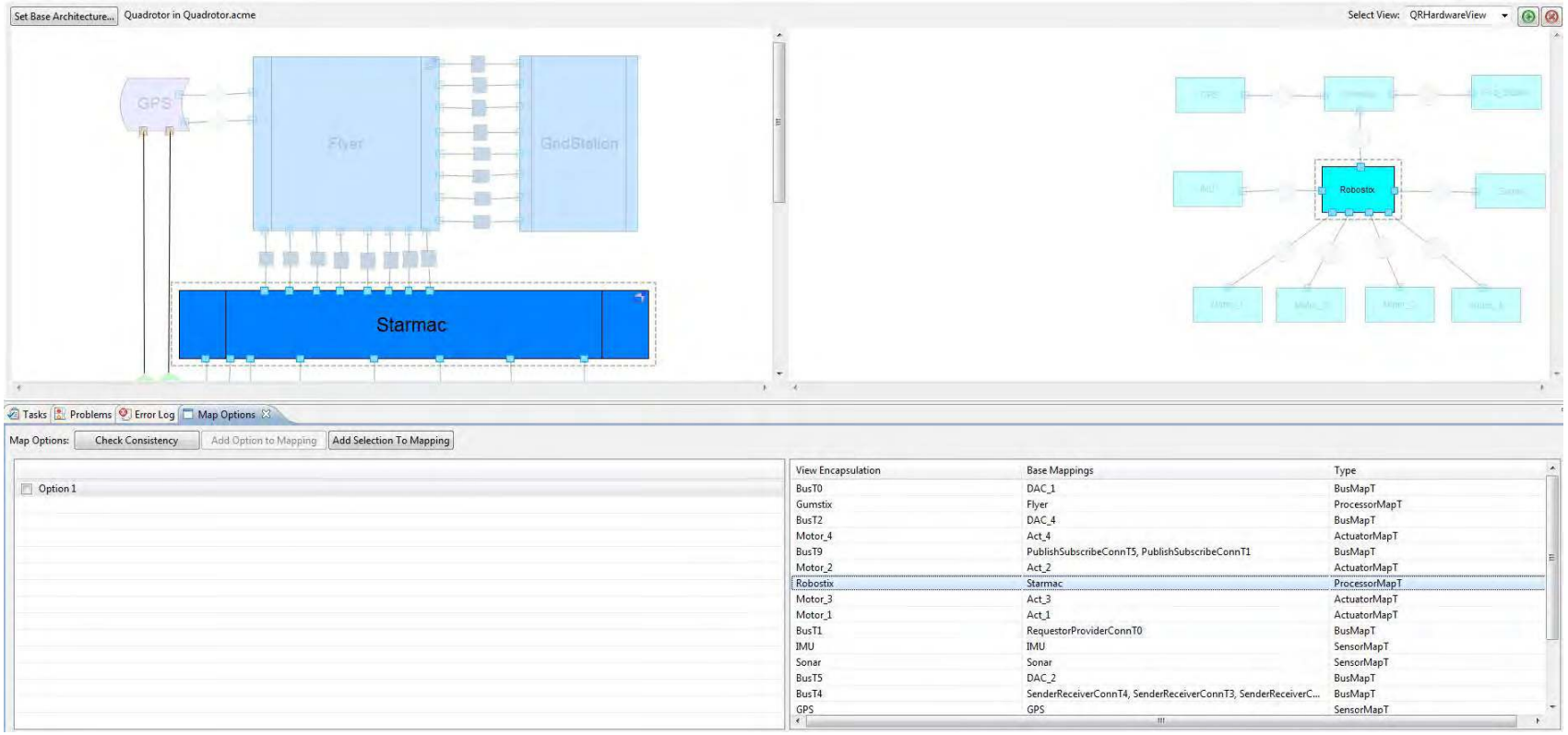

Figure 6.25: Successful conformance check between hardware view and BA of quadrotor.

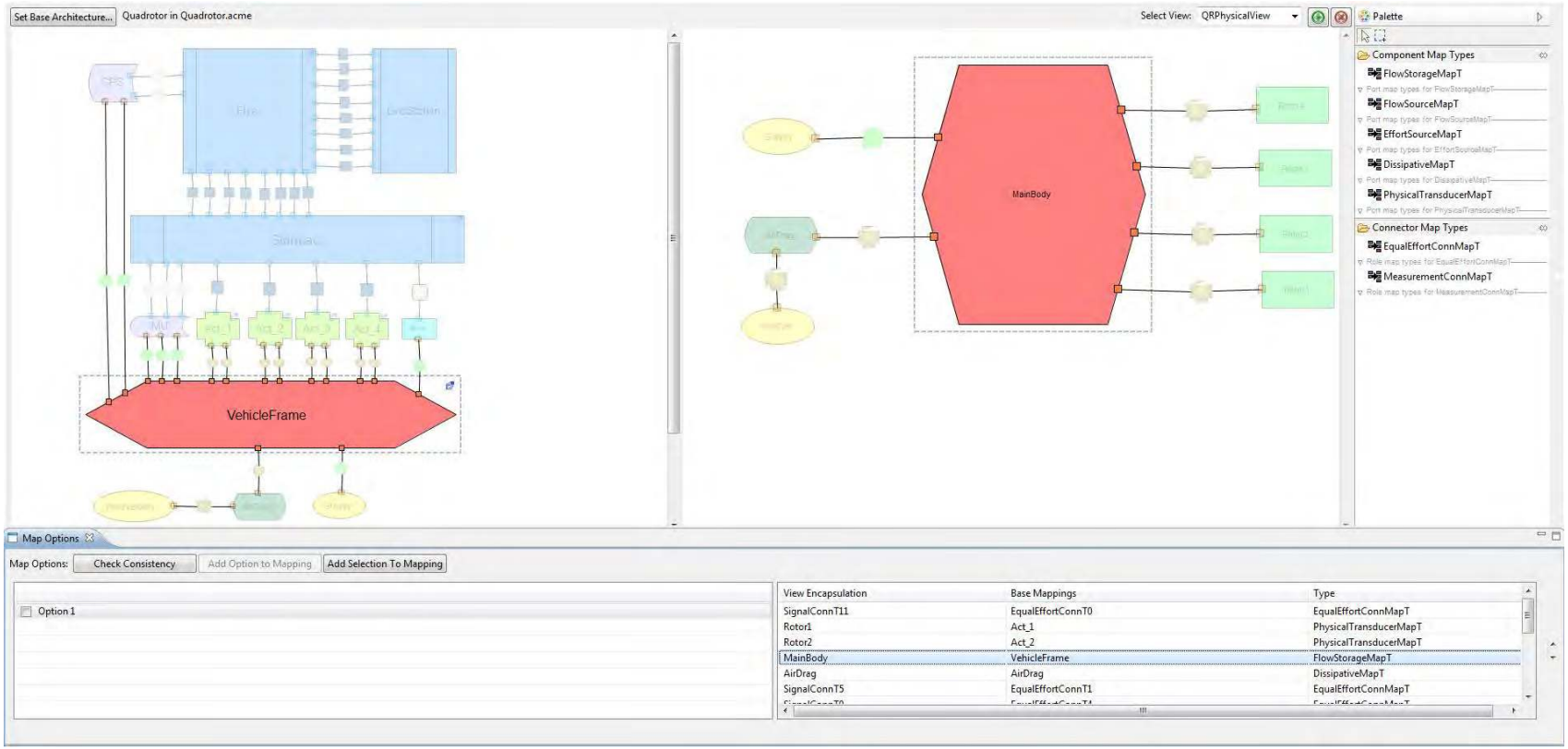

Figure 6.26: Successful conformance check between physical view and BA of quadrotor.

Table 6.2: Consistency analysis for STARMAC quadrotor.

View	Inconsistency Type	Inconsistent Elements	Impact
Software	View Conformance	Additional data connectors in view representing trigger packets between ground station and onboard controllers	Represents important safety protocol in implemented control software that is missing in both control design models
Control I	View Conformance	Misplaced connectors in view between GPS sensor and attitude controller	Wrong assumption in model about how connectivity of hardware and software components
Control I	View Conformance	Additional connector in view from attitude to position controller	Wrong assumption in model about type of data exchanged between controllers
Control I	View Completeness	Additional connectors in BA for telemetry between position controller and gnd station	Affects wireless bandwidth and execution time of position controller
Control I	View Completeness	Additional connector in BA for safety protocol between gnd station and controllers	Affects stability and safety guarantees from control model for implemented system
Control I	View Completeness	Additional connectors in BA for control data sent between position and attitude controller	Data for control actions in model and in software different. Possible behavior mismatch
Control I	View Completeness	Additional connector in BA between sonar and attitude controller for height readings	Wrong assumption in model about source of sensor readings. Possible behavior mismatch
Control II	View Completeness	Additional connectors in BA for feedback data between position and attitude controllers	Data used for control actions in model and in software different. Possible behavior mismatch

Chapter 7

Case Study II : XILS

In this chapter, we demonstrate the use of architectural views to manage variants of models used in "In-Loop-Simulation" testing scenarios in the automotive industry. We apply the concept of an architectural view in a novel way to describe the common structure and interconnection interfaces of the models. Each view is created as an abstraction of a detailed base architecture that is derived from the physical implementation of the closed-loop vehicle testbed. This case study illustrates the design flow where we have the freedom to create the base architecture and each view of the system from a clean slate, based on the requirements of the problem. Each system model is constructed so that the structure of the model conforms to the corresponding view architecture.

We check each XILS view for consistency and conformance with the BA of the system. However, we do not check any existing simulation models for conformance with the view, since the focus of the case study is on defining standardized architectures for the design models in various XILS scenarios.

7.1 Introduction

An urgent concern in current engine control development is the significant number of man-hours being spent in calibration and testing due to the increasing complexity of control software. In the current process, control algorithm validation is performed with the pro-

duction software running on the actual Engine Control Unit (ECU), physically interfaced with the vehicle's engine in most cases. This makes it difficult to separate the impact of the software, the ECU hardware, and the physical engine while evaluating the performance of the closed-loop system. The tight coupling of these elements is also a major source of delay during the debugging phase when engineers are trying to isolate the source of a problem. One solution is to create a closed-loop simulation environment that uses an engine model in place of the actual engine from the early stages of design. Being required to incorporate the physical engine much later in the design flow makes it possible to achieve rapid prototyping and faster turnaround time for engine control testing and validation.

