Integration of Modeling Methods for Cyber-Physical Systems

Ivan Ruchkin

CMU-ISR-18-107
November 2018

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
David Garlan (Chair)
André Platzer
Bruce Krogh
Dionisio de Niz
John Day (NASA Jet Propulsion Lab)

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Software Engineering.*

This research was supported in part by the National Science Foundation (NSF) under grants CNS-0834701 and CNS-1035800, by the Air Force Research Laboratory (AFRL) and the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8750-16-2-0042, the National Security Agency (NSA), the U.S. Department of Defense (DoD) under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute (a federally funded research and development center), and through the Office of the Assistant Secretary of Defense for Research and Engineering (ASD(R&E)) under Contract HQ0034-13-D-0004. Any views, opinions, findings, conclusions, or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSF, AFRL, DARPA, NSA, DoD, ASD(R&E), or the SEI.

Keywords: cyber-physical system, embedded system, integration, model, analysis, contract, architecture, specification, verification

Abstract

Cyber-physical systems (CPS) incorporate digital (cyber) and mechanical (physical) elements that interact in complex ways. Many safety-critical CPS, such as autonomous vehicles and drones, are becoming increasingly widespread and hence demand rigorous quality assurance. To this end, CPS engineering relies on *modeling methods*, which use models to represent the system and design-time analyses to interpret/change the models. Coming from diverse scientific and engineering fields, these modeling methods are difficult to combine, or *integrate*, due to implicit relations and dependencies between them. CPS failures can lead to substantial damage or loss of life, and are often due to two key integration challenges: (i) *inconsistencies between models* — contradictions in models that do not add up to a cohesive design, and (ii) *incorrect interactions of analyses* — out-of-order executions in mismatched contexts, leading to erroneous analysis outputs.

This thesis presents a novel approach to detect and prevent integration issues between CPS modeling methods during the design phase. To detect inconsistencies between models, the approach allows engineers to specify *integration properties* — quantified logical statements that relate various elements of multiple models — in the *Integration Property Language* (IPL). IPL statements describe verifiable conditions that are equivalent to an absence of inconsistencies. To interface with the models, IPL relies on *integration abstractions* — simplified representations of models for integration purposes. Two abstractions are proposed in this thesis: views (annotated component-and-connector models, inspired by software architecture) and behavioral properties (expressions in model-specific languages, such as the linear temporal logic). Combining these abstractions enables engineers to mix model structure and behavior in IPL statements. To ensure correct interactions of analyses, I introduce *analysis contracts* — a lightweight specification that captures inputs, outputs, assumptions, and guarantees for each analysis, in terms of the integration abstractions. Given these contracts, an *analysis execution platform* performs analyses in the order of their dependencies, and only in contexts that guarantee correct outputs.

My approach to integration was validated on four *case studies* of CPS modeling methods in different systems: energy-aware planning in a mobile robot, collision avoidance in a mobile robot, thread/battery scheduling in a quadrotor, and reliable/secure sensing in an autonomous vehicle. The validation has shown that the approach supports expressive integration properties, which can be soundly checked within practical constraints, all while being customizable to diverse models, analyses, and domains.

Acknowledgments

The journey of this PhD is not over yet…

Contents

Chapter 1

Introduction

An emerging class of systems, called *cyber-physical systems (CPS)*, incorporate digital/software (aka cyber) and mechanical/hardware (aka physical) elements. These elements interact in particularly complex ways due to their higher autonomy and greater decentralization than in classic embedded systems. These interactions enable increased socioeconomic benefits, leading CPS to be increasingly ubiquitous and important. For example, fully automated self-driving cars promise efficiency of traffic movement that outperforms human drivers by an order of magnitude [83]. Another example is that incorporating renewable energy sources into smart power grids could lead to large reductions in emissions and environmental damage [148, 232]. Similar effects are expected in other industries and domains [94, 127, 169, 220, 221, 225].

To design interactions between physical and digital elements of CPS, engineers often need to use multiple *modeling methods* — approaches to system-building that rely on structured representations (*models*) of the system and its environment [205]. The models that are used in CPS design describe a broad range of structural and behavioral aspects that often include program execution, hardware design, and mechanical dynamics [29, 76, 225]. The main advantage of models over informal descriptions is that engineers can perform *analyses* over models [108, 178]. Broadly, analyses are any operations that interpret and/or modify models, ranging from manual safety inspections to algorithms that produce new artifacts (e.g., generate code) and change the existing ones (e.g., to find a satisfactory arrangement of components). For example, a bin packing analysis [178] may be used to allocate threads to processors based on processor utilization. Each analysis produces some outcome that has engineering value — be it a guarantee of safety, a controller implementation, or a set of optimal parameter values.

CPS are difficult to engineer correctly. Safety-critical contexts are particularly demanding of correctness and call for rigorous up-front verification and validation — as opposed to informal, post-factum, and ad-hoc quality assurance. Reasoning about large systems' quality is made complicated by models that use continuous, discrete, and probabilistic constructs to represent the physical and digital worlds [144]. Another factor that makes CPS engineering hard is timing of various dynamics: computations, networking, and physical actions must be synchronized as intended by the system designers [143]. Such synchronization is difficult to achieve in the face of non-determinism and randomness of the physical world. Nevertheless, a combination of formal modeling, simulation, and testing promises exhaustive and high-confidence quality assurance [76, 122].

CPS modeling methods originate in many disciplines, such as mechanical, control, and software engineering. This heterogeneity makes it hard to combine, or *integrate*, several modeling methods for a given system. Even when the analyses for different models are independent (i.e., do not affect each other's inputs), their outputs may not be compatible. For instance, if two models make conflicting assumptions, a theoretical guarantee provided by one model may not extend to the code generated by the other. Generally, we observe that absence of contradictions in models (i.e., *model consistency*) is required for analyses to remain modular and have compatible downstream results. Inconsistencies between models threaten the outcomes of analyses and may lead to complex errors, which can take a substantial amount of time, effort, and funds to discover. In some cases these errors are not discovered at all, causing system failures and catastrophic events, such as the Mars Climate Orbiter Mishap [8], in which a mismatch between imperial and metric units led to a trajectory miscalculation and subsequent disintegration of the spacecraft.

Another challenge of combining CPS modeling methods is ensuring *correct analysis interactions*. Since some analyses depend on each other, they need to be executed only in the order of their dependencies and only under appropriate circumstances. For example, accurate control simulation depends on how real-time scheduling analyses allocate computational tasks to processors [61] and should be performed after the allocation. Incorrect analysis interactions, on the other hand, could lead to erroneous outcomes, such as implementation bugs and unsatisfied safety constraints. Such an interaction between electrical and mechanical aspects of the GM ignition switch led to multiple ignition failures, car crashes, and deaths [229]. A change in the electrical aspect made the ignition switch unexpectedly mechanically unstable, which has led to accidental engine shutdowns while driving. If analysis interactions had been explicitly considered, the change would have triggered an analysis of the mechanical aspect and detected this bug prior to deploying the faulty version of the switch.

With a goal of ensuring *both* model consistency and correct analysis interactions, we arrive at *modeling method integration (MMI)* — an activity of combining modeling methods without model inconsistencies or incorrect analysis interactions. The specific desirable outcomes of MMI are defined for an *integration scenario* — a context of engineering a system with certain requirements, where interactions between multiple models or analyses may violate those requirements. In practice, MMI is typically performed informally and without firm guarantees of satisfying critical requirements. Often, the results of such integration are inflexible, fragile, and rarely reusable in a different context. Most importantly, undiscovered errors frequently remain in deployed cyber-physical systems, potentially leading to costly failures in safety-critical contexts. Performing MMI in a sound, modular, and practical way is the central problem addressed in this thesis.

Although partial MMI approaches exist, the body of CPS research does not not yet provide general, sound, and effective solutions. Currently there are two major ways to integrate modeling methods. The first way is to create a single language or formal system with universal semantics that can serve as a lingua franca of all modeling methods that need to be integrated. All existing models are then mapped into this universal language where inconsistencies and dependencies can be directly discovered. The analyses then need to be reimplemented for the universal language or connected to it through model transformation. While this approach is intuitively straightforward, it may lead to large and complex models, exploding the verification state space [44], and thus rendering the approach inapplicable to systems of realistic size. For example, in hybrid systems, verification times grow exponentially with the number of variables [79], and explicitly composing hybrid automata with source code would make verification infeasible [13]. For timed automata, analyzing a combined model can take 60+ times longer than analyzing its parts individually [136]. Compositional approaches can improve scalability, but still time out on large inputs [35], and state space explosion remains a major obstacle in formal methods [179]. Another risk is that a universal formalism might not exist due to incompatible discipline-specific modeling assumptions. For instance, discrete time and continuous time models are difficult to unify in a way that supports tractable automated analysis of both formal systems [144].

The second way of performing integration is to connect modeling methods through intermediate *integration abstractions*, thus preserving the diversity of models and analyses. As detailed in Chapter 9, this approach is supported by several existing multi-model CPS frameworks, like architectural views (hierarchical graphs of components and connectors, annotated with custom types and properties) [23], behavior relations (mapping traces of states between models) [199], CyPhyML/OpenMETA (connecting models through logical interfaces) [223], Ptolemy II (simulation of heterogeneous computations) [146]. These frameworks are more practical for industrial applications than single-formalism solutions. However, most of these frameworks offer no support for analysis execution (and hence, do not offer guarantees of satisfying critical properties for designs under change); many of them are highly domain-specific due to their use of tailored abstractions (and hence, not general or expressive enough to address the broad variety of modeling methods for CPS); finally, some assume top-down development from requirements to implementation using fixed modeling methods (and hence, make it difficult or impossible to incorporate modeling methods that had not been considered during the framework design).

To advance the state-of-the-art of CPS modeling and verification, this thesis addresses three limitations of the existing approaches in the context of MMI:

A. *Limited expressiveness of consistency verification*, which is often confined to the architectural level of abstraction and unable to verify richer properties in a multi-model environment. Specifically, previous work has considered structural consistency of views [24], static constraints on view parameters [200], and directly relating behaviors from multiple models [198]. None of these approaches have provided a way to relate and constrain structure and behavior for an arbitrary number of models. An example of such a constraint would be to limit the allocation of threads to CPUs (a fixed structural choice in one model) based on a charge of individual battery cells (a behavioral quantity that changes over time in another model). Moreover, most of the existing approaches use fixed definitions of consistency, which cannot be tuned to a desired level of precision or allow some amount of inconsistency.

B. *Ad-hoc integration abstractions* that limit soundness and customizability of integration frameworks. One issue is that these abstractions are created based on the designer's intuition, thus confining the formalization to the abstractions and making it impossible to formalize the mapping between the models. For example, architectural views have so far been created by intuitively grouping model parts as components and connectors [23]. Another issue is that once committed to a given abstraction (e.g., EAST-ADL for timed automata [158]), the framework is difficult to extend for other, potentially more convenient abstractions. Finally, many integration abstractions, such as logical interfaces [218] and architectural views [23], often require substantial manual effort throughout the engineering process.

C. *Ad-hoc analysis interactions* that may lead to violating model consistency and introducing errors into models, due to execution of analyses in an incorrect order or in a context that does not match the expectations of analyses. An example of an interaction is that an analysis adding redundant sensors for reliability to re-run the analysis of security for these new sensors — otherwise the consistency between reliability and security models may be violated. Previously, interactions between model-based analyses have not been considered in most frameworks. Whenever such analysis interactions have been considered, verification of their order and execution context has suffered from the two limitations above, leading to limited expressiveness and soundness of modeling method integration.

This thesis overcomes these three limitations by advancing a novel approach to modeling method integration, shown in Figure 1.1. The approach considers models and analyses as its primary inputs, aiming to verify the consistency relation between the models and organize interactions of the analyses. Expressing and checking consistency between heterogeneous models is done with a two-step "bridge" between the models. The first step is to create one of the two integration abstractions as intermediate "interfaces" for the models: *views* (annotated component-and-connector models that represent any given structures in the original model) and *behavioral properties* (verifiable statements in model-specific logics that constrain behaviors in the original model). In the second step, these abstractions are connected with *integration properties* — logical formulas written in a novel specification language (the *Integration Property Language (IPL)*) to express the desired consistency relation. IPL enables a customized approach to each integration scenario, letting engineers tailor integration properties to describe the necessary level of consistency and possibly bounded inconsistency. To control analysis interactions, each analysis is augmented with a *contract* that specifies the inputs, outputs, assumptions, and guarantees of the analysis. Analyses read and write to the models (or integration abstractions, for improved generality and reuse of the analyses) in a correct order, and their execution context is verified to be appropriate.

More specifically, this thesis makes the following advancements to address the respective limitations above:

I. A method of specification and verification of multi-model consistency properties that combine structure and behavior using IPL.

II. Integration abstractions (views and behavioral properties) that serve as a foundation for sound integration.

III. An analysis execution platform for integration of model-based analyses, which manages their ordering and execution context.

As I detail in the remainder of this thesis, the combination of these three advancements provides the essential support for modeling method integration in the context of CPS.

Part I of this thesis addresses the limited expressiveness of state-of-the-art consistency checking. To co-constrain structure and behavior of heterogeneous models, I have developed IPL — a customizable formal language based on the first-order logic. This language allows one to plug in arbitrary model-specific (e.g., modal) logics as sub-languages. I have also developed a verification algorithm that combines satisfiability solving and model checking in order to determine whether an IPL statement is valid by extracting the necessary information from the models.

Part II of this thesis provides a formal description of two integration abstractions: views

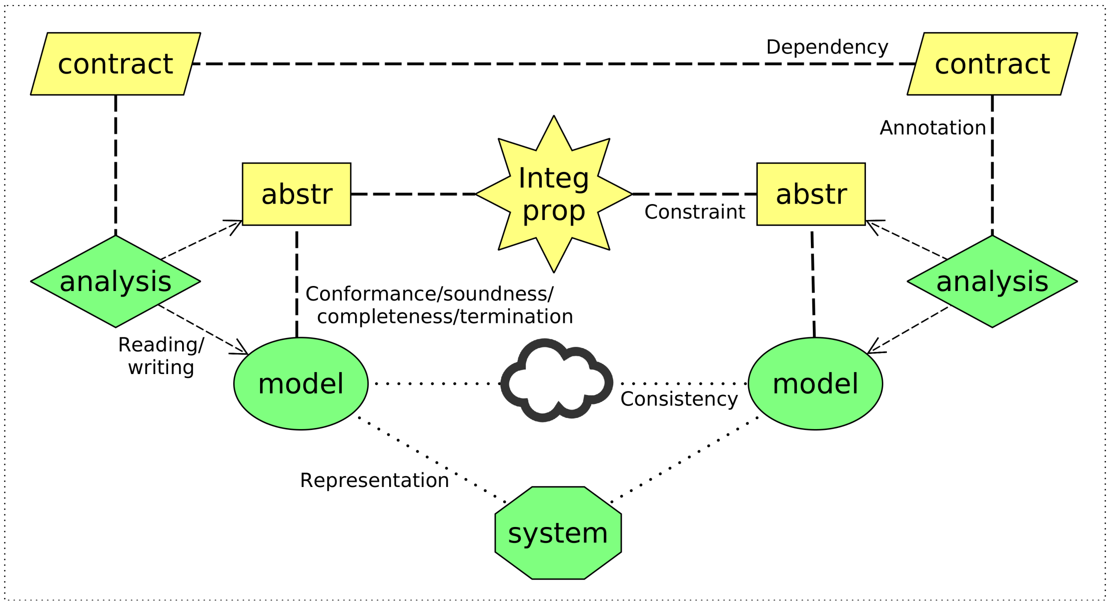


Figure 1.1: My approach to CPS modeling method integration.

and behavioral properties. Views serve as a common language to represent any structures from a model, giving integration checks a uniform way to access these structures — as opposed to tailoring them to the idiosyncratic syntax of each model. I formalize the important properties of views (soundness and completeness) that are necessary for correct integration. Behavioral properties are statements in model-specific languages that are plugged into IPL. The models can be queried to interpret these statements and return their values. Similar to the case of views, I formalize the conditions of correct integration (query soundness and termination) for behavioral properties.

Part III of my thesis is an execution platform for systematic and automated integration of model-based analyses. I have developed a lightweight specification method of "analysis contracts" that, for each analysis, describes its inputs (the information required by the analysis), outputs (the information produced by the analysis), assumptions (IPL statements that must be valid before the analysis executes), and guarantees (IPL statements that must be valid after the analysis executes). The inputs and outputs determine the correct order of analysis execution, guaranteeing that no stale information is consumed and no newer information is overwritten. Furthermore, this platform prevents execution of analyses in a mismatched context by automatically checking their assumptions and guarantees.

Notice that all three parts of my approach leverage existing models and analyses to perform MMI, instead of developing new modeling methods from scratch. By relying on existing modeling methods, my approach reuses the modeling technology and reduces the integration effort. For instance, to verify an integration property over several models, I may use a third-party model checking analysis associated with one of these models. However, this reliance leads to an assumption that the models are syntactically well-formed, and that their analyses are performed correctly with respect to the models' semantics.

I have validated my approach on four case studies to show its expressiveness, soundness, practical applicability, and customizability. The case studies include an energy-aware mobile robot, a collision-avoiding mobile robot, a quadrotor with real-time thread and battery scheduling, and an autonomous vehicle with redundant sensors. Each case study exercises various parts of the approach to provide evidence for the research claims described in the following section. To enable this validation, the specifications and algorithms were implemented in two architectural design environments: AcmeStudio [215] and OSATE2 [70].

The remainder of the introduction presents the thesis statement and its elaboration in terms of claims and qualities of MMI. Following that, I describe the organization of the rest of the thesis.

1.1 Thesis Statement and Claims

This dissertation seeks to advance the state of the art in integration of CPS modeling methods. Specifically, my approach aims to improve four qualities of integration outcomes:

- *Expressiveness:* an expressive MMI approach captures properties that depend both on the structure of models and their semantics/behavior. Therefore, it is suitable for detection of mixed structural-behavioral inconsistencies. That means the integration properties and analysis contracts have to be specified and checked in a way that takes both of the aspects into account.
- *Soundness:* a sound MMI approach ensures that a collection of CPS models is consistent, and that execution of analyses on these models will not lead to inconsistencies. My approach provides these guarantees with respect to mixed inconsistencies mentioned above (and described in more detail in Chapter 2). First, the guarantees include that, when the approach reports no inconsistencies (i.e., no cases of violations of integration properties), then the models are guaranteed to be consistent. Second, analyses should be executed in a correct sequence and appropriate context. As with inconsistencies, if the sequence is marked as correct, it should be so. Similarly, if context is marked as appropriate by the approach, then it should indeed produce no errors when the analyses are run.
- *Applicability:* an applicable MMI approach can be successfully used in the context of a real-world CPS. Although not precisely defined, some rules of thumb help evaluate this quality. Specifically, the approach should support correct integration within *the practical constraints of the scenario*. For instance, it should handle corner cases of behavior that occur in practice and scale to models of common sizes in a given scenario. Further, discovering errors that are contextually meaningful and difficult to detect indicates greater applicability.
- *Customizability:* a customizable MMI approach can be tailored in two dimensions: CPS modeling methods and application domains. In terms of modeling formalisms, it should include component-based models (e.g., signal-flow diagrams), various families of automata (state machines, hybrid automata, probabilistic automata, etc.), and semi-formal models (e.g., equations interleaved with textual descriptions). In terms of application domains, the approach should apply to multiple CPS domains (automotive, aerospace, energy, medical, and others), otherwise it may be relying on domain-specific assumptions that do not transfer to another domain.

The following thesis statement summarizes the principle claim of this dissertation:

Thesis Statement. Four qualities of modeling method integration for cyber-physical systems — *expressiveness, soundness, applicability,* and *customizability* — are enabled by an approach that is based on the following three parts:

I. Specification and verification of multi-model integration properties, supporting mixed structural-behavioral consistency of models (Part I, Chapter 5),

II. Two integration abstractions: views and behavioral properties, supporting Part I and Part III of the approach in their interactions with CPS models (Part II, Chapter 6),

III. Specification and checking of contracts for model-based analyses, supporting correct execution of the analyses (Part III, Chapter 7).

To highlight the mapping between the qualities of interest and parts of the approach, I decompose the thesis statement into research claims below.

Claim 1. *Expressive* MMI is enabled by specifying mixed structural-behavioral integration properties across multiple models (Part I), which are based on views and behavioral properties as abstractions of the models (Part II).

It is difficult to express and check properties that refer to models' elements of different nature (structural or behavioral). It is an advance in the state-of-the-art to enable engineers to specify and verify such integration properties.

Claim 2. *Sound* MMI is enabled by (i) formally verifying multi-model integration properties using a logic-based specification language (Part I), (ii) correctly executing sequences of analyses annotated with analysis contracts (Part III), and (iii) creating appropriate abstractions (Part II) of models to support (i) and (ii).

Across all three parts of the approach, soundness is achieved by rigorously defining the meaning of correct integration and developing algorithms to detect and/or ensure this correctness. Sound verification of multi-model consistency always correctly determines whether some property holds. Sound integration of multiple analyses always executes them in a correct order (according to the dependencies) and within an appropriate context.

Claim 3. *Practically applicable* MMI is enabled by using flexible abstractions to account for corner cases and delegating verification subtasks to model-specific tools (Part I, Part II, Part III).

The integration approach of this thesis is designed to accommodate the idiosyncrasies of systems in practice. That is, the two abstractions allow for unexpected corner cases, arbitrarily complex models, and automation opportunities; also, the verification algorithm is designed to utilize efficient model-specific reasoning (e.g., by using specialized model checking tools that are supplied with the models).

Claim 4. *Customizable* MMI is enabled by embedding arbitrary behavioral property languages into integration properties (Part I) and tailoring views to various domains using architectural styles (Part II).

In my approach, views are based on architecture languages and use customized vocabularies of types and constraints (known as architectural styles). Behavioral languages can be customized to fit models and domains as well — as long as these languages enable sound queries that always terminate, as described in Chapter 4. For instance, one can use a modal logic that is the most appropriate in the context (e.g., CTL for a system with branching computations).

1.2 Thesis Organization

Chapter 2 describes the background of CPS modeling methods, giving the reader the necessary vocabulary to understand the issues of modeling method integration. To exemplify the challenges of modeling method integration, the next chapter describes the four systems that were used for validation in this thesis. These systems are used as illustrations throughout the following chapters, and also serve as contexts for validation studies (described in Chapter 8):

- *System 1:* energy-aware adaptation for a mobile robot.
- *System 2:* collision avoidance for a mobile robot.
- *System 3:* thread and battery scheduling for a quadrotor.
- *System 4:* reliable and secure sensing for an autonomous vehicle.

An overview of the integration approach is given in Chapter 4. The following three chapters (Chapters 5 to 7) elaborate on the three technical parts of the approach. Then follows a chapter on validation studies (Chapter 8) that revisits the four claims for each technical part and each context, providing the supporting evidence for each claim. After, I review related work in Chapter 9. The dissertation concludes with a discussion of limitations, design rationale, and future directions in Chapter 10.

To simplify the reader's navigation through this thesis, Table 1.1 indicates how the technical parts relate to claims and validation contexts. Each claim was validated on at least two systems, depending on the context and evaluation opportunities[1].

[1]Claims 1 and 4 are not evaluated for Part III because they do not apply to it directly (instead, these claims are evaluated for the integration abstractions that supported Part III)

Technical part	Claim	Sys 1	Sys 2	Sys 3	Sys 4
Part I: integration property language	Claim 1: expressiveness	✓		✓	
	Claim 2: soundness	✓		✓	
	Claim 3: applicability	✓		✓	
	Claim 4: customizability	✓		✓	
Part II: integration abstractions	Claim 1: expressiveness	✓	✓	✓	✓
	Claim 2: soundness	✓	✓	✓	
	Claim 3: applicability	✓	✓	✓	✓
	Claim 4: customizability	✓	✓	✓	✓
Part III: analysis contracts	Claim 2: soundness			✓	✓
	Claim 3: applicability			✓	✓

Table 1.1: Mapping between technical parts, claims, and validation systems. A check mark indicates that a claim (row) for a technical part of the thesis (row) was evaluated on a system (column).

Chapter 2

Background: Modeling Methods for Cyber-Physical Systems

This chapter gives the necessary background on CPS modeling and integration. First, it establishes the basic terminology. Then it elaborates on the challenge of modeling method integration by refining the ideas of model consistency and analysis interactions. At the end of this chapter, I frame the problem addressed in this thesis using three conditions of successful modeling method integration.

2.1 Models, Analyses, and Methods

Cyber-physical systems are often engineered using models [60, 107, 145]. A *model* is a formal representation of a system or its part [205]. For example, a common CPS model is a linear hybrid automaton (LHA): it has a well-defined mathematical form that combines discrete jumps and continuous evolutions [100]. The meaning of the model, often given mathematically in some formal system, is known as the *semantics* of the model. From the engineering perspective, a model's semantics is the ultimate source of the information (e.g., decisions and assumptions) that the model contains about the system and its environment. The semantics can be given in terms of behaviors that the model allows, in which case it is called *behavioral*. For example, a model may specify a set of possible traces/executions of a program, which depend on the program's inputs. Behavioral aspects of CPS models are heterogeneous because behaviors depend on the models' concepts of state, computation, and time [76].

Models are typically specified using *formalisms* [29] — languages with a formal *syntax* that is based on explicit rules for generating sentences in a language. To give meaning to a language, its syntax is mapped to its semantics. For example, the input language of the hybrid system reachability tool SpaceEx [79] is a syntax that maps to the LHA semantics. Each model also has a *referent* — the part of the system it intends to represent. For example, an LHA can be used to model a system's continuous mechanical movement with discrete decisions to activate acceleration and braking. Multiple models of the same system often have partially overlapping referents, leading to multiple descriptions of the same system parts (e.g., a controller). This redundancy may lead to conflicts/inconsistencies, as discussed below.

Integration of heterogeneous models often requires creating special representations, or *abstractions* (similar to the concepts of wrappings [20], model aspects [222], integration adapters [63], and semantic interfaces [224] in related work). For the purposes of integration, I assume that the models consist of *elements*. To make my integration approach customizable, I make only minimal assumptions about the nature, characteristics, and relations of elements. Model elements may be syntactic or semantic, and their examples may include statements, blocks, modules, states, traces, and so on. An abstraction of a model is said to extract (or expose/represent) some of the model's elements if these elements are present in the abstraction. For instance, if a property written in the Linear Temporal Logic (LTL) [191] is used as an abstraction, it exposes the behavioral elements (traces) that are constrained, e.g., by LTL. If, on the other hand, the abstraction focuses on the syntactic aspects of the model, it can be called *structural*. For instance, several system modes (e.g., inactive, active, and safe) encoded in a model may be represented as a set of discrete entities in the abstraction. Generally, when discussing the fidelity of an abstraction, a central consideration is the extent to which the abstraction exposes the elements of a model.

A model is useful if it enables an operation with a valuable outcome, such as checking if a system is deadlock-free, since the information about the presence of deadlocks is valuable. To consider these operations on models, I use the concept of an *analysis* — an algorithm or a procedure carried out using a model. In this work, analyses are treated as functions from models to models. Thus, models are produced manually by engineers or (semi-)automatically by analyses. CPS analyses come in a variety of forms: some check properties of models, others generate code, and yet others modify models by finding improved designs. For instance, a bin-packing analysis [178] allocates threads to processors in a model to make it schedulable.

Analyses may depend on each other. By definition, analyses take (read) elements of models as inputs and produce a variety of outcomes (which includes modifying elements of models). When analysis $\mathcal{A}_1$ changes the same (type of) model elements that analysis $\mathcal{A}_2$ consumes as inputs, I say that $\mathcal{A}_2$ *is dependent* on $\mathcal{A}_1$. The granularity of elements in this definition may vary from individual numbers (e.g., parameter values) in a model to large parts of models (e.g., a specification of the environment). This thesis uses types of components (e.g., CPUs or threads) and component properties (e.g., the frequency of a CPU) to determine dependencies: an analysis that reads any CPUs or frequencies from a model is considered dependent on any analysis that changes the set of CPUs or any of their frequencies in that model. The relation of analysis dependency forms a partial order on any set of analyses without circular dependencies (i.e., without analyses that transitively depend on their dependents).

Some analyses are applicable only in a certain *context* — a "condition" of the model(s) that the analysis reads. This condition can be described with logical constraints over the model(s). When checked before the execution of an analyses, these context constraints are called *assumptions*. For instance, some thread model-checking analyses assume that the system is using rate-monotonic scheduling [34]. Applying analyses outside of their appropriate contexts may lead to incorrect results; for instance, the aforementioned model checking analysis may label a faulty system as correct. The conditions of models after an analysis has been performed successfully without errors are called the *guarantees* of the analysis.

Tying models and analysis together, a *modeling method* is an approach to modeling and analyzing a system using a set of related models and analyses. For instance, one can conceptualize an approach to real-time schedulability as a modeling method: it has standard schemas (i.e., models) for schedulability-related information like periods and deadlines, as well as design-time algorithms (i.e., analyses) to optimize system designs, like static voltage-frequency scaling.

The field of CPS uses modeling methods from multiple scientific and engineering disciplines: control theory, electrical engineering, mechanical engineering, energy and power modeling, cybersecurity, and so on [195, 201]. These methods rely on formalisms that differ in their level of abstraction, computational model, notion of time, and so on [201]. For example, a synchronous dataflow program [142] describes discrete, ordered, and time-unaware computations. On the other hand, an ordinary differential equation (ODE) [234] represents a continuous and acausal physical process in continuous time. CPS models also vary in their degree of mathematical formality; for instance, a Simulink signal-flow diagram has a standardized syntax, but its formal semantics is not publicly accessible.

The use of diverse modeling methods in a CPS project has several advantages over using a single modeling method: (i) a broader scope of requirements can be explored and satisfied; (ii) higher degree of quality assurance; (iii) reduced engineering effort due to domain-specific optimizations; and (iv) reduced training costs since engineers can use modeling methods that they are most proficient with. In the long-term perspective, as more advanced modeling methods are being developed, it is natural to expect their combined use for CPS engineering.

The list below illustrates several prominent CPS modeling methods[1]:

- Signal-flow modeling using such toolsets as Matlab/Simulink [53] or SCADE Suite [66] for control design, and SPICE2 for circuit design [177]. These models are widely used for control design and tuning via simulation in many industries including automotive, industrial, and aerospace engineering [117] [152].
- Discrete state-based modeling using state machines, process algebras, statecharts, labeled transition systems, and timed automata. The notations and tools include Promela/Spin [105], FSP/LTSA [154], Matlab/Stateflow [53], MontiArcAutomaton [203], UML statechart [67], and UPPAAL [140]. These models describe discrete computations, event-based and real-time designs. One example of using these models is an analysis of concurrent thread communication to check for absence of deadlocks and race conditions. Discrete state-based models are used across various application domains [47].
- Hybrid system modeling with linear hybrid automata [7] or hybrid programs [189]. These formalisms represent a system as a combination of discrete jumps and continuous evolutions, and are used to verify properties on the boundary of discrete and continuous dynamics. Hybrid models are often used to discover (or establish provable absence of) system executions that falsify safety requirements, as in the Toyota powertrain benchmark [117]. Hybrid programs have been used to formally verify in aerospace and automotive domains [171] [150] [151]. One of the extensions for hybrid modeling is the Sphinx toolset [173] that specifies hybrid programs in terms of UML class and activity diagrams, and enables collaborative proof engineering.
- Differential equation modeling, often simplified to ODEs [234]. A commonly used notation that encapsulates ODEs is a lumped element model: a set of discrete entities that

[1]This list is not complete, but it represents the state-of-practice in CPS engineering. For more detail, see Section 9.1

approximate the behavior of the whole system. Modelica and SimScape are popular toolsets to build and analyze acausal lumped element models (e.g., a model of heat dissipation of multiple independent heating nodes), each element of which has a set of differential equations associated with it. Lumped element models are used to represent continuous dynamics, such as mechanical movement, fluid dynamics, and electrical operation (for example, in rechargeable batteries in electric vehicles [109]).

- System architecture modeling with languages such as AADL [71], SysML [58], UML structure diagrams [46], and Acme [85]. These models focus on the system elements and their interactions, relations, and properties, and can be used for component-based fault analysis, product line management, and checking conformance to design space constraints [72] [84].

Yet another dimension of diversity for CPS (in addition to disciplines and modeling methods) is the *domain of application*: automotive, aerospace, medical, and energy systems differ significantly in their purpose, but rely on common CPS modeling methods [195]. For example, car engine control [51] and infusion pump control [160] rely on the same core principles of control theory, even though their standards of safety and efficiency are different. Therefore, integration approaches need to be applicable to CPS-related application domains.

2.2 Modeling Method Integration

The tension between separating and combining engineering concerns is prominent for CPS. It is required, on the one hand, to separate referents and dimensions of modeling (e.g., modeling the electrical dynamics separately from the software functionality) in order to reduce complexity and apply domain-specific analyses. Thus, CPS design and implementation are modularized, often along the boundaries of different disciplines. On the other hand, it is necessary to assemble the results of different methods to create a cohesive system. Since most modeling methods have been created in isolated disciplines, separation of concerns has been relatively easy and extensively practiced in CPS [143]. In contrast, combining the results of different methods remains an outstanding challenge [61, 143, 201].

CPS modeling methods may be combined in various ways. Models from different methods may be directly composed, translated into hybrid/combined representations, or kept separate. Analyses may be composed to form larger analyses, modified to satisfy mutual assumptions or use the same data format, or used as-is. Regardless of how the methods are combined, they need to be *integrated* — used together in a way that does not lead to errors in the design. What constitutes an integration error is defined by *integration scenarios* — descriptions of a system, its requirements, and the interactions between models/analyses that may violate the requirements. Today, modeling method integration for CPS is often manual, informal, ad hoc, and error-prone [118]. Below I discuss two classes of integration issues: *model consistency* and *analysis interactions*.

Informally, *consistency* of models means that the models describe the same system without conflicts or contradictions. Consistency is threatened by potentially conflicting related information across multiple models. Models of the same CPS are usually not completely independent since their referents may overlap, leading to descriptions of the same system part in multiple models. For instance, controllers appear in many CPS models in various forms: a mathematical function, a hardware chip, a collection of signal blocks, or a piece of source code. *Inconsistency* is, then, the condition when the related information from multiple models leads to a contradiction or a design flaw. These contradictions/flaws are defined relative to the requirements of the system: an error in one design may be a non-issue in another. For example, an electrical model of a battery (the number, location, and connection of battery cells) needs to be describe the same geometry as its thermal model (which analyzes heat transfer via conduction, convection, and radiation). If the two models disagree on the battery's geometry, some battery cells may be overused, overheated, and eventually catch fire. However, if the battery is submerged in a coolant, the exact consistency of these two models is not necessary and can be relaxed to bound the inconsistency (e.g., by how much the cell positions and sizes differ on average).

Consistency of models is necessary to combine the results of model-based analyses. Since models contain information related to each other, an inconsistency would lead to incompatible analysis outputs — similar to the case of running the analyses on two *different* systems. Consistency is necessary even when the analyses do not exchange data directly because their outputs may be combined in downstream development. For example, safety guarantees from one analysis may need to apply to the code generated by another analysis, and if the models are inconsistent, the safety guarantees may be violated. To remove the need for model consistency, each analysis would have to check all related models to ensure compatibility of the outputs. Since such checking is infeasible in practice, modular analyses rely on consistency of their models.

The consistency conditions across CPS models can be difficult to express directly. One challenge comes from CPS models using different abstractions of time, control, state, and data. Therefore, it may be difficult for an engineer to check if one model conflicts with another in terms of these abstractions. For instance, at what point in time, in terms of a real-time controller thread, is a control decision described by a Simulink model taken? Answering such questions manually is a tedious and error-prone process, and automation requires integration abstractions, which are discussed in Chapter 6.

The second integration issue, *analysis interaction*, is typically due to analyses being developed independently of each other and reused in new circumstances. Therefore, each analysis may have implicit *data dependencies*, of which the engineers running the analyses might be unaware. Suppose one analysis changes types and placement of sensors in an autonomous car, while another checks the current set of sensors for security vulnerabilities. If the latter analysis is run first and approves the set of sensors, running the former later may invalidate the conclusion and potentially deliver an unsecure design.

Finally, analyses may be run in a *mismatched context*, in which the expectations of an analysis are not satisfied by the models it reads. Mismatch of execution context can occur in two situations. First, if the meaning of inputs does not match the expectation of the analysis, the outputs may have errors. This situation can occur, for instance, when an analysis is specialized to work only for a certain class of systems (e.g., only for rate-monotonic scheduling). Second, if the analysis is not created for a multi-model context, the changes by the analysis may violate the consistency of models. For example, an analysis can introduce an inconsistency by failing to update all the related information in multiple models (e.g., CPU voltages in one model and associated power draws in another). In both situations, the conditions of context mismatch are similar to consistency relations in that they lack the means of formal specification, let alone automatic checking.

The causes of these two integration issues are two-fold. On the one hand, many integration issues arise from miscommunication between teams and mistakes of individual engineers. For instance, one team might make unsupported assumptions about a model/design that is produced by another team. This invalid assumption would lead to inconsistency of their models. Detecting such issues is difficult because often no single person has a complete perspective on both models. As a result, it may be difficult to even formulate what it means for these models to be consistent. Besides, it can be a tedious and time-consuming process to debug models side-by-side.

On the other hand, sometimes inconsistency can be introduced intentionally in the process of model refinement. For instance, certain optimizations or design choices may be desirable to speed up or simplify the analysis, but may potentially introduce inconsistencies. In such cases, an engineer might be aware that the models do not match, but lack the tools to rigorously define and quantify the mismatch, and ultimately forgo the integration efforts.

Regardless of the cause, poor integration may lead to high engineering costs: errors that are not discovered during model integration rapidly increase in cost since these errors lead to major redesigns, recalls, and failures of the system. One well-publicized example is a recall of General Motors cars due to an unstable ignition switch [229]. In the Chevy Volt case, the electrical aspect of the switch was iteratively redesigned several times, but the mechanical properties of the switch were neglected, leaving the ignition switch physically unstable. Another neglected dependency was that turning off the switch leads to turning off the airbags. Not taking these design dependencies into account led to tragic consequences: a driver could accidentally turn off the ignition with his knee, rendering the car unsafe and poorly controllable [229]. Therefore, better support for integration of electrical and mechanical aspects of the switch could possibly have prevented the fault and consequent costly recalls.

In summary, integration issues occur due to human errors and intentional modifications to models. They are difficult to detect and costly to fix due to the complexities of CPS engineering. In this work I am concerned with integration issues related to analysis interactions and model consistency. Thus, based on the description above I formulate *three conditions of successful integration*:

1. *Model consistency:* the models should not contradict each other; in other words, their related information should contain no conflicts that would lead to violating the system's requirements.

2. *Satisfaction of data dependencies:* the analyses from different modeling methods should only be run in the order of the dependencies, without using stale inputs or overwriting newer outputs with older ones.

3. *Matching context for analyses:* analyses should be executed only in the context (i.e., the models that are read) that the analyses are engineered for, and only when their outputs do not violate consistency of the models.

The next chapter describes the high-level approach to satisfy these conditions.

Chapter 3

Case Study Systems

The research in this thesis was validated by performing case studies on four systems. This chapter briefly describes these systems, related concepts and challenging integration scenarios, so that they can be referred to from the downstream chapters. The systems are as follows:

- *System 1:* energy-aware adaptation for a mobile robot.
- *System 2:* collision avoidance for a mobile robot.
- *System 3:* thread and battery scheduling for a quadrotor.
- *System 4:* reliable and secure sensing for an autonomous vehicle.

3.1 Energy-aware adaptation for a mobile robot

This system was built in a DARPA-funded research project "Building Resource-Adaptive Software Systems" (BRASS). I was part of the team of robotics and software engineering researchers who worked on designing and implementing the MARS (Model-based Adaptation for Robotic Systems) system. MARS is an adaptive mobile robot based on the TurtleBot 2 platform (`http://turtlebot.com`), which navigates to a goal location through a physical environment using its map. The environment contains charging stations for the robot to replenish its battery. In addition to the standard navigation stack of the Robot Operating System (ROS) [194], MARS has an adaptive software layer that monitors and adjusts the robot's configuration and mission plan in response to changes in the environment, with the goal of minimize mission time and power consumption. For instance, if an obstacle blocks the chosen path, the robot needs to re-plan, potentally changing its configuration to reduce power consumption or re-charging along the way.

This system was chosen for validation because it is a CPS of realistic complexity, hence allowing me to investigate the applicability of my integration approach. Furthermore, it includes multiple models of different formalisms, which helps evaluate expressiveness and customizability. The models included utility functions, configuration models, system architectures, planning models, maps, power models, and simulations. Conveniently, I had direct access to the engineers of this project, making it easier to interpret the outputs of integration.

To adapt, the robot uses multiple models, including the environment, mission, and the robot's architecture models. The validation study (described further in Section 8.1 focused on two models with a complex relationship: a power prediction model and a planning model. The power prediction model (M_{power}) is a parameterized set of linear equations that estimates the energy required for motion tasks, such as driving straight or turning in place. The model is a statistical generalization of the data collected from the robot's executions. Given a description of a motion task, the model produces an estimate of required energy.

The planning model (M_{plan}) finds a path to a goal by representing the robot's non-deterministic movements on a map, along with their time and power effects, in a Markov Decision Process (MDP) [106]. The model's state includes the robot's location and battery charge. The robot's actions are modeled as non-deterministic transitions, and the environment's actions are modeled as probabilistic transitions. Whenever (re)planning is required, the PRISM probabilistic model checker [134] produces a plan that is optimal with respect to a utility function that indicates the relative priorities of time and energy. The resulting plan resolves non-determinism with action choices in each state, and these choices are fed to the robot's motion control. Although the energy-related coefficients in M_{plan} are derived from M_{power}, these two models are is not equivalent to each other because of various modeling choices, optimizations, and compromises. For example, M_{plan} does not explicitly represent turns, combining them with forward motion tasks in individual transitions, in order to reduce the state space and make planning feasible in real time.

The power and planning models interact during execution: M_{power} acts as a safeguard against the plan of M_{plan} diverging from reality and leading to mission failure. M_{plan} only needs to be triggered when the current plan is infeasible (e.g., the robot cannot go past an obstacle or does not have enough battery to complete the mission). Otherwise, the robot avoids running the planner to conserve power[1]. If M_{plan} has overly conservative energy estimates compared to M_{power}, it may miss a deadline due to excessive recharging or taking a less risky but longer route. With overly aggressive estimates, the robot may run out of power.

A map model (M_{map}) specifies locations, their adjacency (i.e., the possibility of moving from one location to another directly), location coordinates, and availability of charging stations at each location. From this information one can derive the distances between each pair of adjacent locations. The map model serves as a foundation for M_{plan}: the states and transitions in M_{plan} are created for a specific map. M_{power} does not directly relate to map, although, as discussed in Section 8.2, creating integration abstractions for M_{power} requires a known map.

In this scenario, modeling method integration should assist the *power safety argument* ("the robot never runs out of power"). This argument asserts that if the robot always keeps at least a certain amount of energy (say, $\overline{err\_total}$) in the battery, then it will not run out of power. This amount is dependent on the total error between the planning model's estimate of required energy and the real energy expenditure of the robot. The total error is some function of three errors:

$$\overline{err\_total} = f(\overline{err\_pow}, \overline{err\_mdp}, \overline{err\_cons}),$$

where:

- $\overline{err\_pow}$ is the error of energy expense estimation by M_{power}, related to experimental noise and imperfect fit of the regression function.
- $\overline{err\_mdp}$ is the error of approximation in the description of M_{plan}, as well as floating point operations when finding an optimal policy in the MDP.

[1]The planner's own power consumption is not modeled, contributing to its inaccuracy.

- $\overline{err\_cons}$ is the error of consistency between M_{power} and M_{plan}, which may arise due to heterogeneity, errors, and optimizations in modeling. Often, this error is ignored and assumed to be 0. As will be shown in 8.1, that assumption is not realistic for these models.

While $\overline{err\_pow}$ and $\overline{err\_mdp}$ can be estimated using conventional techniques (analysis of variance and residuals, bounding based on floating/fixed point representations), neither model can reliably estimate $\overline{err\_cons}$ since it depends on both models.

The challenge of integration here is to verify that the value of $\overline{err\_cons}$ is within a certain bound, thus limiting the inconsistency between the two models. Multiple sources of inconsistency between M_{power} and M_{plan} (i.e., high values of $\overline{err\_cons}$) are possible: mismatch in the maps (e.g., the distances between locations not matching M_{map}), mismatch in the actions (e.g., the same actions taking different amounts of energy), mismatch in the initial or final conditions (e.g., different starting orientations of the robot), and other mismatches. If these mismatches are present, the power safety argument would be flawed, and the robot might unexpectedly run out of power on some missions.

Section 8.1 presents a study of checking integration properties for this system, and Section 8.2 provides more detail about the abstractions used in that study.

3.2 Collision avoidance for a mobile robot

Collision avoidance for wheeled robots and vehicles is a classic safety problem in CPS, used to illustrate the need for hybrid discrete/continuous modeling [28, 153, 170]. However, reasoning about collision avoidance remains a challenge for design and verification: in the past, even the most sophisticated autonomous vehicle systems (e.g., those delivered by Cornell and MIT in the DARPA Urban Challenge [77]) do not achieve flawless practical safety; lately, accidents continue to happen due to the increasing scale of deployment of such systems [49, 93]. One of the challenges is absence of modeling methods that combine formal safety guarantees with means to manage the system's complexity and connect with heterogeneous models.

To help motivate the problem and illustrate our approach, consider MDE of an autonomous wheeled robot moving in a 2D space with other obstacles [172]. The robot's goal is to reach its final destination. The robot can determine its own position and sense obstacles in its vicinity using, e.g., a camera, laser scanner, or a sonar. A planning algorithm determines a sequence of waypoints that lead to a global goal, and then a tactical planner selects the best tactic to the next waypoint depending on the environment, e.g., an intersection or a corridor. The robot's movement controller then executes the selected tactical move. The engineering goal is to model a robot that avoids collisions with obstacles and walls. For this system, I concentrate on modeling the subsystems that are most relevant to collision avoidance: tactical planning, movement control, and movement itself.

The collision safety requirement can be operationalized in three ways (ordered from more aggressive to more conservative):

- *Passive safety:* collisions must not happen when the robot is moving, but are allowed when it is stopped.
- *Passive friendly safety:* collisions are allowed only if the robot is stopped and the colliding moving obstacle was given an opportunity to stop [153].

Concern	Variations
Tactic	Avoiding obstacles, passing intersection, arriving at goal.
Physical space	Unconstrained, constrained box, intersection.
Desired property	Passive safety, passive friendly safety, liveness.
Robot trajectory	Grid, lines, arcs, spirals.
Obstacle behavior	Stationary, moving non-deterministically, moving friendly.
Obstacle knowledge	Bounded speed, bounded acceleration.
Sensing precision	Precise, bounded error.
Sensing timing	Immediate, bounded delay.
Actuation	Precise, bounded error.
Dimensionality	1D (line), 2D (plane).

Table 3.1: Concerns and variations in modeling robotic collision avoidance.

- *Absolute safety:* collisions must not occur under any circumstances [28].

A trade-off between these requirements is that stricter notions of safety may lead to unnecessarily conservative behaviors or unrealistic assumptions. For example, a robot may remain stationary forever in a crowded area if it follows the absolute safety requirement. It is essential for an engineer to experiment with combinations of acceptable safety notions and verifiable algorithms. Thus, typically, several models are required to address such variations in the notion of safety.

Safety definition isn't the only varying concern that affects modeling of robotic collision avoidance. Another is the set of assumptions about the robot's mechanical and embedded systems: What kind of trajectories can the robot move in? How well can acceleration be controlled? How precise and immediate is sensing of obstacles? Yet another concern is obstacles' behavior: Can obstacles move with arbitrary speed or acceleration? Can obstacles switch between stationary and moving? Are obstacles trying to avoid a collision? Variability in answering these questions affects the robot's decisions and guarantees that can be obtained from models. Therefore, an engineer needs to explore a large modeling space when developing collision avoidance systems. Table 3.1 summarizes some of the high-level concerns that underlie modeling of robotic collision avoidance.

The dynamics of collision avoidance protocol was modeled (separately from this thesis) in a combination of hybrid programs and Differential Dynamic Logic ($\mathsf{d}\mathcal{L}$) [189]. This modeling method allows for formal proofs of safety and liveness properties of programs with discrete jumps and continuous evolutions. To manage the complexity, the overall problem of verification has to be split into multiple independent *model variants*, each of which addresses some combination of modeling concerns. For example, one model variant may tackle liveness in an intersection with imprecise sensing, while another may model safely avoiding obstacles using precise sensing.

Due to their unique syntax and semantics, hybrid programs are difficult to relate to other (especially non-hybrid) models and check for consistency. Without integration abstractions, it is difficult to guarantee consistency with other models, so the formal guarantees of hybrid programs may not transfer to the implementations that were created with those models. Therefore, in this context the challenge is to find integration abstractions for the hybrid programs that satisfy several requirements:

- For *expressiveness*, the abstractions should expose the dimensions of variability between the hybrid programs (see Table 3.1) and allow reasoning about hybrid programs in $\mathsf{d}\mathcal{L}$.
- For *soundness*, the abstractions should preserve the soundness of $\mathsf{d}\mathcal{L}$ reasoning (achieved with a formal mapping to hybrid programs).
- For *customizability*, the abstractions should be tailorable to specific programs, possibly representing their common parts in a reusable way (achieved via a common customizable representation).
- For *applicability*, it should be possible to encode common HP models and their $\mathsf{d}\mathcal{L}$ properties, and possibly enable automatic generation of the abstractions and/or hybrid programs.

The validation data for this system was comprised of model variants from an independent robotic collision case study [173, 174], with 15 hybrid programs and 12 $\mathsf{d}\mathcal{L}$ formulas over these programs in total. Since the models were created *prior to and without consideration* of their componentization or integration, these models constitute an acceptable evaluation context for the modeling research of this thesis.

The preliminaries on hybrid programs are presented in Section 6.1. The integration abstractions for hybrid programs are presented in Subsections 6.2.3 and 6.3.2. Section 8.2 discusses a validation study of integration abstractions for hybrid programs.

3.3 Thread and battery scheduling for quadrotor

This validation context is centered on a reconnaissance quadrotor. It is controlled by a set of threads (a.k.a. tasks) with different security levels executing on several processors (a.k.a. CPUs). Each thread executes an infinite sequence of periodic jobs. A job is a finite computation, e.g., a control correction for aircraft stability. The system has dynamic multi-cell batteries with configurable connections between cells so that some cells recharge while others are discharging [126].

This system has multiple design parameters in two engineering domains: thread scheduling and battery scheduling. The thread scheduling domain is related to real-time scheduling of threads along with their allocation to processors, and its concepts specify properties related valid thread allocations and priority assignments, as well as checking schedulability according to a selected scheduling policy, determining processor frequency, etc. The important model elements threads (Threads), which are characterized by periods, deadlines, execution times, and thread security classes (SecCl), of which I consider three: **normal**, **secret**, and **topsecret**.

The CPUs of the quadrotor (CPUs) are characterized by frequencies (CPUFreq) and thread scheduling policies (SchedPol). Each CPU dynamically executes threads that were bound to it at design time (with a CPUBind function). Each thread arrives to the execution queue with its period (Per) and has to be executed before its deadline (Dline). This execution may take up to the thread's worst-case execution time (WCET). The selection of threads for execution is governed by a scheduling policy, of which I consider three: rate-monotonic scheduling (**rms**), earliest deadline first (**edf**) [149], and deadline monotonic scheduling (**dms**) [11]. Each policy determines the priority (Prior) of the thread differently, and threads with higher priorities preempt (i.e., take over the processor from) threads with lower priorities.

The battery domain concerns electrical and thermal aspects of battery design. The central domain element is a set of batteries ($\mathsf{Batteries}$) that are mounted on the quadrotor. Each battery consists of a rectangular array of cells that, in combination, maintain a fixed voltage ($\mathsf{Voltage}$), and each cell has a varying charge. Dynamic battery scheduling allows changing which cells are connected for charging and discharging at run time. This process is governed by a *battery scheduling policy* ($\mathsf{ConnSchedPol}$) [125, 126], and I consider three policies for this system: unweighed round robin with fixed cell groups (**FGuRR**), weighed kRR with fixed parallel cell groups (**FGwRR**), and weighed kRR with cell group packing (**GPwRR**).

Informally, a battery execution consists of continuous charging, discharging, and resting of cells. An important run-time characteristic of the battery is *thermal neighbors* — cells that exchange heat conductively through a connector. [2] This concept is motivated by related work in battery design: there is a close connection between thermal neighbors and thermal runaway [124]. Thermal neighbors are encoded with a function TN: in each state of battery b, $\mathsf{TN}(b, i)$ denotes the number of cells with i thermal neighbors. Relatively large numbers of thermal neighbors would indicate high thermal connectivity within the battery, which may lead to a thermal runaway.

The quadrotor has to satisfy five requirements, each addressed by a different model:

- *Thread schedulability:* all computational jobs must meet the deadlines required by the control algorithms.
- *Data security:* threads with different security levels must not run on the same CPU.
- *Energy efficiency:* CPUs must operate on the minimal frequency possible, thus maximizing battery life.
- *Safe concurrency:* threads must be free of deadlocks and race conditions.
- *Thermal safety:* even if a battery cell overheats, it must not trigger a chain reaction called *thermal runaway* [39].

To satisfy these requirements, six models from different engineering domains are used to represent the quadrotor:

- A scheduling model (M_{sch}) is a discrete cyber model that captures the behaviors of a scheduler and several threads. Implemented in Spin, M_{sch} encapsulates the logic of the thread scheduling policy and the rules behind preemption, which are encoded in a relation $\mathsf{CanPrmpt}$.
- A data security model (M_{sec}) analyzing each thread's source code (access to resources like network, I/O, and third-party libraries) to mark it with various levels of trust.
- A CPU model (M_{cpu}) is a physical model of the computing hardware that describes the electrical dynamics of a processor — the relationship between $\mathsf{CPUFreq}$, maximum frequency ($\mathsf{CPUFreq}_{\mathsf{max}}$), voltage, and current. M_{cpu} also provide the algorithms to reduce voltage and, hence, power consumption.
- A safe concurrency model (M_{rek}), which contains source code assertions on correct concurrent behavior (e.g., no deadlocks or race conditions). These assertions are checked by a bounded model checker Rek [35]. This checking is only applicable when the system uses

[2]As opposed to electrical neighbors – cells that are connected to each other electrically, no matter how far apart physically they are.

implicit deadlines (i.e., periods are equal to deadlines) and fixed-priority scheduling.

- A thermal runaway model (M_{tr}) encodes the thermal dynamics of the battery. This model comes with an algorithm that checks whether overheating in one cell would a thermal runaway.
- A battery scheduling model (M_{bsch}) encodes the electrical dynamics of charge/discharge for individual cells. This model comes with an algorithm to determine the optimal scheduling policy for discharging and charging battery cells.

These models serve as the basis for six analyses ($\mathcal{AN}$):

- Bin packing [56]: assigns threads on CPUs to ensure schedulability;
- Secure thread allocation: computes permissible thread co-locations based on security levels;
- Frequency scaling: minimizes the CPU frequency given the threads assignment;
- Rek model checking [35]: checks if threads satisfy user-specified safety properties like absence of race conditions and deadlocks;
- Thermal runaway checking: determines patterns of battery cell connections that lead to thermal runaway;
- Battery scheduling: determines a battery scheduler given the required operation time and battery size.

Arbitrary independent use of these analyses can lead to designs that do not satisfy the requirements. For example, running bin packing before thread allocation could violate secure co-location constraints set by the latter, but not present at the time we run the former. Similarly, a system may miss deadlines if the frequency of CPU with the $\mathbf{edf}$ policy is determined by a frequency scaling algorithm that assumes the $\mathbf{dms}$ policy. Therefore, the integration goal for this system is to ensure correct cooperative usage of the analyses, with satisfaction of their dependencies and their invocation only in appropriate contexts.

Section 8.3 demonstrates systematic integration of these analyses, ensuring satisfaction of their dependencies and assumptions. The assumptions are formalized using IPL in Subsection 8.1.3. An evaluation of the integration abstractions supporting this study can be found in Subsection 8.2.3.

3.4 Reliable and secure sensing for an autonomous vehicle

This validation system is a fleet of wirelessly connected self-driving cars on a highway. Specifically, the focus is a braking scenario: two cars are cruising in the same lane at highway speeds, and the leader car is slowing down. The follower car is equipped with adaptive cruise control. The leading car is about to stop, and the follower needs to make a safety-critical decision: at what point and how hard to actuate the brakes. This decision to brake is informed by several sensors that estimate velocity and position relative to the leading car.

The car systems use velocity and distance sensors for braking. Two distance sensors each use different technologies to measure distance: a Lidar for laser ranging and a car-to-car (C2C) communication network [3] to exchange position information. The Lidar is physically internal

[3] `www.car-2-car.org`

Sensor variable	Technology	Placement
Distance	Lidar	Internal
Distance	C2C	External
Velocity	Speedometer	Internal
Velocity	GPS	External

Table 3.2: Sensor type, technology and placement

to the car, and the network be accessed from the outside. The car is also equipped with two velocity sensors using a different technology: a GPS and a traditional magnetic speedometer. The speedometer is physically accessible only from inside the car, while the GPS is accessible outside. Table 3.2 shows the sensed variable, technology, and placement for the distance and velocity sensors in self-driving cars.

The sensors send data to the braking controller through the CAN (Controller Area Network) bus. Based on this data, the controller decides the moment and power of braking at each periodic execution. Since the controller has no perception of the physical world except through the sensors, it is important to know which sensors are more trustworthy than others. Thus, *trustworthiness* is another important sensor parameter [159], indicating whether a sensor can potentially be compromised by an attacker. A sensor's trustworthiness is evaluated within the context of an adversary model, described below.

For this validation system, the adversary is limited to attacks on the sensors. Thus, the other components of the system (such as controllers) are assumed to be trustworthy. This scope restriction is made to focus on potential vulnerabilities due to analysis interactions. Other analyses for general component trustworthiness would be complementary and are out of scope for this scenario.

Consider three adversary profiles:

- A *powerful adversary* that can attack any sensor, regardless of whether the sensor is located internally or externally. One known case of such an adversary is one with access to CAN bus [131]. By forging CAN packets, the attacker can cause a variety of system failures. However, full internal network access is not always a realistic assumption for a moving vehicle.
- An *external adversary* that can successfully attack external sensors via physical channels, such as infrared [228] or short-range wireless [36].
- An *internal adversary* that has access to internally placed devices like a radio, USB reader, or speedometer.

The last two profiles are more realistic: these adversaries are less powerful, but intelligently manage to exploit a vulnerability using limited resources. We make several assumptions about these adversaries. They have a technical capability to get information about the structure, properties (such as in Table 3.2), and operation of system components by exploring similar systems. For example, an adversary knows that a Lidar sensor does not work in the presence of fog. A realistic adversary can gain such system knowledge by either examining a target system or obtaining such information from third parties. We assume that the adversary does not have the computational capabilities to break strong cryptographic security measures, e.g., encryption. An adversary can

Sensor	Available in mode			
	nominal	fail 1 (fog)	fail 2	fail 3
Lidar	✓	✗	✓	✓
C2C	✓	✓	✗	✗
Speedometer	✓	✓	✓	✗
GPS	✓	✓	✓	✓

Table 3.3: Configurations output by the FMEA analysis. ✓ indicates that the sensor is functioning properly. ✗ indicates that the sensor is malfunctioning and not providing data.

attack sensors in any order, and we do not make any limiting assumptions about duration of attacks.

In this context we consider three analyses:

- Failure Modes and Effects Analysis (FMEA, $\mathcal{A}_{fmea}$).
- Sensor trustworthiness analysis ($\mathcal{A}_{trust}$).
- Control safety analysis ($\mathcal{A}_{ctrl}$).

The goal of *FMEA* is to incorporate redundancy into the design to handle random failures. To achieve this, FMEA considers the probabilities of random sensor malfunction. It further assumes that failures of different sensors are independent. In the braking scenario, FMEA could output the three configurations shown in Table 3.3. The nominal mode indicates the default situation when all sensors function properly. Consider the example of ”Fail mode 1” configuration. FMEA outputs this configuration after considering foggy conditions. Since Lidar may not work under foggy and rainy conditions, the configuration indicates that the Lidar sensor may not function properly. The remaining sensors function properly. The system may have several probable failure modes depending on the technologies used. FMEA may also change the sensor set if the probability of random system failure is too high.

FMEA in AADL uses the *Error Annex* [57], a standardized sublanguage, that defines error state machines where failure modes and recovery transitions are specified for each component. For example, a wireless network error model can have two states – nominal and failed – and change between them via transitions that have particular probabilities. In addition, error and recovery propagation patterns describing, for instance, how a processor failure propagates to networks, devices, and the software components that run on them are affected. Using the outputs of FMEA, engineers improve the system's reliability (e.g., by making some components redundant). In this study, I take a broad view of FMEA, which includes the identification of failure patterns and changing the design to make it more reliable.

The *sensor trustworthiness analysis* determines whether a sensor can be compromised by an attacker. This analysis takes the following inputs: sensor placement (internal or external to the vehicle, connections to networks and controllers), technical characteristics (technology, communication protocol, encryption, manufacturer, and component version) and an adversary model (formulated in terms of possible actions on components). Note that, unlike FMEA, $\mathcal{A}_{trust}$ does not consider the probabilities of sensor malfunction due to random failures. Instead, it takes into account that the probability of an adversary attacking two similar sensors is interdependent.

Developing new trustworthiness evaluation methods is out of scope of this work. Instead, I

Sensor	Placement	Powerful Adversary	External Adversary	Internal Adversary
Lidar	Internal	✗	✓	✗
C2C	External	✗	✗	✓
Speedometer	Internal	✗	✓	✗
GPS	External	✗	✗	✓

Table 3.4: Sensor trustworthiness for the three adversary models.

target the existing design-time and run-time analyses [159, 168]. Design-time methods can be applied directly to a model, and run-time methods can be used in a simulation, and the produced data can be used to infer trustworthiness. I assume that there exists an appropriate trustworthiness evaluation method and do not place specific constraints on it.

Table 3.4 shows the output of the trustworthiness analysis for three adversary models. In the case of a powerful adversary that can attack both external and internal sensors, trustworthiness analysis would determine that all four sensors in our scenario are not trustworthy. In the case of an adversary that can attack only external or internal sensors, it outputs that respectively only the external or sensors are not trustworthy.

The *control safety analysis* decides whether control is functionally correct, stable and meets the required performance level. This analysis needs to consider various control quality metrics, such as settling time and overshoot. In braking controllers for autonomous vehicles it is important to find a balance between a smooth response that is comfortable for the passengers and a sufficiently low rise time so that the braking process completes in time.

Similar to FMEA and trustworthiness analysis, control analysis makes assumptions about the sensors. As an example of this analysis, consider an algorithm by Fawzi et al. [68] that interprets data from potentially compromised sensors in order to estimate the system state. This algorithm assumes that at least half of the sensors are trustworthy; otherwise, it cannot estimate the state properly. This is an important security assumption required by the control analysis to evaluate the safety of controllers.

The system is represented by three models related to their respective analyses: reliability model, trustworthiness model, and control model. The reliability model (M_{fmea}) captures reliability-related design information: whether devices are powered and available (i.e., have not failed), and the chance of random failure for each sensor. It also serves as the basis for the FMEA analysis. The sensor trustworthiness model (M_{trust}) determines whether sensors can be compromised by the selected attacker, and serves as the basis for the sensor trustworthiness analysis. The control model (M_{ctrl}) captures the necessary variables for each controller and determines whether the overall control is safe (by the means of the control safety analysis).

The integration goal for this system is to execute the analyses in a way that creates a reliable, secure, and safe design. Interactions between analyses, in particular the lack of mutual knowledge between $\mathcal{A}_{fmea}$ and $\mathcal{A}_{trust}$, present a challenge: unsatisfied assumptions behind any of the analyses can lead to vulnerabilities, which can be exploited by an adversary. In this scenario, control safety analysis makes an assumption that at least half of the sensors are sending trustworthy data. This assumption can be broken in two ways. The first way may occur at design-time, when the most error-prone sensors are also the ones that not trustworthy. FMEA may try to replicate

			Available in mode:			
Variable	Sensor	Trustworthiness	nominal	fail 1 (fog)	fail 2	fail 3
Distance	Lidar	✓	✓	✗	✓	✓
Distance	C2C	✗	✓	✓	✗	✗
Velocity	Speedometer	✓	✓	✓	✓	✗
Velocity	GPS	✗	✓	✓	✓	✓
Control safety assumption			✓	✗	✓	✗

Table 3.5: External attacker exploiting inter-domain vulnerabilities.

untrustworthy sensors to increase reliability, thus decreasing the proportion of the trustworthy (and not error-prone) sensors below 50%. As a result, the system's controller can be misled by its untrustworthy sensors, which provide more data than the trustworthy ones.

The second possibility for the assumption of $\mathcal{A}_{ctrl}$ to be broken is at run time. Even if an external attacker isn't powerful enough to compromise all the sensors in the nominal mode, it is possible to exploit the system when one of sensors is not available, e.g., due to fog. In foggy conditions, the (trustworthy) lidar sensors are not available, and the control algorithm has to rely on the (untrustworthy) C2C network, which can be exploited to spoof distance readings with larger values. Assuming the car still has time to brake, the misled controller may miss the deadline for braking and potentially cause a crash. The cause of this vulnerability is that the assumption of $\mathcal{A}_{ctrl}$ doesn't hold in all likely failure modes.

Table 3.5 illustrates an external adversary using the unsatisfied assumption failure modes to cause system failures in two out of four modes. In the nominal mode both distance and velocity sensors have the trustworthiness proportion of 50%. In fail mode 1 distance sensing is compromised because the only distance sensor C2C is untrustworthy. Fail mode 2 has the required proportion of trustworthy sensors. Fail mode 3 violates the assumption because the only available velocity GPS sensor is compromised. Thus, an external attacker may be harmless in the nominal mode, but is still capable of exploiting the vulnerability that comes from not considering failure modes.

I performed a study of using analysis contracts (Chapter 7) to prevent the aforementioned vulnerabilities from being introduced by the analyses. This study is described in Subsection 8.3.2, while the integration abstractions for it are described in Subsection 8.2.4.

Chapter 4

Approach to Modeling Method Integration

This chapter presents a high-level overview of the approach to modeling method integration. Given a set of modeling methods (specifically, models and analyses for a CPS), this approach addresses the three conditions of successful integration, which were formulated at the end of Section 2.2. I briefly present the three parts of the approach and discuss integration arguments that rely on these parts.

The approach uses several elements to formally represent complex relationships between heterogeneous models and analyses. The central element in formalizing this relationship is the notion of an *integration property* — a checkable logical statement over several models. Integration properties express conditions of model consistency or matching analysis contexts. Unlike most existing integration approaches, these properties target specific contexts and requirements, allowing engineers to tailor formalizations of consistency to accommodate diverse CPS modeling methods.

The approach prescribes three steps to integrate the models and analyses (for an overview, see Figure 4.1):

1. *Create integration abstractions:* construct intermediate representations (views and behavioral constraints) that provide the basis for multi-model integration properties.
2. *Specify and verify integration properties:* express the intended relationships among the models and check if the models indeed relate in the specified ways.
3. *Execute analyses:* carry out a well-ordered sequence of analyses on consistent models and ensure that the effects of the analyses do not introduce errors or violate model consistency.

In the rest of this chapter, I give a brief description of each step and summarize the integration argument.

4.1 Integration Abstractions

To reduce the syntactic and semantic gap between models, analyses, and integration properties, my work uses two kinds of abstractions:

- *Views* — hierarchical component models annotated with properties. These models are inspired by architectural descriptions [46].

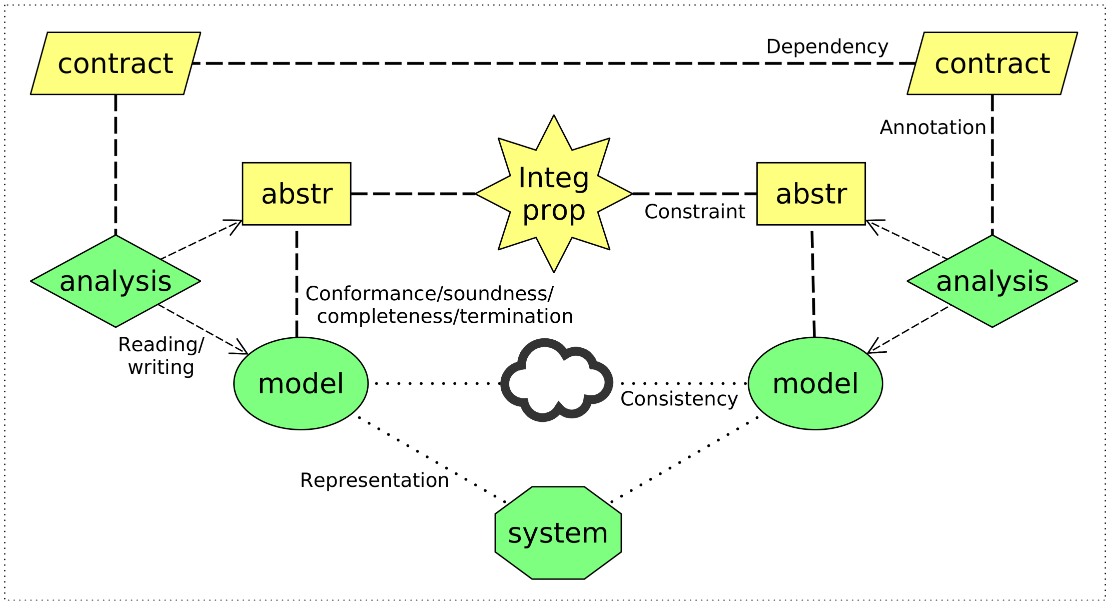


Figure 4.1: Approach to integration of CPS modeling methods.

- *Behavioral properties* — expressions that constrain or quantify behaviors written in model-specific languages. These expressions are inspired by specification languages based on modal logics [6].

Views are used to directly expose discrete structures of models as architectural elements. These elements are defined in a custom vocabulary called an *architectural style*. For example, a view may encode active agents specified in a model as components of a certain type [209]. In my approach, views can be instantiated in two architecture description languages — Acme [86] and AADL [71] — for the following reasons. Acme is domain-agnostic and contains general and flexible constructs (styles, multiple inheritance, first-class connectors), which support customization for CPS constructs and multiple views [23]. Tailored to the architectural style of embedded systems, AADL contains a substantial library of analyses and a mechanism for sub-language extensions, called *annexes* (which are useful for encoding contracts for analyses, as discussed below in Section 4.3).

Unlike views, behavioral properties act as black-box interfaces through which models can be accessed by querying: one can request an evaluation of an expression, which evaluates to a concrete value only in the context of the model, and the model returns this value. For example, a model of a mobile robot can be queried whether the robot eventually reaches the goal. In my approach, behavioral properties are specified in languages that are rooted in two logics: linear temporal logic (LTL) [191], and probabilistic computation tree logic (PCTL) [98]. Aside from these two languagues having different operators and underlying models, their choice was due to their use in the case study systems (see Chapter 3).

These two abstractions are complementary. Views expose structural elements of the model, bypassing the syntactic idiosyncrasies of its model. Views are convenient when integration depends on a relatively small number of the model's discrete elements. In contrast, behavioral properties are used to (indirectly) access the model's semantics (e.g., characteristics of its behaviors), which are difficult to represent in views because they may be infinite, continuous, and non-trivially related to the model's syntax.

For integration of models and analyses to be sound, integration abstractions need to satisfy several requirements. These requirements characterise the fitness of integration abstractions as proxies of models. Views expose model elements as view elements, so the requirements are *soundness* (i.e., every view element is based on some model elements) and *completeness* (i.e., every relevant[1] model element is accounted for by view elements). Behavioral properties are computable functions of model elements, so the requirements for them are *soundness* (i.e., the returned value is correct with respect to the semantics of the behavioral language and the model) and *termination* (i.e., the computation of the query terminates).

Chapter 6 describes views/behavioral properties and their use for model integration.

4.2 Integration Properties

Integration properties are logical specifications that connect several models through their abstractions. These specifications formalize two conditions of successful integration (as described at the end of the previous chapter): model consistency (i.e., models are consistent if integration properties are satisfied) and appropriateness of analysis contexts (i.e., a context is appropriate if its integration properties are satisfied). In practice, if an integration property fails, it may be due to a true negative (an inconsistency between models or an inappropriate analysis context) or an incorrect abstraction (a false negative, like an incomplete view).

In my approach, integration properties are specified in the *Integration Property Language* (IPL). To enable expressive properties over multiple models, IPL combines behavioral semantics with static system-wide reasoning. Specifically, IPL formulas use first-order quantification for static view constraints and modalities for model behavior constraints. To verify integration properties, IPL relies on an SMT solver and several model checkers. IPL is also designed to be extensible with new models and behavioral logics.

An example of an integration property is that, for a given mission of a mobile robot, one model's estimate of the mission's power consumption does not disagree with another model's estimate. If these models are written in different formalisms, it would be difficult to connect them directly. Instead, suppose one model exposes atomic tasks of a mission in a view, and the other allows queries of behavioral expressions to it. Then it is possible to write the following assertion in IPL: *"if one model considers a certain starting power budget sufficient for some mission, then the other model will also consider this budget to be sufficient for the same mission."* If this property and its converse hold, the two models have consistent power dynamics.

To support correct integration, two requirements need to be satisfied in this step. First, IPL should be *expressive* enough to capture the intended complex relationship between the elements of models. Second, verification of IPL specifications should be *sound* (i.e., if IPL returns a result, this result should be correct with respect to the semantics of IPL). IPL verification is not necessarily *complete* (i.e, IPL does not return a result for any expression) because guaranteeing it would limit the expressiveness to less than that of the first-order logic.

[1] A model element is relevant if it can affect satisfaction of an integration property.

Chapter 5 describes specification and verification of integration properties in IPL.

4.3 Integration of Analyses

Each analysis operates on one or several models by reading and changing their elements (e.g., finding an optimal set of control gains). The last step of my approach ensures that the analyses are integrated by automating their execution. Here, it is required that analyses are executed only when their execution does not cause integration issues. That is, the order of execution should *respect the data dependencies* (i.e., analyses do not use stale information or overwrite new models with old) and the *context of execution should be appropriate* (i.e., analyses are invoked in an appropriate context).

A correct analysis ordering is one where all analyses are in order of their dependencies. For example, if analysis A_1 depends on analysis A_2 (i.e., one of A_2 outputs is one of A_1 inputs), then A_2 should be executed before A_1. For instance, a CPU scheduling analysis, which determines the voltage required by CPUs, should be followed by a battery design analysis, which uses the voltage as a requirement.

I provide a specification called *analysis contracts* to ensure that analyses are executed in an appropriate context. Every analysis is annotated with its contract. A contract specifies the inputs of the analysis in terms of the elements of the model(s) that the analysis reads, and outputs — in terms of the elements of the model(s) that the analysis writes. To determine dependencies, elements of views can be used instead of model elements: if analyses are dependent on the same view elements, they are also dependent based on the same model elements. The inputs and outputs help determine a correct dependency order, which is built by creating an analysis dependency graph and, given the desired analyses, selecting any sequence that leads to it.

Further, each contract specifies assumptions and guarantees of the analysis in IPL. The assumptions are binary statements that need to be valid before an analysis executes, while the guarantees need to be valid afterwards. If the assumptions of an analysis are not valid, the analysis may produce incorrect results and should not be executed. If the guarantees of an analysis are not valid, the analysis results are incorrect and should be reversed by restoring the pre-analysis versions of models. Here IPL is reused for specifying assumptions and guarantees because IPL enables constraints over a set of models, similarly to model consistency constraints.

Assumptions and guarantees of an analysis are useful in three cases (formalized in the next section):

1. To ensure that the analysis is executed in an appropriate context, and its inputs match its expectations (as intended by the creators of the analysis).
2. To ensure that the analysis outputs fit into the multi-model context and do not violate model consistency. In this case, the analysis may need to assume that the models are initially consistent because it may not be able to restore consistency if it was not already present[2].
3. To ensure that the analysis is implemented correctly. If an analysis has an error in its implementation, it may lead to errors in models (not necessarily related to MMI). Specifying the

[2]An analysis may waive this assumption if its goal is to repair model consistency that has been violated.

declarative conditions of correct analysis execution may prevent analyses from introducing errors to models.

Chapter 7 describes how analysis contracts ensure correct execution of analyses.

4.4 Integration Argument

Here I provide a general formalized argument for integration of models according to the described approach. This argument identifies the obligations of integration abstractions and IPL necessary to show that the models are related as intended. This argument addresses the first and third requirements from Section 2.2. The second requirement (data dependencies for analyses) is not related to integration of models and is addressed separately in Chapter 7.

Consider n models $\mathsf{M}_1 \dots \mathsf{M}_n$ with their respective integration abstractions $\mathbb{A}_1 \dots \mathbb{A}_n$, which are created in the first step of the approach. The goal is to demonstrate that models are in a consistency relation (written as a predicate, $\mathrm{consistent}(\mathsf{M}_1 \dots \mathsf{M}_n)$). This relationship is formalized based on the abstractions created in the first step of the approach, using an IPL statement f (written by an engineer, as detailed in Chapter 5) and checked in the second (for consistency of models) and third step (for analysis context) of the approach (Section 4.2).

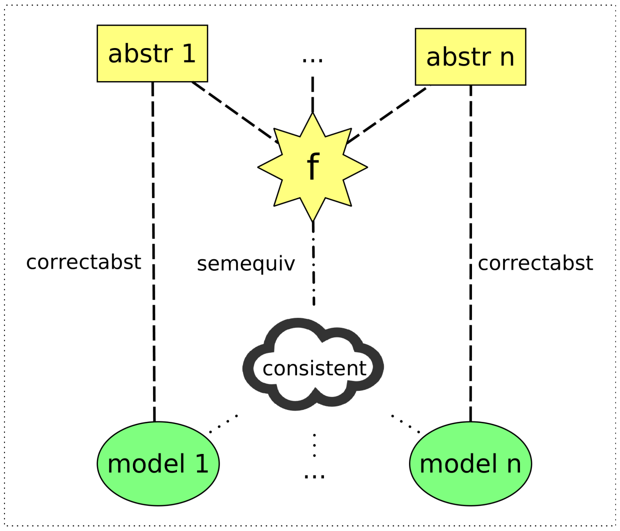


Figure 4.2: Parts of the integration argument.

Suppose the abstractions are correct, the IPL statement expresses the consistency predicate accurately, and the verification of this property is sound (with respect to the IPL semantics). In this case, the IPL statement should hold if and only if the models are consistent. Thus, the following implication (illustrated in Figure 4.2) holds for any models, abstractions, and formulas:

$$\begin{gathered}\mathrm{correctabst}(\mathsf{M}_1, \mathbb{A}_1) \land \dots \land \mathrm{correctabst}(\mathsf{M}_n, \mathbb{A}_n) \land \\ \mathrm{semequiv}\left[f, \mathrm{consistent}(\mathsf{M}_1 \dots \mathsf{M}_n)\right] \land \mathrm{soundverif}(f) \rightarrow \\ \left[\, \mathsf{M}_1 \dots \mathsf{M}_n, \mathbb{A}_1 \dots \mathbb{A}_n \models f \iff \mathrm{consistent}(\mathsf{M}_1 \dots \mathsf{M}_n) \,\right],\end{gathered} \tag{4.1}$$

where the predicates have the following meaning:

- $\mathrm{correctabst}(\mathsf{M}, \mathsf{IA})$ means that IA is a correct abstraction of M. If IA is a view, it should be a sound and complete view of M. If IA is a behavioral property, its query to M should terminate and return a sound result.
- $\mathrm{semequiv}(f, r)$ means that formula f is semantically equivalent to predicate/relation r. That is, f is satisfied if and only if r holds.
- $\mathrm{consistent}(\mathsf{M}_1 \dots \mathsf{M}_n)$ means that models M_1 through M_n are consistent among each other — or generally are in an intended relationship to each other.
- $m_1 \dots m_n \models f$ means that for models/abstractions $m_1 \dots m_n$ formula f evaluates to truth.

The implication of Equation (4.1) holds because, due to the $\mathrm{semequiv}$ assumption, the elements of all abstractions $\mathsf{IA}_1 \dots \mathsf{IA}_n$ are constrained by f according to the same semantics as the model elements from $\mathsf{M}_1 \dots \mathsf{M}_n$ in constraint $\mathrm{consistent}$. Since the abstractions are assumed to be correct, the abstraction elements represent the model elements precisely. Finally, since the verification of f is assumed to be sound, it evaluates to the same true/false value as the consistency predicate.

Now, assuming that models can be checked for consistency, let us turn to integrating analyses and examine the three cases from Section 4.3. Consider an analysis $\mathcal{A}$ that changes a given set of models ($\mathbb{M}$) to a different set ($\mathbb{M}'$):

$$\mathcal{A}(\mathbb{M}) = \mathbb{M}'.$$

An engineer is required to specify the assumptions (A) and guarantees (G) in the contract (C) of $\mathcal{A}$. Although the content of A and G may differ depending on their use (listed at the end of the previous section and discussed below), it is required that the assumptions and guarantees are satisfied by the respective models:

$$\big[\mathbb{M} \models \mathsf{A}\big] \wedge \big[\mathbb{M}' \models \mathsf{G}\big].$$

In the first case (checking appropriate context), the assumptions specify the (otherwise informal) conditions of an analysis matching its intended context. If this specification is complete (i.e., covers all the expected conditions), it is guaranteed that the analysis will not experience context mismatch. Nevertheless, partial specification can also be useful when targeting the context mismatches that are more likely or difficult to notice.

The second case (preserving model consistency for $\mathbb{M}'$ after it has been established for $\mathbb{M}$) can be achieved in two ways. First, a contract can used to ensure consistency directly, every time an analysis is run. That is, the assumptions presuppose consistency before the analysis, and guarantees demand consistency after the analysis:

$$\big[\mathsf{A} \equiv \mathrm{consistent}(\mathbb{M})\big] \wedge \big[\mathsf{G} \equiv \mathrm{consistent}(\mathbb{M}')\big].$$

The other way to approach the second case is to logically derive the auxiliary conditions that imply consistency of models after analysis execution. Specifically, one can write assumption and guarantee conditions to complement the pre-analysis consistency and imply the post-analysis consistency:

$$\mathrm{consistent}(\mathbb{M}) \wedge \big[\mathbb{M} \models \mathsf{A}\big] \wedge \big[\mathbb{M}' \models \mathsf{G}\big] \rightarrow \mathrm{consistent}(\mathbb{M}').$$

For the third case, the contract is written based on the logic internal to the analysis (i.e., not available to other analyses or for integration). The assumptions declare the necessary constraints on the inputs, and guarantees specify the correctness conditions (not necessarily complete):

$$\left[\mathbb{M} \models \mathsf{A}\right] \rightarrow \left[\mathbb{M}' \models \mathsf{G}\right].$$

The three cases are not mutually exclusive: all three cases can be used simultaneously, thus checking for an appropriate context, preserving consistency, and ensuring correct implementation. The approach does not constrain engineers to a particular workflow or use of contracts, giving freedom to enforce partial constraints that are deemed necessary in a project. The tradeoff of this freedom is that analysis contracts require models to be evaluated and, hence, do not permit abstract reasoning about analyses separately from models.

To summarize, this chapter described the three parts of the approach (which correspond to the following three chapters) and a formal argument for integration.

Chapter 5

Part I: Integration Property Language

This chapter presents the first part of the approach to modeling method integration — the *Integration Property Language* (IPL). In terms of the overall approach, the goal of IPL is to express and check integration properties (step 2 of the approach, see Chapter 4), given that appropriate abstractions are available. Creation of these abstractions is covered in Part II (Chapter 6). IPL properties can be used for integration of model-based analyses (Part III, Chapter 7).

For illustration purposes, the next section introduces a motivating integration property for the energy-aware robot (described in Section 3.1). Next, Section 5.2 gives an overview of the IPL design. Afterwards, I describe the syntax of IPL in Section 5.4, semantics in Section 5.5, and the verification algorithm in Section 5.6. Validation studies of IPL can be found in Section 8.1.

5.1 Motivating Integration Property

In the context of an energy-aware mobile robot, consider a potential inconsistency between M_{power} and M_{plan} that threatens the soundness of the power safery argument described in Section 3.1. These models may be inconsistent in energy estimates for turns of the robot due to their difference in representing these turns: M_{plan} models turns implicitly, combining them with forward motions into single actions to reduce the state space and planning time. In M_{power} however, turns are explicit tasks, separate from forward motion. The potential inconsistency between turning energies between M_{power} and M_{plan} can be checked with an integration property, informally stated as "*the difference in energy estimates between the two models should not be greater than a predefined constant* $\overline{err\_cons}$". This property would enable sound power safety argument by putting a bound of $\overline{err\_cons}$[1].

This integration property is difficult to verify for two reasons. First, the abstractions are different: M_{plan} describes states and transitions (with turns embedded in them), whereas M_{power} describes a stateless relation. Second, there is no single means to express such integration properties formally: PCTL (Probabilistic Computation Tree Logic [134]) is a property language for M_{plan}, but M_{power} does not come with a reasoning engine. Finally, even if these obstacles are

[1]As detailed later, we use $\overline{\text{overlines}}$ to mark static entities (not changing over time), and $\underline{\text{underlines}}$ to mark behavioral entities (changing over time in model states).

overcome, the models are often developed by different teams, so they need to stay separate and co-evolve.

To verify this property, I take the approach described in Chapter 4. PCTL-based behavioral properties of M_{plan} will be used to reason about the probabilistic and temporal aspects of M_{plan} traces, such as checking whether a robot eventually reached a goal. A view V_{power} will be used for reasoning about the static/stateless elements of M_{power}. V_{power} is used as a *task library*, containing all atomic tasks (going straight and rotating in the motivating example) in each location/direction in the given map. Each task is annotated with its properties, such as start/end locations, distance, required time, and required energy. Each task in V_{power} is encoded as a component and has several properties associated with it, thus enabling composition of missions as constrained sequences of components. Using this approach, an informal version of the integration property is shown below.

Property 1 (Consistency of M_{power} and M_{plan})**.** For any three sequential M_{power} tasks $\langle$go straight, rotate, go straight$\rangle$ that do not self-intersect and have sufficient energy, any execution in M_{plan} that goes through that sequence in the same order, if initialized appropriately, does not lead to the robot running out of power (allowing for the charge error of $\overline{err\_cons}$).

The challenge of expressing and checking this property is that missions in V_{power} need to correspond to missions in M_{plan}; e.g., the initial charge of M_{plan} needs to be within $\overline{err\_cons}$ of the expected mission energy in M_{power}. However, the two models use different abstractions (a task with pre/post state values in M_{power}/V_{power}, and an explicit state in M_{plan}) and are specified by different logics (first-order predicate logic for M_{power}, and PCTL for M_{plan}). To enable specification of such properties, I introduce the design and syntax of IPL in the coming sections.

5.2 IPL Design

For applicability to real-world model integrations, the design of IPL is based on three principles:

1. *Expressiveness.* To improve expressiveness over state-of-the-art static abstractions, IPL formulas must combine reasoning over views with behavioral analysis of models (e.g., using modal logics). IPL should combine information from several models using first-order logic (quantification, custom functions).
2. *Modularity.* To be customizabile to diverse CPS models, IPL should neither be tied to a particular property language or form of model behavior (discrete, continuous, or probabilistic), require the reengineering of constituent models. Thus, IPL should enable straightforward incorporation of new models and property languages.
3. *Tractability.* To enable automation in practice, verification of IPL specifications must (in addition to being sound) be implementable with practical scalability.

To support these principles, IPL is based on the following four design decisions:

- *Model integration by logically co-constraining models.* IPL rigorously specifies integration conditions over several models. Logical reasoning is an expressive and modular basis for integration because it allows engineers to work with familiar concepts and tools that are specific to their domains/systems. Moreover, logical specifications allow engineers to vary the precision of integration conditions, potentially allowing bounded inconsistency between models. This thesis targets two modal logics that are common in model-based engineering:

LTL and PCTL.

- *Separation of structure and behavior.* IPL explicitly treats the static (rigid) and dynamic (flexible) elements of models separately. Static elements refer to *views* that serve as projections of static aspects of behavioral models, while dynamic elements occur in *behavioral properties* and refer to model traces. This separation enables tractability because static aspects can be reasoned about without the temporal/modal dimension. IPL enhances expressiveness of integration specifications by syntactically combining rigid and flexible elements. More details about views and behavioral properties can be found in Chapter 6.
- *Multi-step verification procedure.* We combine reasoning over static aspects in first-order logic with "deep dives" into behavioral models to retrieve only the necessary values. We preserve tractability by using tools only within individual well-defined semantics, without direct dependencies between models.
- *Plugin architecture for behavioral models.* To create a general framework for integration, we specify several *plugin points* — APIs that each behavioral model has to satisfy. While the model itself can remain unchanged, IPL requires a plugin to use the model's property formalism for verification. This way, IPL does not make extra assumptions on models beyond the plugin points, hence enhancing modularity.

5.3 IPL Concepts and Preliminaries

Here I formalize the important concepts used in IPL: views, models, and model property languages.

Definition 1 (Architectural View)**.** An *architectural view* V is a hierarchical collection of architectural elements (i.e., components and connectors). Each element has fixed-valued properties, the set of which is determined by its type and values set individually for each element.

IPL uses views for modeling static, behavior-free projections of models. For example, a 2D map encoded in an XML (M_{map}) is a model from which locations can be exposed in a view (V_{map}) as a set of interconnected components ($\overline{Locs}$). Each component is a location, and connectors indicate direct reachability between them. We use views as an abstraction because of their composability, typing, and extensible hierarchical structure. Views cannot change during execution, so no dynamic information (e.g., the current battery charge) is put in views, confining the behavioral semantics to models.

Definition 2 (Formal View)**.** A *(formal) view* V is a tuple of a view signature (Σ^{V}), a semantic interpretation of the symbols in the signature (I^{V}), and a structure of architectural elements ($\mathbb{E}$) on which the signature is interpreted. Σ^{V} contains the symbols of types of view elements ($\mathbb{T}$) and properties of view elements ($\mathbb{P}$). $\mathbb{E}$ includes the sets of elements (annotated with the above types/properties), along with sorts/constants and functions/predicates. I^{V} gives static meaning to the elements in the view signature, independent of state or time.

I use formal views to define the syntax and semantics of IPL. For example, the signature of a view may contain a symbol for the component type of batteries ($\mathsf{Batteries}$) and a property that defines its maximum charge (modeled as a function $\overline{maxBat} : \mathsf{Batteries} \to \mathcal{Z}$). A view may provide a specific structure of architectural elements with two batteries. Then, I^{V} would map the symbol $\mathsf{Batteries}$ to a set with the two batteries, as well as the symbol $\overline{maxBat}$ to a function that returns the maximum charge for each of these two batteries.

I establish an isomorphic relationship between the two definitions by converting architectural models to SMT specifications, as done in my prior work [208]. In short, the elements from the architectural view make up the $\mathbb{E}$ structure of the formal view, and the types/declarations of the architectural view make up the signature of the formal view. The definitions differ in that the architectural view focuses on the concrete contents of the view (e.g., specific component instances), whereas the formal definition focuses on specification symbols (e.g., component types) and uses the view contents as an interpretation of these symbols. Both definitions of views are used throughout this thesis: Definition 1 — for applied modeling (e.g., representing views in case studies), and Definition 2 — for theory behind IPL's syntax, semantics, and verification[2].

Definition 3 (Model)**.** A *(behavioral) model* M is a tuple of a signature (Σ^{M}), an interpretation (I_q^{M}), and a structure to which the interpretation maps the signature symbols. Σ^{M} defines symbols of state variables, modal functions/predicates, and a list of name-type pairs for initialization parameters. I_q^{M} gives meaning to the symbols in Σ^{M} in each state. The parametric structure determines the model's set of behavior traces ($\mathsf{M}.traces$) [12, 96].

An example of a behavioral model is M_{plan}: its signature contains the battery charge ($\underline{bat}$) as a state variable that changes dynamically. The structure is a set of possible states (e.g., a Markov chain), and I_q^{M} defines the mapping between the $\underline{bat}$ symbol, a state in the structure, and a concrete value of battery charge. Such interpretation are called *modal* because they depend on the state (or, mode) of the system.

Definition 4 (Model Property Language)**.** For a class of models, a *model property language* is a language for specifying expressions about a model of that class. From these expressions I_q^{M} produces a value of a type interpretable by views.

For M_{plan}, the property language is based on PCTL for express constraints and querying probabilities over executions of M_{plan}. An example of a statement in this language is $P_{min=?}[\mathsf{F}\underline{bat} = 0]$, which returns the minimum possible probability of the robot eventually running out of power.Model property languages can contain modalities (like F) and operators (like $P_{min=?}$) over statements that include state variables.

Shared by models and views, background interpretation I^B evaluates common sorts, constants (e.g., boolean $\top$ and $\bot$), functions (e.g., addition), and predicates (e.g., equality) from background theories (e.g., the theory of equality or linear real arithmetic). Formally, IPL allows only theories that are decidable [132] and form decidable combinations [180], but in practice it is acceptable to use undecidable combinations for which available heuristics resolve relevant statements.

For the rest of this chapter, I consider a given set of behavioral models $\mathbb{M}$, with some of them abstracted by a given set of views ($\mathbb{V}$). Each view can be seen as some (implicit) function of a model. This thesis focuses on two specific model property languages (LTL and PCTL), although in principle IPL is not limited to them. IPL formulas are written in a context of views and models. Syntactically, IPL formulas are written over a *signature* (Σ) that contains symbols from $\mathbb{V}$ and $\mathbb{M}$: $\Sigma = \Sigma^{\mathsf{V}} \cup \Sigma^{\mathsf{M}}$. Semantically, a formula's context is determined by a *structure* (Γ) that contains

[2]The readers who are familiar with the typical uses of software architectures may find some views in this work familiar (e.g., a view with threads and processors as components), whereas other views may be unusual (e.g., a view where components represent tasks that a robot can do). All these views are used as a standardized notation for integration between heterogeneous models.

interpretations I^{M}_q, I^{V}, and I^B along with the view and model structures.

To proceed, four additional assumptions are needed: (i) for verification purposes, views are fully determined and stay up-to-date with models; (ii) views can be translated into finite SMT specifications; (iii) once initialized, any model can check/query any statement in its property language; and (iv) models and views share the background interpretation. All of these assumptions are satisfied in all the systems/models studied in this thesis (see Chapter 3), and the implications of their non-satisfaction are discussed in Chapter 10.