Such a simulation environment is called an *X-In-The-Loop Simulation* (or XILS) environment, where the 'X' stands for either 'Model', 'Software', or 'Hardware'. The three variants (shown in Fig. 7.1) are created to test the control algorithm specification, controller software, and integration with an ECU, respectively.

- **MILS**: In this environment, the controller and the plant both exist as models in the native simulation tool. For example, a typical MILS would contain a Simulink model of the controller connected in a feedback configuration with a Simulink model of the engine. The execution semantics in this case is that of the selected Simulink differential equation solver. Such a configuration is used in the early design phase for control algorithm development. The goal is to check whether the functional behavior of the controller and the dynamics of the plant conform to the given specifications. The models are generally simple and are idealized abstractions of the underlying system to be designed.
- **SILS**: In this environment, the controller model created in MILS is converted into controller software, either manually but more commonly with an automatic code generator. For example, a common way to convert Simulink models to C or C++ code contained in an S-function block is by using Simulink Coder in MATLAB [58].

The software can include code to model simple platform-level details such as fixed-point or integer data types. The plant model at this level can be carried over as is from the MILS environment, or refined to add more behavioral details, or changed to a completely different type of model if required by the specifications to be validated. The interface between the controller and the plant is still made in software. However, the types of signals exchanged between them can (and generally do) change because of the details added to both entities at this level. The purpose of SILS is to test a major part of the controller software as early as possible in the design cycle.

- **HILS**: In this environment, the controller software (integrated with an operating system or scheduler if required) is run on the production ECU, and a detailed plant model is run on a real-time simulator. The purpose is to validate the complete controller integrated with ECU hardware without requiring the physical plant to be ready at this stage. The connections between controller and plant are made through hardware I/O boards between the ECU and the computer running the simulator, and the wiring harnesses are commonly custom-made for each test scenario. HILS is a time-consuming process because of the customization of hardware as well as the need to change the plant model input-output signals to match the signals accepted by the ECU's hardware interface.

7.2 Limitations of Current XILS Environments

A majority of automotive companies currently use XILS environments in their control development process for rapid prototyping and early testing of design. Widespread adoption of XILS is limited because of the necessity of maintaining controller and plant models of different levels of fidelity and different types. We are working with Toyota to formulate the architectural requirements for their XILS environments. Currently, the plant models that Toyota creates do not have a well-defined structure, with a clear separation of the core dynamics, sensors, and actuators. Because of this, an engineer has to modify the

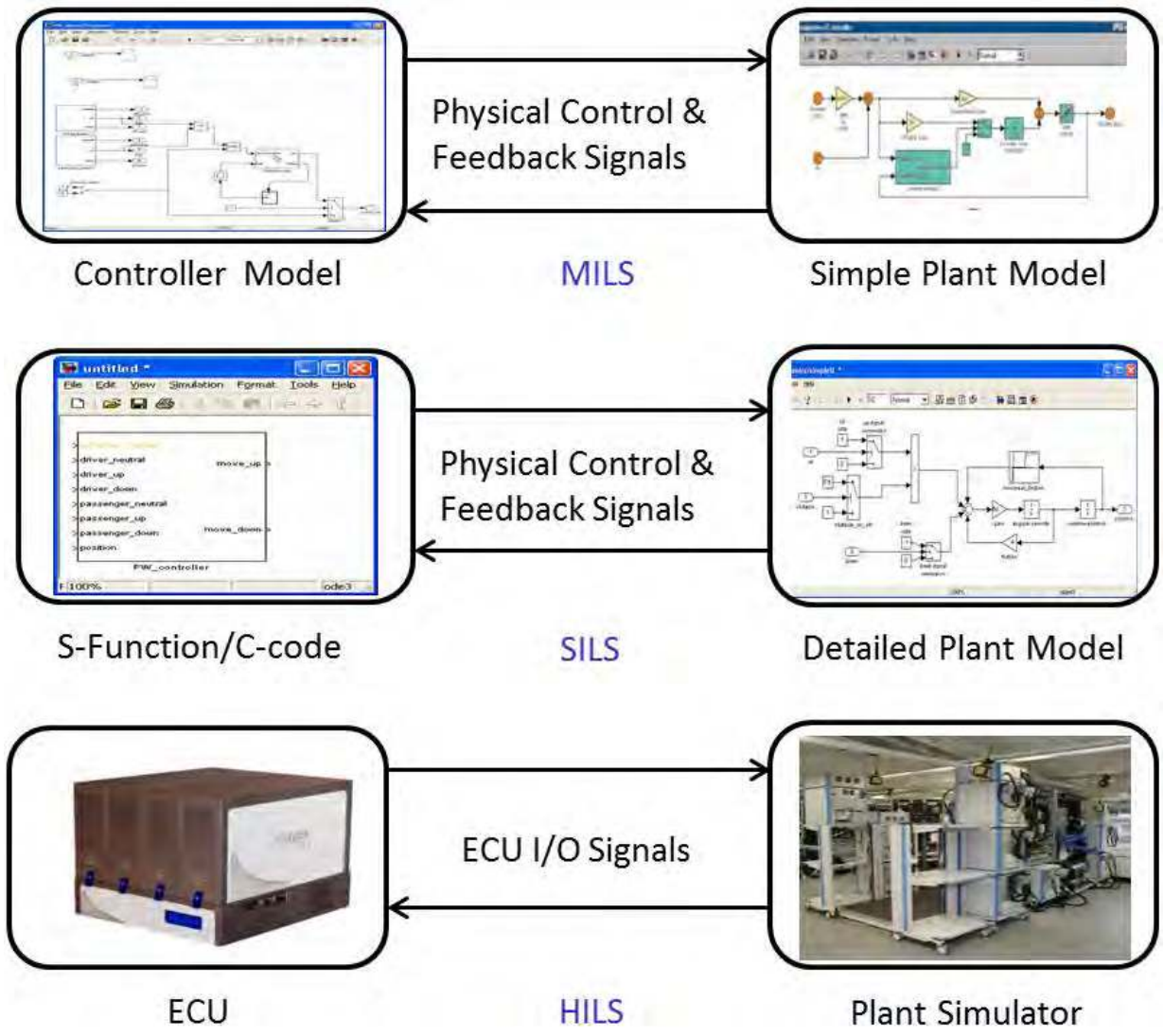

Figure 7.1: XILS scenarios for controller-plant testing.

existing models substantially if certain sensors or actuators are encapsulated in the plant dynamics. Similarly, Toyota's controller models have varying structure and missing or different components between XILS environments. For example, an engine model may need to be simulated with the assumption of varying air-fuel ratio. An injection control scheme is then necessary to keep the air-fuel ratio constant when the air charge model and the fuel behavior model are implemented independently. However, an engine may be modeled under the assumption that the air-fuel ratio is constant. In this case, the controller model is created without a fuel injection control module.

Multiple models also lead to variation in the number and types of signal connections between the controller and plant in different XILS scenarios. For example, typical control signals between an engine controller and the engine are the throttle opening angle (TOA) and fuel injection rate. At the MILS and SILS level, these are represented as physical signals with the units for TOA in *degrees* and the fuel injection amount in $mm^3/cycle$. However, in HILS the signals to be received at the ECU and the plant model are given by the TOA converted to voltage and the time for which the fuel injector remains open (On-Time) in μs.

Managing the different controller and plant models (and their interconnections) for each XILS environment, and converting models between environments is currently a manual, time-consuming, and error-prone process. For example, at Toyota a pair of spreadsheets is created for each environment, one for signals going to the ECU from the plant simulator and the other for signals coming from the ECU to the plant simulator. The spreadsheets describe the set of all signals that could be present for that XILS scenario as well as the subset of signals that are valid for particular controller-engine instances being tested. As a result, the spreadsheets are huge and very difficult to decipher. Before the start of a test, an engineer manually checks to see if the signal connections between the controller and the plant are valid (whether in software or hardware), and makes the necessary changes

to the model's internal structure and input-output blocks if there is incompatibility. The spreadsheets serve as a checklist for the engineer, but there is no automatic flagging of incorrect combinations of signals, nor any aid to make appropriate changes to the models.

The automotive industry requires a framework to manage the model variants in the XILS environments in a consistent manner. In particular, the following issues should be addressed:

- a way to standardize the physical architecture and input-output interface for plant models based on physical variables; and
- a way to enable controller and plant models to be structurally consistent between environments and to automate consistency checking as much as possible

The first requirement allows the sharing of the same plant model between different XILS environments with minimal or no change required. It also allows plant models with the same input-output behavior (but with varying degrees of detail) to be interchanged within the same environment, and exchanged between different environments. The second requirement reduces the error-prone manual process of modifying the internal structure of each new controller-plant instance, when testing in a XILS environment. Comparing simulation results with different controllers using the same plant model enables the degree of error in the algorithm to be clearly indicated, to determine if the simulation results are within range. This also makes it possible to determine if the cause of an error is on the controller or plant side because plant models would have been functionally checked earlier in the design flow.

7.3 Architectural Approach to XILS Testing

We address Toyota's requirements by using architectural views and view consistency to manage models in XILS environments for engine control testing. We have extended the CPS style to include thermal and fluid elements (described in Sec. 7.4) to create well-defined physical architectures for engine plant models. The physical architecture has a

standardized input-output interface based on physical effort and flow variables and their interconnections. The input-output physical interface is defined at the actuators and sensors of the plant architecture. Plant models with the same functional specification but varying in implementation detail are created using different actuator and sensor components, so that the model's input-output signals conform to the standardized physical interface defined for the associated XILS view.

We model a particular XILS level as an architectural view of the underlying system's base architecture (BA). The level of abstraction increases as we move from HILS to MILS, with the HILS environment being closest in detail to the BA. We use view relations to allow the engineer to define mappings between the elements in each XILS view and elements in the BA. Checking that an XILS controller-plant model is structurally consistent with the system becomes a question of whether the corresponding XILS architectural view is consistent with the system's BA. The architectural requirements for the controller and plant models for each XILS environment are provided by Toyota, based on their experience with in-house vehicle testing.

This case study illustrates the design flow where we have the freedom to create the base architecture and each view of the system from a clean slate, based on the requirements of the problem. There are no pre-existing legacy models that the BA or views are derived from. In fact, in this case, each model should be created with conformance to the associated view by construction. If there are existing models that need to be integrated, then their architecture can be checked with the appropriate view for conformance and completeness. Formally defining the architecture of controller and plant models in this fashion, along with the components that are different when switching between different XILS environments, allows different models to be plugged into an XILS test scenario, as long as the models maintain the same structure and input-output signal interface.

To apply our approach to XILS scenarios for engine control, we proceed as follows. To

create detailed plant architectures for the engine, the CPS family is extended to include thermodynamic and fluid elements, as described in the next section. In Sec. 7.5, we create the BA for the complete engine control system that includes the controller software and engine dynamics. The three variants of XILS environments are created as views of the BA in Sec. 7.6 for MILS, Sec. 7.7 for SILS, and Sec. 7.8 for HILS. We demonstrate how the structure of the controller-plant models changes as we move from a MILS view to a HILS view, and create mappings to capture the differences in element configurations.

7.4 ThermoFluid Family

The engine components belong to both the thermodynamic and fluid physical domains. In the thermodynamic domain, the effort variable is temperature T (in $^\circ$ Celsius) and the flow variable is heat flow rate q (in Joules per second). In the fluid domain, the effort variable is pressure P (in Pascal) and the flow variable is mass flow rate $\dot{m}$ (in kilograms per second). We have created a new physical subfamily, called the *ThermoFluid* family, that represents components and connectors from both the physical domains. Each component contains a single four-variable port, with the set of pairs of conjugate variable $\{(T, q), (P, \dot{m})\}$ from the two domains associated with each port. Since the semantics of the physical architecture is based on interconnection of elements and coupling of physical variables only, there is no restriction that the product of conjugate variables has to represent energy or power flow in general. This is less restrictive than the semantics of bond graph or port-Hamiltonian system representation, and allows us to create flexible architectural elements as needed by the application scenario.

Based on the commonly occurring physical phenomena in the two domains, we have created a set of component types that are summarized in Table 7.1. Some examples are: *Pipe* to represent transport of fluid (and associated heat energy) from one point to another, *Vessel* to represent the storage of fluid in an enclosed volume, *Valve* to model a physical valve whose opening and closing is controlled with an external signal, *Reservoir* to

represent a flow source or flow sink in the *ThermoFluid* domain. The ability to create a new physical architectural family flexibly to suit the requirements of new modeling formalisms and systems is useful in scenarios like the XILS case study. A *Thermofluid* connector between two components represents an incompressible fluid flow from one component to the other. The semantics of connectors enforce that the pressures and temperatures at the connected ports are equal, and the mass flows and heat flows from all connected ports should sum to zero. Hence, the connector semantics represent the laws of conservation of both mass and energy for the fluid flow.

We can connect a *Pipe* component to a *Vessel* component by a thermo-fluid connector. The connector semantics enforce that the pressure and temperature at the connection interface of the pipe and vessel are equal, and the fluid and heat flowing out the pipe is transferred into the volume without any losses. Hence, the semantics of the *ThermoFluid* family are physically meaningful and allow the representation of the heat exchanged due to the air-fuel mixture flowing between engine components, as well as the temperature and pressure changes occurring due to this flow in the complete engine system. All existing elements from the physical family are inherited in the new *ThemoFluid* family. In particular, elements from the *Mechanical* family, with torque (or force) and angular (or linear) position as the conjugate variables, are also included since these are needed to model the mechanical load to the engine.

7.5 Base Architecture

The base architecture for the underlying system consists of the physical architecture of the actual engine, the architecture of the control software running on the ECU, and the interface elements converting the software commands into physical signals and sensed physical variables into cyber readings.

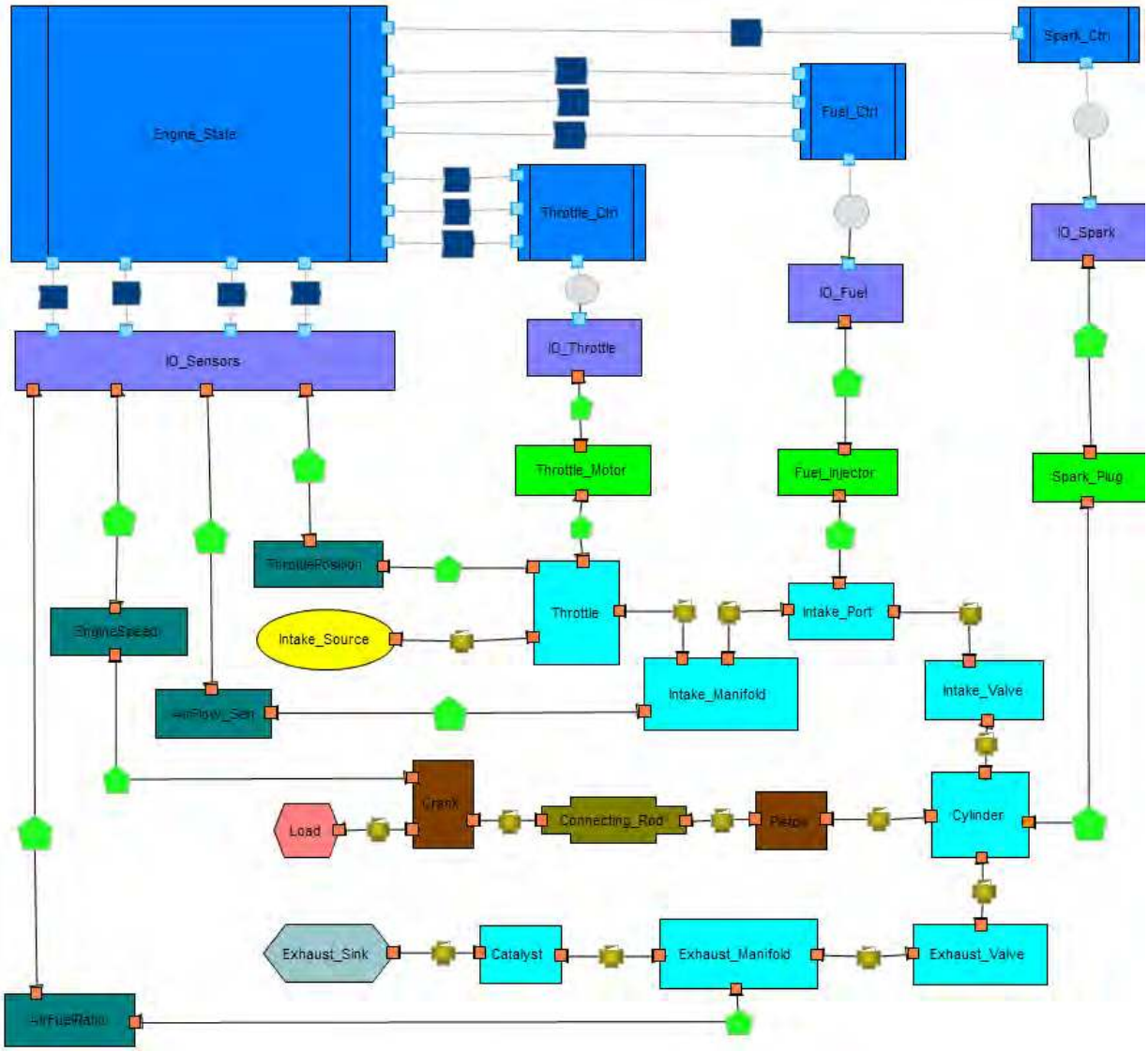

Figure 7.2: Base Architecture for the engine control system.

Table 7.1: Elements of Thermo-Fluid family.

Element	Description
Pipe	Transport of non-compressible fluid between two points
Vessel	Storage of fluid in an enclosed volume
Valve	Physical valve controlled with an external signal
Reservoir	Infinite volume to model flow source or flow sink
Cylinder	A *Vessel* with mechanical piston and input and output ports for fluid flow
Manifold	A *Pipe* with behavior annotations for different types of fluid flow
ThermoFluid Connector	Equates pressure and temperature between connected ports. Enforces conservation of mass and energy between connected components.

7.5.1 Engine Architecture

The engine architecture shown in Fig 7.2 represents a generic version of a single cylinder internal combustion (IC) engine. The intake system consists of *Intake_Source*, *Throttle*, *Intake_Manifold*, *Intake_Port*, and *Intake_Valve* components, and is a simpler representation of the real-world system. For example, the complex dynamics of the air and fuel mixing are not modeled and a homogenous mixture is assumed throughout the intake stage. The *Intake_Source* represents the flow of the air-fluid mixture from the environment at a constant air-fuel ratio (AFR) into the engine. The connector between the *Intake_Source* and the *Throttle* represents fluid flow as well as pressure equalization between the components. The *Throttle* regulates the flow of air-fuel into the *Intake_Manifold*, based on the throttle position set by the *Throttle_Motor* actuator. The *Intake_Manifold* represents a physical transport for the mixture into the *Cylinder* through the *Intake_Valve*. The *Intake_Valve* does not store any flow quantities such as mass or heat energy. However, the temperature and pressure can be different at each end of the valve. The *Intake_Port* represents the injection of fuel into the engine, based on the fuel quantity set by the *Fuel_Injector*

actuator.

The *Cylinder* has a fluid input port for air-fuel flowing into the component and a fluid output port for exhaust gases flowing out. The *Spark_Plug* actuator is connected to the input port of the *Cylinder* by a physical signal connector. The connection represents the timing of the spark in units of *degrees Before Top Dead Center (BTDC)*. In addition, the *Cylinder* is connected to the moving *Piston* by a mechanical (translational) connector that represents the conversion of chemical energy released during the burning of fuel into translational mechanical motion of the *Piston*. A *Connecting_Rod* is a physical transducer component with a translational port on one end and a rotational port on the other. It represents the conversion of translational motion to rotational motion. The *Crank* represents the crank shaft and all connected subsystems, and is a purely rotational mechanical component. The *Crank* delivers mechanical power to the *Load*, which models all vehicle systems (including air-conditioning) that place power demands on the engine. The set of mechanical connectors from the *Cylinder* to the *Load* represent the torque (force) and angular (linear) displacement at various points in the mechanical subsystems of the engine model.