5.4 IPL Syntax

To support plugging of new models, I keep track of syntactic terms that can be interpreted only by views or models. By isolating the model-specific terms, I allow new model property languages to be plugged into IPL. Hence, I introduce the rigid/flexible separation: *flexible* terms (denoted with underlines, like $\underline{loc}$) are interpreted by I^{M}_q, and *rigid* terms (denoted with overlines, like $\overline{Tasks}$) are interpreted by I^{V}. Terms of I^B are used by both models and views (no special notation; e.g., $<$). To embed model property languages into IPL, the IPL syntax allows model-specific formulas to be defined as flexible "plugins" in the grammar. The rigid part of the IPL syntax is, then, considered *native* because it does not change for different behavioral models.

One challenge is that the relation between IPL and model languages is not hierarchical: native formulas contain plugin formulas, but native terms can also appear in plugin formulas. When an IPL interpreter prepares a model-specific subformula for verification, it should only evaluate the native parts without being tied to the model-specific semantics. For instance, in $\forall x : X \cdot \overline{P}(x) \rightarrow \underline{Q}(\overline{R}(x))$, only $\overline{P}(x)$ and $\overline{R}(x)$ should be evaluated natively, after which $\underline{Q}(\overline{R}(x))$ can be passed to a behavioral model for checking.

The native/plugin syntax is combined according to Figure 5.1. We define each syntax element (box) on top of symbols in Σ and quantified variables ($\mathbb{V}$). We build two types of subformulas: rigid atomic formulas (RATOM) from rigid terms (RTERM), and flexible atomic formulas (MATOM). Our strategy is to keep flexible and rigid syntax separate until they merge in the topmost syntactic term — IPL formulas (FORMULA). In this way, modularity is preserved: compound formulas can be deconstructed into simpler ones that are evaluated by either models or views. The production rules for rigid atoms and terms are given below.

Definition 5 (Rigid term)**.** *Rigid terms* of the language are defined as follows:

$$\text{RTERM} ::= \ \text{VAR} \mid \text{CONST} \mid \text{ELEM} \mid \text{RTERM.PROP} \mid \text{BFUNC}(\text{RTERM}, \ldots \text{RTERM}).$$

A *rigid term* RTERM is either a variable VAR, a constant CONST, an architectural element type ELEM, a property of a rigid term RTERM.PROP[3], a background function BFUNC, or a view function VFUNC.

Definition 6 (Rigid atom)**.** *Rigid atoms* are logical formulas over rigid terms:

$$\text{RATOM} ::= \text{RATOM} \wedge \text{RATOM} \mid \neg\text{RATOM} \mid \text{RTERM}.$$

[3]Properties are only applicable to architectural elements, references to which can be accessed through a variable or a function. All expressions are assumed to be well-typed.

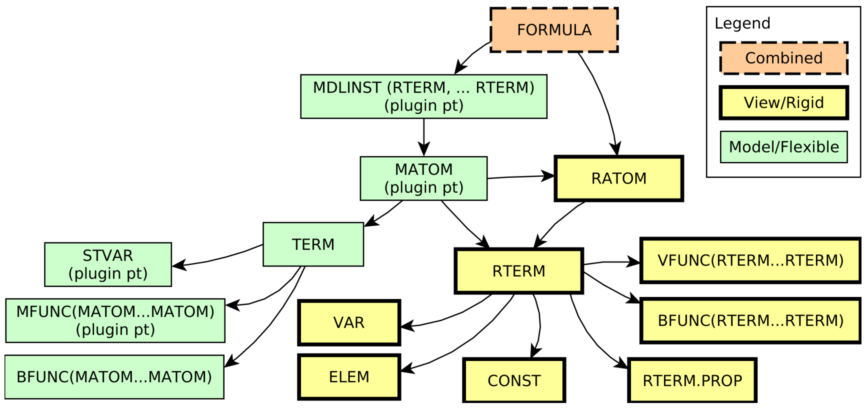


Figure 5.1: IPL abstract syntax. Boxes are syntax elements, and arrows are syntactic expansions.

Thus, a *rigid atom* RATOM is a logical expression over rigid terms. Now we proceed to the flexible part of the syntax.

5.4.1 Plugin Points for Behavioral Properties

To integrate multiple behavioral formalisms into IPL, the syntax defines four plugin points for model-specific constructs. Each plugin point can be instantiated either with an extensible syntactic form (e.g., a modal expression) or a reference to an existing form (e.g., RTERM). Each behavioral model provides its own syntactic elements for plugin instances.

At the level of *flexible terms* (TERM), two plugin points are *state variables* (STVAR) and *model functions* (MFUNC). Each state variable (e.g., the robot's current location $\underline{loc}$) is declared as a pair (name, type) to be referenced from IPL. Each model function declares its name, type, and list of arguments, each of which is name-type pair.

The third plugin point is *model atom* (MATOM), e.g., the expression $P_{max=?}$. It requires one or several syntactic forms with production rules. In addition to model-specific productions (e.g., temporal modalities), MATOM can use elements RATOM and RTERM from the grammar's rigid side (but not vice versa). A model can, for example, plug in an LTL modal expression and use rigid terms in it.

Behavioral models often have parameters defining their configuration and initial conditions (e.g., the starting battery charge of the robot $\underline{initbat}$). To specify such parameter values, I introduce the fourth and outermost plugin point:

Definition 7 (Model Instantiation Clause)**.** *Model instantiation clause* binds rigid terms to model parameters, wrapping MATOM:

$$\text{MDLINST} ::= \text{MATOM}\{\!|\text{RTERM}_1 \dots \text{RTERM}_n|\!\}.$$

The values of RTERM_i are passed as parameters to the behavioral model. Finally, quantification is added on top of the rigid and flexible syntax defined so far:

Definition 8 (IPL Formula)**.** *IPL formulas* are logical formulas with first-order quantification over an instantiated model formula or a rigid atom.

$$\text{FORMULA} ::= \forall \text{VAR} : \text{RTERM} \cdot \text{FORMULA} \mid \text{MDLINST} \mid \text{RATOM} \mid$$

$$\text{FORMULA} \wedge \text{FORMULA} \mid \neg\text{FORMULA}.$$

To demonstrate the customizability of IPL, I provide two extensions of the grammar: first with Linear Temporal Logic (LTL) [191], and second with Probabilistic Computational Tree Logic (PCTL) [134]. Plugins for these logics expand MATOM in different ways.

5.4.2 LTL Plugin Syntax

Linear Temporal Logic is a logic to express temporal constraints on traces [191]. This plugins uses the two usual modalities (until U and next X), and the other modalities expressed through until in a standard way: $\mathsf{G}p \equiv p\mathsf{U}\bot$; $\mathsf{F}p \equiv \top\mathsf{U}p$.

A common model for LTL is a labeled transition system M_{ts} (LTS). State variables are interpreted modally, and more complex elements of state (i.e., modally evaluated functions and relations) are exposed as MFUNC. To express temporal formulas, the LTL plugin introduces several syntactic elements (including five behavioral atoms):

- State variable: STVAR $::= \underline{v}$.
- State function: MFUNC $::= \underline{g}(t_1 \dots t_n)$, where $t_1 \dots t_n \in$ MATOM.
- Background function: BFUNC $::= g(t_1 \dots t_n)$, where $t_1 \dots t_n \in$ MATOM.
- Until: $\text{TATOM}_u ::= \text{TATOM}\mathsf{U}\text{TATOM}$.
- Next: $\text{TATOM}_x ::= \mathsf{X}\text{TATOM}$.
- Conjunction: $\text{TATOM}_a := \text{TATOM} \wedge \text{TATOM}$.
- Negation: $\text{TATOM}_n := \neg\text{TATOM}$.
- A wrapper replaces the MATOM plugin point:

$$\text{MATOM} ::= \text{TATOM} ::= \text{RATOM} \mid \text{TERM} \mid \text{TATOM}_u \mid \text{TATOM}_x \mid \text{TATOM}_a \mid \text{TATOM}_n.$$

5.4.3 PCTL Plugin Syntax

As the second behavioral property language I use extended PCTL (i.e., the variant of PCTL used in the PRISM model checker). It expresses probabilistic constraints over a computation tree, and its models are MDPs and discrete-time Markov chains (DTMCs) [134]. Flexible terms are the same as in the LTL plugin, but MATOM expands into several layered behavioral atoms:

- PATHPROP is a logical expression on a model path using temporal modalities, similar to TATOM in LTL but using bounded until.
- RWDPATHPROP is a logical expression combining co-safe LTL and certain operators to predicate paths on which rewards are calculated.
- PPROP is a boolean check of a probability of a path given by PATHPROP.
- PQUERY is a value query of a probability of a path given by PATHPROP.
- RWDPROP is a boolean check of a reward of a path given by RWDPATHPROP.
- RWDQUERY is a value query of a reward of a path given by RWDPATHPROP.

$$
\begin{aligned}
\text{PATHPROP} &::= \text{RATOM} \mid \text{TERM} \mid \text{PATHPROP} \wedge \text{PATHPROP} \mid \neg\text{PATHPROP} \mid \\
&\quad \text{PATHPROP}\mathsf{U}^{\leq \mathsf{k}}\text{PATHPROP} \mid \mathsf{X}\text{PATHPROP}, \\
\text{RWDPATHPROP} &::= \text{RATOM} \mid \text{TERM} \mid \text{RWDPATHPROP} \wedge \text{RWDPATHPROP} \mid \\
&\quad \text{RWDPATHPROP} \vee \text{RWDPATHPROP} \mid \mathsf{X}\text{RWDPATHPROP} \mid \\
&\quad \text{RWDPATHPROP}\mathsf{U}^{\leq \mathsf{k}}\text{RWDPATHPROP} \mid \mathsf{C}^{\leq \mathsf{k}} \mid \mathsf{I}^{=t} \mid \mathsf{S}, \\
\text{PPROP} &::= P_{o\sim p}[\text{PATHPROP}],\ \text{PQUERY} ::= P_{o=?}[\text{PATHPROP}] \\
\text{RWDPROP} &::= R^{r}_{o\sim v}[\text{RWDPATHPROP}],\ \text{RWDQUERY} ::= R^{r}_{o=?}[\text{RWDPATHPROP}], \\
\text{MATOM} &::= \text{PPROP} \mid \text{PQUERY} \mid \text{RWDPROP} \mid \text{RWDQUERY},
\end{aligned}
$$

where $p \in [0,1], \sim \in \{<, \leq, >, \geq\}, o \in \{max, min, \emptyset\}, t \in \mathbb{N}, \mathsf{k} \in \mathbb{N} \cup \{\text{inf}\}, v \in \mathbb{R}$, and r is a character string (the name of a reward structure).

To summarize the syntax description, IPL formulas express quantified modal constraints over symbols in Σ. To preserve modularity of models and specifications, quantification is used only outside of flexible atoms. So far, IPL contains two extensions with model property languages (LTL and PCTL), and more can be added in the future.

5.4.4 Syntactic Examples

To illustrate the intuition about the IPL syntax, Table 5.1 provides examples of acceptable and unacceptable formulas in the LTL plugin.

#	Example formula	Description	In IPL
(1)	$\overline{f}(1+2) = 3$	View function over a rigid term	Y
(2)	$\forall x : \overline{X} \cdot P(x)$	Quantification over a view-defined set	Y
(3)	$(\mathsf{G}\underline{y} = 10)\{\|\|\}$	Model instance over a modality w/ a state variable	Y
(4)	$\forall x : \overline{X} \cdot (\mathsf{G}P(x, \underline{y}))\{\|\|\}$	Quantification over a model instance	Y
(5)	$\exists x : \overline{X} \cdot P(x) \rightarrow (\mathsf{F}Q(x, \underline{y}))\{\|\|\}$	Quantification over an atom w/ a model instance	Y
(6)	$(\mathsf{G}(\exists x : \overline{X} \cdot P(x, \underline{y})))\{\|\|\}$	Model instance over a quantified formula	N
(7)	$(\mathsf{F}(\underline{y} = \underline{z}))\{\|\|\}$	Mixed models in one term: M_1 owns $\underline{y}$ and M_2 owns $\underline{z}$	N

Table 5.1: Examples of acceptable/unacceptable IPL syntax.

Examples (1)–(3) illustrate base cases of formulas to be supported: unquantified, quantified, and modal. These formulas target only one interpretation (and therefore are trivially modular), although do not improve expressiveness over existing tools. Examples (4) and (5) show the main use case of IPL — quantification over model instances with modalities.

Example (6) is unacceptable because it violates modularity. The existential quantifier cannot be interpreted by the behavioral model that is necessary to interpret the modality. On the other hand, the modality cannot be interpreted by a view. To check such formulas, views and models would need to be merged, which goes against our design for modularity and tractability. Example (7) also violates modularity: no single model can interpret flexible variables from two different models. To check such formulas, one would need to compose the behavioral models, violating the modularity and tractability principles.

Finally, I encode the motivating integration property (Property 1) in IPL below, using quantification to bind constraints on task sequences in V_{power} (with task attributes $start$, end, and expected $energy$) and a PCTL query for M_{plan}. Other integration properties for the energy-aware mobile robot can be found in Subsection 8.1.2.

Property 2. For any three sequential M_{power} tasks $\langle$go straight, rotate, go straight$\rangle$ that do not self-intersect and have sufficient energy, any execution in M_{plan} that goes through that sequence in the same order, if initialized appropriately, does not lead to the robot running out of power (allowing for the charge error of $\overline{err\_cons}$).

"For any three tasks from M_{power} *in a sequence* $\langle$*go straight - rotate - go straight*$\rangle$*"*

$$\forall t_1, t_2, t_3 : \overline{Tasks} \cdot t_1.type = t_3.type = \text{STR} \wedge t_2.type = \text{ROT} \wedge \tag{5.1}$$

"that are well-aligned, do not self-intersect, and have sufficient energy,"

$$t_1.end = t_2.start = t_3.start \;\wedge t_1.start \neq t_3.end \wedge \Sigma_{i=1}^{3} t_i.energy \leq \overline{maxBat} \rightarrow$$

"any execution in M_{plan} *that visits every point of that sequence in the same order,"*

$$P_{max=?}[(\underline{loc} = t_1.start \mathsf{U} (\underline{loc} = t_2.start \mathsf{U} \underline{loc} = t_3.end)) \wedge (\mathsf{F} \underline{loc} = t_2.start)]$$

"if initialized appropriately, is a power-successful mission (modulo $\overline{err\_cons}$*)."*

$$\{|\underline{initloc} = t_1.start, \underline{goal} = t_3.end, \underline{initbat} = \Sigma_{i=1}^{3} t_i.energy + \overline{err\_cons}|\} = 1.$$

5.5 IPL Semantics

Now I give meaning to the IPL syntax in terms of structure Γ by reducing parts of any IPL formula to either Γ's model part (I_q^{M}) or Γ's view part (I^{V}) — but not both for a given subformula.

5.5.1 Semantic Domains and Transfer

IPL syntax elements are interpreted within *semantic domains* – collections of formal objects (e.g., numbers) in terms of which syntax elements can be fully interpreted. For IPL we define two domains: the *model domain* ($\mathsf{D_M}$) and the *view domain* ($\mathsf{D_V}$). $\mathsf{D_M}$ is associated with I_q^{M}, and $\mathsf{D_V}$— with I^{V}.

Definition 9 (Belonging to semantic domain)**.** Syntactic element s *belongs to a semantic domain* D if there exists an interpretation I such that $I(s) \in \mathsf{D}$.

$\mathsf{D_M}$ and $\mathsf{D_V}$ are defined in Table 5.2: the first and third columns contain syntax elements that belong to them. For example, models interpret state variables using their structures, and views can interpret quantified statements using satisfiability solvers. Both domains interpret symbols from background theories (I^B).

The middle column of Table 5.2 indicates if a syntax element, once interpreted, can be *transferred* to the other domain, i.e., if a bijection between its evaluations and some set in the other domain exists. "By value" means mapping to a constant in the other domain. "By reference"

View domain $\mathsf{D_V}$	**Is transferable**	**Model domain** $\mathsf{D_M}$
VAR	Yes, by value	
ELEM	Yes, by reference	
PROP	Yes, by value	
VFUNC	Yes, by value, if all arguments are transferable. Otherwise, no.	
RTERM	Yes, by value	
$\forall x : X \cdot f$	No	
	No	STVAR
	No	MFUNC
	No	MATOM
	Yes, by value	MDLINST
Constants and BFUNC from background theories. Interpretation I^B.		

Table 5.2: Two semantic domains and transfer between them in IPL.

means mapping to an integer ID (e.g., for ELEM, unique integer IDs are generated for referencing in the model). Notice that view domain elements are mostly transferable to the model domain (except quantification). To support modularity, models can transfer only values of MDLINST.

Native semantics IPL formulas are interpreted in the following context: Γ ($\mathbb{V}$, M_i with I_q^{M}, I^{V}, and I^B), states q, potentially infinite sequences of states $\omega \equiv \langle q_1, q_2, \ldots \rangle$, and mapping μ of variables to values. Starting from the bottom of Fig. 5.1 with rigid terms (RTERM), we gradually simplify the semantic context (denoted as the subscript of $[\![]\!]$ and on the left of $\models$).

$$[\![\text{CONST}]\!]_\Gamma = I^B(\text{CONST});\ [\![\text{VAR}]\!]_\mu = \mu(\text{VAR});$$
$$[\![\text{VFUNC}(r_1, \ldots r_n)]\!]_{\Gamma,\mu} = I^{\mathsf{V}}(\text{VFUNC})([\![r_1]\!]_{\mathbb{V},\mu} \ldots [\![r_n]\!]_{\mathbb{V},\mu}),$$
$$\quad\text{where } r_1 \ldots r_n \in \text{RTERM};$$
$$[\![\text{ELEM}]\!]_{\Gamma,q,\mu} = I^{\mathsf{V}}(\text{ELEM}) = \{e\} \subseteq \mathbb{E};$$
$$[\![\text{RTERM.PROP}]\!]_{\Gamma,q,\mu} = I^{\mathsf{V}}(\text{PROP})([\![\text{RTERM}]\!]_{\mathbb{V},\mu});$$
$$\Gamma, \omega, \mu \models \neg f \textit{ iff } \Gamma, \omega, \mu \not\models f, \text{ where } f \in \text{FORMULA or } f \in \text{RATOM};$$
$$[\![\text{STVAR}]\!]_{\Gamma,q} = I_q^{\mathsf{M}}(\text{STVAR});$$
$$[\![\text{MFUNC}(t_1, \ldots t_n)]\!]_{\Gamma,q,\mu} = I_q^{\mathsf{M}}(\text{MFUNC})([\![t_1]\!]_{\Gamma,q,\mu}, \ldots [\![t_n]\!]_{\Gamma,q,\mu}),$$
$$\quad\text{where } t_1 \ldots t_n \in \text{TERM};$$
$$\Gamma, \omega, \mu \models f_1 \wedge f_2 \textit{ iff } \Gamma, \omega, \mu \models f_1 \text{ and } \Gamma, \omega, \mu \models f_2,$$
$$\quad\text{where } f_1 \text{ and } f_2 \text{ are either FORMULA or RATOM};$$
$$\Gamma, \omega, \mu \models \forall x : r \cdot f \textit{ iff } \Gamma, \omega, \mu' \models f, \text{ where } r \in \text{RTERM},$$
$$\quad f \text{ is either FORMULA or RATOM}, \mu' = \mu \cup \{x \mapsto v\} \text{ for all } v \text{ in } [\![r]\!]_{\Gamma,\mu};$$
$$\Gamma, \mu \models (a)[|p_1 \ldots p_n|] \textit{ iff } \mathbb{V}, \mathsf{M}([\![p_1]\!]_{\mathbb{V},\mu} \ldots [\![p_n]\!]_{\mathbb{V},\mu}), \mu \models a, \text{where } a \in \text{MATOM}.$$

5.5.2 LTL plugin semantics

The model of LTL sentences is a state transition system M_{ts} (a state set, an action set, a transition function, an initial state, and a state interpretation I^{M}_q to determine valid propositions in state q) [45]. A transition system defines a set of execution traces ($\mathsf{M}_{ts}.traces$). $\omega^{i,j}$ means a substring of ω from element i to element j inclusively. We evaluate TATOM and FORMULA on a sequence of states (ω). Logical operations and quantifiers are evaluated natively, as defined above.

$$\begin{aligned}
&\Gamma,\omega,\mu \models f \;\; \textit{iff} \;\; \Gamma,q,\mu \models f, \text{ where } q \in \omega^{1,1} \text{ and } f \in \textsc{term};\\
&\Gamma,\omega,\mu \models \mathsf{X}\textsc{tatom} \;\; \textit{iff} \;\; \Gamma,\omega^{2,\infty} \text{ and } \mu \models \textsc{tatom};\\
&\Gamma,\omega,\mu \models \textsc{tatom}_1\mathsf{U}\textsc{tatom}_2 \;\; \textit{iff}\\
&\qquad \exists i : \mathbb{N} \cdot (\Gamma,\omega^{1,i},\mu \models \textsc{tatom}_1 \wedge \Gamma,\omega^{i+1,\infty},\mu \models \textsc{tatom}_2).
\end{aligned}$$

The meaning of FORMULA is described only in terms of Γ by iterating over all traces in the model without any up-front variable mappings:

$$\Gamma \models \textsc{formula} \;\; \textit{iff} \;\; \forall\omega : \mathsf{M}_{ts}.traces \cdot \Gamma,\omega,\emptyset \models \textsc{formula}.$$

5.5.3 PCTL plugin semantics

To evaluate a PCTL formula, we use an MDP (M_{mdp}) as a behavioral model in Γ (or a DTMC (M_{dtmc}) if the model does not have non-deterministic transitions) [134]. It is characterized by finite state set S, finite action set A, an initial state, probability transition function $P : S \times S \to [0,1]$, a discount factor $\gamma \in [0,1]$, reward structures $r_i : S \times S \to \mathbb{R}_{\geq 0}$, and a state interpretation I^{M}_q to determine valid propositions in a given state (q). Temporal modalities in PATHPROP and RWDPATHPROP are characterized for model state q and path ω in the same way as in LTL, with an addition of a bounded until modality ($\mathsf{U}^{\leq \mathsf{k}}$):

$$\begin{aligned}
&\Gamma,\omega,\mu \models f_1\mathsf{U}^{\leq \mathsf{k}}f_2 \;\; \textit{iff}\\
&\qquad \exists i : \mathbb{N} \cdot (i \leq k \wedge \Gamma,\omega^{i,\infty},\mu \models f_2) \wedge (\forall j : \mathbb{N} \cdot j < i \to \Gamma,\omega^{1,j},\mu \models f_1).
\end{aligned}$$

Given policy $\pi : S \to A$, P (one of policies Π) induces a probability measure Pr^{π}_q over paths $Paths(q)$ starting in state q. In turn Pr^{π}_q induces a probability function over formulas that determines the probability of taking a path that satisfies formula f from state q: $Prob^{\pi}(q,f) = Pr^{\pi}_q\{\omega \in Paths(q) \mid \omega \models f\}$.

We can now define formula satisfaction for PPROP and valuation for PQUERY:

$$\begin{aligned}
&\Gamma,q,\mu \models P_{o\sim p}[f] \;\; \textit{iff} \;\; \text{opt}_{\pi\in\Pi}\, Prob^{\pi}(q,[\![f]\!]_{\Gamma,\mu}) \sim p,\\
&[\![P_{o=?}[f]]\!]_{\Gamma,q,\mu} = \text{opt}_{\pi\in\Pi}\, Prob^{\pi}(q,[\![f]\!]_{\Gamma,\mu})),
\end{aligned}$$

where $f \in$ PATHPROP, $\sim \in \{<,\leq,>,\geq\}$, $\text{opt}_{\pi\in\Pi}$ stands for $\sup_{\pi\in\Pi}$ if $o \equiv$ max, and $\inf_{\pi\in\Pi}$ if $o \equiv$ min, and no operator if $o \equiv \emptyset$.

Rewards formulas are evaluated analogously:

$$\Gamma,q,\mu \models R_{o\sim p}[f] \text{ iff } \text{opt}_{\pi\in\Pi}\, \text{Exp}^{\pi}(q,X_{[\![f]\!]_{\Gamma,\mu}}) \sim p,$$

$$[\![R_{o=?p}[f]]\!]_{\Gamma,q,\mu} = \text{opt}_{\pi\in\Pi}\, \text{Exp}^{\pi}(q, X_{[\![f]\!]_{\Gamma,\mu}}) \sim p,$$

where $X_f : Paths^{\pi}(q) \rightarrow \mathbb{R}_{\geq 0}$ is a random reward variable for paths that satisfy f (defined canonically for co-safe LTL and special formulas [134]), and Exp^{π} is its expectation with respect to Pr^{π}_{q}; other variables mean the same as above.

With the semantics of IPL now fully defined. This definition leads to interpreting Equation (5.1) in a way that satisfies the intent of Property 1 (Page 38). Formulas are evaluated by their reduction to subformulas (modularized to individual models/views), and each of them is interpreted by either I^{V}, I^{M}_{q}, or I^{B}.

5.6 IPL Verification Algorithm

Suppose an engineer needs to verify an integration formula f with a signature Σ against Γ, i.e., check if f is a sentence in the *IPL theory* for Γ.

Problem 1 (IPL formula validity)**.** Given $f \in$ FORMULA in Σ and a corresponding Γ, decide whether $\Gamma \models f$.

Intuitively, the goal of IPL verification is to determine whether an IPL formula holds for a given set of views and behavioral models. The first step is determine the extent to which the information from each model is needed, using SMT. This scope of information is represented as values of quantified variables. Second, this information is extracted from each model by evaluating a corresponding subformula on it. Finally, the information is added to the SMT problem with the negation of the original quantified formula, and a satisfiability check determines whether the formula holds.

To formalize the algorithm to solve Problem 1, I introduce several transformations of IPL formulas below. I start with a formalization of the quantifier removal ($RemQuant$) transformation. It is executed until it is not applicable to the input formula.

$$RemQuant \equiv Qx \cdot f \rightarrow f\{x/\hat{x}\}.$$

The transformation to the prenex normal form ($ToPNF$) is formalized as the following rewrite system:

$$\begin{aligned} ToPNF \equiv\ & (Qx \cdot f_1) \wedge f_2 \rightarrow Qx \cdot (f_1 \wedge f_2), \\ & \neg(Qx \cdot f) \rightarrow \overline{Q}x \cdot \neg f, \end{aligned}$$

assuming f_2 does not contain free occurrences of x (otherwise they will be uniquely renamed).

The transformation of constant abstraction ($ConstAbst$) can be described in the following manner:

$$ConstAbst \equiv\ f\{|p_1 \ldots p_n|\} \rightarrow C,$$

where C is an uninterpreted constant of the same type as f.

The transformation of functional abstraction ($FuncAbst$) is performed as follows:

$$FuncAbst \equiv\ f\{|p_1 \ldots p_n|\} \rightarrow F(x_1 \ldots x_n),$$

Algorithm 1 IPL verification algorithm

```
1: procedure VERIFY($f$, $\mathsf{M}$)
2:     $f \leftarrow ToPNF(f)$    ▷ Put the formula into the prenex normal form
3:     $f^{FA} \leftarrow FuncAbst(\hat{f})$    ▷ Replace model instances with functional abstractions
4:     $f^{CA} \leftarrow ConstAbst(\hat{f})$    ▷ Replace model instances with constant abstractions
5:     $\hat{f}^{FA} \leftarrow RemQuant(f^{FA})$    ▷ Remove FA quantifiers
6:     $\hat{f}^{CA} \leftarrow RemQuant(f^{CA})$    ▷ Remove CA quantifiers
7:     $sv \leftarrow$ all $\mu$ s.t. $\exists I \cdot I, \mu \models \hat{f}^{FA} \not\Leftrightarrow \hat{f}^{CA}$ ▷ Saturation: find all variable values that satisfy non-matching abstractions
8:     $I^F_{sv}(F_i(\mu)) \leftarrow [\![\text{MDLINST}_i]\!]_{\mathsf{M},\mu}$ for each $\mu \in sv$ ▷ Model checking: run model instances to interpret functional abstractions on the above values
9:     if $\exists I \cdot I^F_{sv} \subseteq I \wedge I \models \neg f^{FA}$ then return $\bot$  ▷ If the FA formula's negation is satisfiable given the constructed interpretation, return false
10:    else return $\top$    ▷ Otherwise, return true
```

where f is a formula and F is an uninterpreted function of the same type as f. Its parameters $x_1 \ldots x_n$ are all free variables that occur in formula f and terms $p_1 \ldots p_n$.

The verification procedure for IPL formulas is presented in Algorithm 1. The first step is equivalently transforming f to its prenex normal form (PNF, i.e., all quantifiers occur at the beginning of the formula), denoted $ToPNF(f)$. The next step is to replace occurrences of instance terms MDLINST_i (interpretation of which is yet unknown to views/SMT) with two kinds of abstractions:

1. *Functional abstraction (FA).* FA replaces MDLINST_i with uninterpreted functions F_i. The arguments of these functions are the free variables that are present in the syntactic subtree of MDLINST_i. (Below, $\boldsymbol{x} \equiv x_1 \ldots x_n$.)

$$f^{FA} \equiv FuncAbst(f) = Q_1 x_1 : \mathrm{D}_1 \ldots Q_n x_n : \mathrm{D}_n \cdot \hat{f}(\boldsymbol{x}, F_1(\boldsymbol{x}) \ldots F_m(\boldsymbol{x})).$$

2. *Constant abstraction (CA).* CA replaces MDLINST_i with uninterpreted constants.

$$f^{CA} \equiv C(f) = Q_1 x_1 : \mathrm{D}_1 \ldots Q_n x_n : \mathrm{D}_n \cdot \hat{f}(\boldsymbol{x}, C_1 \ldots C_m).$$

Next, we remove all quantifiers ($RemQuant(f^{FA}) = \hat{f}^{FA}$, $RemQuant(f^{CA}) = \hat{f}^{CA}$), replacing all bound quantified variables with free ones.

$$f^{FA} \equiv Q_1 x_1 : \mathrm{D}_1 \ldots Q_n x_n : \mathrm{D}_n \cdot \hat{f}^{FA}(\boldsymbol{x}), f^{CA} \equiv Q_1 x_1 : \mathrm{D}_1 \ldots Q_n x_n : \mathrm{D}_n \cdot \hat{f}^{CA}(\boldsymbol{x}).$$

We look for interpretations (I^F_{sv}) of model instances that affect validity of f. I^F_{sv} are characterized by valuations μ of free variables that are arguments for F_i. These interpretations are also subsumed by I^F— a full interpretation of F_i on all possible variable assignments that coincides with semantic evaluation of model atoms: $I^F(F_i(\mu)) = [\![\text{MDLINST}_i]\!]_{\mathsf{M},\mu}$ for any $\mu \in \mathrm{D}_1 \times \ldots \mathrm{D}_n, i \in [1, m]$.

Instead of constructing full I^F (which requires exhaustive model checking), we determine I^F_{sv} by looking for μ that make the values of FA and CA differ. In other words, these are such valuations that it is possible to interpret the two abstractions so that one formula is valid and the other is invalid. That is, the algorithm constructs a set sv that contains all μ satisfying the *search formula* for f: $\exists I \cdot I, \mu \models \hat{f}^{FA} \not\Leftrightarrow \hat{f}^{CA}$.

In the process of *saturation*, the algorithm enumerates all such μ by iteratively finding and blocking them. With a finite number of μ, it will terminate once the sv is saturated. To terminate, it is sufficient that each D_i is finite, but not necessary: a constrained formula may have finite sv with infinite D_i.

Once variable assignments sv are determined, the algorithm constructs I^F_{sv} (a subset of I^F) by directly executing behavioral checking of $\textsc{mdlinst}_i$ on concrete values:

$$I^F_{sv}(F_i)(\mu) = [\![\textsc{mdlinst}_i]\!]_{\mathsf{M},\mu} \text{ for all } \mu \in sv \text{ and all } i \in [1, m]. \tag{5.2}$$

Finally, the algorithm performs a validity check by checking satisfiability of the negation of f^{FA}. f is valid iff the check fails to find an interpretation that agrees with I^F_{sv} and satisfies $\neg f^{FA}$.

Now, I illustrate the application of the IPL verification algorithm to Equation (5.1). The formula is already in its PNF. The next step is to remove quantifiers and replace t_1, t_2, and t_3 with their free counterparts t^f_1, t^f_2, t^f_3:

$$\begin{aligned}
&t_1.type = t_3.type = \text{STR} \wedge t_2.type = \text{ROT} \wedge \\
&t_1.end = t_2.start = t_3.start \ \wedge t_1.start \neq t_3.end \wedge \Sigma^3_{i=1} t_i.energy \leq \overline{maxBat} \rightarrow \\
&\quad P_{max=?}[(\underline{loc} = t_1.start \mathsf{U} (\underline{loc} = t_2.start \mathsf{U} \underline{loc} = t_3.end)) \wedge (\mathsf{F} \underline{loc} = t_2.start)] \\
&\quad\ \{|\underline{initloc} = t_1.start, \underline{goal} = t_3.end, \underline{initbat} = \Sigma^3_{i=1} t_i.energy + \overline{err\_cons}|\} = 1.
\end{aligned} \tag{5.3}$$

Then $\textsc{mdlinst}$ is abstracted away twice, once with a real-valued function $f(t^f_1, t^f_2, t^f_3)$ and once with a real constant CA:

$$\begin{aligned}
&t_1.type = t_3.type = \text{STR} \wedge t_2.type = \text{ROT} \wedge \\
&t_1.end = t_2.start = t_3.start \ \wedge t_1.start \neq t_3.end \wedge \Sigma^3_{i=1} t_i.energy \leq \overline{maxBat} \rightarrow \\
&\quad \text{FA}(t_1, t_2, t_3) = 1;
\end{aligned} \tag{5.4}$$

$$\begin{aligned}
&t_1.type = t_3.type = \text{STR} \wedge t_2.type = \text{ROT} \wedge \\
&t_1.end = t_2.start = t_3.start \ \wedge t_1.start \neq t_3.end \wedge \Sigma^3_{i=1} t_i.energy \leq \overline{maxBat} \rightarrow \\
&\quad \text{CA} = 1.
\end{aligned} \tag{5.5}$$

Following the abstraction, the saturation process would determine all tuples of free variable values that satisfy the following search formula:

$$\begin{aligned}
&t_1.type = t_3.type = \text{STR} \wedge t_2.type = \text{ROT} \wedge \\
&t_1.end = t_2.start = t_3.start \ \wedge t_1.start \neq t_3.end \wedge \Sigma^3_{i=1} t_i.energy \leq \overline{maxBat} \rightarrow \\
&\quad \text{FA}(t_1, t_2, t_3) = 1 \\
&\not\Leftrightarrow
\end{aligned} \tag{5.6}$$

$$t_1.type = t_3.type = \text{STR} \land t_2.type = \text{ROT} \land$$
$$t_1.end = t_2.start = t_3.start \ \land t_1.start \neq t_3.end \land \Sigma_{i=1}^{3} t_i.energy \leq \overline{maxBat} \rightarrow$$
$$\text{CA} = 1.$$

For each tuple (t_1, t_2, t_3) satisfying Eq. 5.6, M_{plan} (initialized according to MDLINST, with the values derived from the tuple) is queried for interpretations of PATHPROP:

$$P_{max=?}[(\underline{loc} = t_1.start \mathsf{U}(\underline{loc} = t_2.start \mathsf{U} \underline{loc} = t_3.end)) \land (\mathsf{F} \underline{loc} = t_2.start)].$$

Finally, the obtained interpretations will be conjoined with the negation of Equation (5.4) for the ultimate satisfiability check:

$$\neg\ (\forall t_1, t_2, t_3 : \overline{Tasks} \cdot t_1.type = t_3.type = \text{STR} \land t_2.type = \text{ROT} \land \qquad (5.7)$$
$$t_1.end = t_2.start = t_3.start \ \land t_1.start \neq t_3.end \land \Sigma_{i=1}^{3} t_i.energy \leq \overline{maxBat} \rightarrow$$
$$\text{FA}(t_1, t_2, t_3) = 1).$$

Receiving UNSAT from Eq. 5.7 would mean that the original formula is valid, while receiving SAT would mean that it is invalid.

The soundness proof for the algorithm is found in Subsection 8.1.1.

5.7 IPL Implementation

The IPL implementation is based on the Xtext language framework (`https://www.eclipse.org/Xtext`) in the Eclipse IDE. The sources of this IPL implementation have been archived [213] and are also available online (`https://github.com/bisc/IPL`). Xtext automatically generates an IPL parser, an object model of the constructs, and other supporting software infrastructure from the IPL grammar file. This infrastructure supports view-SMT translators and verifiers for IPL statements. Development of IPL specifications is done in a modified version of Eclipse bundled with the IPL implementation.

To make the grammar described in Section 5.4 unambiguous for parsing, I “flattened out” rules regarding logical and background operators. As a result, the implemented syntax is more permissive that the abstract one. However, to preserve the restrictions of the abstract grammar, we implemented typechecking to detect violations. For instance, even though the implemented grammar allows stacking multiple behavioral models, typechecking flags such cases as errors.

As a basis for architectural views, we use the Architecture Analysis and Description Language (AADL, version 2.1) [71]. AADL is an increasingly popular modeling tool for embedded system, featuring an SAE standard designation and multiple extensions. It also fits the assumptions on views stated in the end of Section 5.3: the views are statically defined and can have custom properties with fixed values. IPL’s implementation relies on OSATE2 (version 2.3.0) [69] — an open-source IDE for AADL. Based on Xtext as well, OSATE2 provides a capability of parsing and instantiating AADL models, which serve as views.

I also implemented an AADL-to-SMT translator to convert from architectural to formal views, as per their respective Definitions 1 and 2. The SMT solvers are interfaced done through the SMT-LIB v2.6 [17] syntax to abstract away from specific implementations. The backend solvers are Z3 (version 4.5.0) [55] and CVC4 (version 4.1.5) [16].

Chapter 6

Part II: Structural and Behavioral Integration Abstractions

This chapter describes the second part of the approach to modeling method integration — *integration abstractions*. These abstractions are simplified representations of models used for integration. The motivation for integration abstractions is the need of integration tools, such as IPL (Chapter 5) and analysis contracts(Chapter 7) to interact with heterogeneous models by uniform yet customizable means. Thus, integration abstractions serve as an "interface" through which IPL and analysis contracts access information in models.

Integration abstractions are crucial in integration arguments: the claims about models' consistency or dependencies of analyses depend on what proxies are used by an integration mechanism in place of models. The final part of the integration argument is formulated at the levels of IPL (in Part I) and analysis execution (in Part III respectively). The integration abstractions are responsible for the initial premises in these arguments regarding how accurately the abstractions represent the models.

I present two types of integration abstractions: *structural* and *behavioral*. The structural abstractions are based on architectural views that reflect the static structures of a model. The behavioral abstractions rely on specifying and checking behavioral properties of models in appropriate property languages. This chapter defines these abstractions, explains how they are constructed, and how their accuracy (or, *conformance* to models) fits into the model integration argument (presented in Section 4.4).

The scope of integration abstractions is illustrated in Figure 6.1: several models represent the system under design, and to establish consistency of models, one or several integration abstractions are used. The argument for model consistency relies on characteristics of abstractions and their relationship to models (for instance, *completeness* in terms of representing all relevant entities in a model). The same characteristics are used in the argument about correct analysis execution (to be presented in Part III, Chapter 7).

Due to variation among CPS models, it is infeasible to fit a single type of integration abstractions for all formalisms and integration scenarios. Several factors determine an appropriate integration abstraction:

- The aspect or quality of concern in an integration scenario. The aspects/qualities may include timing, energy, convergence, approximation error, refinement, or other dimensions

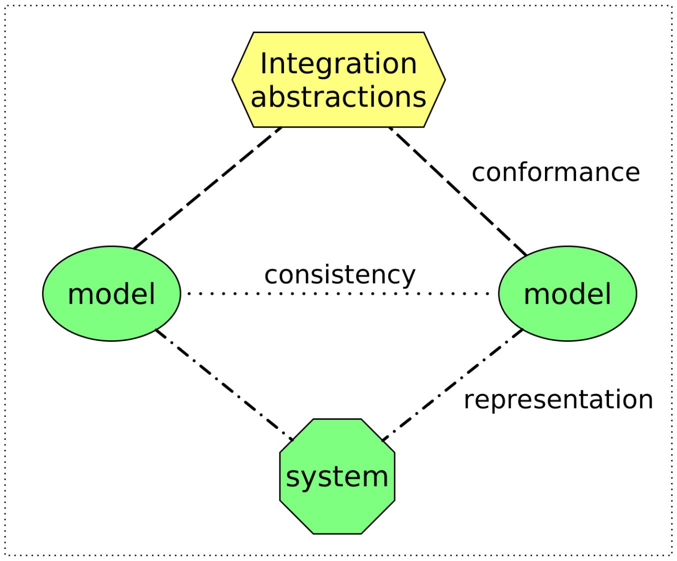


Figure 6.1: The role of integration abstractions.

of modeling.

- The formalism of models. Differences in formalisms may affect the choice of abstraction because the same information may be encoded in different ways, thus affecting the convenient ways of representing that information. For instance, infinite behavior traces are easier to represent with modal properties than with component-and-connector models.
- Integration capabilities. These capabilities are determined by the characteristics of the integration mechanism (compositional or relational, declarative or procedural, etc.). An abstraction needs to provide appropriate object to be manipulated by the integration. For example, verification typically manipulates models with non-determinism, and abstractions for verification should represent allow for non-determinism.

The above factors mean that the abstraction needs to be chosen and specialized for a particular integration scenario, which indicates a specific integration aspect, a set of formalisms, and the integration mechanism[1]. This approach is taken for every system in the validation chapter (Chapter 8): every scenario leads to customized abstractions and their evaluation.

This chapter is organized as follows. To illustrate the proposed integration abstractions, the next section introduces an example integration scenario. After, the two integration abstractions are described in separate sections, each containing its concept definitions, application examples, and important characteristics for integration arguments. In the end, I highlight the practical advantages and disadvantages of these two abstractions.

[1]This thesis uses two integration mechanisms: verification of declarative logic-based properties, described in Chapter 5, and contract-based analysis execution, described in Chapter 7.

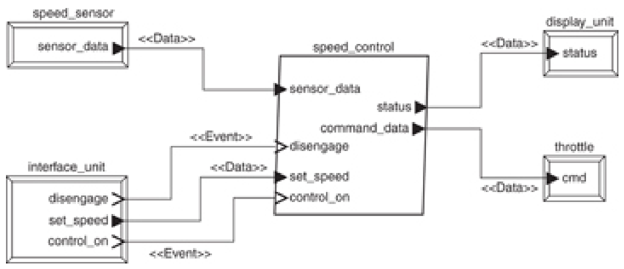


Figure 6.2: An AADL hardware model for a speed control subsystem.

6.1 Running Example: Hybrid Program and Hardware Model

This running example is inspired by the robotic collision avoidance scenario (System 2), more details on which can be found in Section 3.2. Consider that an autonomous mobile robot is represented using two models:

- A *hardware model* that captures the hardware elements of a system in a declarative language, such as Verilog or AADL (Architecture Analysis and Design Language) [73]. The model includes sensors, actuators, processing units, batteries, and connections between them — electrical and wireless.
- A *hybrid program* [189] (HP) that describes interleavings of discrete transitions and continuous evolutions of a system's state (encoded in variables). [2]

Suppose these two models were created for a given system and need to be integrated — i.e., related to each other without contradictions. For example, one could guarantee the hybrid program relies only on the sensors described in the hardware model. Another example is showing that the hybrid program is safe given the sensing error bounds in the hardware model.

Hardware models in AADL represent a system as a set of components with ports (data, physical, ...) that interact with each other through connections. The components have types that describe the nature of the components. In the case of hardware models, the component types include various sensors, actuators, mechanical devices, and electronic devices on which controllers are executed. Often AADL models are used for analysis of system qualities such as reliability and performance. Another use of AADL models is to generate source code structures for implementation.

An example of a hardware model for a speed control subsystem is given in Figure 6.2. The model shows a speed control component that gets data from a speed sensor and an interface with the higher-level planning. The speed controller outputs the speed commands going to the throttle actuator and data to show on a display. This model captures the types of data exchanged between components.

Hybrid programs are built using operators in Tab. 6.1. The flow of programs is controlled by the sequential composition (;), non-deterministic choice ($\cup$), and non-deterministic repetition (*). The semantics of a HP is formally defined over the state represented by its variables. The state changes via value assignments and continuous evolutions. Continuous evolutions advance

[2]To abstract away irrelevant details (e.g., the exact behavior of robot surroundings or the exact timing of events), HPs employ non-determinism in variable values and control transitions.

variable values along differential equations within an evolution domain F, continuing for an arbitrary amount or stop immediately. A test operator cuts off execution branches; it is commonly used in conjunction with non-deterministic assignment: $x := *; ?x > 0$ cuts off all non-positive values of variable x.

Statement	Informal meaning
$\alpha; \beta$	Sequential composition; first executes α and then β
$\alpha \cup \beta$	Non-deterministic choice; executes either α or β
α^*	Non-deterministic repetition; executes α 0 or more times
$x := \theta$	Assignment of value θ to variable x
$x := *$	Assignment of an arbitrary value to variable x
$x'_1 = \theta_1, \ \ldots$ $x'_n = \theta_n \ \& \ F$	Continuous evolution of x_i along differential equations $x'_i = \theta_i$ restricted to an evolution domain specified by formula F
$?F$	Test if formula F holds; proceed if yes, otherwise abort.

Table 6.1: Syntactic constructs of hybrid programs.

Suppose that a robot has position x, velocity v, and acceleration a tries to reach its goal g in a one-dimensional space (along a line). The robot can arbitrarily choose an acceleration ($a := *$) between full throttle ($a \leq A$, where A is the maximum possible acceleration) and full braking ($-b \leq a$, where b is the maximum possible braking power), but cannot drive backwards ($v \geq 0$). The robot's control alternates with physical dynamics of kinematic movement ($v' = a, x' = v$) in a non-deterministic loop. The following hybrid program models possible motion of the robot:

$$\alpha_{robot} \equiv (a := *; ? - b \leq a \leq A; \{v' = a, x' = v, v \geq 0\})^*. \tag{6.1}$$

The *differential dynamic logic* ($\mathsf{d}\mathcal{L}$) [189] is a logic for expressing properties of hybrid programs. Given a hybrid program α, one can write logical assertions with a $\mathsf{d}\mathcal{L}$ formula ϕ:

$$\phi ::== \theta_1 \sim \theta_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \forall x \phi \mid [\alpha]\phi \mid \langle\alpha\rangle\phi, \tag{6.2}$$

where θ_1 and θ_2 are linear real arithmetic expressions and $\sim \ \in \{<, \leq, =, \geq, >\}$. Other operators like $\wedge$ and $\rightarrow$ are derived from the operators in (6.2). The meaning of $[\alpha]\phi$ is that property ϕ holds for every possible execution of α. $\langle\alpha\rangle\phi$ means that there is at least one execution of α that satisfies ϕ.

Simple $\mathsf{d}\mathcal{L}$ formulas often take a form of $\varphi \rightarrow [\alpha]\phi$ or $\varphi \rightarrow \langle\alpha\rangle\phi$. For example, the following formula expresses that if a robot hasn't yet reached its goal ($x < g$), there exists an execution (expressed with the $\langle\rangle$ modality) where the robot (modelled as in Equation (6.1)) reaches its goal ($x \geq g$):

$$x < g \rightarrow \langle\alpha_{robot}\rangle(x \geq g). \tag{6.3}$$

This goal is achieved in two steps. The first integration goal would be to relate hybrid programs and AADL hardware models. The second integration goal is to allow mutual constraints of the models and analyze the properties of these constraints, such as soundness of their satisfaction checking.