The *Exhaust_Valve* component is identical in function to the *Input_Valve*, except that the burnt products from the *Cylinder* flow through it. The *Catalyst* represents the filtering of emissions from the exhaust before they reach the *Exhaust_Sink* component. The *Exhaust_Sink* is a *Reservoir* because it represents the unidirectional flow of gases into the environment following the exhaust stroke of the engine operation.

The dynamic behavior of each component is represented through mathematical equations, expressed in terms of rates of change of conserved quantities such as mass, momentum and energy, and other relevant physical parameters such as density, and mass fractions of the species. These equations can be annotated as properties of individual components and connectors for different kinds of analyses. The engine model reflects the four strokes of

the engine cycle: intake, compression, power, and exhaust strokes. The interacting components represent the following physical processes: the flow of air-fuel mixture from the intake system to cylinder, and that of the exhaust gas from the cylinder to the exhaust, the combustion of the air-fuel mixture into the exhaust product and the subsequent release of energy in the cylinder due to combustion, and the mechanical action of the crank mechanism.

7.5.2 Controller Architecture

The engine controller architecture shown in Fig 7.2 represents the software running on the ECU, and is made up of four main components. The *Engine_State* component receives all sensor values from the engine and determines the current state, i.e., whether the engine is starting up, accelerating, or in warm or cold state. The engine state is given as an output to the *Spark_Ctrl*, which controls the timing of the spark plug (in degrees BTDC) based on the current state. The *Engine_State* component also passes all sensor readings (including the engine state) to the *Throttle_Ctrl* and *Fuel_Ctrl* controller components. The *Throttle_Ctrl* calculates a new throttle position (in degrees) based on the engine state, throttle position, and engine speed. The *Fuel_Ctrl* calculates how much fuel should be injected (in $mm^3/cycle$) based on the engine state, air-fuel ratio, and air-flow.

The *IO_Throttle* C2P transducer converts the throttle position command (in degrees) from *Throttle_Ctrl* into a corresponding voltage signal that is sent to *Throttle_Motor*. This component represents the I/O software and hardware that translates setpoints in physical units from the controller into the electrical signals required by the throttle hardware. Similarly, the *IO_Fuel* component converts the commanded fuel injection amount (in $mm^3/cycle$) to an electrical signal that remains high for a time proportional to the setpoint. The *IO_Spark* C2P transducer converts the controller command in *degrees BTDC* to a proportional voltage signal. The *IO_Sensors* transducer components perform the function of converting physical sensor readings from the plant into physically meaningful units

that the *Engine_State* and controller components recognize. For example, the *EngineSpeed* sensor outputs the speed as a series of electrical impulses per second, while the controller software assumes the units to be in revolutions per minute (RPM). The *IO_Sensor* is responsible for all such conversions so that the input to the controllers is in physical units that are standardized across controller models.

7.5.3 Standardized Input-Output Interface

The *IO* transducer components between the controllers and the plant's actuators and sensors structure the control software to maintain a consistent set of input-output signals based on physically meaningful units. For this engine control case study, the controller outputs are defined as the throttle position (in degrees), the fuel injection amount (in $mm^3/cycle$), and the spark timing (in degrees BTDC). The controller inputs are defined as throttle position (in degrees), engine speed (in RPM), air-fuel ratio (dimensionless), and air flow rate (in kg/sec). The *IO* components represent the software (and associated hardware) that is responsible for converting between the types of signals needed by the controller software and the types of signals present at the actuator-sensor interface of the plant (physical or model).

Similarly, the architectural separation of a plant's core dynamics from the actuators and sensors makes it possible for all plant models adhering to the physical architecture to be interchangeable, as long as they maintain the set of physical input-output variables defined at the actuator-sensor interface. For the engine control example, the plant outputs are defined as throttle position (in degrees), engine speed (in RPM), air-fuel ratio (dimensionless), and air flow rate (in kg/sec). The plant's inputs are defined as the throttle position (in degrees), the fuel injection amount (in $mm^3/cycle$), and the spark timing (in degrees BTDC). For different XILS environments, the actuator components define the transformation between the types of control signals received and the types of physical signals used in the plant model.

In a MILS scenario, the throttle control signal is in degrees, i.e., the same physical units that are used in the MILS plant model. Hence, the *Throttle_Motor* actuator in a MILS model will not contain any dynamics to convert the input command to degrees. However, for a HILS setup, the control signal for throttle position is a voltage. Hence, the actuator component in a HILS plant model will contain dynamics that relate how the input voltage affects the movement of the throttle into the desired position (in degrees). In a similar way, the sensor components allow consistent physical variables to be used across all XILS plant models, and model the translation of measured quantities into output signals that are required for the particular XILS environment. For example, the engine speed sensor in a MILS model will output the value in RPM, as required by the controller. However, for a HILS, the sensor component will have to model the translation of RPM into an impulse train, which is the input required by the ECU hardware.

In the following three sections, we create the architecture of each XILS environments as a view of the base architecture, and identify the changes to the structure of each view through element mappings.

7.6 MILS View

The MILS view is shown in Fig. 7.3 with those elements highlighted that are different with respect to the BA. The controller architecture in the view is chosen to be the same as the underlying BA. The plant model for a MILS is generally simpler than in SILS or HILS. This is reflected by the encapsulation of the *Throttle* and *Intake_Manifold* in the BA to a single *Intake_System* in the view. Similarly, the dynamics of the crankshaft, rod, and piston assembly have been encapsulated in a single *Crank_System* view component, as shown in Fig. 7.4. We note that the MILS architecture enforces that the *IO* components and the actuator-sensor interface are included in every model that conforms to (or is created from) the MILS view. This makes it possible to use different models with varying levels of details in the same MILS environment, provided that each model's structure and input-output

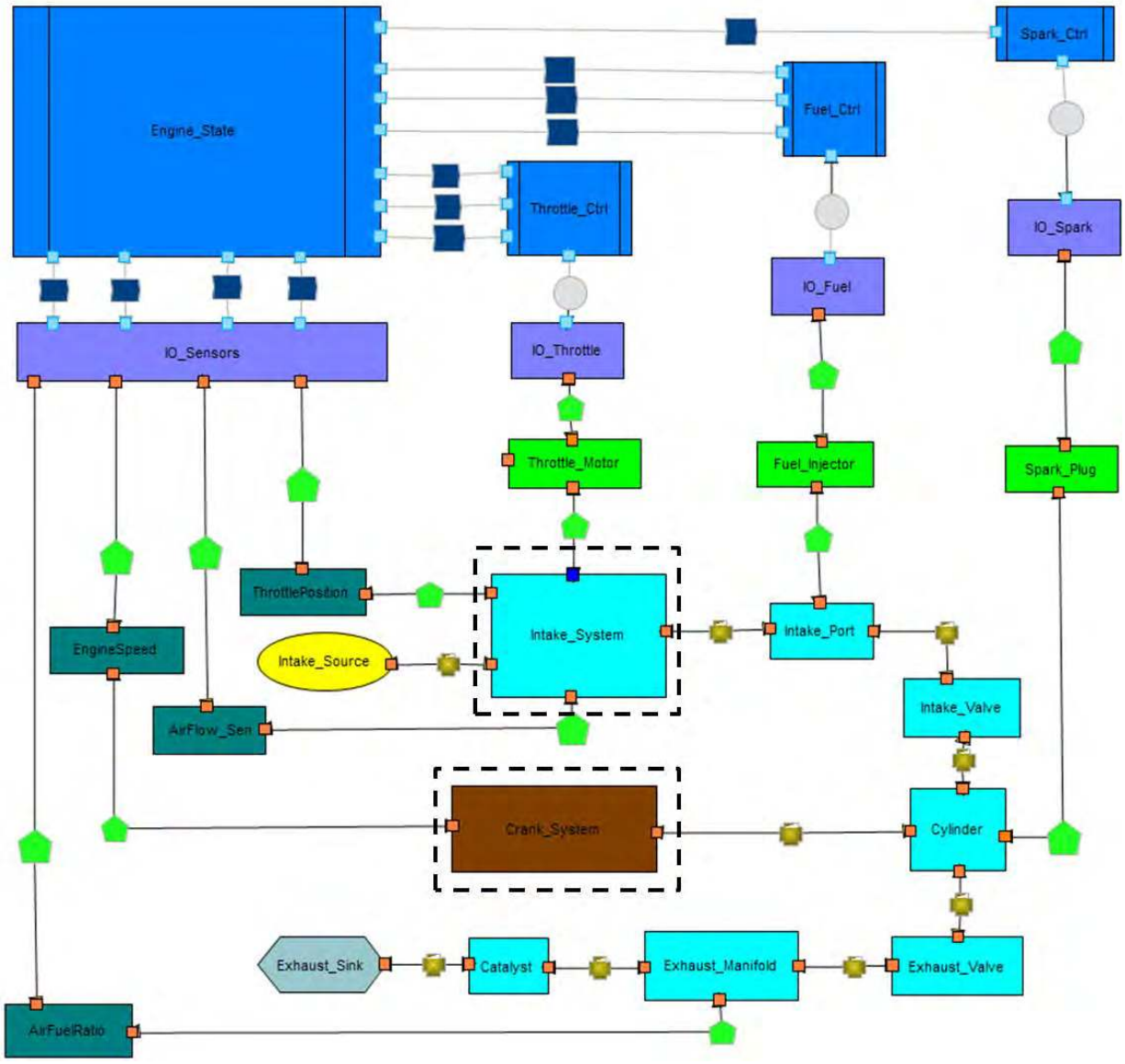

Figure 7.3: MILS view of the engine control system.

interface adheres to the architecture of the MILS view.

The signals exchanged between the controllers and the plant are in physical units, because a MILS environment does not include any platform-level details. Hence, the *IO* components, and actuator-sensor components simply pass through all input signals, since no transformation has to be performed on the signal set.