These two steps are difficult to perform directly because of the models' different structure: AADL hardware models represent a system as an assembly of components, whereas hybrid programs represent a system as a sequence of operators. Furthermore, hardware models do not have an explicit behavioral interpretation, whereas the meaning of hybrid programs is defined directly through behaviors.

To bridge the gap between the two types of models, I introduce integration abstractions. In the next section, I present views as a integration abstraction, define their integration characteristics (conformance, soundness, and completeness), and use views to represent HPs.

6.2 Structural Integration Abstraction: Views

First, I introduce important concepts for architectural views. Then I formalize model-view conformance, describe its supporting activities, and relate it to the integration argument laid out in Section 4.4.

In this thesis, a *view* is a integration abstraction derived from the customizable formalism of *architectural views*. Architectural views have been historically used in software engineering to represent a software system from multiple perspectives, with each view corresponding to a certain viewpoint [46, 133, 157]. In model integration for CPS, architectural views have been extended to represent (sometimes implicit) architectural structures encoded in the models, or the models' assumptions about these structures [23]. In this sense, views are *structural* integration abstraction. An example of a view for a controller model can capture what inputs a controller receives, from what other components, and what properties of these inputs (timing, precision, etc.) are expected.

Prior work[3] has shown that architectural views have three integration capabilities:

1. Architectural views can represent static high-level structures in a broad range of CPS models [24].
2. Given a complete architectural model of the system ("the base architecture"), architectural views can detect inconsistencies in these structures of several models, thus detecting bugs in the system under design [23].
3. Architectural views can check constraints over values of parameters across several models [200].

Thus, architectural views can serve as integration abstractions for integration based on structural consistency. However, the prior application of views is not directly compatible the IPL-based integration, which combines structural and behavioral elements of models. In particular, IPL does not rely on a base architecture — a system's comprehensive architecture, which necessary for structural consistency checking from prior work [24]. Instead, consistency and completeness of views are defined in relation to model elements. Moreover, IPL requires a logical definition of views, to enable custom logical properties, extending the pre-conceived property of structural consistency. Below I reformulate the fundamentals of views for their use in IPL integration.

[3]For a more detailed discussion, see Chapter 9.

6.2.1 Concepts: Views and Viewpoints

As described in the background chapter (Chapter 2), models are representations of the system under design that are amenable to analysis. Engineers follow domain-specific processes to create models of the system that are valuable for domain-specific analyses. A complete set of models at some point in the system's development is denoted as $\mathbb{M}$. Below I focus on the contents of views, defining the views from the architectural perspective, which corresponds to Definition 1 in Section 5.3 for IPL. The formal definition of views (Definition 2, as a 3-tuple of a signature of symbols, a structure of architectural elements, and an interpretation mapping the symbols to the structure) also applies, but it focuses on the distinction between specification/verification, and is therefore less convenient for this chapter.

Definition 10 (View)**.** An *architectural view* V is a tuple $(\mathbb{E}, \mathbb{T}, \mathsf{T}, \mathbb{P})$, where $\mathbb{E}$ is a set of architectural elements, $\mathbb{T}$ is a set of architectural types, $\mathsf{T} : \mathbb{E} \to \{\mathbb{T}\}$ is a typing function that maps every architectural element to potentially several types, and $\mathbb{P}$ is a set of property functions $p : \mathbb{E} \to \mathbb{O}$ that map architectural elements to a value from any given set of values ($\mathbb{O}$).

First, consider views as-is, separately from models. A system is characterized by a set of views ($\mathbb{V}$). Unlike behavioral models, views focus on static (i.e., behavior-free) and potentially higher-level aspects of the system. The internal organization of views, displayed in Figure 6.3, follows the classic works of software architecture [3, 85, 216]. A view contains architectural elements that include components ($\mathbb{C}$, e.g., a controller or a sensor), connectors ($\mathbb{O}$, e.g., visual sensing), ports/roles ($Ports$and $Roles$, i.e., interfaces through which components and connectors, respectively, are attached).

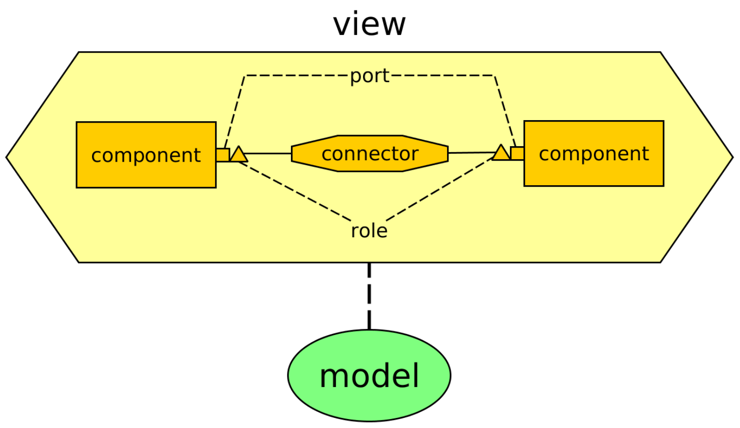


Figure 6.3: Elements of architectural views.

Each architectural element has a finite number of types, with four primitive types for $\mathbb{C}$, $\mathbb{O}$, $Ports$, and $Roles$. Organized in hierarchies, architectural types are used by system engineers to distinguish elements of various nature. For instance, cyber types (e.g., a software thread) represent digitally interacting entities, while physical types (e.g., a motor) represent objects present to physical interactions. Views use standard type safety rules of architectural models: each element has at least one type, and the four aforementioned element types are mutually exclusive. Thus, these four subsets form a complete partitioning of $\mathbb{E}$:

$$\mathbb{C} \cup \mathbb{O} \cup Ports \cup Roles \equiv \mathbb{E},$$
$$\mathbb{C} \cap \mathbb{O} \cap Ports \cap Roles = \emptyset.$$

In this thesis, architectural elements use the standard containment hierarchy: the topmost element is called the "system" element of an architectural view, and the rest of the elements are the system's sub-elements at various depths. Each level (including the topmost) contains components and connectors, which may in turn contain sub-elements, which can also be components or connectors. Components contain ports, while connectors contain roles, and roles attach to ports.

Properties ($\mathbb{P}$) are annotations of architectural elements. Each property function PROP maps an architectural elements to a value of some type (either a primitive type like integers or an architectural type from $\mathbb{T}$): $\mathsf{P} : \mathbb{E} \rightarrow \mathbb{O}$. Syntactically, I denote the value of a property PROP of an element ELEMas ELEM.PROP. Properties are often declared for architectural types, thus propagating the property declaration to all instances of that type; the property values, however, may differ between elements. For example, in an embedded system's hardware view, if the CPU component type specifies the "frequency" property, then all CPU component instances should have the property, but may have different values of it.

Now I turn to using views as abstractions of models, introducing the concept of an integration viewpoint. This concept is inspired by the ISO standard 42010 [112] for architectural description, as well as the seminal works on viewpoints in software architecture and cyber-physical systems [29, 46, 74, 102]. This 42010 standard uses viewpoints as descriptions of each view's perspective. In model integration, a viewpoint defines the specific concern of integration that needs to be carried out. For example, a timing viewpoint would expose timing-related information (such as delays, deadlines, execution times) from each model. Multiple viewpoints are necessary for realistic systems. In this chapter I treat viewpoints separately, without considering their interactions.

Definition 11 (Viewpoint)**.** An *integration viewpoint* (VP) is a function that represents a method of deriving a view from a given model: $\mathsf{VP} : \mathbb{M} \rightarrow \mathbb{V}$. A viewpoint defines vocabulary for the view (i.e., the types $\mathbb{T}$ and property signatures $\mathbb{P}$ that constitute its signature) and a set of algorithms/processes to derive the view elements from the model. Thus, a view is the result of applying a viewpoint applied to a model: $\mathsf{VP}\ (\mathsf{M}) = \mathsf{V}$.

An integration viewpoint characterizes which model elements and properties/topology need to be extracted for integration. An viewpoint can be instantiated as an algorithm of mapping a model to an appropriate view. For instance, it can be implemented in code, with a model as an input. This definition restricts viewpoints to be unambiguous: when applied to a given model, viewpoint produces only one view. This assumption does not limit the generality of the approach: in cases when a model requires several views, these views can be represented with different viewpoints.

From the model's perspective, a viewpoint operates on certain parts or objects in a model, which I call *model elements* ($\mathcal{E}_{\mathsf{M}}$). These elements vary widely depending on the formalism: they can be blocks, modules, statements, or certain values in the model. Scoping the relevant model elements is also a responsibility of a viewpoint. Once the scope/types of the model elements are defined in a given viewpoint, creation of a view is a transformation of model elements into architectural elements annotated with types/properties. Creation of views can be automated, whereas creation of a viewpoint is a manual engineering task.

6.2.2 Conformance of Views to Models

Conformance of views to models is defined based on a viewpoint.

Definition 12 (Model-View Conformance)**.** A view V *conforms* to a model M with respect to some viewpoint VP, written as $(\mathsf{V},\mathsf{M}) \in R^V_M$, iff $\mathsf{V} = \mathsf{VP}(\mathsf{M})$. The relation $R^V_M : \mathbb{V}\times \models$ is a *conformance relation*.

A view conforming to a model means that it is an appropriate view for a given integration context. To rely on model-view conformance for soundness of checking IPL specifications, a view needs to capture appropriate model objects. However, the definition so far does not reflect which model objects ($\mathcal{E}_\mathsf{M}$) are captured by the view. To distinguish different types of views based on how well they represent model objects, I introduce two characteristics of views: *soundness* and *completeness*.

Intuitively, a view is *sound* if every element in the view is "representing" some model element or combination thereof. For instance, every CPU in a sound view maps to an actual CPU in a hardware model. To express this notion without making assumptions on the relation between $\mathbb{E}$ and $\mathcal{E}_\mathsf{M}$, I require that any subset of a view (supposedly with fewer "non-fake" $\mathbb{E}$) conforms to some strict subset of the model (thus leading to a non-zero reduction of model elements, which would then map to the removed view elements). Since not all view subsets are valid in terms of the architectural language and the domain (e.g., a sensing connector cannot be attached to only one actor in the HP view), I restrict such sub-views to those that conform to at least one model (relative to VP). Definition 13 formalizes this intuition.

Definition 13 (View Soundness)**.** A view V (which contains $\mathbb{E}$) is a *sound* view of model M (which contains elements $\mathcal{E}_\mathsf{M}$) with respect to viewpoint VP if for any valid sub-view V' of V (i.e., containing $\mathbb{E}' \subset \mathbb{E}$), there exists a sub-model M' of M that the sub-view conforms to. Formally:

$$\begin{aligned}\mathrm{sound}(\mathsf{V},\mathsf{M},\mathsf{VP}) \equiv\ & \forall \mathsf{V}' \cdot \mathsf{V}' \subset \mathsf{V} \rightarrow (\exists \mathsf{M}_1 \cdot \mathsf{VP}(\mathsf{M}_1) = \mathsf{V}') \rightarrow \\ & (\exists \mathsf{M}_2 \cdot \mathsf{VP}(\mathsf{M}_2) = \mathsf{V}' \wedge \mathsf{M}_2 \subset \mathsf{M}).\end{aligned}$$

Informally, a view is complete if every element in the model is "represented" by some architectural element. Thus, adding more relevant model elements to the model bigger would lead to a strict increase in the elements of the view. For instance, a view for a hardware model is complete if it intends to capture all the CPUs from it and successfully does so. Definition 14 formalizes this intuition.

Definition 14 (View Completeness)**.** A view V is a *complete* view of model M with respect to viewpoint VP if for no super-view of V is a sound view of M with respect to VP. Formally:

$$\mathrm{complete}(\mathsf{V},\mathsf{M},\mathsf{VP}) \equiv\ \nexists \mathsf{V}' \cdot \mathsf{V}' \supset \mathsf{V} \wedge \mathsf{V}' = \mathsf{VP}(\mathsf{M}) \wedge \mathrm{sound}(\mathsf{V}',\mathsf{M},\mathsf{VP}).$$

The concepts of view soundness and completeness have been studied in prior work [23], albeit in a different way. Bhave defined soundness and completeness with respect to the *base architecture* — the complete model of the system's architecture that contains elements from all views. The advantage of using base architecture is that views can be completely agnostic of models they represent. On the other hand, creating a full and up-to-date base architecture may be burdensome or even unrealistic. My work, however, does not rely on having the base architecture and defines view soundness/completeness in terms of model elements (without adding any assumptions on what the elements might be).

Soundness and completeness of a view define the link between models and views in the approach of this thesis (see Figure 1.1) and the integration argument (Section 4.4). If a model is precisely and comprehensively represented by a view, than any mechanism operating on a view is equivalent to the same mechanism operating on the model. This is consideration holds assuming that the viewpoint focuses on the model elements that are targeted by the integration mechanism. For instance, if all energy-consuming devices are represented in views, and each view element that consumes energy corresponds to some energy-consuming device in the system, then a property of consistent energy consumption between different models (to which these views conform) can be reliably verified with IPL. The discussion of the integration argument for views is continued later in this chapter, in Subsection 6.2.5.

Definition 15 extends the definitions of soundness and completeness to viewpoints.

Definition 15 (Viewpoint Soundness/Completeness)**.** A viewpoint VP is *sound* (*complete*) for a given class of models if for any model in this class, its view produced by VP is sound (complete).

Soundness and completeness of viewpoints is an ideal condition for integration: having a sound and complete viewpoint a priori discharges the obligations of views in terms of the integration argument. When a viewpoint is fully formalized and automated, these properties can be demonstrated. However, in most cases in practice (e.g., in studies described in Section 8.2), view creation is at least partially manual, which makes it impossible to guarantee any properties of a viewpoint, and soundness/completeness of views needs to be evaluated case-by-case.

The concepts of a view and a viewpoint are summarized in an entity-relationship diagram in Figure 6.4. In short, the integration viewpoint is determined by the integration approach, the formalism, and the system (in particular, its context and requirements). A viewpoint is used to create a view, which is related to a model, and the view is required to conform to the model whenever it's used.

Generally, defining and maintaining conformance between views and models is challenging for three reasons:

- *Diverse concepts and decomposition hierarchies:* models differ in how they conceptualize the system and its operation. For instance, an assignment in regular programs may be related to an evolution of a differential block in hybrid programs. Models also differ in how they are decomposed into smaller units. For instance, models may be decomposed based on components (e.g., architectural models), operations (e.g., hybrid programs), and logical clauses (e.g., a $\mathsf{d}\mathcal{L}$ formula over hybrid programs). It may difficult to reconcile such diverse concepts and hierarchies with component-based structures in views.
- *Distributed information:* one model may syntactically consolidate certain kinds of data, whereas other models may disperse that data across many locations in the model's description. For instance, sometimes HPs interleave actions from multiple components of a system, such as a robot and its environment. If done manually, it is a tedious and error-prone task to consolidate such dispersed information in views.
- *Continuous change:* models undergo changes throughout the engineering process. For example, one may refine a hybrid program from event-triggered to time-triggered control (the latter mimics the system more closely). Some of these changes would require views to

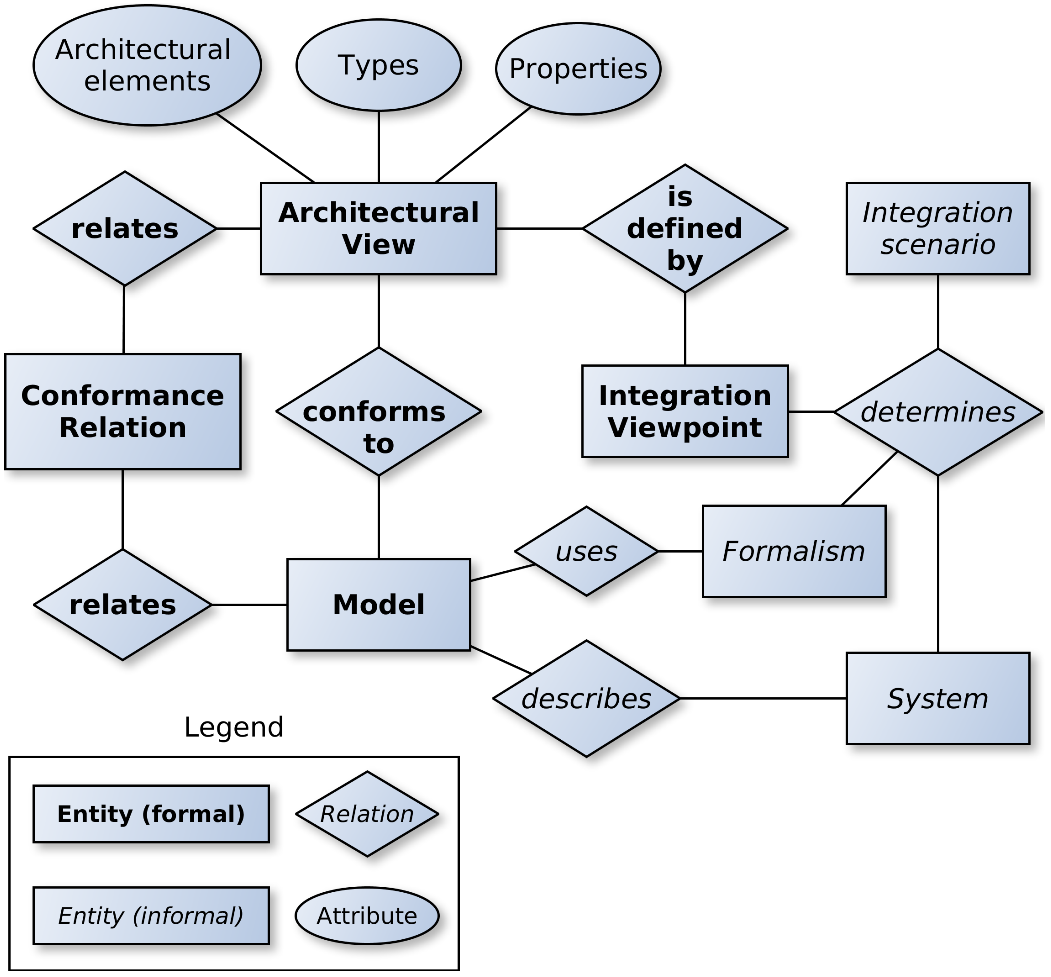


Figure 6.4: View abstraction concepts. Bold is formalized, italicized is informal.

change as well, so that conformance relation is maintained.

However, three special cases of viewpoints require less effort to implement than the general case. The first case is when a model is already written in an architecture description language, containing the information from Definition 10 and, hence, being a (trivially) sound and complete view of itself. For instance, a hardware model in AADL can be considered a hardware view. In such cases the extra information may be redacted from the view to make it simpler to process in the subsequent steps.

The second special case is when, for every element type in $\mathbb{T}$, a viewpoint implements an independent incremental transformation using a model and a partially constructed view. For instance, one function can determine a set of components in a hybrid program. Then another function, using the HP and the set of components, determines their ports. Finally, yet another function takes the HP and ports and returns the connections between ports, according to the HP. In this case, soundness can be guaranteed by construction, whereas completeness may depend on the mapping from architectural elements to sets of model elements, which may be non-local in the model syntax.

The third special case is when a model has discrete elements that can be directly mapped to elements of views. This case usually occurs with block diagrams and component-based models, where view elements are directly derived from certain blocks/components and connections between them. For instance, prior work approached Simulink and Verilog models this way [23]. In this case, a viewpoint explicitly defines a relation between elements of a view and a model, and this relation enables direct validation of view soundness and completeness.

Next, I illustrate view creation on the example of hybrid programs. Then I discuss the strategies for view automation, and conclude the view section with the implications for the integration argument.

6.2.3 View Abstractions for Hybrid Programs

To illustrate using views as abstractions, I created a viewpoint VP^{HP} for hybrid programs. The main idea behind this viewpoint is to capture the primary agents and their interactions, along with relevant properties (such as a control algorithm and a set of physical equations). To illustrate the mapping between hybrid programs and views, we highlight three central concepts: *HP actors*, *HP connectors*, and *HP composers*.

Definition 16. *Hybrid program actor HPA* is a component instance that is characterized by a tuple:

$$HPA \equiv (q, Ports, Ctrl, Phys).$$

The state of an actor is a set of variables and constraints: $q \equiv (\mathbb{V}, Constr)$, where $\mathbb{V} \equiv \{v_i\}$ is set of a typed variables[4], and $Constr \equiv \{\varphi_i\}$ is a set of state constraint formulas, defined by Equation (6.2), over variables in $\mathbb{V}$. For example, for a robot that moves in a one dimension (along a line), $q \equiv (\{x, v, a, o\}, \{o \in \{-1, 1\}\})$.

A port of is an external interface of an actor – a variable v that regulates interaction between actors: $Ports \equiv \{v_i\}$. For example, if a robot is sensing an obstacle's x coordinate, we denote this as a port variable p_{x_o}, which is separate from the obstacle's variable x_o. Unless a state variable is exposed through a port, it is considered hidden from other actors. We do not require that $\mathbb{V} \cap Ports = \emptyset$: a port p may expose a state variable ($p \in \mathbb{V}$) or define its own ($p \notin \mathbb{V}$).

An actor's control is a hybrid program: $Ctrl \equiv \alpha$, defined as a sequence of operators from Tab. 6.1 over variables in $\mathbb{V}$ and $Ports$. $Ctrl$ describes computations executed by the actor. An actor's physics is a set of differential equations with an evolution domain constraint: $Phys \equiv \{x_i' = \theta_i \& F\}$ over variables in $\mathbb{V}$ and $Ports$. The goal of separating physics from control is that the former is often shared among many model variants, while the latter may be more specific the variant's combination of concerns.

Definition 17. Given $\{HPA_i\}$, ***HP connector** HPC* is a connector instance that is characterized by a tuple:

$$(Roles, RTP, \mathsf{Trf}).$$

Roles $Roles \equiv \{r\}$ distinguish different responsibilities of ports attached to a HP connector, such as a sender or a receiver. A mapping between roles and ports RTP associates each role with a port on an actor: $RTP \equiv Roles \rightarrow \bigcup HPA_i.Ports$. The transformation function Trf captures

[4]HPs natively support only $\mathbb{R}$, so we encode $\mathbb{Z}$ and $\mathbb{B}$ as reals with constraints in $Constr$.

the connector's effect on the attached actors so that the connector can be reused in multiple model variants with different actors. Unlike $Roles$ and RTP that define *what a connector is*, Trf defines *how it works*. Formally, Trf maps a set of actors and their attachments to a set of transformed actors:

$$\mathsf{Trf} : \{HPA\} \times Roles \times RTP \to \{HPA\}\,.$$

Consider a simple immediate precise sensing connector (IPSC) that senses the precise value of a variable and returns the result immediately. It has two roles: $Roles = \{Sense, Sensed\}$. Let actor a_1 use its port p_1 to sense the value of p_2 from actor a_2 through an IPSC. The IPSC transformation derives $\dot{a_1}$ from a_1 and $\dot{a_2}$ from a_2[5]. IPSC replaces the readings of variable p_1 in a_1 with readings of p_2 in $a_1.Ctrl$:

$$\begin{aligned} IPSC.\mathsf{Trf}(\{a_1, a_2\}, Roles, RTP) &\equiv \{\dot{a_1}, \dot{a_2}\} \text{ s.t.} \\ \dot{a}_1.q = a_1.q, \qquad & \dot{a}_2.q = a_2.q, \\ \dot{a}_1.Ports = a_1.Ports \setminus \{p_1\}, & \dot{a}_2.Ports = a_2.Ports \setminus \{p_2\}, \\ \dot{a}_1.Ctrl = a_1.Ctrl\,[p_1/p_2], & \dot{a}_2.Ctrl = a_2.Ctrl, \\ \dot{a}_1.Phys = a_1.Phys, \qquad & \dot{a}_2.Phys = a_2.Phys. \end{aligned} \tag{6.4}$$

To enable automated generation of HPs, we need to bridge the gap between HP actors and hybrid programs. To this end, we will compose the actors until there is a single mega-actor, which then generates a HP. There are, however, several ways to compose actors. Therefore, we encapsulate a mechanism of composition in a *composer* (similar to component glue [18], director [146], and coordinator [31] in related work):

Definition 18. A *composer* CPR is a pair (Compose, ToHP), where Compose is a function that maps several actors into one actor: $\{HPA\} \to HPA$, and ToHP is a function that maps HPA to a hybrid program.

A composer implements a method of creating aggregate actors with Compose until the system is represented by a single actor, which is then converted into a hybrid program using ToHP. In the general case Compose can be arbitrarily complicated and is out of this paper's scope. Instead, we focus on the *sequential composer* $SeqC$ that is implicitly used throughout all collision avoidance models in [172]. This composer orders actors' executions in a given sequence:

$$SeqC.\mathsf{Compose}(a_1, \ldots, a_n : HPA) \equiv \dot{a} \text{ s.t.}$$

$$\begin{aligned} \dot{a}.q &= a_1.q \cup \cdots \cup a_n.q, \\ \dot{a}.Ports &= a_1.Ports \cup \cdots \cup a_n.Ports, \\ \dot{a}.Ctrl &= a_1.Ctrl; \ldots; a_n.Ctrl, \\ \dot{a}.Phys &= \{a_1.Phys, \ldots, a_n.Phys\}. \end{aligned} \tag{6.5}$$

To create a HP from $a : HPA$, $SeqC$ sequentially composes control with physics and puts them into a non-deterministic loop:

$$SeqC.\mathsf{ToHP}(a) \equiv (a.Ctrl; a.Phys)^* \tag{6.6}$$

Putting HP actors, connectors, and composers together, we obtain a hybrid program view.

[5]We use $\alpha[a/b]$ to mean substitution of a for b in HP α.

Definition 19. A *HP view* V^{HP} is a tuple $(\{HPA\}, \{HPC\}, CPR)$.

Using the above concepts, conformance of an HP view to a hybrid programs mean that a view's actors and connectors, when composed by the view's CPR, synthesize a program that is equivalent to the given one. The viewpoint (VP^{HP}) is implemented as an algorithm that takes an HP view and synthesizes a program from the view's elements. To obtain a hybrid program α from V^{HP}, the first step is to transforms components using connectors with the TC procedure. Next, HPC compose the actors to form a single HPA and generate a HP. In other words:

Definition 20. An HP view V^{HP} *conforms* to a hybrid program α (i.e., $\mathsf{V}^{HP} = \mathsf{VP}(\alpha)$), if

$$CPR.\mathsf{ToHP}(CPR.\mathsf{Compose}(\mathsf{TC}(\{HPA\}, \{HPC\}))) = \alpha.$$

6.2.4 Automating Model-View Conformance

Creating or updating views manually is time-consuming and error-prone. This circumstance can be mitigated by automating view-related engineering processes, of which I consider two:

- *Automated bootstrapping:* creating a new view that conforms to a given model, or creating a new (potentially partial) model given a view. This bootstrapping would reduce the initial effort of model integration. For example, one could synthesize a hybrid program template that is consistent with a given hardware model.
- *Automated co-evolution:* updating a view after a model change, or updating a model after a view change. In both cases the goal is to repair conformance of the view to the model. Co-evolution would reduce the maintenance effort of keeping the models integrated. For example, whenever a new sensor is added to a hardware model, an appropriate state variable could be added to the hybrid program.

Bootstrapping

First, a integration viewpoint VP needs to be established. A viewpoint requires a mapping between the concepts/elements of a model (e.g., a state variable or operator) and the concepts/elements of a view (e.g., a component or connector). This mapping is specific to the viewpoint, allowing different viewpoints to utilize the same parts of the same model differently.

When implementing viewpoints as algorithms, two options[6] are possible: (i) the algorithm receives a view and outputs a model (or a template), and (ii) the algorithm receives a model and outputs a conforming view. Below we discuss these options and exemplify Option (i) with views for hybrid programs. Examples that illustrate Option (ii) is reviewed in Chapter 8.

Consider hybrid program $CollAvoid$ (Equation (6.7)) below that specifies a unidimensional collision avoidance problem for a robot. It states that, assuming that a robot and an obstacle are far enough apart initially, they will not ever crash, assuming that their control algorithms ($\mathsf{RobotCtrl}$ and $\mathsf{ObstCtrl}$) and their continuous physics ($\mathsf{RobotPhys}$ and $\mathsf{ObstPhys}$) execute in an indefinite loop. During robot's turn, its controller (Equation (6.8)) chooses between braking or non-deterministic acceleration. The acceleration is chosen only if the distance to the obstacle is greater than the worst-case braking distance. This hybrid program is difficult to map to view elements

[6]In some cases, a combination of both options may be possible.

because it contains complex hybrid dynamics and a logical statement over these dynamics, and is not explicitly separated into components or connectors.

$$\begin{aligned} &\mathit{CollAvoid} \equiv \\ &|x_r - x_o| > \Delta \rightarrow [\ (\mathsf{RobotCtrl}; \mathsf{ObstCtrl}; \\ &\qquad (\mathsf{RobotPhys}, \mathsf{ObstPhys})\ )*\ ](x_r \neq x_o), \end{aligned} \tag{6.7}$$

$$\begin{aligned} \text{where } \mathsf{RobotCtrl} \equiv (&a := -b \\ &\cup\ (?|x_r - x_o| > v^2/b;\ a := *)); \end{aligned} \tag{6.8}$$

A view conforming to the above model is shown in Figure 6.5. It factors the control and physics of $\mathit{CollAvoid}$ as two HP actors: a robot and an obstacle. The robot uses the sensed value of the obstacle position x_s instead of directly using the obstacle position x_o. The components are connected with the "immediate precise sensing connector" (IPSC) that replaces the sensing variable (x_s) with the sensed variable (x_o) when the view is transformed into $\mathit{CollAvoid}$. Thus, the view represents hybrid program (without inherent component structure) in terms of the vocabulary of HP actors (components) and sensing between them (connectors).

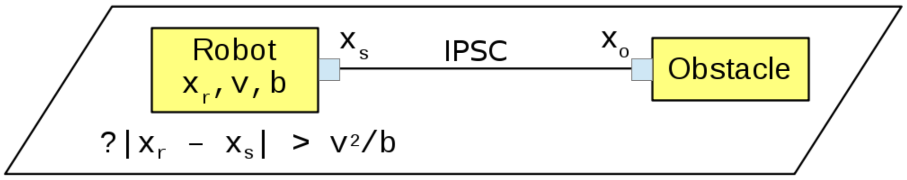


Figure 6.5: A view for the $\mathit{CollAvoid}$ program.

An appropriate model (Equation (6.7)) is automatically synthesized from this view by directly using the definitions of HP view concepts. Thus, conformance between the view and its generated model is established by construction. As a result, HP views become first-class artifacts that engineers create and change.

Notice that in the case of HP views all information required to generate a model is contained in the view. However, it may not always be the case: the view could be a coarse abstraction and, while conforming to the model, miss some details required for synthesizing a complete model. In such cases, the approach should be to generate a template of a model, which will be completed manually.

Generally, the process of synthesizing a model from a view can be described by the steps below. This process is not guaranteed to be fully automated or result in a complete model, so manual effort may be required to complete the model. Thus, Option (i) follows these steps:

1. Create an empty model in the formalism required by the viewpoint.
2. Instantiate all model objects determined by the view's components. For instance, in Simulink, this step may include instantiating the controller and plant subsystems.
3. Instantiate all model objects/code determined by the view's connectors. This may require altering the existing model objects, depending on the formalism.
4. Set the direct properties in the model according to the view.

5. Verify that the emergent properties of the model match their description in the view. If not, then the model generation process failed and will need manual error correction or updating the view/model as described in the next subsection.

Option (ii) of implementing viewpoints (i.e., generating a conforming view given a model) keeps the role of models as first-class artifacts, creating views from models as needed. Generally, the process of generation follows the following steps (illustrated with power models in Chapter 8):

1. Determine the types of view elements that would best represent the model (for instance, by using the cyber, physical, and cyber-physical type hierarchies from prior work [23]). This step includes determination of properties required by the elements.
2. Determine the instances of components (with ports) that populate the view based on the model elements ($\mathcal{E}_\mathsf{M}$).
3. Determine connector instances between components (as well as specific roles, if necessary). The connectors may encode relations between model objects (e.g., reachability) or information exchange mechanisms in a model (e.g., shared variables).
4. For each property of each view element, determine what its value should be. If no value can be inferred from the model, it will not be specified in the view. Some properties take values directly from model objects (e.g., a time step for simulation), whereas others may emerge indirectly (e.g., energy required for a certain move may need to be computed as a function of several model objects).

Co-evolution

The first scenario of co-evolution is as follows: a model M changes (becomes M'), and a previously conforming view V needs to be updated to a view V' that conforms to M'. Assume that the viewpoint VP is fixed and implemented.

I envision two strategies to construct V'. The first one is generating a fresh view $\mathsf{V}' = \mathsf{VP}(\mathsf{M}')$ by applying the viewpoint algorithm, thus abandoning V and starting from scratch. This approach is appropriate when the view generation is fully automatic and does not require substantial computational resources or time.

The second strategy applies when view generation is computationally intensive or not fully automatic (e.g., may require subjective judgment of an expert). The strategy is to reuse parts of V to simplify the generation of V', and it proceeds in two steps. The first step is to identify and localize the model elements that *differ* between M and M'. For instance, in an HP for a robot and an obstacle, if only the robot's control changes in M', the physical dynamics and the obstacle's control can remain the same in V' as in V. The second step is to generate elements of V' that correspond to only updated elements of M'. In the third and final step, these new view elements are merged with the unchanged elements of V, thus forming V'. To be automated, this approach requires a formalized mapping between model and view elements (not available in all cases); otherwise, the second step has to be executed manually.

The second scenario of co-evolution is when a view V changes (becomes V'), and the previous M needs to be updated to M' such that $\mathsf{VP}(\mathsf{M}') = \mathsf{V}'$. This is a more challenging scenario because V' may lack the information to create a complete model. This scenario can either be addressed with bootstrapping a new model (template) with a given view (see Subsection 6.2.4), or with an approach similar to the aforementioned two-step strategy of reusing view elements. The first step is to identify the *identical* elements of V and V'. The second step is to generate a template for M' from V', filling in the syntax from M objects that correspond to the identical elements of V and V'. This approach is not guaranteed to produce a fully-specified M', which would only happen when the change between V and V' involves a rearrangement of view elements.

Changes of the viewpoint do not fall under the co-evolution process. Instead, one needs to follow the bootstrapping process, as described in Subsection 6.2.4.

6.2.5 Integration Argument with Views

Now I examine views as means of integration/consistency reasoning. From the perspective of IPL, a view is an abstraction of a model that enables its quantification over its elements. The outputs of IPL verification are therefore dependent on how accurately and completely views represent underlying models. Here, I assume that the other links of the integration chain work as intended. That is, the integration property accurately reflects the desired consistency relation, IPL checking is sound, and other types of abstractions (i.e., behavioral properties) are sound and always terminate (see Subsection 6.3.3 for a similar discussion about behavioral properties).

The integration argument is concerned with three characteristics of views:

- *Expressiveness:* ability to encode static elements of various models. This ability is determined by two factors: the language of views and the customized types used in a particular view. The language is drawn from the family of architecture description languages (e.g., Acme or AADL), whereas the types are typically specific to a particular VP. For instance, the actor subtypes *robot* and *obstacle* are only used for the set of models concerned with obstacle avoidance for mobile robots.
- *Soundness:* according to Definition 13, sound views do not contain elements that do not correspond to any model elements.
- *Completeness:* according to Definition 14, complete views do not fail to represent any model elements.

The expressiveness of views is largely determined by ADLs. Generally, ADLs are capable of representing finite collections of hierarchical annotated objects [162].The annotations are named finite values of common types (like integers or strings) or lists/sets of values. The types determine convenient distinctions between classes of view elements, which partly enable mapping to $\mathcal{E}_{\mathsf{M}}$. However, types do not substantially contribute to expressiveness because they can be implemented through properties. The expressiveness of views is intentionally limited to support the efficient search/saturation procedure of IPL, as well as dependency resolution for analyses (presented later in Section 7.3). The rest of model integration is delegated to behavioral reasoning over individual models.

View soundness and completeness are necessary for correct reasoning about quantified IPL formulas. For a universally quantified formula, an unsound view with "fake" elements may lead to an incorrect counterexample of a valid integration property (hence, a false negative). An incomplete view in this case may lead to falsely accepting an invalid property as valid (hence, a false positive). For an existentially quantified formula, an unsound view may provide a "fake" element to satisfy an invalid property (hence, a false positive). An incomplete view may miss the element to satisfy the valid property (hence, a false negative).

The challenge of creating expressive, sound, and complete views lies in the complexity of the mapping between view and model elements. With simple mappings, like between a hardware model and its view, an automated algorithm straightforwardly generates a view. However, for more complex mappings, it is important to define the viewpoint as precisely as possible to not generate unsound elements or miss any model elements. For example, a hybrid program can be split up into actors in different ways, some of which do not account for some of the discrete instructions and continuous dynamics, leading to an incomplete view.

6.3 Behavioral Integration Abstraction: Properties

Here I discuss the second integration abstraction — *behavioral properties*, which rely on model-specific property languages to describe constraints or queries over these models. The intent is to take advantage of the out-of-the-box reasoning engines for these languages (e.g., the PRISM model checker [134] for Markov chains and decision processes), instead of developing new engines from scratch for the purposes of model integration. The goal of this section is to explain how these languages are used as integration abstractions in the context of IPL.

Behavioral properties are used as an independent abstraction, separate from views. Typically, in a given integration scenario only one abstraction is used for each model. However, there is no fundamental obstacle to using both abstractions of the same model side-by-side, or even layering them (as briefly illustrated in Definition 23), although developing approaches to such composition is outside of the scope of this thesis.

6.3.1 Concepts

In the definitions below, I build on the definitions (of a model and related concepts) from the background chapter (Chapter 2) and the IPL preliminaries (Section 5.3).

Definition 21 (Behavioral Language/Property)**.** An *behavioral language* $\mathcal{L}$ for a given model M is a set of sentences that are interpretable by M to a value ($\mathbb{B}$ or otherwise). A *behavioral property* P is a sentence in $\mathcal{L}$.

In the running example, $\mathsf{d}\mathcal{L}$ is an example of $\mathcal{L}$, and Equation (6.3) is an instance of P. For the hardware model, we can consider logical constraints on the hardware architecture a kind of $\mathcal{L}$.

Definition 22 (Behavioral Query)**.** An *behavioral query* Q is a function that maps a model $\mathsf{M} \in \mathbb{M}$ and a sentence from its behavioral language $\mathcal{L}$ to a value; i.e., $Q : \mathbb{M} \times \mathcal{L} \to \mathbb{V}$, where $\mathbb{V}$ is some domain of values (booleans, integers, reals, ...).

In cases when the query value is boolean, behavioral queries represent verification on models, such as model checking and theorem proving. For example, determining whether a robot gets to a goal given its program in Equation (6.3) is an example of a query: $Q(\alpha_{robot},$ Equation (6.3)$) \equiv \alpha_{robot} \models x < g \to \langle \alpha_{robot} \rangle (x \geq g)$. If the above holds, $\top$ is returned (otherwise, $\bot$).

In cases when the query value is a non-boolean, the query represents a computational request to the model. For example, computing the total mass of all elements in the system can be done by the hardware model: $Q(\mathsf{M}_{hw}, "mass(*)") \equiv [\![mass(*)]\!]_{\mathsf{M}_{hw}}$. This expression returns a real number provided by the hardware model after a black-box computation.

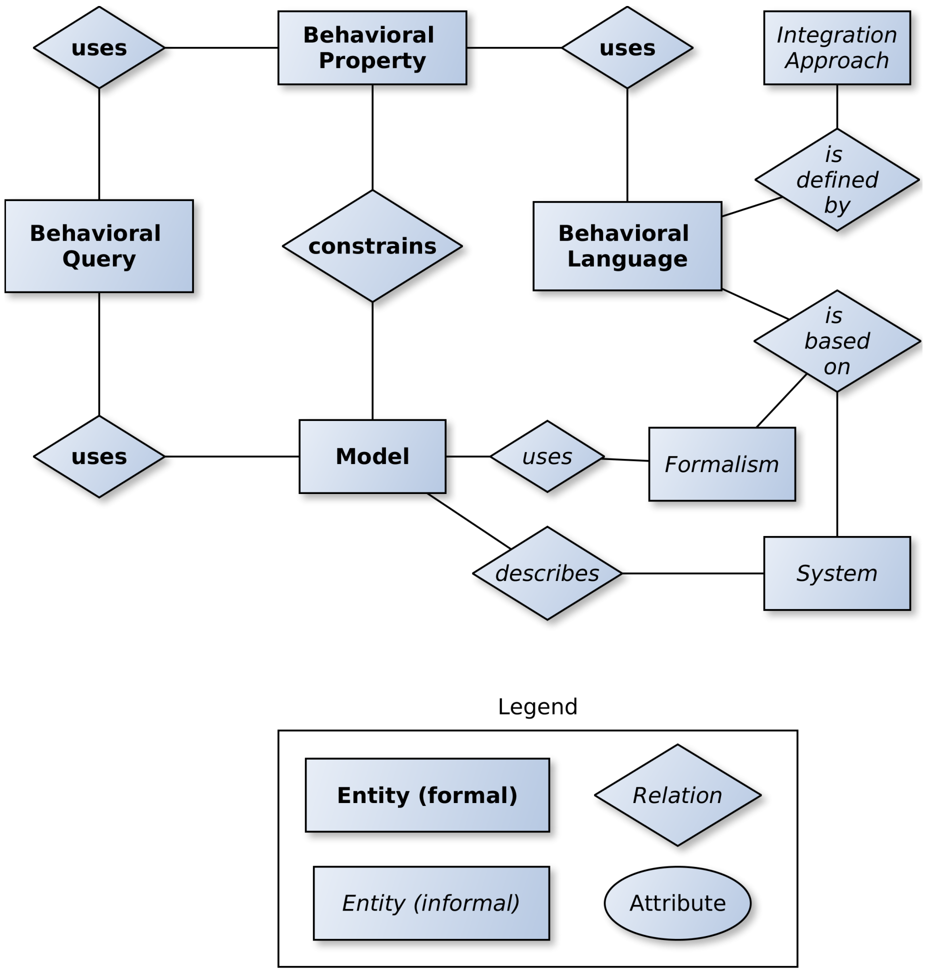


Figure 6.6: Behavioral property abstraction concepts. Bold is formalized, italicized is informal.

The concepts from this section are shown in an entity-relationship diagram in Figure 6.6.

6.3.2 Behavioral Property Abstractions for Hybrid Programs

The differential dynamic logic ($\mathsf{d}\mathcal{L}$) formulas serve as a property language for hybrid programs. Each such formula is either valid or invalid, enabling validity queries with a boolean output. However, to raise the abstraction level, we can define a behavioral property language over views of hybrid programs. One could embed a logical specification in an HP view, but such views would be limited to simple logical formulas. Since $\mathsf{d}\mathcal{L}$ formulas may incorporate several hybrid programs, it is convenient to have a property language separate from the internal details of HP views.

The goal is, thus, to build a logical specification layer on top of HP views, making it possible to use several HPs in a formula. To this end, I define a $\mathsf{d}\mathcal{L}$ *view formula* the following way:

Definition 23. An $\mathsf{d}\mathcal{L}$ *view formula* over HP views $\mathsf{V}_1^{HP}, \dots, \mathsf{V}_n^{HP}$ is a $\mathsf{d}\mathcal{L}$ formula over variables from these views $\mathbb{V}_1 \cup \cdots \cup \mathbb{V}_n$, parametric terms $Constr_1, \dots, Constr_n$, and hybrid programs $\mathsf{V}_1^{HP}, \dots, \mathsf{V}_n^{HP}$.

Given several HP views, one can express a property in a formula that combines these views, their state variables ($\mathbb{V}_i$), and state constraints $(Constr_i)$. To translate a $\mathsf{d}\mathcal{L}$ view formula to a plain $\mathsf{d}\mathcal{L}$ formula, one needs to replace $Constr_i$ with $\mathsf{Compose}(\mathsf{TC}(\mathsf{V}_i^{HP})).q.Constr$ and replace HP view references V_i^{HP} with their synthesized hybrid program code according to Definition 20. The view formulas are equivalent to $\mathsf{d}\mathcal{L}$ in expressiveness, but provide a more abstract way to specify behavioral properties. Just like $\mathsf{d}\mathcal{L}$, queries over this language have a binary output ("valid" and "not valid").

Unlike HPs, hardware models in our running example do not come with a "native" language for properties. However, we can envision an OCL-style [] propositional language with references to individual components (such as cpu_1), their properties (such as $frequency$) and real number operators (such as $<$). Just like in the property languages discussed above, the query outputs are binary in this case. An example property in this language is $cpu_1.frequency \geq cpu_2.frequency$.

6.3.3 Integration Argument with Model Querying

Now I focus on the link between the model and the behavioral property. The behavioral abstraction is responsible for receiving queries in $\mathcal{L}$ and returning values as results. The goal of this link is to support conclusions of IPL via queries that connect the model and the property language. Here, we assume that the other links of the integration chain work as intended. That is, the integration property accurately reflects the desired consistency relation, IPL checking is sound, and other types of abstractions (i.e., views) are sound and complete (see Subsection 6.2.5 for a similar discussion about views).

For integration, we are concerned with three characteristics of queries:

- *Expressiveness:* ability to express properties that are important in a particular integration scenario. This ability is determined by the behavioral language $\mathcal{L}$. For instance, querying temporal relations between a robot's state variables may or may not be possible, depending on $\mathcal{L}$. One way to see expressiveness is by an ability of a query to evaluate to two different values for two similar properties.
- *Soundness:* whether the query value matches the evaluation of the property in the semantics of the model. This is a condition for the computation that performs the query. Formally, $Q(\mathsf{M}, P) = [\![P]\!]_{\mathsf{M}}$. For instance, if $Q(\mathsf{M}_{hw}, "mass(*)")$ returns a real number not matching the total mass, it would be an unsound query. Soundness of queries is an instance the general conformance concept from Figure 6.1: queries in a behavioral language "conform" to the model if they return the correct answers for it.
- *Termination:* whether a query returns a value in a finite amount of time. This is also a condition for the computation that performs the query. If a query does not terminate or returns "unknown," it fails this property and therefore does not serve as a useful integration abstraction.

The expressiveness condition is context-specific and can be assessed in practice. For the rest of this chapter we assume that $\mathcal{L}$ is expressive enough for the purposes of interfacing with M. In other words, we assume that all modal IPL subformulas can be translated into $\mathcal{L}$, and the values in the expressions can be transferred to the domain of $\mathcal{L}$. When this assumption is true, it is guaranteed that behavioral queries will be well-formed and express the intent of the IPL property.

It holds for the proposed IPL plugins for LTL and PCTL (see Chapter 5). Unlike expressiveness, soundness and termination can often (but not always) be evaluated theoretically and a priori. For example, bounded model checking always terminates, but cannot guarantee soundness (with respect to the unbounded semantics) [25].

I consider the above characteristics for two cases: (i) a given individual query and (ii) a set of queries enabled by a given behavioral language. In the first case, for a given integration property in IPL, if every individual query is sound and terminates, then we can conclude that the verification of the integration property is sound. In the second case, if any queries in $\mathcal{L}$ for M is known to terminate and be sound, then IPL verification will be sound for any integration property written over $\mathcal{L}$ (assuming other parts of the reasoning chain are sound).

The three characteristics are challenging to achieve at once because they are often conflicting. A highly expressive language is difficult to create an efficient decision procedure for, if any procedure at all. Sound reasoning requires eliminating the possibility of a false answer, which may lead to an infinite computation. On the other hand, guaranteed finite computations may have to make approximations and accept a possibility of an error, and hence unsoundness.

In the running example, query soundness for hybrid programs is supported by a correct theorem proving: if a proof of a theorem (e.g., that a robot reaches the goal Equation (6.3)) exists, it is always possible to check that it is a correct proof and the theorem holds. Conversely, a counterexample guarantees that the theorem does not hold, hence the query is sound. However, there is no guarantee that a proof would be found, so automated theorem proving does not satisfy the termination condition.

To summarize, this section demonstrated that behavioral properties and queries provide a black-box interface to a model's behaviors, without placing any assumptions on the model's syntactic structure. Three integration-relevant conditions of queries are expressiveness, soundness, and termination. For the running example, the differential dynamic logic is an expressive and sound behavioral abstraction for hybrid programs, although termination of queries is not guaranteed. The hardware model permits abstraction by customized queries and predicate logic formulas, which can be implemented to be sound and terminable.

6.3.4 Shared Background between Abstractions

One of the assumptions in IPL is that the view interpretations (I^{V}) and model interpretations (I_q^{M}) do not contradict over the shared symbols, thus forming a single consistent background interpretation (I^B). This condition is necessary for successful domain transfer — exchange of values values between views and behavioral properties as part of the IPL verification (Subsection 5.5.1). This is the only requirement that mutually limits views and behavioral properties. It needs to be satisfied through construction of the integration abstractions, since a priory assurance of shared interpretation without require further assumptions about the contents of the views and models.

Typically, the exchanged values are sorts like integers or reals, and the domain transfer happens trivially. In some cases, however, the exchanged values are identifiers/references to model/view objects (e.g., software threads or robot tasks). In such cases, the interpretation of references in a view needs to be coordinated with the interpretation in a behavioral property. Even though the references may be represented as integers, their meaning is the components. An example of such a case can be found in the case study of System 3 (Subsections 8.1.3 and 8.2.3): thread IDs are exchanged between the view (which represents threads as components) and the model (which represents threads as Promela processes) by the means of an LTL property.

6.4 Comparison of Integration Abstractions

Now that both integration abstractions have been presented, Table 6.2 summarizes how the views and behavioral properties were used this chapter's running example (Section 6.1).

Model	**View abstraction**	**Behavioral property abstraction**
Hybrid program (Table 6.1)	Hybrid program view (Definition 19)	$\mathsf{d}\mathcal{L}$ formulas (Equation (6.2)), $\mathsf{d}\mathcal{L}$ view formulas (Definition 23)
Hardware model (Section 6.1)	Hardware architecture view (Definition 10)	OCL-like constraint formulas

Table 6.2: Integration abstractions for two models of the running example.

When faced with integration scenarios in practice, engineers need to choice between views and behavioral properties as abstractions. Due to the design of IPL, any abstractions can be combined, so the choice of an abstraction is local to each model (as opposed to choosing one kind of abstraction for all models). Another aspect of this choice is that views can be used for specification of dependencies between analyses, as explained in the next chapter.

Below I provide several rules of thumb for making this choice, based on the research experience described in this thesis. Although these rules of thumb are subjective and not universal, they can help identify the impact of potential choices.

Views is a *convenient abstraction* in the following circumstances:

- The model is written in a component-based or hierarchical formalism, and the component structure or hierarchy needs to be exposed for integration.
- The model's finite and static elements need to be exposed for integration.
- A view, if used, would have few elements (dozens).
- A viewpoint implementation is available.
- The viewpoint can be applied unambiguously and automatically to generate views/models in a negligible time.
- Multiple analyses have overlapping dependencies over to one or several models.

Views are *difficult to use* in the following circumstances:

- The model is written in a formalism that is difficult to map to the component-based structure of a view.
- The model's infinite or numerous elements (e.g., all possible behaviors) need to be exposed for integration.
- A view, if used, would have many elements (hundreds and more).
- A viewpoint needs to be designed and implemented from scratch.

- The viewpoint's application is a manual, ambiguous, and time-consuming process.

Behavioral properties are a *convenient abstraction* in the following circumstances:

- The model's numerous (hundreds and more) or infinite homogeneous elements need to be exposed for integration.
- The hierarchical structure, dependencies, and multiple properties of the model's elements are not important for integration.
- The model's elements are difficult to expose as-is (without querying).
- The model comes with a property language and a reasoning engine for sentences in this language.
- The queries are fully automated, sound, and are guaranteed to terminate.

Behavioral properties are a *difficult to use* in the following circumstances:

- The model has few elements (dozens) that need to be exposed for integration.
- The hierarchical structure, dependencies, and multiple properties of the model's elements are important for integration.
- The model's elements are easy to expose as-is (without querying).
- The model does not have a native property language or a reasoning engine.
- The queries are potentially unsound or do not terminate, and may require manual effort.

To summarize, this chapter has established views and behavioral properties as integration abstractions of models. Each of the abstractions has three characteristics that affect its usefulness for integration with IPL. Should these characteristics be missed, the correctness of integration is threatened, as described above. Various practical circumstances affect the trade-offs between the two abstractions, and the provided rules of thumb are expected to help engineers choose an appropriate abstraction.

Chapter 7

Part III: Analysis Execution Platform

This chapter presents the third, and final, technical part of this thesis — the *Analysis Execution Platform*. This platform supports the third step of the approach proposed in Chapter 4 by executing analyses in a correct way. To this end, the platform restricts execution of analyses to appropriate contexts and orders this execution according to the input/output dependencies of the analyses. The central concept enabling the execution platform is an *analysis contract* — a lightweight specification that captures the essential information about the data flow and the intended context of an analysis.

Analysis execution relies on the previous two technical parts of this thesis. Integration abstractions provide a uniform representation of the system's elements, in terms of which engineers can specify dependencies between analyses. IPL is used for specifying appropriate execution contexts as assertions over multiple models. Using this specification, the IPL verification mechanism ensures that the context for each analysis is appropriate during any execution of that analysis.

This chapter presents a formalization of analysis contracts and the execution platform. As an illustration, I use the analyses for thread/battery scheduling in System 3, a quadrotor (Section 3.3). The validation of the analysis platform, including a full encoding of analysis contracts for Systems 3 and 4, can be found in Section 8.3.

7.1 Domain Signatures and Analysis Contexts

To specify contracts for analyses, one first needs to define the formal basis that captures information about analysis dependencies and execution contexts. Since analyses change models, part of the context may change after executing an analysis. It is impossible, however, to organize execution of analyses that may change anything at any time: if nothing (even at the meta-level) stays fixed, in which terms can the context or dependencies be described or operated on? It is required, then, to distinguish between the part of the context that is guaranteed to stay the same, and the part that may change.

I address the above requirement by separating *signatures* (symbols for specification) of models/views and *interpretations* (mappings to semantic structures) of the symbols in the signatures. The former determines the symbols (e.g., a component type for threads, and a function that returns the period of a thread) used in specification of analysis contracts, whereas the latter determines the values assigned to the symbols (e.g., a specific set of four threads, and concrete integer values for each thread's period). The critical distinction is that the symbols are fixed during analysis execution, whereas the values can change due to analyses. This distinction is similar to the difference between the IPL syntax (Section 5.4) and semantics (Section 5.5).

Consider a set of models ($\mathbb{M}$) and a set of views ($\mathbb{V}$) that conform to these models. Each model in $\mathbb{M}$ contains a signature (Σ^{M}), an interpretation (I_q^{M}), and a structure (Γ), as defined in Definition 3 (Section 5.3). Each view in $\mathbb{V}$ contains a signature (Σ^{V}), an interpretation (I^{V}), and a structure of architectural elements ($\mathbb{E}$), as defined in Definitions 1 and 2. The architectural definition of views (Definition 1) is used for specification and resolution of dependencies, while the formal definition of views (Definition 2) is used for specification and verification of analysis contexts.

Definition 24 (Domain Signature)**.** A *domain signature* (Σ) is a tuple of model and view signatures: $(\Sigma_1^{\mathsf{M}} \dots \Sigma_n^{\mathsf{M}}, \Sigma_1^{\mathsf{V}} \dots \Sigma_m^{\mathsf{V}})$.

Intuitively, a domain is a set of related concepts, definitions, and types (e.g., for thread scheduling) where several analyses can operate. Characterized by the signatures of related models (Σ^{M}) and views (Σ^{V}), a domain signature stays fixed while multiple analyses execute. Model signatures provide syntactic elements for specification of appropriate contexts, to capture important behavioral aspects. Thus, the context contains state variables (e.g., battery charge) and run-time functions, which have their intepretations that change at run time (and are thus determined by a modal interpretation, I_q^{M}). Run-time functions are a more complex way to encode state of a model, with each state determining a concrete function for a symbol. For instance, preemption between threads is a dynamic attribute of a system, encoded as a function $\mathsf{CanPrmpt}$. Another example of a run-time function is dynamic connectivity between battery cells (encoded as thermal neighbors function TN).

View signatures are used in both context and dependency specification. Through view signatures, a domain signature provides architectural elements, which can be referred to either by instance (a specific thread) or by type (thus referring to a set of all elements that have that type), as well as other view-related sorts. For example, threads ($\mathsf{Threads}$) and CPUs (CPUs) are types of architectural elements (components), and thread scheduling policies ($\mathsf{SchedPol}$) is a view-related sort, which is used to characterize CPUs with a property. Views also contain properties, which are represented seen as functions of architectural elements, such as thread periods ($\mathsf{Per} : \mathsf{Threads} \mapsto \mathcal{Z}$) and processor frequencies ($\mathsf{CPUFreq} : \mathsf{CPUs} \mapsto \mathcal{Z}$).

Both view and model signatures also contain standard sorts (such as Booleans $\mathcal{B}$ and integers $\mathcal{Z}$), along with their interpretation, called a background interpretation (I^B) since it is shared between models and views. Standard operators like addition ($+ : \mathcal{Z} \times \mathcal{Z} \mapsto \mathcal{Z}$) are also part of the background interpretation.

Domain signatures are a convenient representation of information that affects multiple models in a given domain. For instance, the scheduling domain can be represented with a tuple of all symbols from scheduling models and views: $(\mathsf{M}_{sch}, \mathsf{M}_{sec}, \mathsf{M}_{cpu}, \mathsf{M}_{rek}, \mathsf{V}_{sch}, \mathsf{V}_{sec}, \mathsf{V}_{cpu}, \mathsf{V}_{rek})$. This way, the dependencies and appropriate contexts of analyses can be documented from the standpoint of this fixed domain, while allowing the interpretations and structures to change due to analysis execution. When using analyses from multiple domains, the domains can be combined into a single larger domain with a union of all symbols (provided that these domains agree on the interpretation of their shared symbols).

Definition 25 (Analysis Context)**.** For a given domain signature $\Sigma = (\Sigma^{\mathsf{M}}_1 \dots \Sigma^{\mathsf{M}}_n, \Sigma^{\mathsf{V}}_1 \dots \Sigma^{\mathsf{V}}_m)$, an *analysis context* (Υ) is a tuple of the following pairs: for each view signature, Υ contains a pair of a view interpretation and a set of architectural elements; and for each model signature, Υ contains a pair of a model interpretation and a model structure:

An analysis context represents the parts of system design that can be changed by analyses in a given domain (represented by Σ). A view interpretation maps the symbols from the signature to the actual values. No view elements are interpreted dynamically/modally. For example, if a set of three threads is denoted T in V, it can be interpreted as $I^{\mathsf{V}}\mathsf{Threads} = \{t_1, t_2, t_3\}$. The periods of these threads (in milliseconds) are determined by a function Per, which can be interpreted as follows: $[\![\mathsf{Per}]\!]_{\mathsf{M}} = \{t_1 \mapsto 40, t_2 \mapsto 50, t_3 \mapsto 60\}$.

Given a runtime state q, modal interpretation give meaning to symbols in Σ^{M}: some model function $f : A_i \times \dots \times A_j \mapsto A_k$ is defined as $q(f) : I^{\mathsf{M}}_q(A_i) \times \dots \times I^{\mathsf{M}}_q(A_j) \mapsto I^{\mathsf{M}}_q(A_k)$, which is the value of the state-interpreted function on state-interpreted arguments, in state q. Each model's interpretation and structure are necessary for checking whether a context is appropriate.

Normally, the structure is defined as a set of potentially infinite traces, with each state assigning the values for symbols in the signature. For LTL, suppose Q is the set of all possible states, and Q^ω is the set of all infinite sequences of states (i.e., executions). Then the structure is $\Gamma \subseteq Q^\omega$. In other words, Γ is the set of executions of the system defined by M. The form of the structure traces varies depending on the modal formasims (e.g., in PCTL the structure would be a set of paths and an induced probability measure; see Subsection 5.5.3 for details).

To simplify further checking, the views are combined into a single view based on the mappings between them, as done in the prior work []. The view signatures, view interpretations, and sets of architectural elements are combined into a single tuple $(\Sigma^{\mathsf{V}}, I^{\mathsf{V}}, \mathbb{E})$:

$$\Sigma^{\mathsf{V}} = \Sigma^{\mathsf{V}}_1 \cup \dots \cup \Sigma^{\mathsf{V}}_n \tag{7.1}$$
$$I^{\mathsf{V}} = I^{\mathsf{V}}_1 \cup \dots \cup I^{\mathsf{V}}_n \tag{7.2}$$
$$\mathbb{E} = \mathbb{E}_1 \cup \dots \cup \mathbb{E}_n \tag{7.3}$$
$$\tag{7.4}$$

For this combined view to be internally consistent, I assume that the views do not contradict each other on their shared symbols. This obligation is discharged by the prior work. The models are, however, not combined and kept separate through the rest of this chapter. For brevity, I refer to the joint interpretation of I^{V} and I^{M}_q as $I = I^{\mathsf{V}} \cup I^{\mathsf{M}}_q$.

7.2 Analysis Contracts

This section describes how analysis contracts are specified. I start with defining an analysis. Functionally, an analysis reads an input context and produces an output context.

Definition 26 (Analysis)**.** Given a domain (Σ), an analysis ($\mathcal{A}$) is a function that maps an input context (Υ^i) of Σ to an output context (Υ^o) in the same domain: $\mathcal{A}(\Upsilon^i) = \Upsilon^o$.

A contract for $\mathcal{A}$ specifies restrictions on valid input contexts and valid output contents, as well as the parts of the context that the analysis reads and modifies.

Definition 27 (Analysis Contract)**.** An *analysis contract* (C) for an analysis ($\mathcal{A}$) in a given domain (Σ) is a tuple $(\mathbb{I}, \mathbb{O}, \mathbb{A}, \mathbb{G})$, where:

- Inputs are a subset of view symbols $\mathbb{I} \subseteq \Sigma^{\mathsf{V}}$ from Σ that $\mathcal{A}$ reads.
- Outputs are a subset of view symbols $\mathbb{O} \subseteq \Sigma^{\mathsf{V}}$ from Σ that $\mathcal{A}$ writes.
- Assumptions are IPL statements (Definition 8) over Σ symbols: $\mathbb{A} = \{\mathsf{A}\}$, where $\mathsf{A} \in$ FORMULA. These statements must be satisfied by every input context to $\mathcal{A}$.
- Guarantees are IPL statements (Definition 8) over Σ symbols: $\mathbb{G} = \{\mathsf{G}\}$, where $\mathsf{G} \in$ FORMULA. These statements must be satisfied by every output context of $\mathcal{A}$.

The purpose of inputs and outputs is to document dependencies between analyses. The inputs should contain the domain signature elements that can potentially affect the outputs of the analysis. Specifically, for each input $\mathsf{i} \in \mathbb{I}$, there should exist two such contexts Υ_1 and Υ_2 that differ only the interpretation of i (i.e., $I_1(\mathsf{i}) \neq I_2(\mathsf{i})$) and lead to different outputs of the analysis: $\mathcal{A}(\Upsilon_1) \neq \mathcal{A}(\Upsilon_2)$. Similarly, for the outputs, each output listed in a contract should vary on some of the inputs. Formally, for each output $\mathsf{o} \in \mathbb{O}$, there should exist such an context Υ^i that the output context ($\Upsilon^o = \mathcal{A}(\Upsilon^i)$ differs from Υ^i in terms of o: $I_i(\mathsf{o}) \neq I_o(\mathsf{o})$).

Definition 28 (Analysis Dependency)**.** An analysis $\mathcal{A}_1$ (with contract C_1) *is dependent* on an analysis $\mathcal{A}_2$ (with contract C_2), denoted $d(\mathcal{A}_1, \mathcal{A}_2)$, if the inputs of $\mathcal{A}_1$ overlap with the outputs of $\mathcal{A}_2$: $\mathsf{C}_1.\mathbb{I} \cap \mathsf{C}_2.\mathbb{O} \neq \emptyset$.

When multiple analyses are executed in a sequence, analysis dependencies need to be respected by the order of the analyses. Given a set of analyses $\mathcal{AN}$ with contracts, an ordering $\mathcal{O} = \langle \mathcal{A}_1 \cdots \mathcal{A}_n \rangle$ of $\mathcal{AN}$ is *correct* if each analysis in the ordering is not dependent on any of the downstream analyses:

$$\forall i \in [1, n] \cdot \forall j \in [1, i) \cdot \neg d(\mathcal{A}_j, \mathcal{A}_i) \tag{7.5}$$

Finally, the appropriateness of a context is checked by satisfaction of assumptions and guarantees. This is done on a per-case basis, for a given input context Υ^i and output context $\Upsilon^o = \mathcal{A}(\Upsilon^i)$. A contract C is satisfied when the input content satisfies the assumptions ($\Upsilon^i \models \mathsf{A}$) and the output content satisfies the guarantees ($\Upsilon^o \models \mathsf{G}$). Both of these satisfactions are meant in the IPL sense (Problem 1). This definition of contract satisfaction is used in all three use cases for analysis contracts (described in Section 4.3), allowing to specify both assumptions and model consistency post-analysis.

Definition 29 (Appropriate Context)**.** A context Υ is *appropriate* for executing an analysis $\mathcal{A}$ with a contract C if $\Upsilon \models \mathsf{A} \wedge \mathcal{A}(\Upsilon) \models \mathsf{G}$.

Notice how only a domain signature is needed to specify a contract and determine the dependencies, with no context required. This way, the contracts are independent from the changes to views/models made by analyses. For instance, an analysis that adds a new thread (e.g., a watchdog) only affects the structure/interpretation of $\mathsf{Threads}$symbol, and does not change the $\mathsf{Threads}$symbol itself in the signature (i.e., the thread component type).

7.3 Analysis Execution

The order of analysis execution is determined by the input-output dependencies between the analyses. To arrange the analyses in a correct dependency order, consider a directed graph of analyses $\gamma = (\mathcal{A}, d(,))$ for a given set of analyses $\mathcal{AN}$. The analyses from $\mathcal{AN}$ constitute the nodes of the graph, and the edges follow the dependency relation.

If γ is a cyclic graph, there is no sound ordering of $\mathcal{AN}$. Indeed, assuming that some ordering $\mathcal{O}$ exists, consider the elements of $\mathcal{O}$ that correspond to the cycle in γ. The analysis in the first node of this subsequence of $\mathcal{O}$ will violate Equation (7.5) because it will be dependent on at least one of its successors (due to these nodes forming a cycle in γ). Without cycles, γ is a Directed Acyclic Graph (DAG), and any topologically-sorted ordering of its nodes is a correct ordering of $\mathcal{AN}$.

Therefore, an ordering $\mathcal{O}$ for a set of analyses $\mathcal{AN}$ is computed by: (i) constructing γ for $\mathcal{AN}$; (ii) checking its cyclicity; (iii) if γ is cyclic, aborting; and (iv) if γ is acyclic, constructing any topological ordering of its nodes. The choice of a specific ordering from the set of possible topological orderings does not affect the correctness of this approach because topological orderings of γ differ only in relative positions of mutually independent analyses.

Now I present an analysis execution algorithm that achieves both goals set at the beginning of the chapter: it respects analysis dependencies during execution (as defined by Equation (7.5)), and executes analyses only in appropriate contexts (as defined by Definition 29). The algorithm takes a domain signature Σ, an input context Υ, a set of analyses $\mathcal{AN}$ annotated with contracts , and one analysis $\mathcal{A} \in \mathcal{AN}$ as the goal analysis. The algorithm performs a correctly-ordered execution of analyses (with their output Ω_o) — or aborts if such execution is not possible.

The analysis execution algorithm consists of the following steps:

1. Construct a dependency graph γ.
2. Determine an ordering $\mathcal{O}$ of $\mathcal{AN}$ that respects all analysis dependencies, setting the next analysis pointer to the first one. If such an ordering does not exist, abort.
3. Execute the next analysis $\mathcal{A}$ (with contract $(\mathbb{I}, \mathbb{O}, \mathbb{A}, \mathbb{G})$) in $\mathcal{O}$.
   (a) Verify that $\forall \mathsf{A} \in \mathbb{A} \cdot \Upsilon \models \mathsf{A}$ (using the IPL verification algorithm). If this statement does not hold or the verification is inconclusive, abort.
   (b) Execute $\mathcal{A}$ on Υ and update $\Upsilon = \mathcal{A}(\Upsilon)$.
   (c) Verify that $\forall \mathsf{G} \in \mathbb{G} \cdot \Upsilon \models \mathsf{G}$ (using the IPL verification algorithm). If this statement does not hold or the verification is inconclusive, abort.
4. Advance to the next analysis in $\mathcal{O}$ and repeat the previous step. If at the end of $\mathcal{O}$, output Υ as the final result.

The above algorithm ensures that all analyses execute only in an appropriate context by proceeding only if $\Upsilon \models \mathbb{A}$ and $\mathcal{A}(\Upsilon) \models \mathbb{G}$, and aborting otherwise. A correct ordering is guaranteed if γ does not have cycles, thus ensuring that downstream analyses do not overwrite results of the upstream ones.

In practice, executing a series of analyses on models may make the models inconsistent. If a sequence of analyses is aborted partway because an assumption/guarantee did not hold, the models may inconsistent. To establish and maintain model consistency with analysis execution, every execution of a sequence of analyses is treated as a transaction. That is, the initial state of the models/views is saved, and if the execution aborts, the initial state is restored. With this technique, any analysis execution can result in either no change, or the final state that satisfies the guarantees (which may include consistency properties) — but not in an intermediate inconsistent state.

7.4 Implementation

The analysis execution platform was implemented as a tool ACTIVE (Analysis Contract Integration Verifier) based on OSATE2 — an open-source environment for AADL modeling [70]. ACTIVE has been archived [211] and is also available online (`https://github.com/bisc/active`). Domain signatures are modeled with AADL component types and property sets, listing the properties that apply to each component type. Analysis contexts are encoded as AADL instance models. ACTIVE handles analysis dependencies and ordering, delegating checking of assumptions and guarantees to the IPL implementation, which is extensible with new behavioral models (see Section 5.7 for detail). Analyses are plugged into ACTIVE through a standardized interface as external tools that consume and output AADL models. Analysis contracts are specified in an AADL sub-language *annex* that captures inputs and outputs over the available types in the workspace, and assumptions and guarantees as IPL formulas over AADL views.

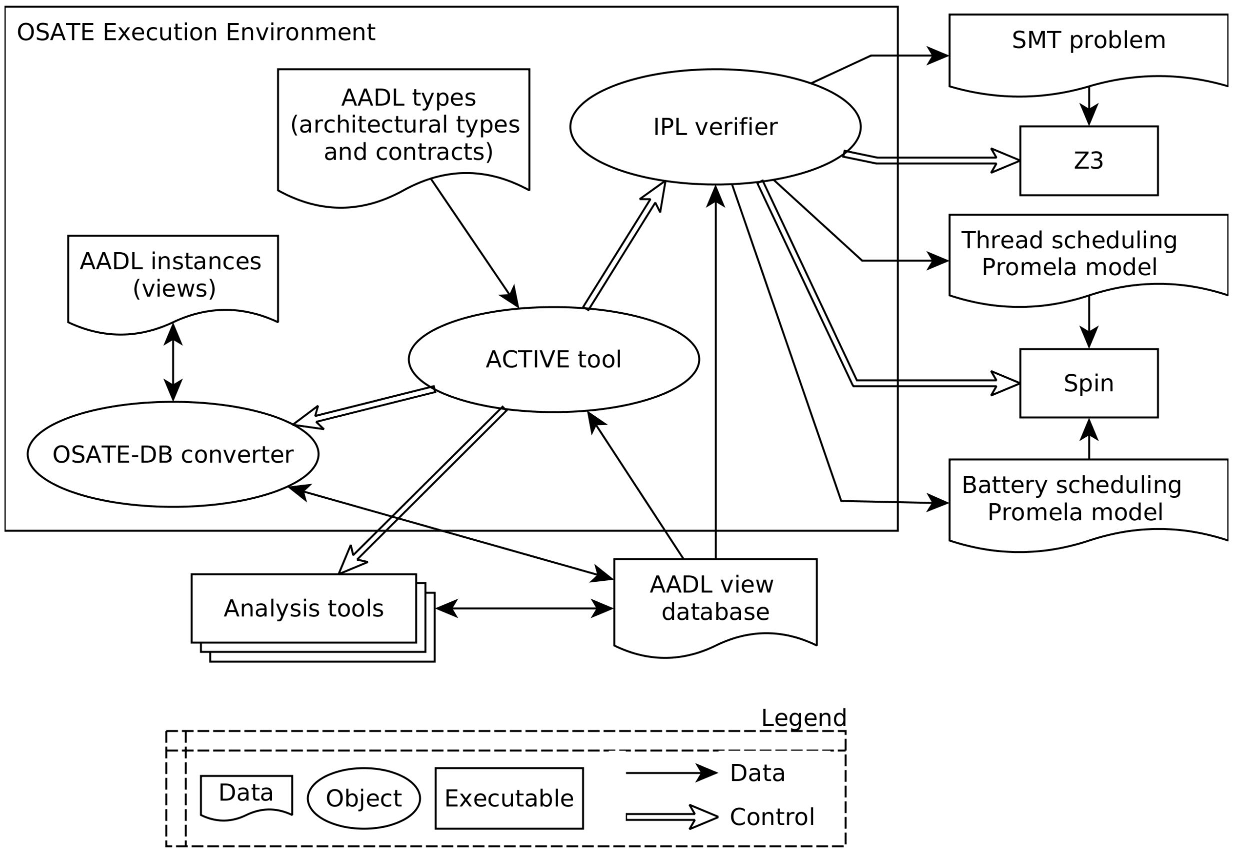


Figure 7.1: The architecture of the analysis execution platform.

Figure 7.1 depicts the architecture of the analysis execution platform. Analysis contracts C are associated with AADL component types, while Υ is represented by the AADL main system instance. Initially, the platform converts Υ from AADL into a database representation using the *OSATE-database converter*. The subsequent analysis and verification steps are performed through this database. ACTIVE constructs the analysis graph γ, as described in Section 7.3, and delegates the checking of $\mathbb{A}$ and $\mathbb{G}$ to an implementation of IPL. This implementation creates an SMT problem from the database and instantiates the behavioral models M_{sch} and M_{bsch} (for details on how IPL verification is implemented, see Section 5.7).

Chapter 8

Validation

This chapter presents the studies and evidence that support the claims formulated in Section 1.1. The studies are listed for each of the three parts of the approach, in the same order as Chapter 5 (Part I), Chapter 6 (Part II), and Chapter 7 (Part III):

1. The validation of Part I (Section 8.1) investigates the four claims related to IPL (expressiveness, soundness, applicability, and customizability). These claims are studied theoretically and practically in two validation contexts: energy-aware adaptation for a mobile robot (context 1) and thread/battery scheduling for a quadrotor (context 3).
2. The validation of Part II (Section 8.2) investigates the four claims of integration abstractions (expressiveness, soundness, applicability, and customizability). These claims are studied practically in all four validation contexts. In the first two contexts (energy-aware adaptation for a mobile robot and collision avoidance for a mobile robot), all four claims are studied. In the other two contexts (thread/battery scheduling for a quadrotor and reliable/secure sensing for an autonomous vehicle), only applicability and customizability claims are studied.
3. The validation of Part III (Section 8.3) investigates the two claims related to analysis contracts (soundness and applicability). Soundness is studied theoretically, and both claims are studied practically in two validation contexts: thread/battery scheduling for a quadrotor and reliable/secure sensing for an autonomous vehicle).

For a visual overview of the claims and how they correspond to validation studies, see Table 1.1 in Section 1.2.

8.1 Validation of Part I: Integration Property Language

8.1.1 Theoretical Evaluation

I start with an abstract, domain-independent analysis of soundness of the IPL verification algorithm (Algorithm 1). To remind the reader, the goal of the IPL algorithm is to solve the IPL validity problem (Problem 1): given an IPL formula and a structure (a set of views and models), determine whether it is valid on this structure. To avoid false positives/negatives, IPL verification should produce sound results. I prove that any answer returned by Algorithm 1 is correct with respect to the IPL semantics (independently of the behavioral plugins). To be valuable in engineering, the verification algorithm should terminate on practical problems, hence I also describe the termination conditions.

First, I show that interpretations of MDLINST over sv determine validity of an IPL formula. Correctness and termination follow directly from this result in Corollary 2.

Theorem 1. Absence of flexible interpretations that agree with I^F_{sv} and satisfy $\neg f^{FA}$ is necessary and sufficient for validity of f^{FA} on I^F:

$$\nexists I \cdot I^F_{sv} \subseteq I \wedge I \models \neg f^{FA} \text{ iff } I^F \models f^{FA}.$$

Proof. Soundness follows from straightforward instantiation. Equivalent transformation of the left side yields $\forall I \cdot I^F_{sv} \subseteq I \rightarrow I \models f^{FA}$. Instantiating I with I^F, we get $I^F_{sv} \subseteq I^F \rightarrow I^F \models f^{FA}$. Since the premise of this implication holds by construction, modus ponens yields $I^F \models f^{FA}$.

Completeness relies on instantiation of μ and contradiction. It is required to show that if $\exists I \cdot I^F_{sv} \subseteq I \wedge I \models \neg f^{FA}$, then $I^F \not\models f^{FA}$. Assume, for contradiction, that $I^F \models f^{FA}$.

Instantiating $\exists$, we have interpretation I' that satisfies $\neg f^{FA}$ and agrees with I^F_{sv} (and, therefore, with I^F on sv).

We will construct a variable assignment μ, starting from $\emptyset$, by unwrapping quantifiers in f^{FA}. By f^{FA}_i we mean a formula resulting from removing the first i quantifiers from f^{FA}. That is,

$$f^{FA}_i \equiv Q_{i+1}x_{i+1} : \mathrm{D}_{i+1} \cdot \ldots Q_n x_n : \mathrm{D}_n \cdot f^{FA}.$$

Consider the two cases of the outermost quantifier Q_1 in f^{FA}. f^{FA}_1 is the result of removing Q_1 from f^{FA}, with x_1 being a free variable in f^{FA}_1.

If $Q_1 \equiv \forall$, then pick v_1 from $I' \models \neg f^{FA}$ by pushing negation over the universal quantifier and instantiating the resulting existential quantifier with v_1. Then, $I', v_1 \models \neg f^{FA}_1$. We also get $I^F, v_1 \models f^{FA}_1$ by instantiating the universal quantifier in f^{FA} with v_1.

If $Q_1 \equiv \exists$, then pick v_1 from $I^F \models f^{FA}$ by instantiating the existential quantifier, leading to $I^F, v_1 \models f^{FA}_1$. Like the above, by pushing the negation for I' we get $I' \models \forall x_1 : \mathrm{D}_1 \cdot \neg f^{FA}_1$. Instantiating the universal quantifier with v_1, we get $I' \models \neg f^{FA}_1$.

Either way, we add $x_1 \mapsto v_1$ to μ and repeat the above process for the remaining $n-1$ quantifiers, arriving at $f^{FA}_n \equiv \hat{f}^{FA}$, an assignment μ for all variables $x_1 \ldots x_n$, and two assertions:

$$I', \mu \models \neg \hat{f}^{FA}, \tag{8.1}$$

$$I^F, \mu \models \hat{f}^{FA}. \tag{8.2}$$

Now we will show that $\mu \in sv$. Consider $\hat{f}^{CA}$ that corresponds to $\hat{f}^{FA}$, $\hat{f}^{CA} = ConstAbst(f)$. Let I^{CA} be some interpretation of the constant abstractions in $\hat{f}^{CA}$.

We consider two cases of $\hat{f}^{CA}$ by the principle of excluded middle. If $I^{CA}, \mu \models \hat{f}^{CA}$, then by combining it with Equation (8.1) we get $I', I^{CA}, \mu \models \hat{f}^{FA} \not\Leftrightarrow \hat{f}^{CA}$. If $I^{CA}, \mu \not\models \hat{f}^{CA}$, then by combining it with Equation (8.2) we get $I^F, I^{CA}, \mu \models \hat{f}^{FA} \not\Leftrightarrow \hat{f}^{CA}$. Thus, $\mu \in sv$.

Since $\mu \in sv$, I' and I^F agree on valuations of F_i for μ because these are determined by I^F_{sv}. Since interpretation of $\hat{f}^{FA}$ only depends on $F(x_1, \ldots x_n)$ and free variables $x_1 \ldots x_n$ (determined by μ), both interpretations (Equation (8.1) and Equation (8.2)) should agree on the validity of $\hat{f}^{FA}$. Since the semantics of IPL is unambiguous, this leads us to a contradiction.

We conclude that $I^F \models f^{FA}$. □

Theorem 1 leads to two corollaries below, one showing soundness of verification and the other listing the termination conditions.

Corollary 1. Validity of formula f is equivalent to unsatisfiability $I^F_{sv} \models \neg f^{FA}$.

$$\mathsf{M} \models f \text{ iff } \nexists I \cdot I^F_{sv} \subseteq I \wedge I \models \neg f^{FA}.$$

Proof. By construction of f^{FA} in the algorithm, $\mathsf{M} \models f$ is semantically equivalent to $I^F \models f^{FA}$. By Theorem 1, the latter is equivalent to $\nexists I \cdot I^F_{sv} \subseteq I \wedge I \models \neg f^{FA}$. □

Corollary 2. Algorithm 1 is sound for solving Problem 1. The algorithm terminates if (i) satisfiability checking is decidable, (ii) behavioral checking with M is decidable, and (iii) search formula $\hat{f}^{FA} \not\Leftrightarrow \hat{f}^{CA}$ has a finite number of satisfying values for free variables (e.g., when quantification domains D_i are finite).

Proof. The algorithm equivalently transforms f to its PNF and performs a functional abstraction, which is an equivalent transformation under full interpretation I^F. Soundness follows from the Corollary 1 that shows that the last step of the algorithm is equivalent to the semantic validity of IPL formulas.

Termination of the verification algorithm follows from termination of the search of sv and construction of I^F_{sv}. The search terminates due to decidability of satisfiability checking (premise (i) above) and finiteness of the free variable values to satisfy the formula under check (premise (iii) above). For each μ in sv, construction of I^F_{sv} terminates because behavioral checking with M is decidable (premise (ii) above). □

Corollary 2 is the central piece of evidence to support the *soundness* claim (Claim 2) for IPL. The IPL specification focuses on expressive first-order sentences over multiple theories (including arithmetic), and the verification algorithm focuses on sound reasoning. As a result, completeness of reasoning is sacrificed: some IPL statements cannot be determined to hold or not on a given model.

8.1.2 Evaluation of IPL on System 1

IPL was used to check integration properties for the energy-aware mobile robot (described in Section 3.1). This study particularly focused on applicability, customizability, and expressiveness of IPL. In the rest of this section I describe the methodology, the set of available models, the integration properties for this study, the integration issues discovered using IPL, and the details of IPL performance.

Methodology

This validation was guided by three questions:

1. *What is the role of integration properties in this system?* Answering this question helped evaluate applicability of IPL, since it is only applicable when integration properties have an important role in the system's development.

2. *Can we specify the integration properties of this system in IPL?* This question evaluated customizability, expressiveness, and applicability IPL: to specify integration properties in a given system, the language has to be customizable to the models and concepts of the domain, expressive enough to capture the intended relationship between models, and applicable to handle corner cases and idiosyncrasies of models and their relationships.
3. *Is verification of these IPL properties tractable in practice?* This question helped evaluate applicability of IPL because the scale of models and properties encountered in practice may be intractable for IPL.

To address these questions, I performed a case study [239] on the robotic system that was described in Section 3.1. Since the system performed adaptation using multiple models, it is appropriate for validation of IPL. To discover the models and their relations, I conducted a historical review of the (completed by then) first phase of the project. Specifically, I investigated the available versions of the design, implementation, and documentation artifacts. Their sampling was determined by availability and convenience. The modeling and analysis of integration properties targeted the project's artifacts as-is, without any modification.

Throughout the case study I executed the following process assisted by an implementation [1] of IPL based on Eclipse (Oxygen 1a) and OSATE2 (version 2.3.0) [69]:

- Browse the available artifacts in search for models.
- Determine a relevant[2] relationship between models.
- Create integration abstractions for the models (details in Section 8.2).
- Specify an integration property in IPL.
- Execute the verification algorithm on the property.
- If the property fails, trace the error to the specific elements of models and abstractions.
- Determine if the verification error constitutes an integration error (or merely an abstraction or instrumentation error).

Evaluation Data

Upon the review of the robot's models, I decided to focus on power-related models[3] because power appeared to be a safety-critical and cross-cutting concern. Specifically, I studied the consistency relationships between three models (more information about which can be found in Section 3.1):

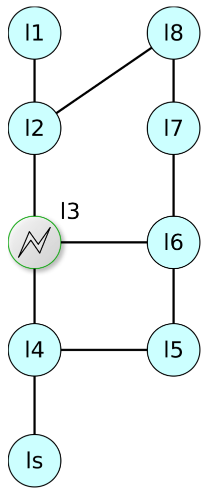
- A planning model M_{plan}. Approximately 10 variants of this model were discovered, each with different features (explained below). Another 10 variants were created to complete the space of models.
- A power model M_{power}. While the system fundamentally used one power model, various features lead to considering up to 6 power model variants per map.
- A map model M_{map}. 2 significantly different maps were discovered, but minor differences

[1]The IPL implementation is available at `https://github.com/bisc/IPL`.

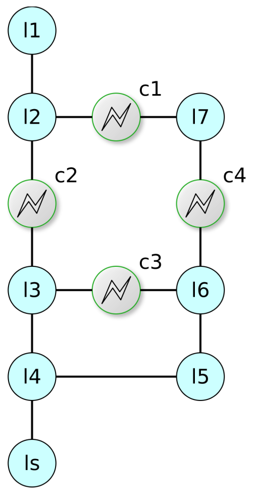
[2]In this context, a relationship is relevant if its violations can lead to system failures.

[3]The case study models have been archived [213] and are also available online: `https://github.com/bisc/IPLProjects`.

between map versions lead to considering 5 variants of this model.



(a) Map variant 1.



(b) Map variant 2.

Figure 8.1: Two variants of maps used in this study, with marked charging stations.

The high-level definitions of the robot's behavior are shared across these three models. Each behavior of the robot consists of the smallest behavior primitives called atomic tasks:

Definition 30 (Atomic task)**.** An *atomic task* is an indivisible action of the robot with fixed start/end map locations and other characteristics (time, energy, . . .). Atomic tasks can be of the following types:

- *Forward tasks:* the robot moves forward until the next checkpoint on the map.
- *Empty tasks:* the robot does nothing and stays in place. Empty tasks are used to model for missions of variable length by stuttering in the goal location.
- *Rotation tasks:* the robot rotates in place (changes its orientation while keeping the location the same).
- *Charging tasks:* the robot replenishes (some of) its battery charge while staying at a location with a charging station.
- *Other actuation tasks:* turning sensors on/off, changing speed, reconfiguring, and others.

The atomic tasks are combined to form missions:

Definition 31 (Mission)**.** A *mission* is a finite sequence of atomic tasks with contiguous and non-self-intersecting[4] locations. A *power-successful mission* can be completed without draining the battery using a given initial energy budget (which varies from 0 to $\overline{maxBat}$), with potential charging tasks.

The planning and power models exhibited similar variations over several dimensions. I term these dimensions *features*, and models that select concrete values for each feature are termed *model variants*. The following features are considered in this study:

[4]No implementations of M_{plan} allowed for self-intersecting trajectories.

- Missions of variable length. If this feature is disabled, only missions of a fixed length (e.g., of 5 tasks) are considered. Otherwise, all missions up to a certain length (e.g., from 1 to 5 tasks) are considered.
- Missions with rotations. If this feature is disabled, the robot's orientation is not part of the robot's state, and no rotation tasks are available or necessary.
- Missions with charging. If this feature is disabled, the robot does not charge at any station and has to complete the mission with the initial energy budget. If this feature is enabled, the robot can stop at any station and replenish its battery.

More details on how missions are modeled with views can be found in Subsection 8.2.1, devoted to the evaluation of integration abstractions used for System 1.

Integration Properties

As discussed in Section 3.1, the intent of modeling method integration for this system is to verify consistency between M_{power} and M_{plan}. Since the map model may cause inconsistencies, two goals need to be achieved:

- Check that, for any mission, the disagreement in power estimates between M_{power} and M_{plan} is no greater than $\overline{err\_cons}$.
- Check that, given M_{map}, both M_{plan} and M_{power} are operating over the same locations with the same adjacencies.

Notice that the first property above is dependent on the second one: if M_{power} and M_{plan} use a different set of locations and distances, they would disagree on the energy requirements for some missions. To simplify the integration, the second property is factored out. Thus, consistency between M_{plan} and M_{map} means that M_{plan} has been constructed to plan in the exactly same map as M_{map}. For M_{power}, consistency with M_{map} means that its view V_{power} (introduced below, and explained in more detail in Subsection 8.2.1) was constructed for the same map as in M_{map}.

To perform these integration checks, I created integration abstractions for the above models. First, structural views V_{power} for M_{power} and M_{map} list all atomic tasks possible in M_{map} and their expected energy expenditures based on M_{power}. Second, behavioral properties for M_{plan} to constrain the robot's behavior to specific missions from V_{power}. Multiple views and behavioral properties are necessary to integrate models with different features.

Thus, to perform the model integration for M_{power} and M_{plan}, three types of integration properties should be satisfied:

1. If M_{power} considers a mission power-successful, then M_{plan} should do so.
2. If M_{plan} considers a mission power-successful, then M_{power} should do so.
3. M_{plan} and M_{power} agree on the map.

In the rest of this section, these integration properties are expressed with several IPL formulas, with their verification results presented in the two sections to follow. Below is an IPL formula that illustrates the first type of integration properties for a three-step mission with a rotation in the middle. This property uses M_{plan}, a behavioral property, and V_{power}, as shown in Figure 8.2.

Property 3. If M_{power} considers a mission (of 3 tasks, straight-rotate-straight) power-successful, M_{plan} should consider this mission power-successful as well — allowing for $\overline{err\_cons}$ error.

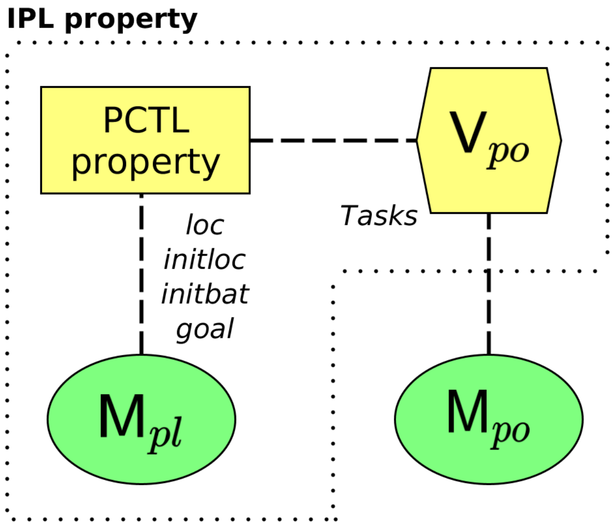


Figure 8.2: The context of integration properties for M_{power} and M_{plan}. The dotted line indicates the scope of the property.

"For any three tasks from M_{power} *in a sequence* $\langle$*go straight - rotate - go straight*$\rangle$*"*

$$\forall t_1, t_2, t_3 : \overline{Tasks} \cdot t_1.type = t_3.type = \text{STR} \wedge t_2.type = \text{ROT} \wedge \tag{8.3}$$

"that are well-aligned, do not self-intersect, and have sufficient energy,"

$$t_1.end = t_2.start = t_3.start \ \wedge t_1.start \neq t_3.end \wedge \Sigma_{i=1}^{3} t_i.energy \leq \overline{maxBat} \rightarrow$$

"any execution in M_{plan} *that visits every point of that sequence in the same order,"*

$$P_{max=?}[(\underline{loc} = t_1.start \mathsf{U} (\underline{loc} = t_2.start \mathsf{U} \underline{loc} = t_3.end)) \wedge (\mathsf{F} \underline{loc} = t_2.start)]$$

"if initialized appropriately, is a power-successful mission (modulo $\overline{err\_cons}$*)."*

$$\{|\underline{initloc} = t_1.start, \underline{goal} = t_3.end, \underline{initbat} = \Sigma_{i=1}^{3} t_i.energy + \overline{err\_cons}|\} = 1.$$

The above property uses quantification and constraints to construct a mission from the atomic tasks of V_{power} (accessed via the variable $\overline{Tasks}$). The mission is also constrained in terms of the energy budget, with all three tasks having to be executed (in the estimates of M_{power}) with at most one battery's worth of energy (charging is disabled for this property). In the second part of the formula, that mission constrains M_{plan} by prescribing a sequence of locations that the robot has to go through with an LTL subformula. An important part of this formula is that the initial value of the robot's battery in M_{plan} equals the energy estimate from M_{power} plus $\overline{err\_cons}$, thus adding a margin of acceptable consistency error. Then, the PCTL probability query operator returns the maximum possible probability (by picking the robot's actions in the non-deterministic transitions of the MDP) of the robot completing the mission. If the probability is 1, then the mission is power-successful by M_{plan}.

The above property can be augmented by allowing missions of variable length, and following the same general pattern:

Property 4. If M_{power} considers a mission (up to four go-straight tasks long) power-successful, M_{plan} should consider this mission power-successful as well — allowing for $\overline{err\_cons}$ error.