7.7 SILS View

The SILS view is shown in Fig. 7.5 with those elements highlighted that are different with respect to the BA. The SILS view contains the same types and number of controller elements as the MILS, since the controllers in SILS are created by generating code from the corresponding MILS models. However, the plant models can be significantly different, depending on what requirements for the controller are being validated. In Fig. 7.6, the *Throttle* and *Intake_Manifold* components in the BA have been encapsulated as the single *Intake_System* component in the view. The mapping captures the abstraction that the associated SILS model makes about the physical elements in the underlying system, and is used to check the structural consistency of every model used for SILS testing. The crank shaft assembly has a more detailed structure compared to the MILS view. The signals exchanged between the controller and plant are still in physical units. However, some platform-level details, such as fixed-point data types, may require the *IO* components to contain logic to convert between data formats in going from controller to plant.

As demonstrated for MILS and SILS, the standardization of the physical signals for plant models is implemented by defining the same set of sensor-actuator components for each XILS view. The components have different implementations in each XILS scenario and are responsible for converting between the physical signals in the plant model and the signals required by the connected controller software for that scenario. Similarly, the standardization of controller signals is implemented by defining *IO* transducer components that are responsible for converting between signals required by the controller software and

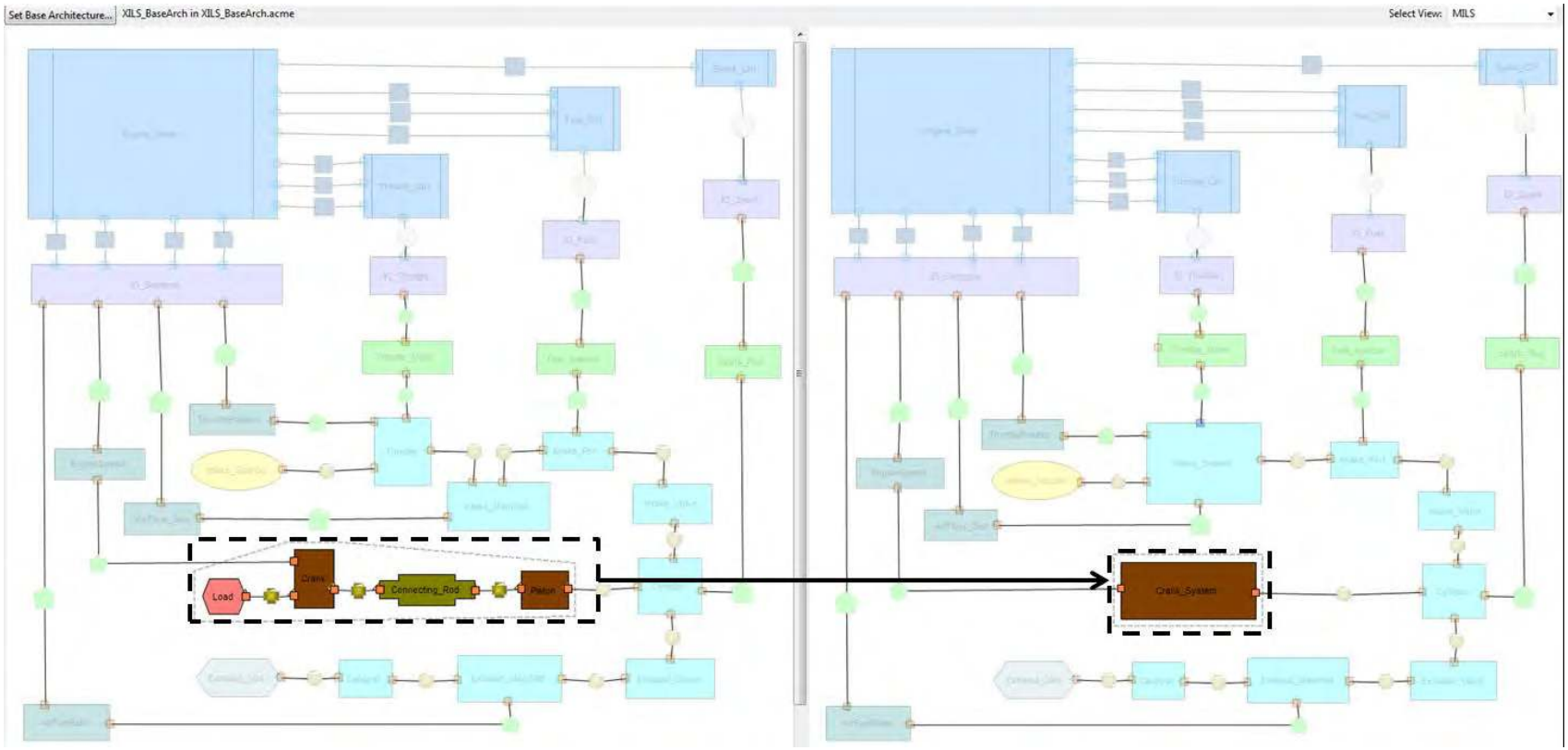

Figure 7.4: Encapsulation of crank shaft subsystem in MILS view.

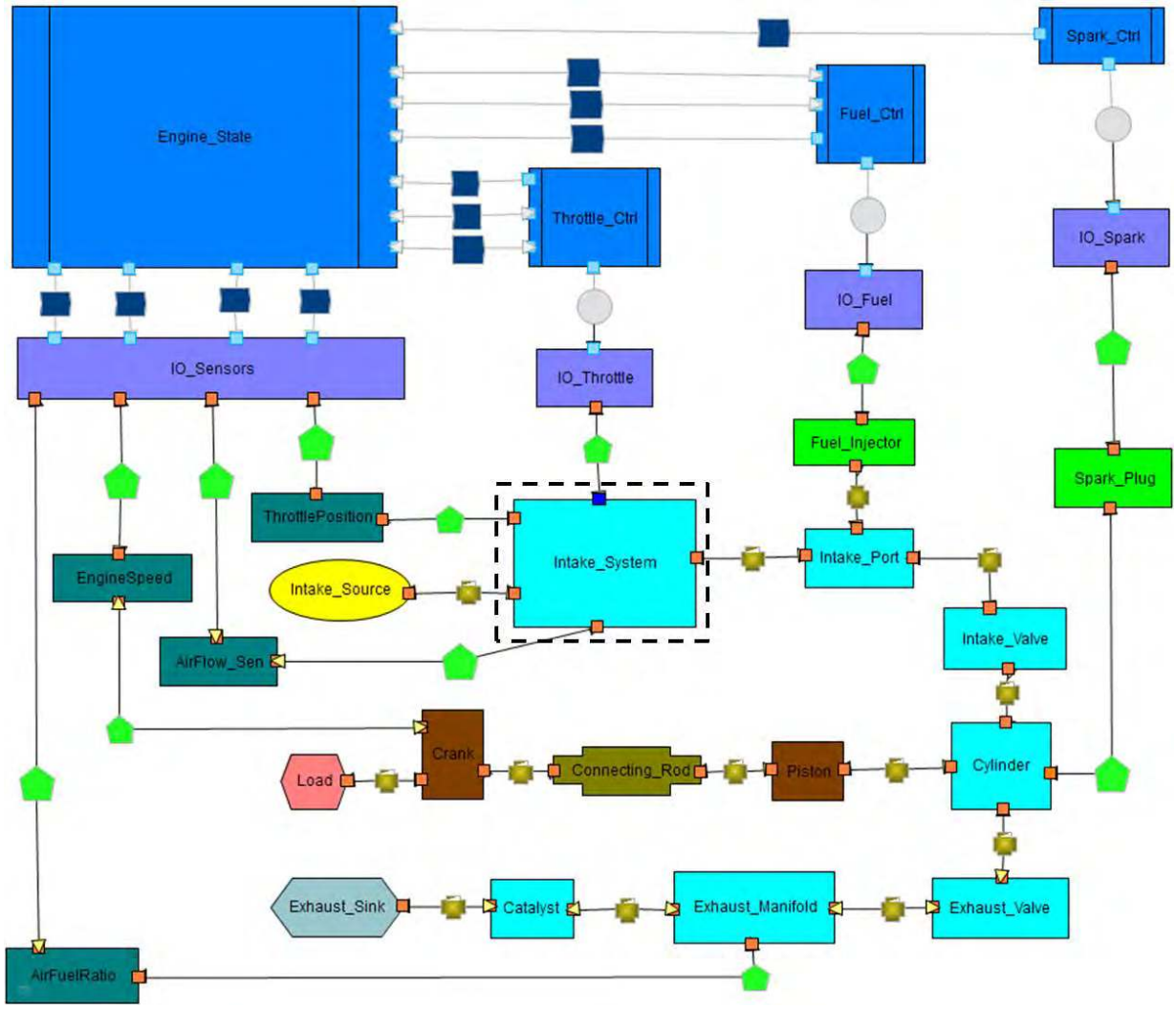

Figure 7.5: SILS view of the engine control system.

the signals exchanged with the hardware and plant.

7.8 HILS View

The HILS view is created from the requirements given to us for typical models used in the HILS test scenarios. The controller components (including the *IO* transducers) are identical to the BA, since the code used in HILS is supposed to be exactly the same as that used in the production vehicle. This assumption is reflected in the HILS view shown in Fig. 7.7 with those elements highlighted that are different with respect to the BA. The number and types of cyber components, and their interconnections, are unchanged from the corresponding elements in the BA.

However, the structure of the plant model used in a HILS test is, in general, different from the actual plant in the production vehicle. This is due to several reasons. One reason is the plant model has to be simulated in real-time to keep up with the sampling rates of the ECU controllers. Hence, many detailed subsystems are replaced by lookup tables or coefficients from test data to speed up the performance of the model. The second reason is the use of actual hardware to replace some parts of the plant model while testing. This is done when certain physical parts of the engine cannot be modeled with the desired fidelity and performance together, and replacing those subsystems in the plant model with the actual hardware is easier/more accurate for controller validation. An example of this is highlighted in Fig. 7.8, where the *Throttle* and *Intake_Manifold* in the BA is replaced by an *Air_Estimation* component in the HILS view. In addition, there is an additional physical component *Throttle_Hardware* to represent the actual throttle. The throttle position is now measured from *Throttle_Hardware* actuator, which makes the HILS view inconsistent with respect to the BA.