"Any mission with up to four straight motion tasks"

$$\forall t_1, t_2, t_3, t_4 : \overline{Tasks} \cdot$$

"connected to each other in a sequence"

$$t_1.endloc = t_2.startloc \wedge t_2.endloc = t_3.startloc \wedge t_3.endloc = t_4.startloc \wedge$$

"that is non-empty, can have empty tasks only in the end,"

$$t_1.type \neq \text{EMP} \wedge (t_2.type = \text{EMP} \rightarrow t_3.type = \text{EMP}) \wedge$$
$$(t_3.type = \text{EMP} \rightarrow t_4.type = \text{EMP}) \wedge$$

"contains no self-intersecting tasks"

$$(\nexists i : \overline{Tasks} \cdot (i = t_1 \vee i = t_2 \vee i = t_3 \vee i = t_4) \wedge i.type = \text{STR} \wedge$$
$$((i \neq t_1 \wedge t_1.endloc = i.endloc \wedge t_1.type = \text{STR}) \vee$$
$$(i \neq t_2 \wedge t_2.endloc = i.endloc \wedge t_2.type = \text{STR}) \vee$$
$$(i \neq t_3 \wedge t_3.endloc = i.endloc \wedge t_3.type = \text{STR}) \vee$$
$$(i \neq t_4 \wedge t_4.endloc = i.endloc \wedge t_4.type = \text{STR}))) \wedge$$

"and that is a power-successful mission in M_{power}"

$$\Sigma_{i=1}^{4} t_i.energy \leq \overline{maxBat} \rightarrow$$

"will correspond to such executions in M_{plan} that visit all sequence points"

$$P_{max=?}[(\mathsf{F}\underline{loc} = t_2.startloc) \wedge (\mathsf{F}\underline{loc} = t_3.startloc) \wedge (\mathsf{F}\underline{loc} = t_4.startloc) \wedge$$

"in the correct order"

$$((\underline{loc} = t_1.start)\mathsf{U}(\underline{loc} = t_2.start\mathsf{U}(\underline{loc} = t_3.start\mathsf{U}\underline{loc} = t_4.end)))]$$

"and, when initialized correctly, will be power-successful."

$$\{|\underline{initloc} = t_1.start, \underline{goal} = t_4.end, \underline{initbat} = \Sigma_{i=1}^{4} t_i.energy + \overline{err\_cons}|\} = 1.$$

In a variant of V_{power} created for this property, $\overline{Tasks}$ contains empty tasks, but no rotation tasks (to simplify this illustration). No connectors are used. Expressing absence of self-intersection with potentially empty tasks is conveniently done by quantifying over the four tasks ($t_1 \cdots t_4$) with another variable (which represents a potentially intersecting task, i) and declaring it non-intersecting with each of the four. This example shows how quantifiers allow IPL to express complex constraints on view elements.

The second type of integration properties does the opposite implication of power success: from M_{plan} to M_{power}. This time, we use an existentially quantified variable for a power budget. We are interested in situations when, given a power budget, M_{plan} successfully completes a mission, but this mission fails in M_{power} even with a large enough budget to offset the consistency error. This property assumes monotonicity of power dynamics: given a mission, its success with a certain budget automatically means its success with any larger budget (the converse is true about insufficient energies). The sv for this property contains 100 vectors of free variable values.

Property 5. If M_{plan} considers a mission power-successful (of four go-straight tasks long), M_{power} should do so.

"For any mission with exactly four straight motion tasks"

$$\forall t_1, t_2, t_3, t_4 : \overline{Tasks} \cdot t_1.type = t_2.type = t_3.type = t_4.type = \text{STR} \wedge$$

"connected to each other in a sequence"

$$t_1.endloc = t_2.startloc \wedge t_2.endloc = t_3.startloc \wedge t_3.endloc = t_4.startloc \wedge$$

"without self-intersections"

$$distinct(t_1.startloc, t_2.startloc, t_3.startloc, t_4.startloc, t_4.endloc) \rightarrow$$

"there exists such an energy budget greater than the energy expected by M_{power}*"*

$$(\exists b : \mathbb{N} \cdot b \geq \Sigma_{i=1}^{4} t_i.energy - \overline{err\_mdp} \wedge$$

"that if M_{plan}*, going through all the sequence points"*

$$P_{max=?}[(\mathsf{F}\underline{loc} = t_2.startloc) \wedge (\mathsf{F}\underline{loc} = t_3.startloc) \wedge (\mathsf{F}\underline{loc} = t_4.startloc) \wedge$$

"in the correct order"

$$((\underline{loc} = t_1.start)\mathsf{U}(\underline{loc} = t_2.start\mathsf{U}(\underline{loc} = t_3.end\mathsf{U}\underline{loc} = t_4.end)))]$$

"and initialized correctly, is power-successful on that budget,"

$$\{|\underline{initloc} = t_1.start, \underline{goal} = t_4.end, \underline{initbat} = b|\} = 1 \rightarrow$$

"then M_{power} *should also be power-successful that budget."*

$$\Sigma_{i=1}^{4} t_i.energy - \overline{err\_cons} < b).$$

The third integration property — consistency of M_{power} and M_{plan} with respect to maps — can be split into two groups of properties: consistency of locations and consistency of edges between locations. It is convenient to take a transitive approach to checking this consistency, showing that M_{power} and M_{map} are consistent, and that M_{map} and M_{plan} are also consistent. It is easier to quantify specific locations by using an explicit model of a map.

Starting with M_{power} and M_{map}, two views are necessary: V_{power} (which has already been used above) and V_{map} (which contains map locations as components, and their connectivity is recorded as a property of nodes, with more detail in Section 8.2). In short, V_{map} contains locations as components in $\overline{Locs}$, and each location is characterized by a unique identifier (id) and a list of its adjacent locations ($edges$). Specification of M_{power} and M_{map} consistency is exemplified with the two properties below.

Property 6. Any location in M_{map} is reachable in M_{power}.

"For any location, there exist an incoming and outgoing tasks."

$$\forall l : \overline{Locs} \cdot (\exists t_{in}, t_{out} : \overline{Tasks} \cdot l.id = t_{in}.endloc = t_{out}.startloc).$$

Property 7. Every straight motion task in M_{power} corresponds to an edge in M_{map}.

"For any straight motion task, there is a pair of locations"

$$\forall t : \overline{Tasks} \cdot t.type = \text{STR} \rightarrow \exists l_1, l_2 : \overline{Locs} \cdot$$

"where the task begins and ends connected by an edge."

$$l_1.id = t.startloc \wedge l_2.id = t.endloc \wedge l_1 \in l_2.edges \wedge l_2 \in l_1.edges.$$

In a similar way, one can assure that V_{map} is consistent with V_{power}. Given that the above properties are satisfied, we can turn to consistency between M_{map} and M_{plan}.

Property 8. Every location in M_{map} exists in M_{plan}.

"Any location from V_{map} *exists in* M_{plan}*"*

$$\forall l : \overline{Locs} \cdot P_{max=?}[\underline{loc} = l.id]\{|\underline{initloc} = l.id, \underline{goal} = l.id, \underline{initbat} = 1|\} = 1.$$

If the above property does not pass, it indicates that the set of locations is not consistent. Further, assuming continuous location IDs, one can ensure that M_{plan} does not have more locations than V_{map} by attempting to get to a location with the ID smaller than the minimum (similarly for larger than the maximum):

Property 9. There are no reachable locations in M_{plan} with IDs smaller than the minimal ID found in M_{map}.

"For any two distinct locations, one starting and one with the smallest ID,"

$$\forall l_{init}, l_{min} : \overline{Locs} \cdot l_{init} \neq l_{min} \wedge (\forall l_o : \overline{Locs} \cdot l_{min}.id \leq l_o.id) \rightarrow$$

"any path in M_{plan} *attempting to get the ID of the smallest minus 1,"*

$$P_{max=?}[\mathsf{F}\underline{loc} = l_{min}.id - 1]$$

"initialized correctly, would fail."

$$\{|\underline{initloc} = l_{init}.id, \underline{goal} = l_{min}.id - 1, \underline{initbat} = \overline{maxBat}|\} = 0.$$

The above property uses a slightly weaker notion of existence (reachability), but it is sufficient for our purposes: if a location is not reachable in the MDP, it would not affect other verification. Finally, we can demonstrate that the edges are consistent between M_{plan} and M_{map} by showing two properties below.

Property 10. Any pair of locations with an edge in M_{map} can be traversed directly in M_{plan}.

"For any task, the behaviors in M_{plan} *going from its start to its end"*

$$\forall t : \overline{Tasks} \cdot t.type = \mathsf{STR} \rightarrow P_{max=?}[\underline{loc} = t.startloc \mathsf{U} \underline{loc} = t.endloc]$$

"initialized correctly, should succeed if given enough battery."

$$\{|\underline{initloc} = t.startloc, \underline{goal} = t.endloc, \underline{initbat} = t.energy + 1|\} = 1.$$

Property 11. Any pair of locations without an edge in M_{map} cannot be traversed directly in M_{plan}.

"For any pair of distinct locations without an edge between them,"

$$\forall l_1, l_2 : \overline{Locs} \cdot l_1 \neq l_2 \wedge l_1 \notin l_2.edges \rightarrow$$

"any behavior in M_{plan} *aiming to go between them,"*

$$P_{max=?}[\underline{loc} = l_1.id \mathsf{U} \underline{loc} = l_2.id]$$

"when initialized correctly, would fail even with a maximum charge."

$$\{|\underline{initloc} = l_1.id, \underline{goal} = l_2.id, \underline{initbat} = \overline{maxBat}|\} = 0.$$

If the above properties are valid, we can conclude that M_{power} and M_{plan} are consistent with respect to a given M_{map}.

The three integration properties were exemplified with specific IPL formulas above, and the full set of IPL specifications for these models is available at `https://github.com/bisc/IPLProjects`. If all of these formulas in this set hold, then models M_{power}, M_{plan}, and M_{map} are considered sufficiently consistent for the purposes of this integration.

Discovered Integration Errors

Each instance of an invalid integration property (except typos and IPL implementation errors) throughout this case study has been documented and analyzed. The analysis traced each failure to the error that caused it and determined the (hypothetical) impact of this error on the running system. The impact of each error was evaluated in separation from the other integration issues.

The 17 discovered errors were assigned the following (mutually exclusive) categories:

- Errors in the models (or model generation code).
    - Leading to aggressive faults: the issues that would lead to possible violation of safety constraints (power in this case).
    - Leading to conservative faults: the issues that would lead to being overly conservative and missing behaviors that are otherwise more optimal.
    - Not leading to faults: these issues are inconsistencies, but do not affect the system's real-world behavior.
- Errors in the integration artifacts: views, behavioral properties, and IPL formulas. These errors are false positive ones.
    - View errors: an error in constructing a view.
    - Behavioral property errors: an error in the specification of a behavioral property.
    - IPL formula errors: an error in an IPL specification (beyond a behavioral property).

Since each error was discovered by running IPL verification of integration properties, this set of errors leads to three takeaways:

- IPL can discover model integration issues in realistic projects. Some of these issues could lead to system failures and safety violations.
- View creation is error-prone and can introduce a significant number of false positives. It is necessary to perform separate quality assurance on views to discover these issues separately from true positives (inconsistencies between models).
- Tools are needed for separate evaluation of parts of IPL statements. E.g., visualizing missions that have been considered in a given property.

#	Error	Category	Origin	Impact	Fix
1	The robot can move despite insufficient energy	Model error, aggressive fault	A modeling decision: transitions in M_{plan} do not explicitly check that $\underline{bat} > 0$ before executing a task (t). Instead, each transition assigns $\underline{bat} := max(\underline{bat} - e)$, where e is the energy required for the transition	The robot would run out of energy on missions where $\Sigma_{i=1}^{n-1} t_i.energy < \underline{initbat} < \Sigma_{i=1}^{n} t_i.energy$, where $t_1..t_n$ are the tasks to move between the initial position (or a recharging station) and the goal (or a recharging station).	A guard checking $\underline{bat}$ >= for every transition in M_{plan}.
2	Negative rotation energy in M_{plan} when facing south and turning to face southeast	Model error, aggressive fault	In generation of M_{plan}, incorrect angle normalization occurs when performing angle operations on non-matching intervals: $(0, 2\pi)$ and $(-\pi, \pi)$	On maps with south-to-southeast edges (no such maps were discovered in this study), the robot would underestimate the required energy and may run out of power	Correct the angle computation
3	Making turns of 0 degrees takes more than 0 mwh of energy	Model error, conservative fault	M_{power} used a linear regression model that maximized the fit to the experimental data, but not the constraint of passing through $(0, 0)$. In contrast, V_{power} assumed that tasks with 0 degrees of rotation required 0 energy.	For missions with empty tasks, the robot would overestimate the required energy budget and discard otherwise optimal plans	Changing M_{power} to a function that maps empty tasks to 0 required energy
4	Inconsistent required energy in M_{plan}	Model error, conservative fault	Unknown, possibly a copying mistake of the model author	The required energy for the $l_3 \rightarrow l_6$ edge was higher than in M_{power}— or than the energy of $l_6 \rightarrow l_3$. Thus, the robot would behave unnecessarily conservatively.	Restoring the energy value in $l_3 \rightarrow l_6$ to be consistent with M_{power} and $l_6 \rightarrow l_3$

5	Slight mismatch (0.1 second) in required times for straight moves between M_{power} and a variant of M_{plan}	Model error, no fault	Unknown, possibly rounding of times or distances by the model author	An integration property of precise consistency ($\overline{err\_cons} = 0$) fails, but no perceivable impact on planning or execution	Adjust the failing integration property to have $\overline{err\_cons} > 0$
6	Unnecessarily wide ranges for location variables in M_{plan}	Model error, no fault	Unknown, possibly a typo of the model author	A model contains 9 locations, but the type of the variable specifying the range of $[0..9]$ (10 locations). This issue affects map consistency properties (since they target the variable type), but does not impact planning and execution.	Replacing the range with $[0..8]$
7	Unnecessarily wide ranges for location variables in another M_{plan}	Model error, no fault	Unknown, possibly a typo of the model author	A model contains 12 locations, but the type of the variable specifying the range of $[0..12]$ (13 locations). This issue affects map consistency properties (since they target the variable type), but does not impact planning and execution.	Replacing the range with $[0..11]$
8	Edge $l_3 \rightarrow l_6$ missing in one of the map views	View error	A typo during view creation	The robot would consider the paths using $l_3 \rightarrow l_6$ infeasible (while these paths are feasible in reality), thus possibly missing optimal plans.	Adding the missing edge to the view
9	Edge $l_s \rightarrow l_4$ missing in another map view	View error	A typo during view creation	The robot would consider the paths using $l_s \rightarrow l_4$ infeasible (while these paths are feasible in reality), thus possibly missing optimal plans.	Adding the missing edge to the view

10	Rotations do not match the robot headings	IPL formula error	An error of the researcher, with the constraints on directions matching in consecutive moves only applying (incorrectly) in cases of empty moves	The integration property failed. No impact on planning or execution.	Apply the constraints on directions to all consecutive moves of of the robot
11	A view of M_{power} did not assign values of starting and ending heading for forward task components	View error	An accidental omission in view generation code	The robot would consider invalid missions with mismatched headings	Add the missing properties
12	Overlapping IDs between charging, empty, and rotation tasks in several views of M_{power}	View error	An error of the researcher during manual view creation	A robot would consider missions with invalid combinations of tasks	Assign all tasks non-overlapping IDs
13	Mismatch between turns in views of M_{power} for two different maps	View error	An accidental omission in view creation code	The robot would miss the available turns in one of the maps, leading to not considering otherwise valid missions when checking integration properties	Add missing turn components to one of the views
14	Empty tasks assigned an incorrect type (charging tasks) in a view of M_{power}	View error	A typo in the view generation code	The robot would not consider missions of variable length, unless they ended at a charging station	Change the type to empty tasks

15	Mismatch between charging behavior in M_{plan} and integration properties	IPL formula error	The specification of charging behavior in the integration property allowed any charging, whereas M_{plan} only allowed charging if $\underline{bat} > 1500$ mwh	The integration property considered sp,e missions that are prohibited in M_{plan}	Add constraints in the integration property that allow charging only when the battery charge is below 1500 mwh
16	The “last task is not charging” constraint specified incorrectly	IPL formula error	The constraint referenced the second last task, instead of the last task	The missions with charging in the second last task were not considered, while the missions with charging as the last task were	Adjust the constraint to reference the last task
17	Initial battery from integration properties was too low to complete tasks in M_{plan}	IPL formula error	The enable charging (as described in the previous inconsistency), integration properties would require that the energy is low enough when arriving to charging stations. This led to requiring the initial energy to be lower than required for the initial sequence of tasks in M_{plan} leading to a charging station	Considering missions that are not valid	Adding a constraint to the integration properties that the energy has to be sufficient to perform tasks between charging stations

Table 8.1: Model inconsistencies discovered in the study.

Performance

I evaluated the performance of an Eclipse-based IPL implementation using variants of the M_{power}-to-M_{plan} property (e.g., Prop. 4). In particular, twenty four verification runs were executed by varying the number of mission tasks and the map, and toggling each of the mission's features — variable length missions, charging, and rotations.

The following dependent variables were observed in these runs: count of sv, total time, saturation time, interpretation time, time in SMT, and time in model checking. We did not find IPL's memory demands limiting since at most one external tool was executing at each point (which, however, indicates potential for parallelizing the model checking process). The performance results are shown in Table 8.2.

The verification runs were performed sequentially on the following platform: Intel Core i7-7600U, Ubuntu 17.04, Eclipse Oxygen 1a, OSATE 2.3.0 (debug mode) [70], Z3 solver 4.5.0 [55], PRISM model checker 4.4.beta [134] with Rabinizer 3.1 [129]. The dataset and its analysis have been archived [213] and are also available online[5].

The high-level performance findings are: (a) verification times vary from dozens of seconds to over 6 hours. Counts of sv vary from dozens to over a thousand; (b) we found that longer missions lead to increase in both saturation and interpretation times, whereas missions with more features primarily affect the saturation process; (c) model checking times grew linearly with increases in mission length across feature groups, with little response to increases in mission features; (d) the saturation times grow substantially with more features, especially when considering rotations due to additional quantified variables and constraints; (e) IPL's mean overhead (across all verification runs) was 0.74% (stdev 0.78%) of the total verification time of each run; (f) IPL's memory demands were not limiting, since at most one external tool ran at a time.

Conclusions for Evaluation on System 1

Returning to the questions posed in Subsection 8.1.2, this study leads to the following answers:

1. *What is the role of integration properties in this system?* In the power-aware mobile robot, integration properties describe complex relationships between structural and behavioral models. These relationships play a role the system's safety arguments, and their violation may lead to system failures. This finding supports Claim 3 (applicability).
2. *Can we specify the integration properties of this system in IPL?* The integration properties can be specified IPL, using views and behavioral properties as abstractions of models. This finding supports Claim 1 (expressiveness), Claim 4 (customizability), and Claim 3 (applicability).
3. *Is verification of these IPL properties tractable?* The verification is tractable, and performance improvements are possible (discussed in Chapter 10). This finding supports Claim 3 (applicability). Claim 2 (soundness) is in part supported by identifying the existing model integration issues using verification.

[5]`https://github.com/bisc/IPLProjects/tree/master/IPLRobotProp/performance-analysis`

Map	# of steps	Variable length?	Charging?	Rotation?	# satvals	Saturation time (s)	Interp. time (s)	overhead (%)	Total time (s)
map0	4	n	n	n	50	4.1	16.1	1.3	20.4
map0	5	n	n	n	40	5.5	22.7	0.9	28.5
map0	6	n	n	n	34	9.7	55.2	0.5	65.2
map0	7	n	n	n	16	7.8	85.9	0.4	94.0
map0	4	y	n	n	142	55.4	37.1	0.6	93.1
map0	5	y	n	n	182	142.6	95.3	0.3	238.5
map0	6	y	n	n	216	234.1	197.5	0.3	432.8
map0	7	y	n	n	232	336.4	446.1	0.2	783.8
map0	4	n	y	n	86	22.7	43.7	0.5	66.8
map0	5	n	y	n	100	37.7	67.0	0.4	105.1
map0	6	n	y	n	108	47.8	116.7	0.3	165.0
map0	7	n	y	n	99	107.9	191.1	0.3	299.9
map0	4	y	y	n	195	71.4	85.0	0.3	156.9
map0	5	y	y	n	295	243.8	171.2	0.2	416.0
map0	6	y	y	n	403	468.9	373.3	0.2	843.5
map0	7	y	y	n	502	949.9	656.1	0.1	1608.0
map0	8	y	y	n	559	1467.7	1407.2	0.1	2876.6
map3b	4	n	n	y	56	213.1	19.7	1.0	235.1
map3b	5	n	n	y	60	315.6	33.1	0.9	352.0
map3b	6	n	n	y	44	450.8	63.6	0.8	518.5
map3b	4	y	n	y	162	2768.1	42.1	0.2	2815.0
map3b	5	y	n	y	222	5692.3	75.5	0.1	5773.5
map3b	6	y	n	y	266	8618.4	168.1	0.1	8793.4
map3b	7	y	n	y	266	10137.3	256.2	0.1	10403.8
map3b	4	y	y	y	440	5663.5	15410.6	0.0	21078.6

Table 8.2: IPL performance results.

8.1.3 Evaluation of IPL on System 3

IPL was secondarily evaluated in the context of thread and battery scheduling for a quadrotor. This evaluation was opportunistic: there was no initial intention of discovering or checking integration properties, and the study focused on integration of analyses (more details can be found in Section 8.3). Nevertheless, two integration properties (part of analysis contracts) required expressive specification, and an early prototype of IPL was used for it. The goal here is to demonstrate that application of IPL generalizes across logics (in this case, LTL) and domains (in this case, aerospace).

Integration Property 1: Thread Scheduling and Frequency Scaling

The first integration property concerns two models: the scheduling model (M_{sch}) and the CPU model (M_{cpu}). These models serve two competing purposes: M_{sch} schedules threads so that they do not miss deadlines (which may result in failures of the whole system), and M_{cpu} reduces frequency of CPU (to make the system more power-efficient). These two models are also not independent: frequency reduction may lead to deadline misses, since threads take longer to compute on CPUs with smaller frequencies. The frequency scaling model only behaves correctly if the scheduling is semantically equivalent to a deadline-monotonic scheduling policy. Note that a scheduling policy can be deadline-monotonic scheduling (DMS) for a specific model, e.g., rate-monotonic scheduling (RMS) for a harmonic model, even thought it is not deadline monotonic in general.

I use two views for this property. The scheduling view (V_{sch}) exposes $\mathsf{Threads}$as components with Dline, their periods, and worst-case execution times as their properties. The CPU view (V_{cpu}) exposes CPUsas components, with $\mathsf{CPUFreq}$, $\mathsf{CPUFreq_{max}}$, and $\mathsf{CPUBind}$ as component properties.

To keep M_{sch} and M_{cpu} non-conflicting, I specify and verify the integration property informally stated as *"when CPU frequencies are reduced by a frequency scaling algorithm, deadlines are not missed if the scheduler and threads* behave *as deadline-monotonic (not necessarily that the prescribed policy is deadline-monotonic)"*. Deadline monotonicity depends on CPU frequencies, bindings, and timing behaviors of the scheduler. To use IPL for this property, I use behavioral semantics of M_{sch} and abstract away the details of M_{cpu} by using V_{cpu}. Thus, the context of this IPL specification is M_{sch}, V_{sch}, and V_{cpu}.

The property, specified below in Property 12, iterates over all CPUs with reduced frequency and demands that all threads allocated to such CPUs behave deadline-monotonically. That is, a thread is allowed to preempt only threads that have deadlines greater than its own. The formula uses two layers of quantification wrapped around two rigid terms and a model instance with a temporal atom inside.

Property 12. All CPUs with reduced frequency behave deadline-monotonically.

"All CPUs whose frequency was scaled down"

$$\forall c : \mathsf{CPUs} \cdot \overbrace{c.\mathsf{CPUFreq} < c.\mathsf{CPUFreq_{max}}}^{\text{RTERM}} \rightarrow$$

"should only bind pairs of threads that"

$$\forall t_1, t_2 : \mathsf{Threads} \cdot \overbrace{\mathsf{CPUBind}(t_1, c) \wedge \mathsf{CPUBind}(t_2, c)}^{\text{RTERM}} \rightarrow$$

"behave deadline-monotonically with respect to each other."

$$\overbrace{\underbrace{(\mathsf{GCanPrmpt}(t_1, t_2) \rightarrow t_1.\mathsf{Dline} < t_2.\mathsf{Dline})}_{\text{TATOM}} \{|\underline{thrdset} = \{t_1, t_2\}, \underline{cpu} = c|\}}^{\text{MDLINST}}.$$

This property can be checked by the IPL verification algorithm. Using V_{sch} and V_{cpu}, the saturation process will find all values of c, t_1, and t_2 satisfying the two instances of RTERM. For these values MDLINST will be behaviorally evaluated on M_{sch}. After obtaining the necessary interpretations of MDLINST, the final satisfaction check will be done to determine the property's validity. Property 12 should be verified every time after CPU frequencies are scaled down. If verification succeeds, it is guaranteed that deadlines will not be missed, and the power consumption has been minimized (i.e., that M_{sch} and M_{cpu} are integrated correctly).

Integration Property 2: Safe Concurrency and Thread Scheduling

Another pair of models includes, again, the model of thread scheduling (M_{sch}) and the safe concurrency model (M_{rek}), described in Section 3.3. The integration goal is to apply M_{rek} to the scheduling, which is only valid under implicit deadlines and fixed-priority scheduling. Satisfaction of these conditions is the integration property that needs to be verified.

To express this integration property, I use the set of threads from V_{rek} (with their properties like period and deadline) and behavioral properties from M_{sch} below. One (fully rigid) formula constrains threads to be implicit deadlines, and another, mixed (rigid and flexible) formula for M_{sch} expresses the fixed-priority scheduling:

Property 13. All threads have implicit deadlines and fixed-priority scheduling.

$$\forall t \cdot \mathsf{Per}(t) = \mathsf{Dline}(t) \wedge \tag{8.4}$$

$$\forall t_1, t_2 \cdot \mathsf{G}(\mathsf{CanPrmpt}(t_1, t_2) \rightarrow (\mathsf{G}\neg\mathsf{CanPrmpt}(t_2, t_1))). \tag{8.5}$$

When run on this property, the IPL verification algorithm finds relevant values of t, t_1, and t_2 from the views and checks the LTL subformula on them using M_{sch}. This property should be verified every time when concurrency safety is checked with M_{rek}. If this property fails, the output of this check may be incorrect and is not to be trusted.

Conclusions for Evaluation on System 3

This application of IPL to a quadrotor has shown that IPL is customizable to new systems, domains, and behavioral logics — thus supporting Claim 4 (customizability). Indeed, the system is unlike a mobile robot in that it has jobs with lower computational complexity, but strong real-time requirements to keep the system stable. In addition, the technical domain is different: schedulability-related models instead of planning/power models. We have also observed that IPL can be customized to LTL (in addition to PCTL for System 1)d that IPL can be customized to LTL (in addition to PCTL for System 1).

In this context, Claim 3 has been demonstrated here by encoding the precise properties of interest for existing models. Furthermore, Claim 1 (expressiveness) has been supported by expressing these properties in terms of the temporal behaviors of the system — as opposed to a common (and less expressive) approach of categorical tags (RMS, DMS, and so on). Experiments with several system designs of varying sizes showed that model integration is checked appropriately and within times acceptable in practice (the details of these experiments are located in Section 8.3), again supporting Claim 3 (applicability). Finally, Claim 2 (soundness) is in part supported by correctly identifying model integration issues using verification.

8.1.4 Conclusions for Evaluation of the Integration Property Language

This section described a theoretical evaluation of the IPL verification algorithm, and the application of IPL to integration properties in Systems 1 and 3. In both of these case studies, multiple integration properties were discovered, specified, and verified.

The following results summarize the validation findings with respect to the qualities of integration:

- *Expressiveness:* IPL has been shown to be sufficiently expressive to capture integration properties between mixed structural-behavioral models. The expressiveness of IPL builds upon the expressiveness of the first-order logic and multiple pluggable modal languages, demonstrated on the examples of LTL (System 3) and PCTL (System 1). These results support Claim 1.
- *Soundness:* the IPL algorithm is shown to be sound and terminate under realistic conditions. The practical application of IPL delivers sound results as well: integration properties fail due to either integration errors between models or modeling errors in views or properties. These results support Claim 2.
- *Applicability:* IPL showed its flexibility in handling corner cases in both case studies. The performance experiments in Systems 1 and 3 have demonstrated reasonable scalability for realistic systems. These results support Claim 3.
- *Customizability:* IPL was successfully customized to two modal logics (LTL and PCTL), two systems (a mobile robot and a quadrotor), and three domains (power-aware planning, thread scheduling, and battery design). These results support Claim 4.

8.2 Validation of Part II: Integration Abstractions

Integration abstractions were validated in the context of all four case study systems. The claims/qualities for evaluation of integration abstractions are summarized in Table 1.1 in Section 1.2. This section describes the aspects of the case studies related to views and integration properties.

8.2.1 Evaluation of Integration Abstractions on System 1

The investigation of integration abstractions for the energy-aware mobile robot (System 1) was conducted as part of the integration study described in Subsection 8.1.2. The main focus of the study was on discovering, specifying, and verifying integration properties (covered in Subsection 8.1.2). Below I address the secondary focus of this study — on whether integration abstractions can support the desired qualities of integration: expressiveness, soundness, applicability, and customizability.

In the context of System 1, these integration qualities take the following interpretations:

- *Expressiveness:* whether the views (specifically, the instances of the power view V_{power}) have the means to express the tasks and missions that the robot can accomplish (particularly in M_{plan}), and annotate them with the required energies from M_{power}; and whether behavioral properties in PCTL have the means to constrain the robot in M_{plan} to the missions described in M_{power}.
- *Soundness:* whether the views are sound and complete with respect to the tasks/missions that the robot can accomplish; and whether the checking of behavioral properties produces a sound result and terminates.
- *Applicability:* whether the views can satisfy the implicit constraints on tasks and their ordering, and accommodate different features of views and modes of the robot; and whether the PCTL checking produces results within the practical limits on time and memory.
- *Customizability:* whether the views and PCTL properties are extensible for new tasks and mission features.

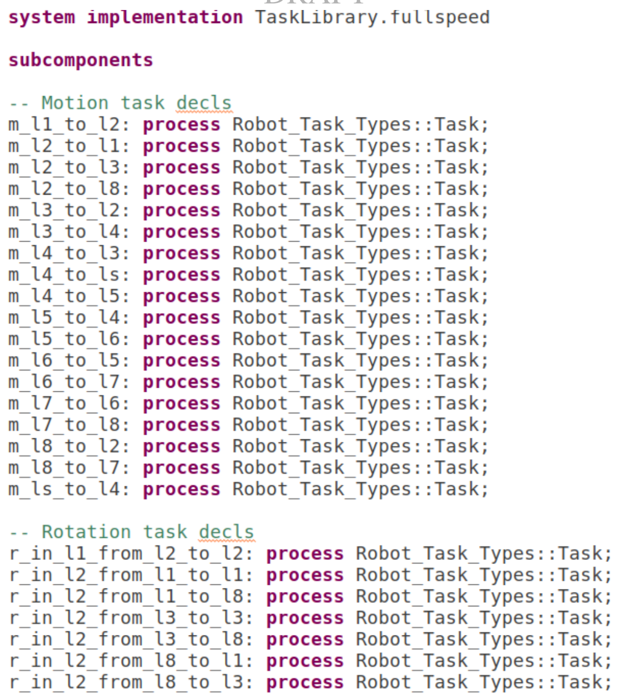
First, I discuss how these qualities are supported by using views to model robot tasks based on M_{power}, and then by using PCTL properties to interface with M_{plan}.

Views for System 1

As described in Subsection 8.1.2, the role of the power view (V_{power}) for the regression power model (M_{power}) is to represent a set of atomic tasks (see Definition 30) that can be performed on a given map. Atomic tasks are sequentially composed into missions (see Definition 31) using quantified variables in IPL formulas. Instances of V_{power} are created automatically by an implementation of the power viewpoint, which creates an instance of V_{power} based on a map (M_{map}). Encoded in AADL, V_{power} is separated into a declaration part, where all task components are declared (Figure 8.3), and a value-setting part, where all property values are set (Subsection 8.1.2).

Once V_{power} is created, an integration property compares the missions constructed from V_{power} and M_{plan} to check that energy is modeled consistently in M_{power} and M_{plan}. The integration qualities supported by V_{power} are evaluated at two levels: for the individual tasks directly represented in the view, and for the missions composed from these tasks by the means of IPL's quantified variables (e.g., t_1, t_2, t_3 in Property 3 described in Subsection 8.1.2).

The *expressiveness* of V_{power} in terms of task types is sufficient for this integration scenario: any task that ends in one of the map's locations can be represented as a component in V_{power}. In cases when a task does not require setting certain properties (e.g., an empty task does not



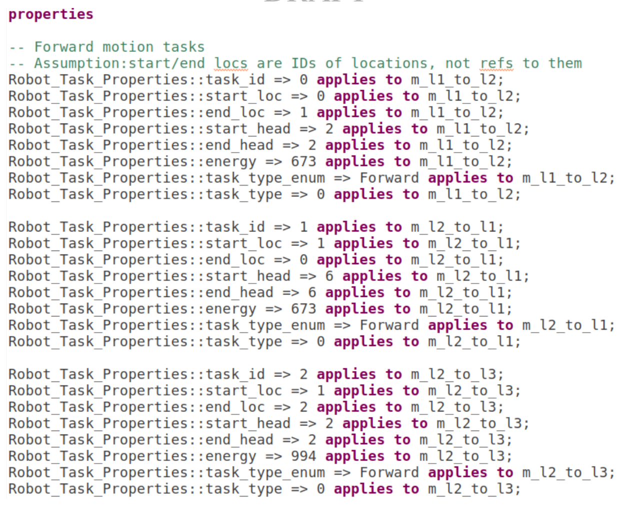
```
system implementation TaskLibrary.fullspeed

subcomponents

-- Motion task decls
m_l1_to_l2: process Robot_Task_Types::Task;
m_l2_to_l1: process Robot_Task_Types::Task;
m_l2_to_l3: process Robot_Task_Types::Task;
m_l2_to_l8: process Robot_Task_Types::Task;
m_l3_to_l2: process Robot_Task_Types::Task;
m_l3_to_l4: process Robot_Task_Types::Task;
m_l4_to_l3: process Robot_Task_Types::Task;
m_l4_to_ls: process Robot_Task_Types::Task;
m_l4_to_l5: process Robot_Task_Types::Task;
m_l5_to_l4: process Robot_Task_Types::Task;
m_l5_to_l6: process Robot_Task_Types::Task;
m_l6_to_l5: process Robot_Task_Types::Task;
m_l6_to_l7: process Robot_Task_Types::Task;
m_l7_to_l6: process Robot_Task_Types::Task;
m_l7_to_l8: process Robot_Task_Types::Task;
m_l8_to_l2: process Robot_Task_Types::Task;
m_l8_to_l7: process Robot_Task_Types::Task;
m_ls_to_l4: process Robot_Task_Types::Task;

-- Rotation task decls
r_in_l1_from_l2_to_l2: process Robot_Task_Types::Task;
r_in_l2_from_l1_to_l1: process Robot_Task_Types::Task;
r_in_l2_from_l1_to_l8: process Robot_Task_Types::Task;
r_in_l2_from_l3_to_l3: process Robot_Task_Types::Task;
r_in_l2_from_l3_to_l8: process Robot_Task_Types::Task;
r_in_l2_from_l8_to_l1: process Robot_Task_Types::Task;
r_in_l2_from_l8_to_l3: process Robot_Task_Types::Task;
```

Figure 8.3: Task declarations in an instance of V_{power}.

have a specific orientation), their values can be left unspecified. Omitting the values leads to underspecified SMT constraints on the uninterpreted functions that model the properties of the view elements. The expressiveness of tasks is constrained by the locations of the map: tasks that start or end not in one of the map's locations cannot be specified. From the perspective of missions, V_{power} allows to specify only missions up to a given finite length. This constraint is not limiting for this case study because each map has an upper bound of missions because of the non-self-intersection requirement (7 for Figure 8.1a and 10 for Figure 8.1b).

The *soundness* of V_{power} in terms of tasks (i.e., containing tasks only that are valid) relies on the correctness of the power viewpoint (i.e., the view generation algorithm), which takes a map and a set of required task types, and outputs a view. The implementation of this viewpoint has been iteratively refined based on the verification failures due to views (the issues are listed in Table 8.1 in Subsection 8.1.2), to the point where the viewpoint produces sound views in practice. In terms of missions, any sequence of tasks that is constrained to be contiguous and non-self-intersecting is considered a mission, in accordance with Definition 31. Missions with certain types of tasks are modeled with additional constraints: charging should only happen if the robot had enough charge to arrive at a charging station, and rotation tasks need to have their starting orientation coincide with the robot's current orientation. All of these constraints are expressed within integration properties in this case study, thus making any sequence of tasks from V_{power} that fits these constraints a valid mission.

The *completeness* of V_{power} in terms of tasks is based on the expressiveness of V_{power}. As discussed above, V_{power} can encode any task that is relevant for the robot on a given map, thus leading to a complete view in terms of tasks. In terms of missions, given a sufficient number of quantified variables, any finite mission with first-order logic constraints can be encoded for a given



```
properties

-- Forward motion tasks
-- Assumption:start/end locs are IDs of locations, not refs to them
Robot_Task_Properties::task_id => 0 applies to m_l1_to_l2;
Robot_Task_Properties::start_loc => 0 applies to m_l1_to_l2;
Robot_Task_Properties::end_loc => 1 applies to m_l1_to_l2;
Robot_Task_Properties::start_head => 2 applies to m_l1_to_l2;
Robot_Task_Properties::end_head => 2 applies to m_l1_to_l2;
Robot_Task_Properties::energy => 673 applies to m_l1_to_l2;
Robot_Task_Properties::task_type_enum => Forward applies to m_l1_to_l2;
Robot_Task_Properties::task_type => 0 applies to m_l1_to_l2;

Robot_Task_Properties::task_id => 1 applies to m_l2_to_l1;
Robot_Task_Properties::start_loc => 1 applies to m_l2_to_l1;
Robot_Task_Properties::end_loc => 0 applies to m_l2_to_l1;
Robot_Task_Properties::start_head => 6 applies to m_l2_to_l1;
Robot_Task_Properties::end_head => 6 applies to m_l2_to_l1;
Robot_Task_Properties::energy => 673 applies to m_l2_to_l1;
Robot_Task_Properties::task_type_enum => Forward applies to m_l2_to_l1;
Robot_Task_Properties::task_type => 0 applies to m_l2_to_l1;

Robot_Task_Properties::task_id => 2 applies to m_l2_to_l3;
Robot_Task_Properties::start_loc => 1 applies to m_l2_to_l3;
Robot_Task_Properties::end_loc => 2 applies to m_l2_to_l3;
Robot_Task_Properties::start_head => 2 applies to m_l2_to_l3;
Robot_Task_Properties::end_head => 2 applies to m_l2_to_l3;
Robot_Task_Properties::energy => 994 applies to m_l2_to_l3;
Robot_Task_Properties::task_type_enum => Forward applies to m_l2_to_l3;
Robot_Task_Properties::task_type => 0 applies to m_l2_to_l3;
```

Figure 8.4: Task properties and their values in an instance of V_{power}.

map. In this case study, for any set of mission features, the intended set of missions were possible to encode with first-order constraints. However, some constraints required direct encoding of multiple conditional branches, leading to large IPL specifications (dozens of lines/logical atoms). For instance, if any task in a sequence of N quantified task variables can be a charging task, then N conditions on the pre-task battery state need to be written, comparing the sum of energies from the preceding tasks to the minimal threshold required for charging. Thus, completeness of views can be in conflict with applicability, leading to large specifications.

The *applicability* of power views encountered two challenges for this system. The first applicability challenge for V_{power} is that certain tasks and missions have implicit constraints (that are automatically satisfied in M_{plan}). For instance, the robot does not always start facing the direction of its first move, and an extra rotation task is needed to encode a mission. Another example is that in some versions of M_{plan} only allow charging once the battery charge is below a certain level, and the aforementioned constraints are needed. These caveats led to V_{power} generating some missions that cannot be executed in M_{plan}, and hence unsatisfied integration properties. However, as mentioned before, IPL allows arbitrary logical specifications over views, these constraints are satisfied with additional variables and constraints for the rigid part of IPL properties.

The other applicability challenge is that views with different mission features (e.g., charging) and robot modes (e.g., the mode of the visual sensing) of the robot may be required for integration with the same M_{plan}. This variability is handled using different instances of viewpoints: each mode and each combination of features is encoded as a separate viewpoint instance, creating multiple views for a combination of M_{power} and M_{map}. As a result, each view of M_{power} has to be compared with an appropriate version of M_{plan} that sets the same mode and uses the same mission features. Thus, this applicability challenge is addressed using the flexibility of views and viewpoints.

Finally, V_{power} is *customizable* in terms of new mission features, which can be represented as new task types or new properties. For instance, if the robot was augmented with a manipulator, one would be able to add manipulation tasks to V_{power} without changing the rest of the tasks. It is also possible to add new properties of tasks, such as the time taken by a task or the total distance traveled while executing a task. This customizability is due to the extensibility of architecture description languages, in particular AADL.

Behavioral Properties for System 1

In the study of model integration for System 1, I used behavioral properties specified in PCTL for M_{plan} to constrain the robot to a particular mission. When constrained to a given mission, the robot would have to use the energy that V_{power} requires for it (i.e., the sum of energies of all the tasks). Then, the mission is power-successful according to M_{plan} if and only if the robot reaches the goal location (within M_{plan}).

As an example of a behavioral property, consider a PCTL query that is part of Property 4. This query expresses the probability (maximized by choosing the optimal actions in non-deterministic MDP transitions) of a robot completing a mission of four consecutive tasks $t_1...t_n$, starting in the beginning of the first task ($t_1.start$) with a sufficiently charged battery.

$$P_{max=?}[(\mathsf{F}\underline{loc} = t_2.startloc) \wedge (\mathsf{F}\underline{loc} = t_3.startloc) \wedge (\mathsf{F}\underline{loc} = t_4.startloc) \wedge \tag{8.6}$$

$$((\underline{loc} = t_1.start)\mathsf{U}(\underline{loc} = t_2.start\mathsf{U}(\underline{loc} = t_3.start\mathsf{U}\underline{loc} = t_4.end)))] \tag{8.7}$$

$$\{|\underline{initloc} = t_1.start, \underline{goal} = t_4.end, \underline{initbat} = \Sigma_{i=1}^{4} t_i.energy + \overline{err\_cons}|\} \tag{8.8}$$

The expectation of *expressiveness* for behavioral properties is as follows: the behavioral language should provide sufficient means to express the integration constraints/queries so that the model does not need to be manually changes outside of the behavioral language. In System 1, the expressiveness challenge for behavioral properties of M_{plan} is two-fold: (i) constraining the robot's actions to the individual tasks from quantified variables, and (ii) enforcing the ordering of the tasks that constitute the mission. For motion-related tasks (forward, rotation, and empty tasks — all but charging tasks), challenge (i) is addressed by expressing the sequence of locations with nested until operators: M_{plan} to constrains the robot to make necessary moves in order to visit the locations and make the nested expression hold. However, the nested until operators do not fully address (ii) because the semantics of PCTL allow the robot to skip intermediate locations of the mission. For instance, the constraint $\underline{loc} = l_1\mathsf{U}(\underline{loc} = l_2\mathsf{U}(\underline{loc} = l_3))$ is satisfied both by the intended mission $l_1 \rightarrow l_2 \rightarrow l_3$ and by an unintended mission $l_1 \rightarrow l_3$ (assuming that there is an edge between l_1 and l_3). Therefore, to address (ii), additional clauses with future (F) modalities are added to the formula.

Charging tasks are expressed implicitly in PCTL properties: if a robot visits a charging station, it gets a choice of charging in M_{plan}; due to the operator $P_{max=?}$, the robot is forced to charge if this charging would enable it to reach the goal. This way, the robot can charge at any location with a charging station without specifying additional constraints in PCTL. Thus, PCTL properties, augmented with an initialization clause, have been found sufficiently expressive to constrain the robot to a mission from V_{power} in this case study.

The *soundness* of behavioral queries is based to the tools that implement the behavioral queries. For M_{plan}, queries are checked by the probabilistic model checker PRISM. Depending on the configuration of PRISM, some queries may not terminate (in practice, running out of time or memory), but any valid terminating query is guaranteed to provide a correct result. Thus, the checking of PCTL formulas with PRISM is sound without guarantees of termination.

The *applicability* of PCTL queries in this study faced the challenge of termination on some properties, particularly with multiple nested until operators. The issue occurred in the process of converting a behavioral property (specifically, its LTL part within a probabilistic operator) into a deterministic automaton, which would be composed with M_{plan}. PRISM uses third-party tools to perform this conversion, and these tools differ in their efficiency on different formula classes. The default tool in PRISM, called *ltl2dstar* [128], showed impractically long times (several minutes) and memory consumption (over one gigabyte) for a conversion of one property. By using a different conversion tool, Rabinizer 3 [129], it was possible to make behavioral checks applicable to all the formulas in this study with, times comparable to SMT checking (see Table 8.2 in Subsection 8.1.2). Thus, the lack of formal termination guarantees has manifested in limited out-of-the-box applicability in the case of PCTL properties.

Customizability of behavioral properties is determined by the behavioral model and the logic that describes it. In the System 1 study, the logical operators were determined by PCTL and the PRISM input language. These operators sufficiently constrained M_{plan} for the purposes of this integration scenario. The initial and state variables of M_{plan} have also been sufficient to express the constraints. Thus, the customization of behavioral properties has not been required in this study. Instead, the ability to plug in modal logics contributes to the customizability of IPL, as discussed in Subsection 8.1.2.

8.2.2 Evaluation of Integration Abstractions on System 2

The investigation of integration abstractions for the collision avoidance system was part of a independent study, the goal of which was to find appropriate integration abstractions for hybrid programs and $\mathsf{d}\mathcal{L}$ properties. In particular, the abstractions should connect hybrid programs to component-based integration approaches. This objective is challenging because the hybrid programs are not designed for an explicit component structure.

The integration abstractions in this study have complementary roles: views represent HPs to enable integration with component-based models (e.g., a hardware model described in Section 6.1), and $\mathsf{d}\mathcal{L}$ view formulas enabling reasoning about hybrid programs through the view abstractions. Thus, integration is achieved without sacrificing model-specific analysis — theorem-proving for $\mathsf{d}\mathcal{L}$ specifications with KeYmaera [81].

The integration abstractions for hybrid programs were defined earlier in this thesis: a description of HP views can be found in Subsection 6.2.3, and a description $\mathsf{d}\mathcal{L}$ view formulas is located in Subsection 6.3.2. Below I explain how these abstractions enable the four qualities of integration considered in this thesis, with the following interpretations in this context:

- *Expressiveness:* whether HP views have sufficient means to represent common hybrid programs; and whether $\mathsf{d}\mathcal{L}$ view formulas are sufficient to represent commonly occurring $\mathsf{d}\mathcal{L}$ properties.
- *Soundness:* whether HP views are a sound and complete representation with respect to operators in HPs; and whether the checking of $\mathsf{d}\mathcal{L}$ view formulas is sound and terminates.
- *Applicability:* whether HP views contain the information necessary to generate HPs in practice.
- *Customizability:* whether views can be customized to represent HP concepts; and whether

$\mathsf{d}\mathcal{L}$ specifications can be tailored for HP views.

This evaluation was performed on a set of model variants from a related robotic collision case study (described in Section 3.2) [173, 174]. To support this evaluation, a prototype tool for creation and analysis of these abstractions was implemented based on AcmeStudio [215]. The tool and models are available in an archive [212].

Expressiveness

The *expressiveness* of integration abstractions for HP is evaluated separately for HP views and $\mathsf{d}\mathcal{L}$ view formulas (a language for behavioral properties of HPs, see Definition 23 in Subsection 6.3.2). Starting with the views, they should to reflect the common varying parts of hybrid programs: actors (i.e., robots and obstacles), sensing, control, physical dynamics, and so on. In other words, the viewpoint for constructing views of HPs (VP^{HP}) needs to be sensitive to variance in the HPs. The important aspects of this variance for robotic collision avoidance are summarized in Table 3.1 in Section 3.2. If these aspects were not shown in abstractions (i.e., the abstractions of different models were the same), then integration would lack the expressiveness necessary to identify inconsistencies. On the other hand, if abstractions were completely different for similar models, they would not adequately display the common patterns in HPs that affect integration.

As an example of variance between HPs, consider the robot's possible physical dynamics in Table 8.3. When relating HPs to other models, it is important to distinguish between these dynamics at a higher level of abstraction. In the simplest case, a robot is moving with velocity v and acceleration a along a line in a binary orientation $o \in \{1, -1\}$. A slightly more complicated case is with movement along a grid net, defined by directions $o_{fb}, o_{hv} \in \{1, -1\}$, and a line with direction defined by $dx, dy \in [0; 1]$. Modeling movement in arcs of fixed radius r requires representing rotational velocity ω and linking it to a. To enable spinning on a single spot ($r = 0$), $w' = \frac{a}{r}$ needs to be rewritten with a new helper variable s as $s' = a, s = wr$, introducing yet another physical model. Finally, the model of spiral movement does not link rotational velocity ω with a.

To represent the differences in control and physics between variants, I used component types for HP actors. In this case, an *actor type* is a partially specified actor (Definition 16). Thus, a fully specific actor can be composed from an arbitrary number of types. Thus actor a satisfies types $a : \mathcal{A}$ and $a : \mathcal{B}$ if[6]:

$$\begin{aligned}
&\mathcal{A}.State \cup \mathcal{B}.State \subseteq a.State,\\
&\mathcal{A}.Ports \cup \mathcal{B}.Ports \subseteq a.Ports,\\
&\mathcal{A}.Phys, \mathcal{B}.Phys \subseteq a.Phys.
\end{aligned}$$

Type extension is equivalent to having both types: $a \in (\mathcal{A} \sqsubseteq \mathcal{B}) \equiv a \in \mathcal{A} \wedge a \in \mathcal{B}$. This approach enables representation and reuse of varying elements of HPs. For example, notice that spiral dynamics (Tab. 8.3) is a more general case of arcs w/o spinning, so we extend the former with the latter: $\mathsf{ArcNoSpinDynT} \equiv \mathsf{SpiralDynT} \cup (\emptyset, \emptyset, \emptyset, \{w' = \frac{a}{r}\})$. Then $\mathsf{SpiralDynT}$ is reused every time an actor is declared with it or $\mathsf{ArcNoSpinDynT}$.

[6]The control property *Ctrl*, however, cannot be composed from multiple types: we demand that there is a single source of controller, be it an actor instance or one of actor types.

1D Line	$x' = ov, v' = a, v \geq 0$
Grid	$x' = \frac{(1+o_{hv})o_{fb}}{2}v, y' = \frac{(1-o_{hv})o_{fb}}{2}v, v' = a, v \geq 0$
Line	$x' = vd_x, y' = vd_y, v' = a, v \geq 0$
Arcs w/o spin	$x' = vd_x, y' = vd_y, d_x' = -wd_y, d_y' = wd_x, v' = a, w' = \frac{a}{r}, v \geq 0$
Arc w/ spin	$x' = vd_x, y' = vd_y, d_x' = -wd_y, d_y' = wd_x, v' = a, s' = a, s = wr, v \geq 0$
Spiral	$x' = vd_x, y' = vd_y, d_x' = -wd_y, d_y' = wd_x, v' = a, v \geq 0$

Table 8.3: Versions of the robot's physical dynamics.

Variance in actor interactions can be represented with an HP connector, which determines the Trf function (see Subsection 6.2.3 for its definition). This connector encapsulates modeler's expertise about common transformations of the actors' programs, such as IPS or IBES. Instead of manually introducing new variables, constraining them, and weaving into the code, a modeler achieves this with a HP connector automatically. For example, in the *ISect* program, an engineer can derive a variant with uncertain awareness of the obstacle's position by replacing IPS with IBES, accomplishing this task more conveniently than by manually changing a large hybrid program.

The evidence of reusing the architectural types in HP views supports the claim of expressiveness. Many primitive fragments of hybrid programs, such as LCT and *SeqC*, contributed in all model variants. An HP view was reused as a whole three times in a model variant where the robot reaching the goal was modeled at different time moments. This demonstrates utility of architectural d$\mathcal{L}$ formulas. Among types that described robot's physics, 1D line, grid, and arc movement types were used three, three, and five times respectively. Thus, physical commonalities are a fruitful target for reuse. Surprisingly for us, HP connectors were used 28 times (IPS being most common) – almost twice in each model – demonstrating the tremendous amount of component interaction that the architecture made explicit. Overall, this reuse shows that HP views are capable of comparing and differentiating the dimensions of variance in hybrid programs.

For d$\mathcal{L}$ view formulas, the expressiveness means the capacity of these formulas to describe the common logical properties of HPs. Among the HP models in this case study, I found that 10 out of 12 properties used one of two simple patterns:

$$\phi \rightarrow [\mathsf{V}^{HP}]\psi$$
$$\phi \rightarrow \langle \mathsf{V}^{HP} \rangle \psi,$$

where ϕ is the precondition for the program, ψ is a postcondition, and V^{HP} is a view of a hybrid program.

The remaining two properties required more than one HP. Consider a property for passive friendly safety (defined in Section 3.2). This property requires that for all executions of a robot and a moving obstacle [RobotObst], a robot should always stop or be far enough from the obstacle to stop ($RobotFar$). Assuming the obstacle ⟨DetailedObst⟩ is far enough from the robot ($ObstFar$), should have an opportunity to stop and avoid collision ($Safe$). The following formula (part of the original modeling effort [172]) captures this property:

$$\begin{aligned} Pre \rightarrow [\mathsf{RobotObst}](&RobotFar \wedge \\ &(ObstFar \rightarrow \langle \mathsf{DetailedObst} \rangle Safe)) \end{aligned} \tag{8.9}$$

This $\mathsf{d}\mathcal{L}$ formula includes two hybrid programs that execute independently: once RobotObst, which contains a robot and a non-deterministic obstacle, stops at some point, program DetailedObst starts executing. DetailedObst does not have robot's code in it explicitly (it is assumed to be stopped), but has a refined model of an obstacle that is capable of braking and accelerating unlike the one in RobotObst. The two hybrid programs share some of their variables, such as the obstacle's position, and the initial constraints of these two programs are mixed in Pre.

Using the HP views and $\mathsf{d}\mathcal{L}$ view formulas, Equation (8.9) can be specified with two HP views RobotObst and DetailedObst, each of which corresponds to a hybrid program, and one formula that conjoins views' $Constr$ in Pre. The other conditions ($RobotFar$, $ObstFar$, and $Safe$) are part of the $\mathsf{d}\mathcal{L}$ view formula and reference the variables from the views. Since views have disjoint state spaces, extra statements are necessary to relate the state of $RobotObst$ after finishes execution, as well as the state of $DetailedObst$ before it starts execution.

Thus, I found that all the properties from the case study could be represented using HP view formulas. This demonstrates that the expressiveness of property specifications is preserved after creating integration abstractions. This finding, combined with the evidence of reuse for actor connector types, supports the claim that the integration abstractions for hybrid programs are sufficiently expressive for integration purposes.

Soundness

Soundness and *completeness* of *HP views* depends on the implementation of VP^{HP}. This viewpoint is implemented according to Option (ii) described in Subsection 6.2.4: a model can be generated from a given view automatically, and views have to be created for given models manually. Soundness and completeness are evaluated based on the mapping between HP model elements (variables and operations) and HP view elements (HP actors, connectors, and composers).

Before assessing soundness and completeness, the first step is to guarantee conformance. Given a view V^{HP} and a hybrid program M, I use V^{HP} to generate a conforming model M' according to Definition 20, and check whether M is equivalent to M'. All 15 views were found conforming to the original models.

Soundness of VP^{HP} means that, informally, every element of an HP view maps to some element in an HP. Since the views have been established by splitting the original HPs into actors and connectors, it is only necessary to check that every variable and operator is present in some view. All 15 views were found sound with respect to VP^{HP}.

When the viewpoint is complete, every model element is guaranteed to be represented in its view. Since the views have been established by splitting the original HPs into actors and connectors, it is only necessary to check that every variable and operator is present in some view. All 15 views were found complete with respect to VP^{HP}.

Soundness and *termination* of checking of *$\mathsf{d}\mathcal{L}$ view formulas* are dependent on the techniques and tools for hybrid programs: HP views generate HP models, thus reducing $\mathsf{d}\mathcal{L}$ view formulas to regular $\mathsf{d}\mathcal{L}$ formulas. This study relied on interactive theorem proving with KeYmaera. This technique does not guarantee termination for arbitrary programs and formulas: human assistance may be needed to complete some proof branches. However, once the proof is completed, it is guaranteed to hold on the HP generated by the view. Thus, if a view is sound and complete, then the checking of $\mathsf{d}\mathcal{L}$ view formulas is sound.

To summarize, the manual process behind creating views makes it impossible to guarantee soundness of VP^{HP} theoretically. However, this study has shown that careful creation of views leads to sound and complete abstractions. With theorem-proving as the technique for checking $\mathsf{d}\mathcal{L}$ view formulas, behavioral queries are guaranteed to be sound, but not necessarily terminate automatically.

Applicability

The central applicability challenge for *HP views* was to find a decomposition of a hybrid program into actors that maximizes cohesion (i.e., having actors with closely related state variables, control, and physics) and minimizes coupling (i.e., the number of connectors between actors). Most hybrid programs in the study had straightforwardly described actors (robots and obstacles), but also had variables and operators that propagated to many parts of the program and, hence, were difficult to encapsulate in any actor.

As an example of such propagating variables, consider different patterns for modeling time in HPs:

- Event-triggered timing (ETT). Time is not represented in a model as a variable. Instead, event conditions are part of an evolution domain constraint F. For example, the system can execute until the robot has to brake, which is when time flow is interrupted and the control is handed to the robot.
- Local continuous timing (LCT) with bounded non-deterministic intervals. The timer is reset in the discrete part of model loop $t := 0$ and increases monotonically longer than ε: $\{t' = 1, t \leq \varepsilon\}$.
- Global continuous timing (GCT). To verify liveness properties global progress towards a goal needs to be tracked. In this case, a global timer is initialized $T := 0$ and evolved continuously without resets $\{T' = 1\}$. Global timing may be combined with event-triggered or local continuous timing.

Each of these patterns impacts multiple parts of a model. If one chooses to use, for instance, local continuous timing, then a number of changes must be made throughout the model: first of all, t needs to be reset in the loop, but the spot needs to be carefully chosen depending on whether other parts of the loop use t. Second, the differential equations and evolution domain constraints need to be updated. Furthermore, t and ε need to be added to the variable and constant declarations, respectively. Finally, control decisions are very likely to change to accommodate a possible delay of ε seconds. Thus, timing is an aspect that is difficult to encapsulate in a single actor, which would need to be connected to all other actors, adding to the view complexity. Nevertheless, timing needs to be represented for views to be complete.

To address this challenge, I introduced a global actor GlobalHPA, represented by the high-level *system* component in Acme, Part of every HP view, GlobalHPA has its variables visible to all other actors. To reuse timing patterns with types $LCT \equiv ((\{t, \varepsilon), \{\varepsilon \geq 0\}), \emptyset, t := 0, \{t' = 1, t \leq \epsilon\})\}$ and $GCT \equiv ((\{T\}, \emptyset), \emptyset, \emptyset, \{T' = 1\})$, we let GlobalHPA : LCT or GlobalHPA : GCT to ensure consistent timing without the need to create a connector to read t, T, or ε. This approach enabled complete views and reusable specifications for timing without introducing a dedicated timing actor with connectors to all the other actors.

However, in some HPs cohesion of actors was limited: large portions of similar hybrid code were "trapped" in the robot controllers because the controllers differed in the way they addressed specific aspects of the variant, such as environment assumptions and uncertainty in sensing or actuation. This limitation is, however, not fundamental: one can use types on top for statecharts that encapsulate control algorithms, as it is done in Sphinx [173].

Application of d$\mathcal{L}$ *view formulas* was more straightforward than that of HP views: once the views were created, the d$\mathcal{L}$ formulas wrapped the original properties around the views, without any customization or redesign. Scalability was also not impacted, retaining the same complexity as the original models.

To summarize, the applicability challenges in this study revolved around fully decomposing hybrid programs into views. It was possible to encapsulate time in a global actor, although large controller code remained part of some actors without further modularization or reuse. Encoded as d$\mathcal{L}$ view formulas, behavioral properties did not encounter substantial applicability challenges in this study.

Customizability

The main focus of this study was whether views can be *customized* to represent hybrid programs. Therefore, in this study customizability was linked to expressiveness. As discussed earlier, this study demonstrated a creation of a customized viewpoint VP^{HP}, which has been tailored to the formalism of hybrid programs. The architectural notions of components and connectors were customized to encode the rules of composition typical for hybrid programs. Furthermore, the types were used to represent and reuse common specification patterns. VP^{HP} may be refined for other HP modeling projects to further tailor the integration abstractions by, for instance, creating new types of actors or connectors may be added to represent domain-specific interactions and dynamics.

This study has also shown that behavioral properties are not mutually exclusive with views, and can be used together. Specifically, the original d$\mathcal{L}$ formulas were customized to specify properties for views. To check d$\mathcal{L}$ view formulas, HPs are generated from the views. Therefore, the same model can have both a view abstraction and a behavioral property abstraction. The practical implication is that views can replace models as engineer-facing artifacts, eliminating the effort required for co-evolution (described in Subsection 6.2.4).

To summarize, the study of integration abstractions for robotic collision avoidance has demonstrated the customizability of views and behavioral properties in the context of modeling hybrid programs and their $\mathsf{d}\mathcal{L}$ properties.

8.2.3 Evaluation of Integration Abstractions on System 3

An investigation of integration abstractions for a quadrotor was conducted as part of the application of analysis execution platform to the analyses applicable to this system. The system description and a list of analyses/models can be found in Section 3.2. The main focus of the study was on specifying and checking the contracts for six analyses in the domains of thread scheduling and battery design, while integration abstractions provided convenient representations of the models. This section focuses only on the abstractions, while the contracts are described in Subsection 8.3.1.

The models were inspired by the literature and constructed manually (see Subsections 8.3.1 and 8.3.1), as opposed to being taken from an existing set of data. This circumstance makes *soundness* of views a priori satisfied. Hence, only and checking-related aspects of soundness are evaluated in this context, along with the other three integration qualities, using the following interpretations:

- *Expressiveness:* whether the views can capture the static information related to thread scheduling and battery design; and whether LTL properties can capture the necessary constraints on behavioral dynamics in order to integrate the analyses correctly.
- *Soundness:* whether the checking of LTL properties can deliver a correct evaluation within a finite amount of time.
- *Applicability:* whether the views and LTL properties can be combined, while maintaining a consistent shared interpretations; and whether the checking of LTL properties can be done within realistic times.
- *Customizability:* whether the views and LTL properties can be tailored to represent the concepts of battery scheduling and battery design.

Views for Schedulability and Battery

Views were applied as a uniform representation of multiple models related to schedulability and battery. I created several views in this study, all of which were specified in AADL:

- The scheduling view (V_{sch}) for M_{sch} contains Threadsas components with Dline, their periods, and worst-case execution times as their properties.
- Data security view (V_{sec}) for M_{sec} contains Threadsas components with their security levels marked.
- The CPU view (V_{cpu}) for M_{cpu} contains CPUsas components, with CPUFreq, $\mathsf{CPUFreq}_{\mathsf{max}}$, and CPUBind as component properties.
- The Rek view (V_{rek}) has the same viewpoint as V_{sch}: it represents Threadsand their properties.
- The thermal runaway view (V_{tr}) contains the thermal parameters of the battery: number of cells and whether it is safe from the thermal runaway.

- The battery scheduling view (V_{tr}) contains the electrical parameters of the battery: required voltage, number of cells, and its scheduling mode.

Some of the above views contain redundant information, e.g., threads are found both in V_{sch} and V_{rek}. To eliminate the redundancy, the views were merged into a single view that is a union of all the architectural elements from each view. This provides a simple consistency check, in case the models have conflicting structural elements. All of the view creation and merging was manual for this system, since automation was not the focus of this study.

A challenge for both *expressiveness* and *applicability* here is that views capture only static information, without recording any behavioral information. This challenge was addressed by separating the behavioral models from views, and expressing behavioral constraints in LTL (which is discussed in the next section). The domains of thread scheduling and battery design turned out to be well-suited for modeling structural information in views, using properties found in Tables 8.6 and 8.7 in Subsection 8.3.1. The behavioral dynamics were modeled separately with Spin models for thread scheduling and battery scheduling, discussed in Subsections 8.3.1 and 8.3.1 respectively. This separation led to views being sufficiently expressive for the static elements.

Performance of SMT checking was not an obstacle for applicability in this study. Due to the small size of views (dozens of elements) and a small space of solutions for rigid parts of IPL formulas (presented in Subsection 8.1.3), SMT performance on the part of IPL was near-instantaneous: an SMT query for a merged view took less than 1 second.

In terms of *customizability*, this study has shown that views can be tailored to two domains: thread schedulability and battery schedulability. As mentioned above, these domains used well-defined structural schemas for design information (e.g., recording periods, deadlines, execution times for each thread, and size parameters for batteries), and views were usable for these domains by customizing the types and properties. Thus, this customization relied on extensible types and properties of architecture description languages, similar to other applications of views.

Soundness has not been evaluated for the views in this case study: they served as an interchange medium for information, rather than an abstraction for other models. Thus, by construction the views represented the ground truth of the system's design, and hence sound and complete.

Behavioral Properties in LTL

To query behavioral aspects of Spin models in this system, I used LTL properties to place constraints on the behaviors of thread schedulers and batteries. One property, which queries M_{sch}, was presented in Subsection 8.1.3. This property provides a way to reason about preemption patterns by using the model's reasoning engine (Promela), without encoding all possible behaviors in the views. Another property for M_{sch}, (fixed-priority scheduling) was expressed in Property 13. Similarly, it is checked using a Promela engine on the model of the scheduler.

Another LTL property was used for querying M_{bsch}. To prevent the thermal runaway, the following IPL specification iterates through batteries to ensure that thermal neighborhoods support sufficient heat transfer (the terms TN and K are defined in *sec:batt-dom*):

Property 14. All batteries have the numbers of thermal neighbors that do not lead to a thermal runaway.

$$\forall b \cdot \mathsf{G}(\mathsf{K}(b,0) \times \mathsf{TN}(b,0) + \mathsf{K}(b,1) \times \mathsf{TN}(b,1) +$$

$$\mathsf{K}(b,2) \times \mathsf{TN}(b,2) + \mathsf{K}(b,3) \times \mathsf{TN}(b,3) \geq 0).$$

Due to using LTL modalities, *expressiveness* of behavioral properties for this system is sufficient for describing temporal changes in the scheduler and the battery. One limitation of standard LTL is that it does not allow comparing values from different states of a trace. This obstacle has been overcome by exposing complex notions as modal functions from M_{sch} (the preemption relation for threads) and M_{bsch} (the thermal neighborhood of battery cells), as opposed to more granular terms, like the current thread executing on a CPU or a cell's current charge. Another assumption has been made that the weights are linear and are a proxy of a thermal runaway, rather than a more complex function of the state that would be harder to express and check in LTL. With a more expressive (and still checkable) behavioral property language, a more precise expression of thermal runaway could be specified.

Soundness of these properties is guaranteed by the Spin model checker: if it returns an answer, it is the correct answer with respect to the model. Theoretically, model checking may not guarantee termination of queries. However, in the models M_{sch} and M_{bsch}, all model checking queries always terminated in this study.

The main challenge of *applying* LTL properties in practice was the model checking time, which grew exponentially with the size of the model (in terms of the number of threads or battery cells). It was found (information below) that the times for verification were adequate to the size of the models used in the study. The advantage of using individual properties is that each model can be checked individually, as opposed to their parallel composition. In this study, the verification time of M_{sch} combined with M_{bsch} would be intractable: various possible interleavings of transitions would lead to state-space explosion.

I evaluated the verification of LTL properties on the two Promela models using a general-purpose Amazon EC2[7] virtual machine with 8 cores and 30 Gb memory. The worst-case exploration times by scheduler for the full state space M_{sch} and M_{bsch} are shown in Table 8.4 and Table 8.5, respectively. For the former the threads with implicit harmonic periods are used, and for the latter the battery size is grown, fixing the output voltage requirement to $\mathsf{SerialReq} = \mathsf{ParalReq} = 3$. Although the complexity and time grows exponentially, $\mathcal{K}(\llbracket \mathcal{T}_{Sched} \rrbracket)$ is verifiable up to 6-10 threads (per CPU), and $\mathcal{K}(\llbracket \mathcal{T}_{Therm} \rrbracket)$ is verifiable up to batteries with 25 cells. These numbers match the scale of a realistic CPS. Moreover, the memory issues can be mitigated by increasing the RAM size, so it is not a fundamental limitation.

A secondary applicability challenge was to ensure a non-contradictory shared interpretation of thread IDs, for checking properties based on M_{sch}. Constructing a consistent background interpretation is a responsibility of integration abstractions described in Subsection 6.3.4, in order to enable domain transfer in IPL (Subsection 5.5.1). In this study, for instance in Property 12 described in Subsection 8.1.3, SMT found pairs threads for evaluating their deadline-monotonicity in M_{sch}. The threads' IDs had to be passed from V_{sch} to M_{sch}. To enable this transfer, I implemented thread IDs as a shared property of view components and Promela processes, part of I^B shared by V_{sch} and M_{sch}, leading to consistent referencing of threads between rigid and flexible parts of IPL properties.

[7]aws.amazon.com/ec2.

[8] All times are in seconds. MEMLIM indicates that the verification exceeded the memory limit of 30Gb.

Threads	DMS/RMS Time[8]	EDF Time
3	0.01	0.01
4	0.01	0.52
5	0.07	33.4
6	0.37	2290.0
7	2.18	MEMLIM
8	12.4	MEMLIM
9	71.2	MEMLIM
10	421	MEMLIM
11	MEMLIM	MEMLIM

Table 8.4: Scalability of the $\mathcal{K}(\llbracket \mathcal{T}_{Sched} \rrbracket)$ program.

Cells	FGURR Time [8]	FGWRR Time	GPWRR Time
9	0.13	0.15	0.15
12	0.61	2.34	3.94
16	44.0	31.4	127
20	1060	619	MEMLIM
25	MEMLIM	MEMLIM	MEMLIM

Table 8.5: Scalability of the $\mathcal{K}(\llbracket \mathcal{T}_{Batt} \rrbracket)$ program.

Customizability, in this system, required finding state variables that represent the dynamics that are relevant to integration. The state variables were not simple scalars in this case (like a battery charge in System 1), but instead they were vectors (like numbers of thermal neighbors). To represent them, modally interpreted functions were used. In the property of behavioral deadline monotonicity (see Subsection 8.1.3), such a modal function had to take references to architecture components, which added the domain transfer challenge: converting component references to integers. Thus, it was possible to customize the LTL properties to the needs of this domain.

To summarize, the study of integration abstractions for thread/battery scheduling in a quadrotor has demonstrated that views and behavioral properties can be used together, to balance structural/behavioral aspects of specifying multi-model properties. Both abstractions were found applicable and customizable for the two domains. These integration abstractions support integration of analyses, presented in Subsection 8.3.1.

8.2.4 Evaluation of Integration Abstractions on System 4

An investigation of integration abstractions for an autonomous car (described in Section 3.4) was conducted as part of the application of analysis contracts and the analysis execution platform to this system. The main focus of the study was on specifying and checking contracts for reliability and security analyses (described in Subsection 8.3.2), while views provided a convenient representation of models for these contracts. This section focuses only on evaluation of views since no behavioral properties were used in the contracts.

The models for the analyses in this study were inspired by the literature (see Section 3.4), as opposed to being taken from an existing set of data. Therefore, the views were a priori a ground-truth representation, yielding no insight into their *soundness* (in relation to other models). Hence, only *expressiveness*, *applicability*, and *customizability* were evaluated in this context, with the following interpretations:

- *Expressiveness:* whether views can express the notions necessary to analyze the reliability and trustworthiness of sensing in a self-driving car.
- *Applicability:* whether views can practically represent the sensors, controllers, and their relevant properties.
- *Customizability:* whether views can be tailored for the needs of specifying analysis contracts in this system/domain.

Views for Reliability, Trustworthiness, and Control

In this study, the modeling goal for views is to represent structural elements that are relevant to inter-domain vulnerabilities. The interactions of analyses in this study are determined by the static information from M_{fmea}, M_{trust}, and M_{ctrl}, which I encode the respective AADL views: V_{fmea}, V_{trust}, and V_{ctrl} (see Subsection 8.3.2 for their definitions). The views are built on top of an existing collision detection and avoidance AADL model for an autonomous vehicle created by McGee et al. [161]. The original model contains a number of sensors, processing units (hardware devices and control threads), actuators, and other car components, organized into several functional subsystems: collision prediction/avoidance/response, networking, user interaction, and physical devices (various sensors, brakes, airbags, radio, and so on). We enhance this model by adding a lidar and C2C sensors for distance and a magnetic speedometer with GPS for velocity measurement [9].

The AADL views consist of architectural elements and their properties, which are defined using AADL data types, component types, and custom properties. I use AADL modes to encode different configurations under the different failures of the system using state machines, as described in Section 3.4. Mode examples are given in the rows of Table 3.3. Each mode m contains a full system architecture: sensors ($m.\mathbb{S}$), controllers ($m.\mathbb{R}$), and actuators [10]. Usage of modes differentiates these views from those for System 3 (Subsection 8.2.3), since the dynamics in System 3 were complex enough to warrant using a model through behavioral properties. In contrast, the dynamics of changing attackers is relatively simple in System 4 and can be encoded as modes.

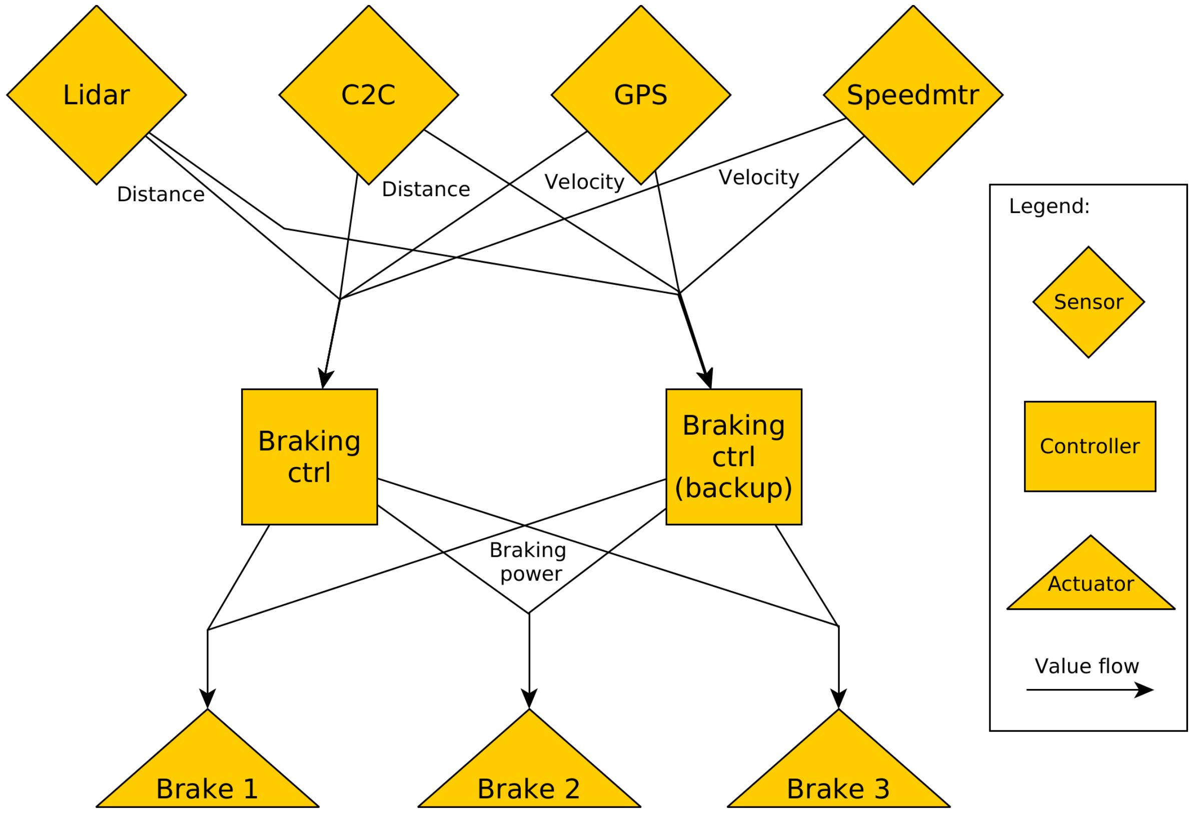


Figure 8.5: Braking architectural view in a self-driving car.

To simplify checking of contracts, all views are merged into a single view (see Figure 8.5), by merging identical elements from various views. This operation has been performed manually. The following elements comprise the resulting architectural model:

[9]The AADL model with analysis contracts has been archived [211] and is also available online (`github.com/bisc/collision_detection_aadl`).

[10]Actuators are critical components of the system, but we do not model them explicitly because our focus is on interaction between sensors and controllers.

- Sensors ($\mathbb{S}$) have the following properties:
  - Sensed variables $\mathsf{VarsS} \subseteq \mathbb{V}$: the variables for which the sensor can provide series of values. For example, a speedometer provides values for velocity. Some sensors may provide several variables, e.g., GPS values can be used to compute both the absolute position and distance to an obstacle.
  - Power status Pow (boolean values: $\mathbb{B} \equiv \{\top, \bot\}$): whether the sensor is turned on by the user or engineer.
  - Availability Avail (■=■: whether the sensor is providing data. This property does not presuppose that the data is trustworthy or compromised.
  - Trustworthiness Trust (■=■: whether the sensor can be compromised by the attacker and is sending untrustworthy data. We use this boolean abstraction of trust for demonstrating how a vulnerability is introduced. For sensors with $\mathsf{Trust} = \bot$ we assume that an attacker can compromise them in any quantity and at any point of time. Even with this relatively simple abstraction we showed an exploitation of vulnerability in Table 3.5. More sophisticated models may consider numeric or multidimensional trustworthiness [185] for more precise estimation of confidence in sensor data.
  - Probability of mechanical failure $\mathsf{P}_{\mathsf{fail}}$ (%): the probability of a sensor mechanically malfunctioning and remaining broken ($\mathsf{Avail} = \bot$) within a unit of operation time (e.g., an hour or a day).
  - Sensor placement Place (internal or external): the sensor may be located on the outer perimeter of the car and facing outwards, or on the inside perimeter and not exposed to the outside world.
- Controllers ($\mathbb{R}$) have the following properties[11]:
  - Required variables $\mathsf{VarsR} \subseteq \mathbb{V}$: the variables for which the controller should receive values from sensors. For example, the automated braking controller should receive velocity and distance to the closest obstacle on the course.
  - Power status Pow (■=■: analogous to sensors, whether the controller is turned on by the user or engineer.
  - Availability Avail (■=■: whether the controller is functioning and providing output to actuators. This property does not presuppose that the control is safe or uncompromised.
  - Safety of control $\mathsf{CtrlSafe}$ (■=■: whether the controller meets the control performance, safety, and stability requirements.
- System modes $\mathbb{M}$ (i.e., different configurations) have the following properties:
  - Required fault-tolerance α_{fail} (%): the maximum acceptable probability of the system's random failure. The final design is expected to malfunction less or equally likely than α_{fail}.

[11] Although controllers are physical elements and can be attacked, for this system I focus on sensor attacks and assume that direct controller attacks do not happen. Since controllers are typically not exposed to the physical world, their attacks would require an access to the internal car network, leading to a powerful attacker and trivial security analysis.

- Attacker model $\mathsf{AttackM}$ (internal or external): the type of the attacker considered in the system design. For simplicity, we consider only one dimension, that is, internal or external attacker. If required, we could model other dimensions such as local or remote attacker. Each attacker model contains a sensor vulnerability evaluation function $\mathsf{IsVuln} : \mathbb{S} \to \mathbb{B}$ that determines whether a particular sensor can be attacked by this attacker. This function abstracts out technical and operational aspects of attacks in order to represent the relationship between attackers and sensors. For example, the vulnerability function for a powerful adversary in Table 3.4 is $\mathsf{IsVuln} \equiv \top$.

Qualities of Integration

The *expressiveness* provided by views is sufficient for this system because the views captured all the information necessary for integration of analysis: no behavioral properties were needed to express appropriate contexts for analyses. The expressiveness of views was extended beyond their standard capacity with modes, which help encode limited dynamism in the system.

This study presents no evidence to support *soundness* of views is not considered in this case because the views are created as a single interchange medium, and are sound and complete by definition.

The views did not face any outstanding *applicability* challenges in this study: the views successfully captured the information and were analyzable using the standard means of SMT solving. In part, this is due to reusing an existing AADL model with appropriate schemas from prior work.

In terms of *customizability*, this study shows how the standard view concepts can be adjusted to yet another system and domain, making views a flexible integration abstraction. Moreover, views can be extended with the notion of modes, if some dynamism needs to be modeled.

To summarize, the application of views to models of an autonomous car has demonstrated that views are tailorable to new domains with custom information schemas and can incorporate limited dynamic information, to avoid using behavioral properties and simplify verification. This application of views supports integration of analyses, presented in Subsection 8.3.2.

8.2.5 Summary for Evaluation of Integration Abstractions

This chapter described application of integration abstractions to four systems. Across these systems, the views have been found useful as a representations for structural information in models, and behavioral properties were successfully used to query behaviors in models.

The following insights were gained with respect to specific integration qualities:

- *Expressiveness:* using both abstractions and combining them has led to sufficient expressiveness in multiple domains, beyond what either of the abstractions could provide. This finding supports Claim 1. Sometimes expressiveness was improved by modeling domain-specific concepts (e.g., modes in views or modal functions), thus taking advantage of customizability. In some cases, expressiveness was limited to guarantee soundness (in particular, for checking behavioral properties).
- *Soundness:* soundness and completeness of views is critical to integration arguments in the case study systems, and has been achieved by a combination of automated generation,

manual creation/review, and iterative debugging. Soundness of behavioral queries is also an imperative, while termination cannot be guaranteed in many cases (although in practice most queries terminate). These findings support Claim 2. It was also observed that in practice soundness can be traded off for higher expressiveness.

- *Applicability:* the main applicability concerns for integration are scalability (termination may be not guaranteed, again linking it to soundness) and debugging (e.g., ensuring that views conform to the models, linking applicability to soundness). Some applicability issues arise when handling complex domain-specific concepts, thus relating to expressiveness. Regardless of their nature, most applicability concerns were satisfactorily addressed in these case studies, supporting Claim 3.
- *Customizability:* in all the case studies, integration abstractions were successfully tailored to the systems and domains. Views are typically tailored through architectural types and customizable properties of architectural elements. Behavioral properties typically build on the specifications that are available and applicable to the domain's behavioral models. Customizability of the abstractions often enabled representation that are sufficiently expressive and checkable at the same time, customizability to soundness and expressiveness. These findings support Claim 4.

8.3 Validation of Part III: Analysis Execution Platform

The analysis execution platform was studied in the context of Systems 3 and 4 since only these systems features multiple dependent analyses with sophisticated execution contexts. This section describes the case studies of analysis intergration in those two systems.

8.3.1 Evaluation on System 3: Quadrotor

The investigation of analysis execution for the quadrotor (System 3, Section 3.3) was an independent case study, with an aim to discover and prevent conflicts between CPS analyses. The discovery process was performed as a literature review of common analyses for embedded systems for correctness, schedulability, and security applicable to a quadrotor. In the second step, the review was extended to analyses in the battery domain because the scheduling and battery domains are linked through voltage, which can be changed, potentially leading to errors in one of the domains. Thus, the literature review has produced the list of analyses presented in Subsection 8.1.3.

Integration of analyses in this case study was performed in four steps. First, for each analysis, a model of relevant elements of the quadcopter was constructed (e.g., the dynamics of the scheduler were described in M_{sch}). Second, the overlapping information between the models was encoded in the views. Then, a contract was written for each analysis in terms of the view elements. Finally, I performed experiments with analysis execution.

This evaluation focused on *applicability* and *soundness* of the analysis execution platform. Applicability in this case refers to the ability of the platform to execute the analyses correctly (with respect to their dependencies and context), avoiding dependency loops and scaling bottlenecks. Soundness means executing the analyses without introducing inconsistencies or design errors. The expressiveness and customizability were not evaluated directly because these qualities were determined by the integration abstractions used by the platform. The validation of these abstractions can be found in Subsection 8.2.3.

The rest of this study is described as follows. First, I present the formalizations of the models and views, organized into two domains: thread scheduling and battery design. Then, I provide full descriptions of the contracts for all the analyses. Finally, I discuss the experiments with execution of the analyses for the quadrotor.

Scheduling Domain

The *scheduling domain* focuses on timed interactions of threads and processors in an embedded system. This domain formalizes the concepts used in specifying the contracts for analyses that find valid thread allocations and priority assignments, check schedulability according to a selected scheduling policy, and determine appropriate processor frequencies. The symbols of the scheduling domain for specification of analysis contracts are represented with a signature Σ_{Sched} (Definition 24). The central sets in Σ_{Sched} are threads ($\mathsf{Threads}$) and CPUs (CPUs), which are modelled as component types in AADL. Thus, the meaning of these symbols is established with I^{V} by mapping them to a set of all components of the respective type in a given architecture, which plays the role of an analysis context (Υ, Definition 25). Σ_{Sched} also provides multiple properties of threads and CPUs, documented in Table 8.6. All of the properties are declared as part of component types, separately from AADL instances that contain specific components. Further, the signature provides two sorts with categorical values for the properties of thread security classes (SecCl) and thread scheduling policies ($\mathsf{SchedPol}$):

$$I^{\mathsf{V}}_{Sched}\mathsf{SecCl} = \{\mathbf{normal}, \mathbf{secret}, \mathbf{topsecret}\} \tag{8.10}$$

$$I^{\mathsf{V}}_{Sched}(\mathsf{SchedPol}) = \{\mathbf{rms}, \mathbf{dms}, \mathbf{edf}\} \tag{8.11}$$

The behavioral model for the scheduling domain (M_{sch}) encodes the dynamics of real-time thread scheduling and execution. In terms of specification, a dynamic/behavioral property interpreted by this model is *preemption between threads*, represented as a function that is evaluated modally (i.e., in every state the function *itself* might be different): $q(\mathsf{CanPrmpt}) : T \times T \mapsto \mathcal{B}$, such that $q(\mathsf{CanPrmpt}(t_1, t_2)) = \top$ iff t_1 can preempt t_2 in state q. To instantiate M_{sch}, one needs to provide a set of threads (sharing the same processor) and a scheduling policy as initialization parameters.

In addition to the above view and model symbols, the domain signature contains background sorts — Booleans $\mathcal{B}$, integers $\mathcal{Z}$, and reals $\mathcal{R}$ — that are shared between views and behavioral models.

Now I will describe how the above symbols are interpreted, and in terms what structures. The view symbols above (components and their properties) are interpreted on concrete instances of

[12]A real number between 0 and 1.

[13]Voltage is a nullary function, or a real constant. I consider a simplified example where the system voltage is the maximum of required individual processor voltages.

Name	Type	Description
Per	Threads $\mapsto \mathcal{Z}$	Thread's period.
Dline	Threads $\mapsto \mathcal{Z}$	Thread's deadline.
WCET	Threads $\mapsto \mathcal{Z}$	Thread's worst-case execution time.
ThSecCl	Threads $\mapsto$ SecCl	Thread's security class.
CPUSchedPol	CPUs $\mapsto$ SchedPol	CPU's scheduling policy.
CPUFreq	CPUs $\mapsto \mathcal{R}$	CPU's normalized frequency[12].
NotColoc	Threads $\mapsto 2^{\text{Threads}}$	A thread t is mapped to a set of threads that should not share the same C
CPUBind	Threads $\mapsto$ CPUs	Thread-to-CPU binding.
ThSafe	$C \mapsto \mathcal{B}$	A flag for whether a CPU's threads are thread-safe.
Voltage	$() \mapsto \mathcal{R}$	Required system voltage[13].

Table 8.6: Properties of threads and CPUs in Σ_{Sched}.

AADL models (which play the role of views and constitute a context Υ). SchedPol and SecCl are interpreted as enums of categorical values.

To define the execution semantics of M_{sch}, I constructed a model in Promela (the input language of the Spin model checker [105]) as follows. Recall that each thread consists of an infinite and periodic sequence of jobs. A state q of the system corresponds to points in time where a new job has just arrived or a currently executing job has just terminated. An execution is an infinite sequence of such states observed at run time. Note that, given a state, multiple executions are possible due to the non-determinism in the time required by each job to be completed. The intent for M_{sch} is to represent all such executions.

The model is a *Kripke structure* composed of one "task" process for each thread. Task processes are periodic and their numeric characteristics – $(\mathsf{Per}, \mathsf{Dline}, \mathsf{WCET})$ – are specified by the view Υ. There are $|[\![C]\!]_{\mathsf{M}}|$ processors, and each running task is allocated to a processor dynamically. For each task process t, a state q interprets the following propositions:

- $\mathsf{Prior}(t) : \mathcal{Z}$ – the priority of t.
- $\mathsf{Run}(t) : \mathcal{B}$ – whether a job of t is dispatched on a processor.
- $\mathsf{InQ}(t) : \mathcal{B}$ – whether a job of t has arrived but hasn't been completed yet.

$\mathsf{Prior}(t)$ is set by the scheduling policy and decides which tasks are executed. The last two propositions encode every possible state of t: idle if $\neg\mathsf{InQ}(t) \wedge \neg\mathsf{Run}(t)$, waiting for processor if $\mathsf{InQ}(t) \wedge \neg\mathsf{Run}(t)$, and executing if $\mathsf{InQ}(t) \wedge \mathsf{Run}(t)$.

For any state q of $\mathcal{K}([\![\mathcal{T}_{Sched}]\!])$, and threads t_1, t_2, $q(\mathsf{CanPrmpt})(t_1, t_2)$ is $\top$ iff the following holds in q:

$$\mathsf{Run}(t_1) \wedge \neg\mathsf{Run}(t_2) \wedge \mathsf{InQ}(t_2)$$

The implementation of M_{sch} in a Promela program computes $q(\mathsf{CanPrmpt})(t_1, t_2)$ appropriately for each state q and pair of threads t_1 and t_2, as described above. Each task t is implemented as a Promela process, and a manager process decides what priorities are assigned to threads and what threads are dispatched to processors. Thus, the manager process plays the role of a scheduler and a dispatcher in this model. The model handles the events of job arrival and termination in an infinite cycle, interleaving each event with the manager execution.

The Promela program represents non-deterministic job terminations without explicit time counters as follows. The manager process calculates possible upcoming events. Time is advanced in a greedy manner (i.e., whenever possible): if an arrival event happens, or the earliest of all the possible job termination events.

To achieve a finite state space, all clock variables[14] are reduced by the minimum value of all clock variables periodically. This model simulates a real-world scheduler execution as long as clock variables are not used in a contract. Since Σ_{Sched} does not expose clock variables as model symbols, the model is a valid representation of a scheduler. This is one of the conditions underlying soundness of this instance of the analysis contract approach.

This definition and implementation of the scheduling model makes it possible to apply IPL, by using LTL properties over $\mathsf{CanPrmpt}$ as a "behavioral interface" to this model. The scheduling contracts are presented below, after the battery domain is defined.

Battery Domain

The *battery domain* focuses on the design and dynamics of a new-generation battery that reschedules the cell connections at run time. The signature of this domain (Σ_{Batt}) is defined as follows. The only view components in this domain is batteries ($\mathsf{Batteries}$), which are rectangular arrays of cells (with $\mathsf{BatRows}$ rows and $\mathsf{BatCols}$ columns). Three mutually exclusive policies for scheduling cell connections ($\mathsf{ConnSchedPol}$) are allowed: unweighed round robin with fixed cell groups ($\mathbf{FGuRR}$), weighed kRR with fixed parallel cell groups ($\mathbf{FGwRR}$), and weighed kRR with cell group packing ($\mathbf{GPwRR}$) [125, 126]. One goal of these policies is to select the cells to discharge to maintain a constant output voltage ($\mathsf{Voltage}$) , which is a property that intersects with the scheduling domain. Note how specifying domain signatures makes it possible to naturally capture overlapping domains through views. A given voltage is maintained by arranging a number of cells in series ($\mathsf{SerialReq}$) and in parallel ($\mathsf{ParalReq}$). Another goal of connection scheduling is to ensure a battery lifetime that matches the product specifications (which is represented as a flag $\mathsf{HasReqdLifetime}$). The background sorts ($\mathcal{B}, \mathcal{Z}, \mathcal{R}$) and their interpretations are identical to the scheduling domain.

A battery execution consists of continuous charging, discharging, and resting of cells. An important run-time property is *thermal neighborhood*, represented in Σ_{Batt} with a function $\mathsf{TN} : \mathsf{Batteries} \times \mathcal{Z} \to \mathcal{Z}$. When a battery b is in state q, $q(\mathsf{TN}(b, i))$ denotes the number of cells with i *thermal neighbors* – cells that exchange heat conductively through a connector [15]. This is motivated by earlier results [124]: there is a close connection between thermal neighbors and thermal runaway. Specifically, there exist constants $\mathsf{K}(b, i) : b \in B, i \in \mathbb{Z}$ such that a state q

[14]Such as the next job arrival or the absolute system time.

[15]As opposed to electrical neighbors – cells that are connected to each other electrically, no matter how far apart physically they are.

Name	Type	Description
$\mathsf{Voltage}$	$() \mapsto \mathcal{R}$	Required system voltage.
$\mathsf{BatRows}$	$\mathsf{Batteries} \mapsto \mathcal{Z}$	Battery's cell rows.
$\mathsf{BatCols}$	$\mathsf{Batteries} \mapsto \mathcal{Z}$	Battery's cell columns.
$\mathsf{BatConnSchedPol}$	$\mathsf{Batteries} \mapsto \mathsf{ConnSchedPol}$	Battery's cell scheduling policy.
$\mathsf{SerialReq}$	$\mathsf{Batteries} \mapsto \mathcal{Z}$	Number of cells required to connect in series[16].
$\mathsf{ParalReq}$	$\mathsf{Batteries} \mapsto \mathcal{Z}$	Number of cells required to connect in parallel[17].
K	$\mathsf{Batteries} \times \mathcal{Z} \mapsto \mathcal{Z}$	Weight of cells with i thermal neighbors.
$\mathsf{HasReqdLifetime}$	$\mathsf{Batteries} \mapsto \mathcal{B}$	Flag whether a battery has the required lifetime.

Table 8.7: Properties of batteries in Σ_{Batt}.

triggers a thermal runaway in battery b if it violates the condition:

$$\sum_i \mathsf{K}(b, i) \times q(\mathsf{TN}(b, i)) \geq 0 \tag{8.12}$$

The exact values of K are not known up-front, and they are determined experimentally. Once they are obtained, they are added to the battery view. The static properties of battery are summarized in Table 8.7.

Now I turn to the structures on which the symbols of Σ_{Batt} are interpreted. As in the scheduling domain, the static properties of batteries are interpreted on AADL views. For interpretation of TN, I have constructed a Promela model of a battery (M_{bsch}). This model is instantiated with the size ($\mathsf{BatRows}$, $\mathsf{BatCols}$) and requirements ($\mathsf{SerialReq}$, $\mathsf{ParalReq}$).

M_{bsch} is defined as follows. A battery b consists of a matrix of cells χ being continuously charged, discharged, connected, and disconnected with each other. A state q of a battery corresponds to a point in time when either the charge or the connectivity status of a cell changes. An execution consists of an infinite sequence of such states observed at runtime. Many such executions are possible due to the non-determinism in the order of charge and discharge. M_{bsch} represents all such executions for a concrete battery.