This change is necessitated for the following reasons. First, the behavior of the throttle for HILS is tested by using the actual hardware to get accurate response times, and the position measured by the throttle sensor is used by the ECU. Second, the controllers in

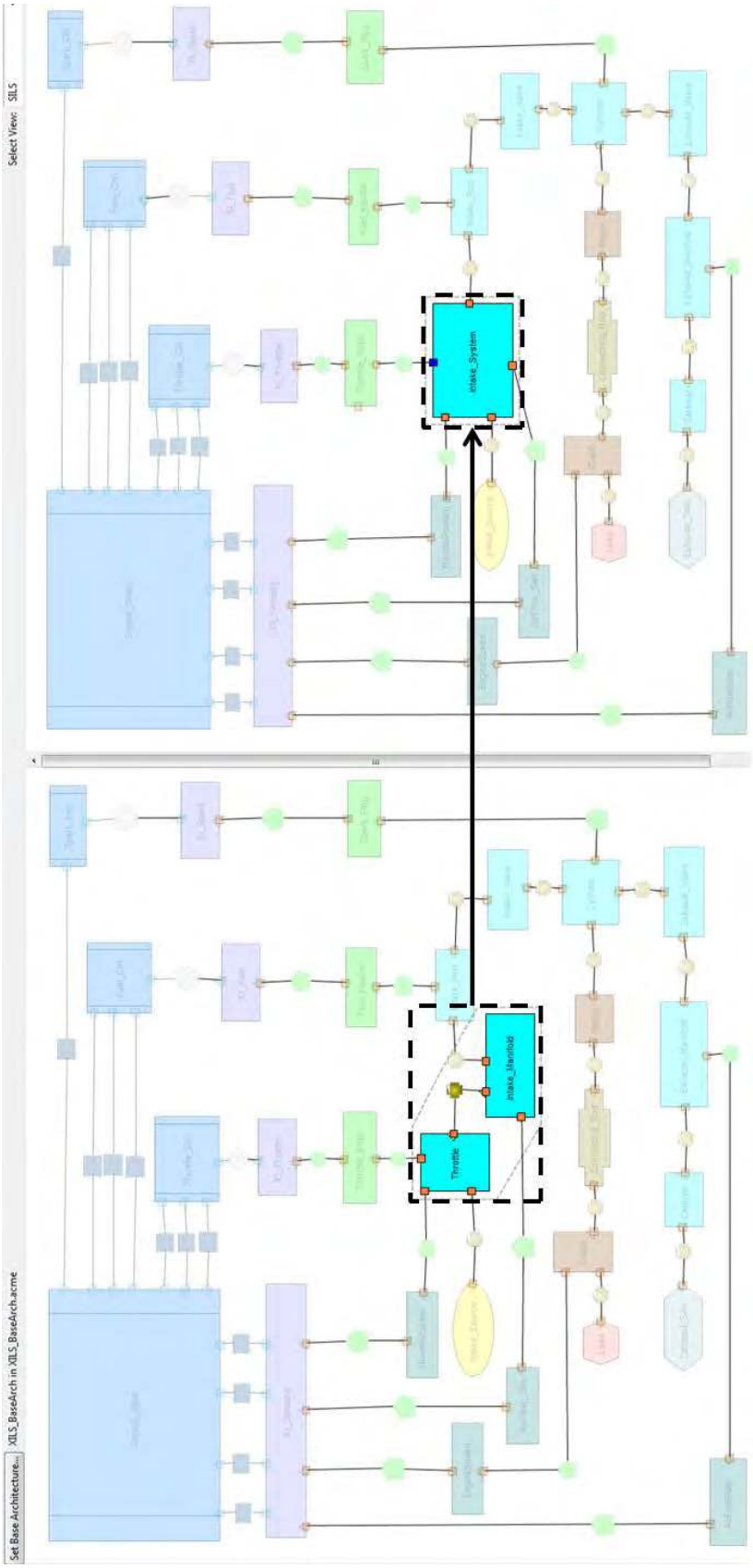

Figure 7.6: Encapsulation of intake system in SILS view.

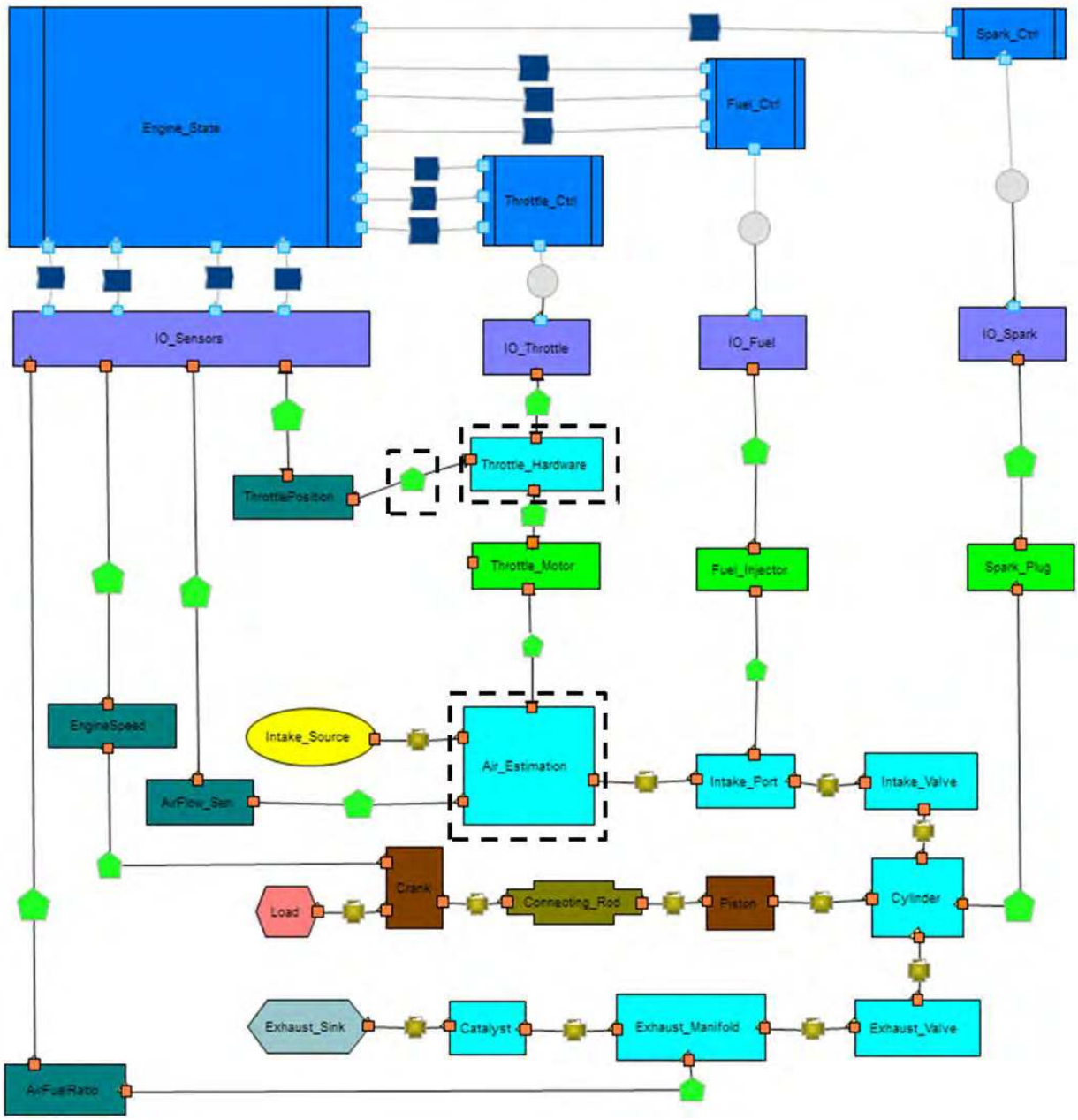

Figure 7.7: HILS view of the engine control system.

the ECU use a lookup table for estimation of airflow, based on the engine state. To ensure that the behavior of the plant model (with respect to air estimation) is consistent with that assumed in the ECU, the intake system in the plant model is replaced by an air estimation component that uses the same calculations as those used in the ECU. The controller software sends the throttle command to the throttle hardware, the hardware sends its position as feedback to the ECU, and also sends a physical throttle command signal to the *Throttle_Motor* component in the HILS plant model. the *Throttle_Motor* contains logic to convert the electrical signal units into physical units for TOA (in degrees) and this is given as input to *Air_Estimation*. The *Air_Estimation* component uses the throttle angle, along with the physical signal from the *Intake_Source*, and calculates the required mass flow rate that flows to the *Intake_Port*. The architecture of the rest of the HILS plant model remains the same as that of the BA.

The HILS view is an example of an architectural view that is inconsistent with the BA by design. The view conformance check highlights the inconsistent elements, which are the throttle hardware component and the throttle position connector. This ensures that when any model that conforms to the HILS view is being tested, the engineer is cognizant of the assumption that an air estimation module is being used in the model in place of the physical dynamics of throttle and intake manifold. The software represented by the *IO* components for HILS is responsible for transforming between the electrical signal units defined by the ECU interface and the physical units required by the controller software. Similarly, the actuator-sensor interface converts between the electrical signals at the I/O interface of the PC running the plant simulator and the physical signals used in the plant model itself, as described for the *Throttle_Motor* component.

7.9 Summary

In this chapter, we demonstrate the use of architectural views to manage variants of models used in "In-Loop-Simulation" testing scenarios in the automotive industry. To create

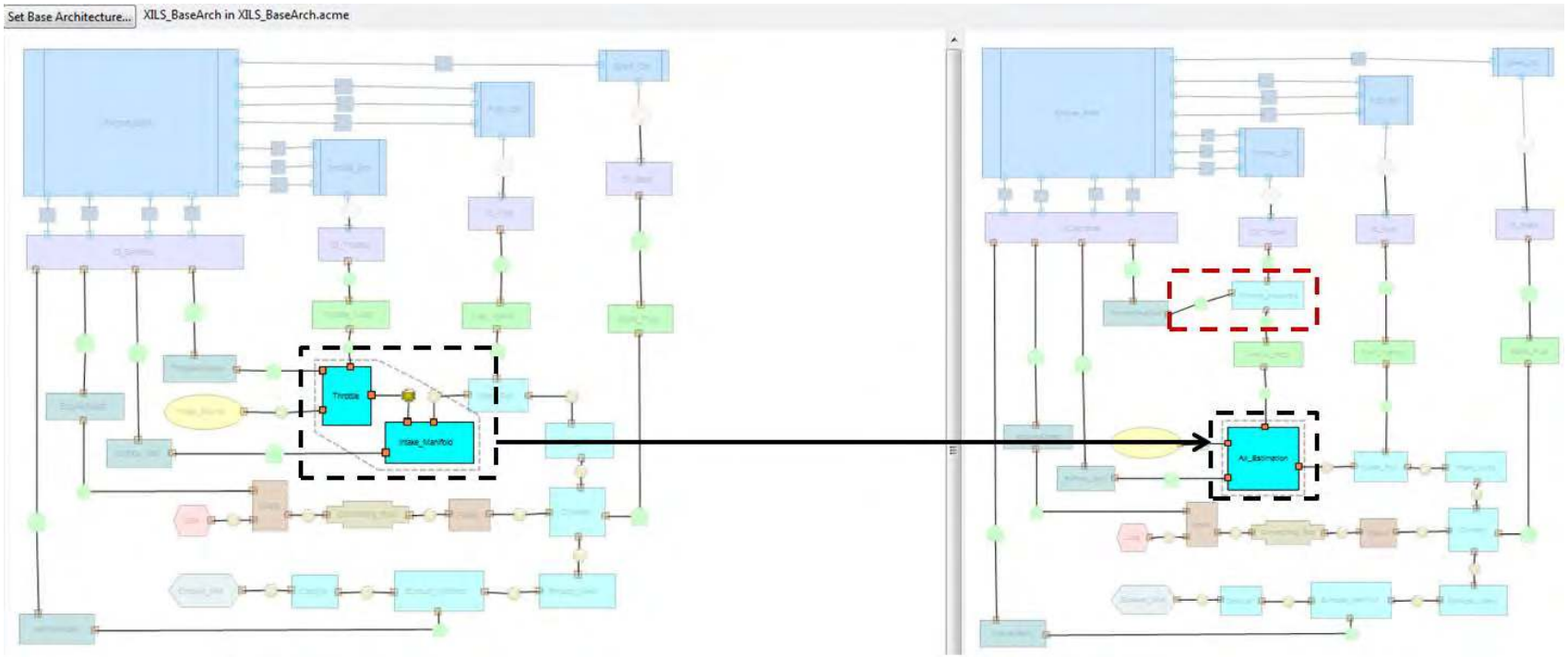

Figure 7.8: Encapsulation of air estimation system in HILS view.

a detailed physical BA of the engine and controller testbed, we extend the physical family to the thermal and fluid domains. We create the architecture for each XILS scenario as a view of the underlying BA of the vehicle testbed. View mappings capture the architectural abstractions between the BA and each XILS environment. We check the structural consistency of each XILS view with the BA to highlight the changes in structure between the different testing scenarios. Currently, we have not created any controller or plant simulation models to validate the XILS approach. However, the assumption we make is that all models will be created with structural conformance with the associated view.

The standardization of the physical signals for plant models is implemented by defining the same set of sensor-actuator components for each XILS view, with different implementations for physical signal conversion for the associated XILS environment. Similarly, the standardization of controller signals is implemented by defining *IO* transducer components that are responsible for converting between signals required by the controller software and the signals exchanged with the hardware and plant.

Chapter 8

Conclusions

In this final chapter, we summarize the technical contributions made in this thesis and suggest several directions for future research.

This thesis presents an architectural approach to relate the structure and semantics of heterogeneous models used in CPS design. There are two main contributors to today's exponential growth in the cost of designing CPSs [9]. The first is the lack of support to detect design defects as early as possibly in the design process. The second is the existence of multiple versions of the system in different analysis models. These impact the following areas of system design:

- The number of system-level faults introduced during the design phase.
- The number of system-level faults detected during the design phase.
- The cost of fault removal during the complete life cycle.
- Limited ability to use analyses from different models to derive system-level properties because of multiple versions of the underlying system.

Our approach impacts each of these areas in the following ways. Because there is a single, unifying representation of the system in terms of its base architecture, there is a single *ground truth* that every system model has to be *structurally consistent* with. The important constraint is that the presence or absence of ports and connectors in either the architectural

view or the BA must be reflected in the other structure through one-to-one or many-to-one associations from BA to view. Such a constraint rules out the possibility that a view can introduce a "back-door" communication channel not present in the reference structure, a property called *communication integrity* in software architectures [20]. This helps in reducing the number of mismatched assumptions introduced in a model about the structure and connectivity of system elements, and directly impacts the *number of faults introduced* during design. This is demonstrated in the XILS case study (refer to Chap. 7), in which the creation of a standardized architecture for system models in different scenarios limits the potential of a model being created with wrong assumptions about the system.

Checking a view for consistency with the BA every time the associated model is modified can catch structural errors early in the design flow, directly impacting the *number of faults detected*. This is demonstrated by the STARMAC case study (refer to Chap. 6), in which multiple wrong assumptions about component connectivity and types of data exchanged were caught by the view conformance and view completeness checks. When errors are caught in the design phase rather than the integration phase, the *cost of fault removal* is impacted and reduces by a large factor [1]. The *ability to use analyses from different models* is also enhanced when each model is structurally consistent with the BA. This is because structural mismatches can point to implementation errors or behavioral mismatches in many cases, as demonstrated by the missing altitude controller being detected by the missing sonar connectors in the STARMAC control view consistency checks.

8.1 Contributions

There are two fundamental shortcomings of current architecture modeling capabilities that limit their potential to fully address the engineering problems of large-scale, heterogeneous CPSs: [i] limited vocabulary to represent physical elements and their interactions; and [ii] inadequate ways to support consistency relations between heterogeneous architecture views of the same system.

This thesis addresses the first shortcoming through the development of an architectural style that serves as a common representation of the complete system, and the second shortcoming through the abstraction of architectural views to compare the structure and semantics of corresponding heterogeneous models to the CPS architecture. In particular, we make the following contributions:

1. **Extending architecture to represent continuous dynamical physical elements and their coupling.** We have created a CPS architectural style that supports a unified representation of both physical and cyber elements and their interactions in the same architectural framework. This ability allows the architect to create a common base architecture (BA) for a CPS that provides a unified point of reference for multi-domain system models.
2. **Using architecture as the common system representation to relate the structure and semantics of heterogeneous models.** The architectural view is used as the mechanism to 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 are used to identify and manage semantically equivalent elements (and their relations) between each model and the underlying system.
3. **Defining and evaluating consistency between architectural views and the system's BA.** View consistency defines when an architectural view conforms to the structural and semantic constraints imposed by components and connectors in the system's BA. Such a notion of consistency ensures that the model elements adhere to the connectivity constraints and physical laws present between elements in the BA. This guarantees that the models used for design and evaluation are not based on assumptions about the system's design that are inconsistent with the actual system as reflected in the BA. We define view consistency as the existence of an appropriate morphism between the typed graphs of a view and the BA. Depending on the type

of morphism present, two notions of consistency are defined: *view conformance* and *view completeness*.

4. **Tools for automated consistency checking of architectural views.** The first tool is a mechanism (Acme Maps) to define the types of maps possible between each view type and the BA as well as map instances between the elements of a view and the BA . Acme Maps is an extension of the core Acme Architecture Description Language (ADL) that allows the creation and type checking of view relations and element correspondences. The second tool is a graphical editor to compare views visually, define element correspondences (including encapsulations) that must be maintained, and display the results of consistency checking to the user. The third tool is a set of graph morphism algorithms that find the largest set of semantically consistent element mappings between a view and the BA, based on the pre-mapped elements and the type compatibility defined by the user. All tools are implemented as plugins in the AcmeStudio architecture design framework [10], so that they can be extended easily for future enhancements.

5. **Evaluation of multi-view architecture framework.** We illustrate the application of our architectural approach with two case studies. The first one demonstrates how heterogeneous models of an avionics system (the STARMAC quadrotor) can be created as views of the baseline architecture. The choice of the modeling domains is motivated by the analysis and verification activities typically found in an embedded control system design process. We apply the consistency check to each view and highlight the mismatches that are detected between the models and the actual implemented system. The second case study illustrates the usefulness of architectural views to manage model variants for *X-In-the-Loop Simulation* (XILS) environments for engine control of vehicles. We show how each simulation scenario can be captured as a view of the system under test, and how view consistency can help the engineer in

checking consistency of controller-plant models between various XILS environments.

8.2 Future Work

The architectural consistency of multiple views (and associated models) is a rich problem with many directions where extensions can be made.

- **Translating from models to views.** Currently, there is limited support for translating from a domain-specific model to the corresponding architectural view in a formal or an automated way. This capability is important to enable the practical use of an architectural framework in multi-domain CPS design. Semantic anchoring [59] is a promising approach to transform between system models that concentrates on the specification of the dynamic semantics of each model's DSML and the relationship to pre-defined DSMLs. One possibility is to leverage the machinery of semantic anchoring to define transformations between the DSMLs of system models and the architectural style of each view.
- **Automatic creation of view maps.** Currently, all element maps between views are created manually, which is not scalable for large systems. To make our approach applicable in practice, support for automated map creation is needed. Graph grammars provide a theoretical foundation to transform from source to target patterns between two graphs and could be useful to generate specific maps between two views. In addition, view maps could be used to focus the search space of the consistency checkers. For example, if all view elements are mapped to a small part of the BA, then the consistency checkers could use this information to limit the search for morphisms over that part alone.
- **Richer analysis of views and relations.** New types of view relations have to be added to support different kinds of view analyses. For example, an important relation between system components is the fact that they should *not* be connected

in a view. This constraint can be useful when two components are replicas of each other, and a structural constraint for not deploying them on the same processor (to avoid common failure) has be added to the architecture.

View relations can also be used to define metrics for model and system-level coverage. For example, a simple metric to evaluate model coverage by a view would be the number of encapsulations being made in the view. Too many encapsulations point to a lot of abstraction about the model. Another intuitive metric for system coverage is to check whether all system elements have been represented in at least one system view. If a set of elements is not present in any view, it might point to incomplete analysis of the underlying system. In a similar way, if a certain set of system elements are present in a majority of the views, this set may need to have stricter behavioral and structural checks, since these elements impact multiple design concerns.

- **Support for variants in base architecture.** Currently, there is limited support in the approach for the BA to represent product line variants. This is especially useful in the automotive industry, where onboard diagnostics, entertainment systems, and safety features all lead to a system architecture with multiple optional elements. All these have to be represented in a unified way in the BA, and each view has to be created with the missing elements made explicit.

- **Analyzing map types.** We currently create the map types between a view style and the base architecture style based on engineering judgement and domain knowledge of the design concern. However, creating map types manually could lead to infeasible or inconsistent map instances between two views. Although there are numerous tools to help in the analysis of architectures for individual systems, relatively less work has been done on tools to help in the design of architectural styles. In particular, there is no support (to the best of our knowledge) to analyze map types between two architectural styles. Kim and Garlan have shown how to analyze an architectural style

formally by mapping it into a relational model that can be checked for consistency properties [60]. Such work could be extended to analyze the consistency of map types between styles as well.

- **Creating a formal semantics for Acme Maps.** Currently, Acme Maps does not have a formalized semantics, in contrast to Acme's semantics. We would like to integrate Acme Maps into Acme, so that relations between views become a first-class construct of the ADL. We believe that the problem of creating well-defined structural maps between architectures holds promise in enabling general consistency checking between different views.
- **Semantic relations between views.** Our approach is limited to structural consistency between architectural views with limited checking of semantic properties. The work has to be extended toward the specification and analysis of inter-view (and inter-model) consistency at the semantic (behavioral) level. In this context, well-defined structural mappings between a view and the base architecture can be used as the basis for identifying and managing the semantic dependencies among the various analysis models, and to evaluate mutually constraining design choices. This concept forms the basis of our initial work in this area [47]. There is also a need to incorporate the assumptions of the different analysis algorithms used in system models for semantic checking. A first step in this direction is the approach proposed by Nam, de Niz et al. [61].
- **Tool support for multi-view consistency.** The current tools for architectural design (including those developed by us) are not yet scalable to large-scale systems in terms of usability and performance. The current morphism and MCS algorithms will have to be implemented to take advantage of today's parallel processing hardware, if they are to be used in real systems. The creation, editing, and display of hierarchical architecture views and their maps have to be enhanced for practical use as well.

Bibliography

[1] L. W. P. Feiler and J. Hansson, "System architecture virtual integration: A case study," in *Proc. of the Embedded Real Time Software and Systems Conf. (ERTS^2 2010)*, 30 Nov. 2010.

[2] D. Garlan, R. Monroe, and D. Wile, "Acme: Architectural description of component-based systems," in *Foundations of Component-Based Systems* (G. T. Leavens and M. Sitaraman, eds.), pp. 47–68, Cambridge University Press, 2000.

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

[4] "NSF Cyber-Physical Systems Program." `http://www.nsf.gov/pubs/2010/nsf10515/nsf10515.htm`.

[5] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing domain-specific design environments," *Computer*, vol. 34, no. 11, p. 4451, 2001. `doi:http://dx.doi.org/10.1109/2.963443`.

[6] D. Galin, *Software Quality Assurance: From Theory to Implementation*. Pearson/Addison-Wesley, 2004.

[7] NIST, "The economic impacts of inadequate infrastructure for software testing," Planning Report 02-3, National Institute of Standards, 2002.

[8] J. Hansson and S. Helton, "ROI Analysis of the System Architecture Virtual Integration Initiative," Technical Report SEI-2010-TR-001, 2010, Software Engineering Institute, Carnegie Mellon University, 2010.

[9] P. Feiler, "Challenges in validating safety-critical embedded systems," *SAE International Journal of Aerospace*, vol. 3, pp. 109–116, 2010.

[10] D. Garlan and B. Schmerl, "Architecture-driven modelling and analysis," in *11th Australian Workshop on Safety Related Programmable Systems, SCS'06*, vol. 69, Conferences in Research and Practice in Information Technology, 2006.

[11] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.

[12] D. Perry and A. Wolf, "Foundations for the study of software architecture," in *ACM SIGSOFT Software Engineering Notes*, vol. 17(4), 1992.

[13] P. Kruchten, "The 4+1 view model of architecture," *IEEE Software*, vol. 2, no. 6, pp. 42–50, 1995.

[14] N. Rozanski and E. Woods, *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, 2005.

[15] "ISO/IEC Standard for systems and software engineering - Recommended practice for architectural description of software-intensive systems," *ISO/IEC 42010 IEEE Std 1471-2000 First edition 2007-07-15*, 2007.

[16] "ISO/IEC 42010, IEEE P42010," 2009.

[17] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[18] N. Bouck, D. Weyns, R. Hilliard, T. Holvoet, and A. Helleboogh, "Characterizing relations between architectural views," in *Software Architecture* (R. Morrison, D. Balasubramaniam, and K. Falkner, eds.), vol. 5292 of *Lecture Notes in Computer Science*, pp. 66–81, Springer Berlin / Heidelberg, 2008.

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

[20] M. Moriconi, X. Qian, and R. Riemenschneider, "Correct architecture refinement," *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, vol. 21, no. 4, p. 356372, 1995.

[21] L. Lavagno, G. Martin, and B. Selic, *UML for real: design of embedded real-time systems*. Kluwer Academic Publishers, 2003.

[22] "SysML." `http://www.sysml.org/`.

[23] P. H. Feiler, D. P. Gluch, 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, Feb 2006.

[24] D. D. Niz and P. Feiler, "Aspects in the industry standard AADL," 2007.

[25] A. Egyed, *Heterogeneous view integration and its automation*. PhD thesis, University of Southern California, 2000.

[26] N. Bouck, A. Garcia, and T. Holvoet, "Composing structural views in xADL," in *Early Aspects: Current Challenges and Future Directions* (A. Moreira and J. Grundy, eds.), vol. 4765 of *Lecture Notes in Computer Science*, pp. 115–138, Springer Berlin / Heidelberg, 2007.

[27] A. Radjenovic and R. Paige, "Architecting dependable systems," ch. The view glue, pp. 66–88, Berlin, Heidelberg: Springer-Verlag, 2007.

[28] T. Denton, E. Jones, S. Srinivasan, K. Owens, and R. W. Buskens, "Naomi — an experimental platform for multi—modeling," in *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, MoDELS '08, (Berlin, Heidelberg), pp. 143–157, Springer-Verlag, 2008.

[29] "Modelica Association.." `http://www.modelica.org`.

[30] "MapleSim." `http://www.maplesoft.com/products/maplesim`.

[31] "Simscape." `http://www.mathworks.com/products/simscape/`.

[32] 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.

[33] 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.

[34] T. Johnson, C. J. J. Paredis, and R. M. Burkhart, "Integrating models and simulations of continuous dynamics into SysML," in *6th International Modelica Conference*, pp. 135–145, Modelica Association, 2008.

[35] J. Willems, "The behavioral approach to open and interconnected systems," *IEEE Control Systems Magazine*, vol. 27, no. 6, pp. 46–99, 2007.

[36] S. S. Bhattacharyya, E. Cheong, and I. Davis, "Ptolemy II heterogeneous concurrent modeling and design in Java," tech. rep., 2003.

[37] G. Agha, "Concurrent object-oriented programming," *Commun. ACM*, vol. 33, no. 9, pp. 125–141, 1990.

[38] 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*, (Washington, DC, USA), pp. 3–10, IEEE Computer Society, 2009.

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

[40] G. Abowd, R. Allen, and D. Garlan, "Formalizing style to understand descriptions of software architecture," *ACP Transactions on Software Engineering and Methodology*, vol. 4, pp. 319–364, Oct 1995.

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

[42] D. Garlan, W. Reinholtz, B. Schmerl, N. Sherman, and T.Tseng, "Bridging the gap between systems design and space systems software," in *29th Annual IEEE/NASA Software Engineering Workshop (SEW-29)*, 2005.

[43] R. Allen and D. Garlan, "A formal basis for architectural connection," *ACM Transactions on Software Engineering and Methodology*, July 1997.

[44] P. Gawthorp, *Bond Graphs and DynamNizic Systems*. Prentice Hall, 1996.

[45] D. Jeltsema and J. M. A. Scherpen, "Multidomain modeling of nonlinear networks and systems," *Control Sytems Magazine*, Aug 2009.

[46] N. Medvidovic and R. Taylor, "A classification and comparison framework for software architecture description languages," *tse*, vol. 26, no. 1, 2000.

[47] A. Rajhans, A. Bhave, S. Loos, B.Krogh, A. Platzer, and D. Garlan, "Using parameters in architectural views to support heterongeneous design and verification," in *50th IEEE Conference on Decision and Control (CDC)*, (Florida), Dec 2011.

[48] 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*, pp. 149–159, 2001.

[49] 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*, pp. 1172–1177, 1999.

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

[51] C. S. D. Conte, P.Foggia and M. Vento, "Thirty years of graph matching in pattern recognition," *IJPRAI*, vol. 18, no. 3, pp. 265–298, 2004.

[52] E. M. Luks, "Isomorphism of graphs of bounded valence can be tested in polynomial time," *J. Comput. Syst. Sci.*, vol. 25, no. 1, pp. 42–65, 1982.

[53] E. B. Krissinel and K. Henrick, "Common subgraph isomorphism detection by backtracking search," *Softw. Pract. Exper.*, vol. 34, pp. 591–607, May 2004.

[54] D. Conte, P. Foggia, and M. Vento, "Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs," *J. Graph Algorithms Appl.*, vol. 11, no. 1, pp. 99–143, 2007.

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

[56] "Open Modelica.." `http://www.openmodelica.org/`.

[57] G. Karsai and J. Sztipanovits, "Model-integrated development of cyber-physical systems," in *SEUS '08: Proceedings of the 6th IFIP WG 10.2 international workshop on Software Technologies for Embedded and Ubiquitous Systems*, (Berlin, Heidelberg), pp. 46–54, Springer-Verlag, 2008.

[58] "Simulink Coder." `http://www.mathworks.com/products/simulink-coder`.

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

[60] J. S. Kim and D. Garlan, "Analyzing architectural styles with alloy," in *In Proceedings of ROSATEA06, ISSTA*, pp. 70–80, ACM Press, 2006.

[61] M. Y. Nam, D. de Niz, L. Wrage, and L. R. Sha, "Resource allocation contracts for open analytic runtime models," in *In Proc. EMSOFT*, 2011.