I represent M_{bsch} as a Kripke structure with the following propositions for each cell $c = (x, y) \in \chi$, which is characterized by its physical coordinates $x \in [0, \mathsf{BatRows} - 1]$ and $y \in [0, \mathsf{BatCols} - 1]$:

- $\mathsf{CellCharge}(c)$ is the charge of c. To simplify model checking we chose a Boolean abstraction for the cell charge, but other abstractions are possible too.
- $\mathsf{CellSt}(c)$ is the status of c with possible values $\mathbf{discharging}$, $\mathbf{charging}$, and $\mathbf{idle}$.

[16]$\mathsf{SerialReq}$ is a battery-specific form of the voltage output requirement.
[17]$\mathsf{ParalReq}$ is a battery-specific form of the electrical current output requirement.

- $\mathsf{Gr}(c)$ is the number of group of cells electrically connected in serial within which c is located. Groups are treated as electrically connected in parallel with each other. Every cell belongs to a group, but not every group or cell is discharging.

TN is encoded as follows. Cells c_1 and c_2 are thermal neighbors, denoted $istnbr(c_1, c_2)$, if: (i) $c_1 \neq c_2$; (ii) $\mathsf{Gr}(c_1) = \mathsf{Gr}(c_2)$; (iii) $|c_1.x - c_2.x| + |c_1.y - c_2.y| \leq \mathit{TNDIST}$ [18]; (iv) $\mathsf{CellCharge}(c_1) = \mathsf{CellCharge}(c_2) = \top$; and (v) $\mathsf{CellSt}(c_1) = \mathsf{CellSt}(c_2) = \mathbf{discharging}$. The number of thermal neighbors of cell c is $ntnbr(c) = |\{c' \in \chi \cdot istnbr(c, c')\}|$. Finally, $\mathsf{TN}(b, i) = |\{c' \in \chi \cdot ntnbr(c) = i\}|$.

The above Kripke structure is implemented as a single-process Promela program. The program maintains the state variables $\mathsf{CellCharge}$, CellSt, Gr, as discussed above. The program execution works in two steps: first, the cells are scheduled for discharging/charging (i.e., changing Gr and CellSt), and second, the charge state is advanced (i.e., $\mathsf{CellCharge}$ is changed).

The first step of M_{bsch} is deterministic: it imitates the logic of the selected cell scheduler. **FGURR** does not change Gr and rotates through groups, setting $\mathsf{ParalReq}$ groups to discharge each time and the rest to idle. **FGWRR** does not change Gr either, but instead of rotating the groups it sorts them in decreasing order of charge (which, for us, is the number of cells with $\mathsf{CellCharge}(c) = \top$) and selects the top $\mathsf{ParalReq}$ groups. **GPWRR** assembles groups by packing as many charged cells into each group as possible. Then it selects the top $\mathsf{ParalReq}$ most charged groups to discharge. Within each group, all schedulers select $\mathsf{SerialReq}$ charged cells.

The second step of M_{bsch} is non-deterministic: every discharging cell non-deterministically becomes discharged; every charging cell non-deterministically becomes charged; idle cells, however, do not change their charges. The program terminates when there is not enough charge for the output requirements. This charging and discharging dynamic is an overapproximation of high-fidelity battery models with precise measurements of the cell charge. An abstract charge state is used as a basis of scheduling the cells. Due to the non-determinism in the second step, our implementation accounts for possible cell failures (i.e., cell gets immediately discharged) and subsubes any high-fidelity model of charge. Thus, by representing the logic of the cell schedulers and abstracting the charge state of the cells, M_{bsch} contributes to the soundness of the approach: the model does not miss potentially dangerous states of the battery.

With the battery domain defined, I move on to specifying analysis contracts.

Analysis Contracts for System 3

Using the domains signatures Σ_{Sched} and Σ_{Batt} from the previous section, I specify the contracts for the analyses that were described in Section 3.3.

Secure thread allocation ($\mathcal{A}_{SecAlloc}$) has contract $\mathsf{C}_{SecAlloc} : \mathbb{I} = \{\mathsf{Threads}, \mathsf{ThSecCl}\}$, $\mathbb{O} = \{\mathsf{NotColoc}\}$, $\mathbb{A} = \emptyset$, and $\mathbb{G} = \{\mathsf{G}\}$ where G is:

$$\forall t_1, t_2 : \mathsf{Threads} \cdot \mathsf{ThSecCl}(t_1) \neq \mathsf{ThSecCl}(t_2) \rightarrow t_1 \in \mathsf{NotColoc}(t_2).$$

Thus, $\mathcal{A}_{SecAlloc}$ makes no assumptions, but guarantees that threads with different security classes are never co-located.

[18] For our calculations we use $\mathit{TNDIST} = 2$.

Bin packing ($\mathcal{A}_{BinPack}$) has contract $\mathsf{C}_{BinPack}$: $\mathbb{I} = \{\mathsf{Threads}, \mathsf{CPUs}, \mathsf{NotColoc}, \mathsf{Per}, \mathsf{WCET}, \mathsf{Dline}\}$, $\mathbb{O} = \{\mathsf{CPUBind}\}$, $\mathbb{A} = \emptyset$, and $\mathbb{G} = \{\mathsf{G}\}$ where G is:

$$\forall t_1, t_2 : \mathsf{Threads} \cdot t_1 \in \mathsf{NotColoc}(t_2) \rightarrow \mathsf{CPUBind}(t_1) \neq \mathsf{CPUBind}(t_2).$$

Thus, $\mathcal{A}_{BinPack}$ makes no assumptions but guarantees that threads that should not be co-located are never scheduled on the same CPU.

Frequency scaling ($\mathcal{A}_{FreqSc}$) has contract C_{FreqSc}: $\mathbb{I} = \{\mathsf{Threads}, \mathsf{CPUs}, \mathsf{CPUBind}, \mathsf{Dline}\}$, $\mathbb{O} = \{\mathsf{CPUFreq}\}$, $\mathbb{A} = \{\mathsf{A}\}$, and $\mathbb{G} = \emptyset$, where A is:

$$\begin{aligned} &\forall t_1, t_2 : \mathsf{Threads} \cdot t_1 \neq t_2 && \wedge \mathsf{CPUBind}(t_1) = \mathsf{CPUBind}(t_2) : \\ &\mathsf{G}(\mathsf{CanPrmpt}(t_1, t_2) && \rightarrow \mathsf{Dline}(t_1) < \mathsf{Dline}(t_2)). \end{aligned}$$

Thus, $\mathcal{A}_{FreqSc}$ makes no guarantees but assumes that the scheduling used is semantically equivalent to a deadline-monotonic scheduling policy. Note that a scheduling policy can be DMS for a specific model, e.g., rate-monotonic scheduling for a harmonic model, even thought it is not deadline monotonic in general. This property has been discussed in more detail for validation of IPL in Subsection 8.1.3, and mentioned for validation of LTL properties as integration abstractions in Subsection 8.2.3.

Model checking with REK ($\mathcal{A}_{Rek}$) has contract C_{Rek}: $\mathbb{I} = \{\mathsf{Threads}, \mathsf{CPUs}, \mathsf{Per}, \mathsf{Dline}, \mathsf{WCET}, \mathsf{CPUBind}\}$, $\mathbb{O} = \{\mathsf{ThSafe}\}$, $\mathbb{G} = \emptyset$, and $\mathbb{A} = \{\mathsf{A}_1, \mathsf{A}_2\}$ where:

$$\begin{aligned} &\mathsf{A}_1 \triangleq \forall t : \mathsf{Threads} \cdot \mathsf{Per}(t) = \mathsf{Dline}(t), \\ &\mathsf{A}_2 \triangleq \forall t_1, t_2 : \mathsf{Threads} \cdot \mathsf{G}(\mathsf{CanPrmpt}(t_1, t_2) \rightarrow \mathsf{G}\neg\mathsf{CanPrmpt}(t_2, t_1)). \end{aligned}$$

The Rek model checker [35] takes threads and their marked source code files (which were not part of the case study) as input and verifies whether the system is safe, where safety is expressed as assertions embedded in the source code. $\mathcal{A}_{Rek}$ assumes implicit deadlines (expressed in A_1) and fixed-priority scheduling (expressed in A_2: if t_1 preempts t_2, then t_2 should never be able to preempt t_1). Prior to this work, the only way to apply Rek was to use RMS. With a contract, this analysis could be applied more broadly, not necessarily to systems that directly use RMS. Thus, contracts can improve applicability of analyses. This property has also been discussed for validation of IPL in Subsection 8.1.3 and mentioned for validation of LTL properties as integration abstractions in Subsection 8.2.3.

Thermal runaway ($\mathcal{A}_{ThermRun}$) has contract $\mathsf{C}_{ThermRun}$: $\mathbb{I} = \{\mathsf{Batteries}, \mathsf{BatRows}, \mathsf{BatCols}, \mathsf{Voltage}\}$, $\mathbb{O} = \{\mathsf{K}\}$, $\mathbb{A} = \emptyset$, and $\mathbb{G} = \emptyset$. Thermal runaway determines the patterns, which, given concrete battery characteristics, would result into a thermal runaway. In this case study, I encode these patterns as $\mathsf{K}(i)$ for $i : \mathcal{Z} \in [0, 3]$. $\mathcal{A}_{ThermRun}$ determines K through experimentation, adjusting K so that acceptable heat propagation patterns satisfy (8.12), and the unacceptable ones violate it. Note that $\mathcal{A}_{ThermRun}$ has no assumptions or guarantees, but it has a dependency with the battery scheduling analysis (defined below) via $\mathbb{I}$ and $\mathbb{O}$.

Battery scheduling ($\mathcal{A}_{BatSched}$) has contract $\mathsf{C}_{BatSched}$: $\mathbb{I} = \{\mathsf{Batteries}, \mathsf{BatRows}, \mathsf{BatCols}\}$, $\mathbb{O} = \{\mathsf{BatConnSchedPol}, \mathsf{HasReqdLifetime}, \mathsf{SerialReq}, \mathsf{ParalReq}\}$, $\mathbb{A} = \emptyset$, and $\mathbb{G} = \{\mathsf{G}\}$ where G is:

$$\forall b : \mathsf{Batteries} \cdot \mathsf{G}(\mathsf{K}(b, 0) \times \mathsf{TN}(b, 0) + \mathsf{K}(b, 1) \times \mathsf{TN}(b, 1) +$$

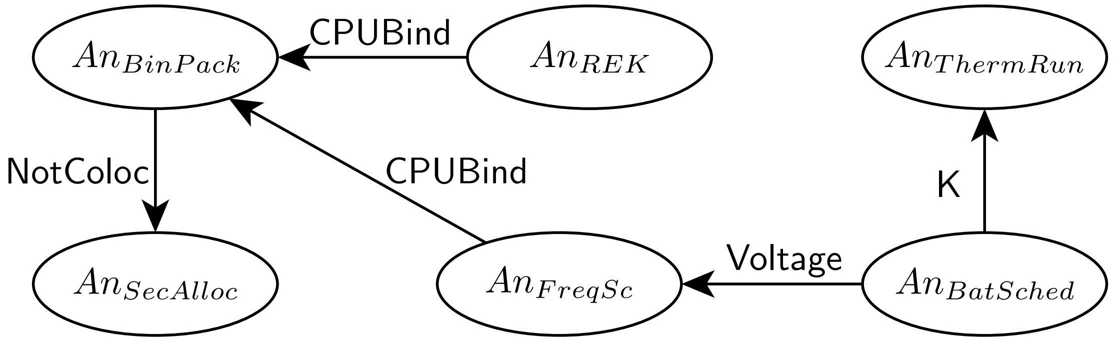


Figure 8.6: Analysis dependency graph ($gamma$) for System 3.

$$\mathsf{K}(b,2) \times \mathsf{TN}(b,2) + \mathsf{K}(b,3) \times \mathsf{TN}(b,3) \geq 0).$$

$\mathcal{A}_{BatSched}$ computes a battery cell connectivity scheduler that maximizes the battery lifetime given the battery characteristics and output requirements. It sets the flag HasReqdLifetime indicating whether the battery, given its selected scheduler, meets the lifetime requirement. Since the scheduling is not aware of the thermal runaway, the determined scheduler needs to be verified against the thermal runaway pattern, hence the guarantee. $\mathcal{A}_{BatSched}$ also sets cell group characteristics SerialReq and ParalReq that are used to verify its guarantee.

The next section explains how the specified contracts assisted the execution of the analyses for System 3, and how this execution affected the integration qualities.

Analysis Execution

I performed several experiments with various designs of the quadrotor, to examine the application of the analysis execution platform to this system. Each experiment involved choosing an initial design of the thread scheduler and the battery, and running a series of analyses on it using the execution platform. The soundness of the initial and modified design were manually compared to the outputs of the analysis execution and contract checking.

Dependency resolution was based on the inputs and outputs in the contracts from the previous section, leading to the dependency graph shown in Figure 8.6. Each edge indicates a dependency (the arrow points *from* a dependent analysis *towards* an independent analysis), labeled with a symbol that causes the dependency. Executing any analysis in the graph follows the algorithm described in Section 7.3: before any goal analysis is executed, its dependencies are executed in the order determined by the graph. If assumptions or guarantees are not satisfied occurs in the middle of the analysis series, the models and views are reverted to the original state. Otherwise, the full sequence of analyses is executed successfully. During the experiments, no outdated information was consumed by the analyses, and no newer information was overwritten by its older version.

Consider the following quadrotor configuration: threads t_1, t_2, t_3 have $[\![\mathsf{Per}]\!]_\mathsf{M} = \{t_1 \mapsto 100, t_2 \mapsto 150, t_3 \mapsto 2$
$[\![\mathsf{Dline}]\!]_\mathsf{M} = \{t_1 \mapsto 100, t_2 \mapsto 90, t_3 \mapsto 200\}$, $[\![\mathsf{WCET}]\!]_\mathsf{M} = \{t_1 \mapsto 10, t_2 \mapsto 15, t_3 \mapsto 20\}$ are allocated to a single CPU. Before analysis $\mathcal{A}_{FreqSc}$ is applied to determine CPU frequencies, its assumption $\mathsf{C}_{FreqSc}.\mathsf{A}$ is verified. Recall that $\mathsf{C}_{FreqSc}.\mathsf{A}$ states that the scheduling policy must be semantically equivalent to DMS.

Suppose, first, that the system uses RMS scheduling, i.e., $\mathsf{Prior}(t_1) > \mathsf{Prior}(t_2) > \mathsf{Prior}(t_3)$. In this case, verification detects a violation of $\mathsf{C}_{FreqSc}.\mathsf{A}$ because in this case DMS would assign $\mathsf{Prior}(t_2) > \mathsf{Prior}(t_1)$, thus preempting not in a deadline-first way. Now suppose that the system uses EDF. The IPL verification procedure indicated that this system would then satisfy $\mathsf{C}_{FreqSc}.\mathsf{A}$. Thus, the analysis execution platform not only prevents incorrect usage of $\mathcal{A}_{FreqSc}$, but also extends the application of this analysis to EDF, beyond its original scope (i.e., DMS).

Next, suppose the quadrotor has a single battery with $\mathsf{BatRows} = \mathsf{BatCols} = 4$, and a voltage requirement $\mathsf{ParalReq} = \mathsf{SerialReq} = 3$. It has been observed in related work [124] that heat-dissipating cells (i.e., those with many thermal neighbors) and heat-isolated cells (i.e., those with no thermal neighbors) tend to prevent thermal runaway, while cells with one thermal neighbor tend to accumulate heat and lead to runaway. An assignment of weights $\mathsf{K}(0) = \mathsf{K}(1) = \mathsf{K}(2) = 2, \mathsf{K}(1) = -1$ in (8.12) simulates this intuition.

After executing $\mathcal{A}_{BatSched}$ on the above design, which picks a battery scheduler, the IPL algorithm verifies its guarantee $\mathsf{C}_{BatSched}.\mathsf{G}$. Since $\mathcal{A}_{BatSched}$ is not aware of thermal runway, not every scheduler meets the guarantee. As the Spin verification on M_{bsch} indicates, $\mathbf{FGWRR}$ and $\mathbf{FGURR}$ satisfy it, but $\mathbf{GPWRR}$ fails because it causes the system to reach a configuration that violates (8.12) with $\mathsf{TN}(0) = \mathsf{TN}(3) = 0, \mathsf{TN}(1) = 8, \mathsf{TN}(2) = 1$. Thus, the platform detects possibility of a thermal runaway even though the existing analysis $\mathcal{A}_{BatSched}$ does not.

The above experiments provide additional confidence in *soundness* of the approach to analysis integration. Specifically, it shows that in practice, the analysis execution platform respects analysis dependencies and executes analyses only in appropriate contexts. As a result, the platform prevents errors due to missed dependencies and context mismatch. The soundness of analysis execution depends on the soundness of abstraction, in particular the completeness of views to avoid unsatisfied dependencies (discussed for System 3 in Subsection 8.2.3) and soundness of behavioral queries to correctly determine whether a context is appropriate (discussed for System 3 in Subsection 8.2.3).

A major *applicability* concern is the performance of the dependency resolution and verification. The dependency resolution was near-instantaneous (from the tool user's perspective), and hence does not present a performance issue for realistic dependency graphs. The verification of analysis contexts is more time-consuming and can be a threat to scalability. The performance results of verification with IPL have been presented in Subsection 8.2.3 (specifically, see Tables 8.4 and 8.5) and found generally adequate to practical needs. The performance of analysis execution depends on the termination quality of behavioral properties (discussed for System 3 in Subsection 8.2.3).

Another applicability concern is existence of dependency loops in a given set of contracts. Generally, an engineer writing the contracts should define the types and properties of architectural elements that allow modeling the dependencies with the level of fidelity that does not result in dependency loops. In this case study, it was possible to avoid dependency loops by modeling multiple properties of threads and CPUs, different for each analysis. Therefore, one factor that helps avoid dependency loops is the expressiveness of views in terms of custom types and properties (i.e., if some property or type is causing a dependency loop, they can be refined into multiple properties/types that characterize the inputs/outputs more precisely). Another factor to avoid dependency loops is the soundness of views: no extra elements appear in views, potentially reducing the set of view elements that may lead to a dependency.

An additional benefit of the analysis contracts in terms of applicability is that some analyses are applicable beyond their original intended context with logical specification of their appropriate contexts. This benefit has been demonstrated for $\mathcal{A}_{Rek}$ and $\mathcal{A}_{FreqSc}$.

To summarize, in this case study, six analyses for a quadrotor design were integrated by the means of contracts. The study has found that the analysis execution platform is applicable to the realistic analyses and can execute them correctly, preventing errors due to missed dependencies or inappropriate contexts. The expressiveness and customizability of the integration abstractions (AADL views and LTL properties) used in this study have been addressed in Subsection 8.2.4.

8.3.2 Evaluation on System 4: Autonomous Car

The investigation of analysis execution for the autonomous car (System 4, Section 3.4) was an independent case study, with the primary focus on *customizability* of the analysis contract approach to new domains — in this case, the domains of reliability and security. Similarly to the System 3 case study, the discovery process of analyses was performed as a literature survey. In this study, the guiding principle was to find analyses that treat failure of the system's components differently: reliability analyses typically consider failure a random event, whereas security analyses may treat failure as a result of intentional activities of attackers. My hypothesis was that due to this difference, analyses from reliability and security domains may make potentially incompatible assumptions. Another condition was that the analyses selected for this study had to be applicable in the context of an autonomous vehicle, motivated by the related work on attacking Jeep through its sensors [], published at the time of this study.

To investigate *applicability* of analyses contracts, a sample model of an autonomous car was constructed. The model contains the sensors, configurations, and adversary profiles as discussed in Section 3.4. Then, the analysis contracts were created and shown to appropriately order the dependencies and prevent invokation of analyses in inappropriate contexts. This section focuses only on applicability and soundness of the analysis execution platform. In this case, soundness means executing analyses according to their dependencies and appropriate contexts. Applicability refers to avoiding dependency loops and following the intent of the analyses in their contracts. The expressiveness and customizability were not evaluated directly because these qualities were determined by the views that were used by the platform. The description of the views for the autonomous car and their evaluation can be found in Subsection 8.2.4.

In the remainder of this chapter, I describe the specification of the contracts for the analyses from Section 3.4 and discuss how the qualities of integration determined by the execution platform.

Specification of Contracts

To specify the contracts, I defined a single domain signature (Σ) based on the views described in Subsection 8.2.4. To remind the reader, the architectural types are sensors ($\mathbb{S}$), controllers ($\mathbb{R}$), and modes ($\mathbb{M}$). These view elements are annotated with properties related to security and reliability. Based on this signature, I specify the contracts for the three analyses below.

FMEA analysis ($\mathcal{A}_{fmea}$): the goal of the FMEA analysis is to find a component redundancy structure [19] that is capable of withstanding the expected random failures of individual components

[19]This analysis is constrained by cost (in terms of funds and available space) of components: the trivial solution of

and provide a system with a probability of failure no larger than α_{fail}. Hence, one output of FMEA is an architecture of sensors and controllers.

Another output of FMEA is a set of likely failure modes. [20] The output contains the failure modes (i.e., system configurations with some unavailable sensors, $\mathsf{Avail} = \bot$) that need to be checked for the system to be safe.

A typical FMEA assumption is that the random mechanical/hardware failures are independent among all of the system's components. That is, a failure of one sensor does not increase the probability of another sensor's failure. This assumption allows for simpler reasoning about failure propagation and failure modes during the analysis. Since the probabilities of failure are usually generalized from noisy empirical data, a correlation tolerance bound $\epsilon_{fail} > 0$ is added to the assumption.

A guarantee of FMEA is that the controllers receive all the required variable (as streams/series of measurements) to actuate the system. This guarantee does not ensure the full correctness of the FMEA analysis (the system may still not be fault-tolerant), but it allows to verify that the analysis has not rendered the system non-functional.

Thus, the contract for $\mathcal{A}_{fmea}$ is as follows:

- Inputs: $\mathsf{P_{fail}}$, α_{fail}.
- Outputs: $\mathbb{S}$, $\mathbb{R}$, $\mathbb{M}$.
- Assumption. *Component failure independence:* if one component fails, another component is not more likely to fail:
$$\forall c_1, c_2 \in \mathbb{S} \cup \mathbb{R} \ \cdot P(\neg c_1.\mathsf{Avail} \mid \neg c_2.\mathsf{Avail}) \leq P(\neg c_1.\mathsf{Avail}) + \epsilon_{fail}.$$
- Guarantee. *Functioning controllers:* some sensor provides each variable that some controller expects:
$$\forall m \in \mathbb{M} \cdot \forall c \in m.\mathbb{R} \cdot \forall v \in c.\mathsf{VarsR} \cdot \exists s \in m.\mathbb{S} \cdot v \in s.\mathsf{VarsS}.$$

Sensor trustworthiness analysis $\mathcal{A}_{trust}$: this analysis determines the possibility of each sensor being compromised (represented with a flag Trust) given their placement, power status, availability, and the selected attacker model. To avoid ambiguity, I assume that unpowered and unavailable sensors cannot be compromised. Therefore, $\mathcal{A}_{trust}$ marks a sensor as untrustworthy if and only if the sensor is powered, available, and vulnerable for the given attacker model:

$$\forall s \in \mathbb{S} : \neg\mathcal{A}_{trust}(s) \iff s.\mathsf{Pow} \wedge s.\mathsf{Avail} \wedge \mathsf{AttackM.IsVuln}(s).$$

The sensor trustworthiness analysis treats failures differently from FMEA. It is expected that some sensors may go out of order together because of a coordinated physical attack or an adverse environment like fog. This leads to the failure dependence assumption with an error bound $\epsilon_{trust} > 0$. While not being a direct negation of FMEA's assumption, failure dependence makes analysis applicable in a different scope of designs. Whether the analyses can be applied together on the same system depends on calibration of the error bound parameters ϵ_{fail} and ϵ_{trust}.

The correctness of the sensor trustworthiness analysis can be expressed declaratively: untrustworthy sensors are the ones that can be attacked by the selected attacker model. I put this

replicating each sensor a large number of times would typically not be acceptable.

[20] The definition of likelihood for failure modes may differ depending on the system requirements. For example, one may consider failure modes with probabilities $\geq 0.1\alpha_{fail}$.

statement in the contract as a guarantee to create a sanity check on the analysis implementation, which may contain unknown bugs.

Given the above, the contract for $\mathcal{A}_{trust}$ is specified as follows:

- Inputs: $\mathbb{S}$, Place, Pow, Avail, $\mathsf{AttackM}$.
- Output: Trust.
- Assumption. *Component failure dependence:* some components are likely to fail together:
$$\exists c_1, c_2 \in \mathbb{S} \cup \mathbb{R} : P(\neg c_1.\mathsf{Avail} \mid \neg c_2.\mathsf{Avail}) \geq P(\neg c_1.\mathsf{Avail}) - \epsilon_{trust}$$
- Guarantee. *Correct trustworthiness assignment:* a sensor is not trustworthy if and only if it is vulnerable for the considered attacker model:
$$\forall m \in \mathbb{M}, s \in m.\mathbb{S} \cdot s.\mathsf{Trust} = \bot \iff m.\mathsf{AttackM}.\mathsf{IsVuln}(s).$$

Control safety analysis $\mathcal{A}_{ctrl}$: this analysis determines whether the control has a required performance, is stable and robust (or, in short, *safe*). I abstract away the details of this analysis and specify that it requires the control model (sensors, controllers, actuators and their variables) and outputs whether the control is safe. More details can be added as necessary for more refined contracts.

A common feedback controller architecture includes a state estimator (e.g., a Kalman filter or a decoder) and a control algorithm, such as PID control. A decoder is used to estimate the genuine system state when an attacker may have falsified some sensor data. According to Propositions 2 and 3 in [68], it is required that at least half of sensors that sense the same variable are trustworthy. Otherwise a decoder cannot discover or correct an intentional sensor attack, leading to the system being compromised. Powered off and unavailable sensors are considered trustworthy, but do not contribute to the trustworthiness estimate.

I specify the assumption about a half of sensors being trustworthy by establishing a mapping function f (for each variable) between trustworthy and untrustworthy sensors. Existence and surjectivity[21] of f mean that for each untrustworthy sensor there exists at least one unique trustworthy sensor. That existence is equivalent to the proportion of trustworthy sensors being at least 50%.

We thus arrive at the following contract for $\mathcal{A}_{ctrl}$:

- Inputs: $\mathbb{S}$, VarsS, $\mathbb{R}$, VarsR.
- Output: $\mathsf{CtrlSafe}$.
- Assumption. *Minimal sensor trust* — for each untrusted sensor there is at least one different trusted sensor [22]:
$$\begin{aligned} &\forall m \in \mathbb{M}\ \forall c \in m.\mathbb{R}, v \in c.\mathsf{VarsR} \cdot \\ &\quad \exists f :\ \mathbb{S} \to \mathbb{S}\ \cdot \forall s_u \in m.\mathbb{S} \cdot \\ &\quad\quad v \in s_u.\mathsf{VarsS} \wedge s_u.\mathsf{Trust} = \bot \to \\ &\quad\quad \exists s_t \in m.\mathbb{S} \cdot v \in s_t.\mathsf{VarsS} \wedge s_t.\mathsf{Trust} = \top \wedge f(s_t) = s_u. \end{aligned}$$
- Guarantee: none.

[21] A *surjective* function covers its full range of values.

[22] This assumption can be written in a simpler form, ”at least half of the sensors are trustworthy”: $\forall m \in \mathbb{M} \cdot |m.S_{trustworthy}|/|m.\mathbb{S}| \geq 0.5$. Unfortunately such statements cannot be verified using state-of-the-art SMT, and theories with set cardinalities have not been implemented for SMT yet.

This concludes the specification of the analysis contracts for System 4. The ultimate design goal is to apply these analyses in a way that guarantees that the sensors trustworthiness is adequate for the considered attacker model ($s.\mathsf{Trust} = \bot \iff \mathsf{AttackM.IsVuln}(s)$), the system's control is safe ($\mathsf{CtrlSafe} = \top$), and that the system's failure probability is not greater than α_{fail}. The next section shows how the analysis execution platform achieves this goal.

Analysis Execution

Before the analyses are executed, their dependencies need to be resolved. Dependency calculation for $\mathcal{A}_{fmea}$, $\mathcal{A}_{trust}$, and $\mathcal{A}_{ctrl}$ yielded the following dependency graph (shown in Figure 8.7):

- $\mathcal{A}_{fmea}$ does not depend on any analyses considered in this paper.
- $\mathcal{A}_{trust}$ depends on $\mathcal{A}_{fmea}$ that outputs $\mathbb{S}$— an input for $\mathcal{A}_{trust}$.
- $\mathcal{A}_{ctrl}$ depends on $\mathcal{A}_{fmea}$ that outputs $\mathbb{S}$ and $\mathbb{R}$— inputs for $\mathcal{A}_{ctrl}$.
- $\mathcal{A}_{ctrl}$ depends on $\mathcal{A}_{trust}$ that outputs Trust— part of an assumption for $\mathcal{A}_{ctrl}$.

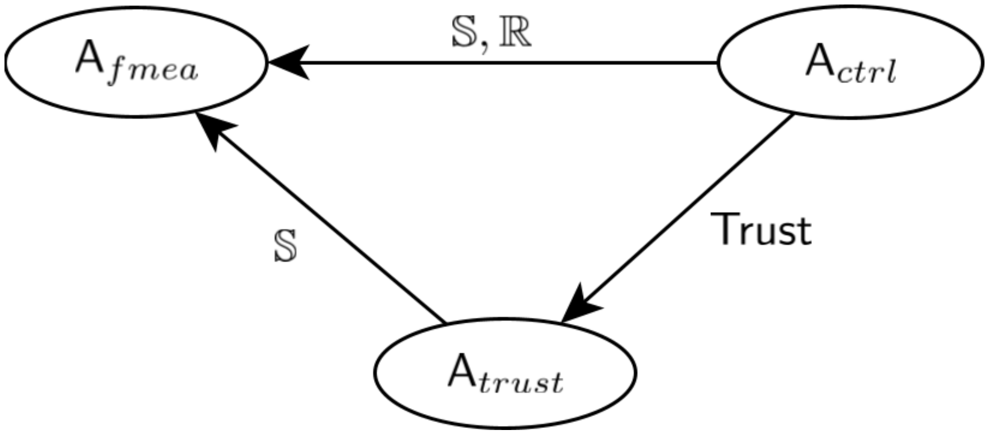

$$\begin{tikzcd}
\mathsf{A}_{fmea} & & \mathsf{A}_{ctrl} \arrow[ll, "{\mathbb{S}, \mathbb{R}}"'] \arrow[dl, "\mathsf{Trust}"] \\
& \mathsf{A}_{trust} \arrow[ul, "\mathbb{S}"] &
\end{tikzcd}$$

Figure 8.7: Dependencies of analyses for System 4.

I executed the analyses according to these dependencies in the ACTIVE tool (see Section 7.4), in the order of $\mathcal{A}_{fmea}$, $\mathcal{A}_{trust}$, and $\mathcal{A}_{ctrl}$. For example, if the user changes $\mathsf{AttackM}$ and tries to execute $\mathcal{A}_{ctrl}$, $\mathcal{A}_{trust}$ is executed first so that the assumption of $\mathcal{A}_{ctrl}$ is verified on values of Trust that are consistent with $\mathsf{AttackM}$. Moreover, before $\mathcal{A}_{trust}$ is executed, $\mathcal{A}_{fmea}$ is executed since $\mathcal{A}_{trust}$ (and $\mathcal{A}_{ctrl}$ as well) depends on it as well.

The *soundness* of analysis integration is determined by handling the dependencies and appropriate contexts of the analyses. The dependencies determined the correct order of the analyses, similar to the System 3 study (Subsection 8.3.1). The contexts, however, were more diverse. This study features three types of context specifications: first-order deterministic, second-order deterministic, and first-order probabilistic. All contracts other than the second-order deterministic ones are expressible and checkable in IPL. The deterministic logical constracts for System 4 that used only *first-order quantification* over variables in bounded sets were translated into SMT programs and checked using the IPL verification algorithm. *Probabilistic quantification* is also supported in IPL, given an appropriate model and a behavioral language (such as PCTL [98]). In this case, probabilistic specifications are convenient to capture statements that go beyond boolean logic, which happens often in domains related to rare or uncertain events and behaviors. Fault tolerance, cryptography, and wireless ad hoc networks are examples of such domains. To check such contracts, one needs to supply probabilistic model checking tools like PRISM [134] or MRMC (Markov Reward Model Checker) [121]. The contracts with the *second-order quantification* were not supported by the IPL verification, lacking a general and sound way to be checked. Thus, the contracts in this study were logically specifiable, and most of the contracts were soundly checkable (since no contract quantified over unbounded sets, like integers or reals). Checking of the contracts (mixed automated and manual) found no violations, providing secondary evidence of soundness.

In this study, the *applicability* of the analysis execution platform has been achieved by specifying the contracts without dependency loops. At the first glance, the trustworthiness and FMEA analyses operate on the same set of sensors, leading to a dependency loop. I resolved the loom by letting $\mathcal{A}_{fmea}$ work on the set of sensors, and have $\mathcal{A}_{trust}$ perform trustworthiness calculations on the set of sensors. Thus, the dependency loop is resolved.

To summarize, the case study of System 4 has shown applicability and soundness of the analysis execution platform by integrating analyses from several domains (other than schedulability/battery, as in System 3). It was shown that the inputs, outputs, assumptions, and guarantees for the domain of reliabiliy/trustworthiness of autonomous car sensing can be specified, and lead to correct execution of analyses. The abstractions used in this study demonstrated the customizability of views, as discussed in Subsection 8.2.4.

8.3.3 Summary for Evaluation of the Analysis Execution Platform

This section described the application of the analysis execution platform to a total of nine analyses across two systems: a quadrotor and an autonomous car. For both of these systems, I specified analysis contracts to organize execution of the analyses.

The following results summarize the findings of the case studies with respect to the qualities of integration:

- *Soundness:* analysis execution is sound with respect to the dependencies in every case without circular dependencies. The soundness of context checking followed from implementing the context checks with the IPL verification. These results provide evidence for Claim 2. Soundness of properties where IPL is not currently applicable has not been studied.
- *Applicability:* the analysis execution platform was applicable to the two case studies. Applicability was enabled by the contract specifications that accurately indicated the scope of each analysis and avoided dependency loops. This result provides evidence for Claim 3.
- *Expressiveness* and *customizability:* these concerns were handled at the level of the integration abstractions that were used in the respective case studies. Thus, the evidence supporting Claims 1 and 4 can be found in Section 8.2.

Chapter 9

Related Work

This chapter discusses the research related to this thesis. I split the related work in several categories:

- Modeling methods for CPS. I describe typical CPS models and analyses, which may need to be integrated with each other.
- Foundations of my approach. I discuss the research that serves as a basis for my approach. In particular, software architecture and views, as well as logic and automated reasoning tools. I also revisit the ideas of dependency, contracts, and model transformations that have inspired my approach.
- Approaches to integration of modeling methods. Here I focus on the approaches that can address the problem of modeling method integration. I compare these approaches to the one described in this thesis.

9.1 Modeling Methods

CPS engineering relies on modeling methods that use a spectrum of discrete and continuous representations of systems. The discrete modeling methods are traditionally used in computer science and electrical engineering, whereas most continuous models originate in control, mechanical engineering, and physics. In this section I discuss discrete modeling methods, continuous modeling methods, and their hybrid combinations.

9.1.1 Discrete Modeling Methods

One category of discrete models focuses on descriptions of complex states and their relations, often expressed as data schemas and object models. Popular formalisms of this category include Alloy [113], TLA+ [137], Object-Z [219], VDM-SL [141], and B [138]. The models in these formalisms can be considered sets of declarative constraints on an abstract structure or state. Such modeling methods typically support extension through refinement and composition, and naturally enable logical analyses, such as checking for contradictions and generating (counter-)example model instances.

Another category of discrete modeling methods focuses on process descriptions, where the primary focus is on transitions and changes to the system state. These modeling methods relies on algorithmic notations and various forms of state machines: process algebras like CSP [103] and FSP [154], transition systems [236] like statecharts [67] and Promela/Spin [105], PlusCal [137], Petri nets [64], dynamic logic (accompanied with JML specifications) [99, 115], and reactive models [5]. For these models, analyses typically check input-output properties of algorithms (e.g., correctness with respect to a specification) and concurrency properties (e.g., absence of deadlocks and race conditions). These models are usually composable in a parallel (i.e., via synchronization over shared actions) or functional way (i.e., applying one algorithm to the output of another).

Some discrete models aim to combine rich specifications of state and first-class process elements, resulting in such formalisms as Event-B [1], UML [67], SysML [58], and CML [237]. These methods can be considered modeling frameworks with several related formalisms. Relating these formalisms makes it easy to check some consistency properties (e.g., referential integrity), but does not fully solve the problem of model integration. Combinations of formalisms allow for multiple modes of analysis, composition, and extension, leading to different modeling methodologies (some of which can be used to address integration issues and are mentioned below in Section 9.3).

Thus, discrete models are useful for verification and several forms of composition (which can be used to address some model integration issues). Some discrete models enable synthesis (e.g., code generation) and execution/simulation. Many of these models also have associated logics (like LTL for Promela/Spin) for abstract properties, which as behavioral property languages (as discussed below in Section 9.2). In a CPS context, a major downside of discrete modeling methods is their treatment of continuous processes [144]: the necessary granularity of discretization is usually unknown, and high enough granularity is often impractical due to limited scalability.

9.1.2 Continuous and Hybrid Modeling Methods

On the other side of the spectrum are classic modeling methods that rely on differential and difference equations [234]. These equations are traditionally used to describe physical processes in mechanics (e.g., motion of bodies), thermodynamics (e.g., exchange of heat), and electromagnetism (e.g., temporal changes in the electromagnetic field). Differential equations describe a system using state variables (e.g., location or temperature), their initial conditions, and laws of their evolution — without a priori prescribing the states in which the system can be. Partial differential equations can be simplified into acausal lumped element models (combinations of discrete entities with shared variables, like in Modelica [80]), described by ordinary differential equations. Generally, these models enable simulation, theoretical analysis (stability, safety, robustness) and, in some cases, admit closed-form solutions that simplify prediction and other analyses.

Continuous models are often used in control engineering. In practice, causal signal-flow formalisms (Simulink[53] and SCADE [66]) are used for designing control and plant (environment) models, enabling simulation analysis of empirical properties like rise time, overshoot, and setting time [43]. Signal-flow models were extended with discrete state descriptions (see StateFlow [97]) for prototyping algorithmic decision (e.g., implementing modes or responding to exceptional behavior). Signal-flow models do directly allow logical reasoning (like many discrete models do), but can be used to as easily, although still can be related to logic by falsifying specifications in temporal logics [119].

Thus, continuous models provide high-fidelity representations of continuous phenomena, and natural ways to analyze dynamic systems. Although these models are well-suited for traditional control settings (like physical process control), it is increasingly difficult to apply such models to complex systems that operate according to discrete algorithms.

To reap the benefits of discrete and continuous representations within one model, the field of hybrid systems has developed models that combine discrete jumps (discontinuous instantaneous changes in state) and discrete evolutions (continuous trajectories according to a set of differential equations). The common hybrid models are a hybrid automaton [7] or a hybrid program [21, 189]. Typical analyses of hybrid systems are based on forward reachability (given a set of initial states, whether a system eventually reaches a safe or unsafe set of states) and backward reachability (what initial states to prefer/avoid to reach/not reach a given set of states). To compute flowpipes, automated tools (e.g., SpaceEx [79] and Flow* [38]) use various geometric approximations of state sets, such as rectangular hulls, polyhedra, and ellipsoids. Another way to analyze hybrid systems is to specifying their invarians and proving them, as done for the differential dynamic logic and hybrid programs [190]. This analysis is used for hybrid program models in this thesis, with the notation introduced in Section 6.1 and the study found in Subsection 8.2.2.

An important subclass of hybrid automata is timed automata, where continuous evolution is restricted to real-valued clock variables. The reachability problem is decidable for timed automata, and the computations are tractable without approximations. These characteristics checking logical safety and liveness properties of timed automata, using such tools as UPPAAL [140]. Timed automata were used as a semantic basis for component-based models such as BIP [18] and EAST-ADL [158], allowing design and synchronization analysis at a higher level of abstraction.

The advantage of hybrid models is that an engineer can choose which dynamics to represent continuously, and which can be discretized. Continuous dynamics do not depend on a fixed discretization schema (like in discrete models). However, the price for this flexibility is the complexity of syntax and semantics, and the consequent difficulties of analysis and relating hybrid models to other models. This thesis addresses alleviates some of this problem by designing component-based integration abstractions (see Section 6.2) [209].

This brief overview of modeling methods for CPS demonstrates a heterogeneous toolbox of models and analyses, supporting the description in Section 2.1. This diversity may lead to model inconsistencies and unforeseen analysis interactions, as explained in Section 2.2.

9.2 Foundations for the Approach

In this section I describe the related work that serves as a foundation or inspiration for this thesis. Some of this work is used directly by the tools, whereas other serves as a conceptual precursor to the ideas of this thesis. First, I review the work that enables integration abstractions: architectural models that (for views) and theories/tools related to modal logics (for behavioral properties). Next, I focus on the research that underlies verification of integration properties: first-order logic and satisfiability solving. Finally, I discuss the related work that inspires the notions of analysis, contract, and dependency.

My approach builds on a special type of discrete models that originate in the field of software architecture [216]. Architectural models are hierarchical collections of components and connectors. The two architecture description languages used in this thesis are AADL [71] and Acme [86]. Traditionally, these models were used to represent parts of software systems. Although the native constructs of these models are high-level (e.g., a database or a memory chip), multiple extensibility mechanisms (profiles, types, property sets, and annexes) allow specialization to include detailed domain knowledge.

Since software systems can be decomposed in multiple ways (e.g., the run-time structures are organized differently that the design-time structures) [226], software architecture relies on a concept of a viewpoint — a perspective from which an architectural model (or, a view) is created [102, 112]. This concept allows me to use views to represent the parts of models relevant to a particular integration scenario, as described in Subsection 6.2.1.

Multiple prior works have investigated the problem of consistency between software views [41, 62, 176, 196]. In these works, views are treated as projections of a single underlying model onto different dimensions. Instead of assuming a single underlying model that aggregates the views, IPL assumes that views have a shared meta-model, which helps simplify diverse models. My approach is an instance of consistency constraints at the meta-model level [186], which is used in views that were extended to represent physical elements of CPS and perform consistency via graph mappings [24] and arithmetic constraints [200]. Thus, this thesis contributes to the line of work that seeks to develop tools for integrate views as needed, not necessarily create a unified model of the full system [92].

A relatively recent modification of architectural models — Distributed Emergent Ensembles of Components (DEECo) [31, 32] — replaces the typical fixed system configurations with dynamic component assemblies defined by membership predicates. In the DEECo model, communication of components in assemblies is indirectly decided by mapping between component states through coordinating state automata. Such models can be used as an abstraction for models with late-binding or frequently-changing membership, appropriate for ad-hoc wireless networks. The idea of predicates over views is used in IPL as well (in its rigid syntax) to select view elements that satisfy certain criteria, on which then behavioral elements are checked.

Another extension of architectural models is an inclusion of hybrid systems, like the Hybrid Annex for AADL [2]. This annex extends AADL with hybrid annotations that capture variables, invariants, and differential evolution and discrete jump behaviors. This annex enables analysis and generation of hybrid automata from architecture. Similarly, when views serve as integration abstractions for hybrid models (see Subsection 6.2.3), analysis and code generation are enabled. The difference between these extensions is that an annex puts the details of a model in a separate sub-language, not allowing to reference them from integration properties. Another difference is that views support rich connectors, enabling reasoning about and reuse of HP transformations, whereas AADL annexes focus on rich component modeling.

The other integration abstraction (behavioral properties) draws on the field of logic, in particular on modal logics and model checking [45, 98, 156]. Model checking is an analysis that takes a model (typically in a form of a transition system) and a logically expressed property, and checks whether the model satisfies the property. Usually this checking is done by converting the property to an automaton (usually of the Buchi or Rabin class) and composing it with the model [12, 134]. In model integration, it is particularly convenient to use models that can be model-checked: the models have been constructed by engineers, and the properties can be built into IPL as flexible plugins.

The design of IPL is inspired by combinations of the first-order logic [27] with various modal logics. The LTL plug-in draws on the seminal work of Manna and Pnueli [156] on first-order LTL, which has been instantiated in many contexts [42, 87]. Another example is the Quantified Computation Tree Logic (QCTL) [52]. Typically, these works focus on classical properties of logics and algorithms, such as decidability and complexity. Using these logics for verification requires complete unified models of the system or model-free deductive reasoning.

IPL differs from the above models in respecting the modularity and independence of models, and hence not requiring a unified semantic model for evaluating its formulas: separability into interpretable subformulas is sufficient for IPL verification. To avoid semantic unification, IPL uses syntactic restrictions to prevent mixing of semantics. For example, IPL differs from the trace language for object models [42] in that we do not create a full quantification structure in each temporal state. Instead, the IPL design follows earlier proposals of combining open reasoning systems [90], by implementing where the interaction part is implemented by building behavioral queries into the rigid IPL syntax.

The verification algorithm for IPL relies on the reasoning implemented in using Satisfiability Modulo Theories (SMT) solving [181]. To guarantee termination of SMT solving, IPL is limited to decidable combinations of background theories (e.g., uninterpreted functions and linear real arithmetic) that admit the Nelson-Oppen combination procedure [180]. The of SMT solving to reason over the contents of views and find variable values for which more information would be needed from models to decide the integration property . IPL is not bound to a specific set of theories, which may change depending on the contents of views from which the SMT specifications are generated. In practice, modern SMT solvers (e.g., z3 [55]) can use heuristics to solve problem instances in undecidable theory combinations. IPL interfaces with SMT solvers using the SMT-LIB textual format [17], which enables seamlessly switching between SMT solvers, depending on their performance.

The concept of an analysis in this thesis has been borrowed from earlier research on open analytic models [178]. This concept generalizes a broad set of operations on models, such as model transformations, which have been studied extensively for software models [88, 95, 135, 139, 202, 231]. A common assumption behind model transformations is that models have a common and known syntax (usually graph-based UML-like or architectural models), in terms of which the transformation can be specified. Due to the vast differences between CPS models, I do not assume any specific model syntax or structure. While views have a fixed syntax, the analyses are not confined to views (only their dependency specification is). Similarly, I do not assume that the effects of analyses can be comprehensively specified as rules or logical statements (as it is done with model transformations). Analysis contracts can be used to specify some analyses effects relevant to errors and model consistency as guarantees, but I do not rely on completeness of these guarantees. These non-restrictive assumptions let me handle a broad range of CPS analyses, but make it more difficult to fix consistencies or synchronize models.

The analysis contracts are inspired by assume-guarantee reasoning that have been used extensively for compositional verification, dating back to application of contracts to programming languages [165]. In development and verification of CPS, contracts between components provide an important alternative to a unified semantic model, and I discuss their use for integration in the next section. The contracts between analyses allow reasoning about combinations of analyses without knowing or having their implementations. An early application of contracts to analyses [178] has been extended in this thesis. Previously, contracts were restricted to a single domain of resource allocation, and their checking was not necessarily unsound since it explored the system statespace only up to a finite depth (using Alloy [113]). In contrast, this thesis uses an extensible integration property language with a sound algorithm for analysis contracts, and dependencies are based on systematic abstractions (views).

The notion of dependency between analyses is inspired by prior work on dependencies in software engineering and CPS [193, 196, 240]. Often, a dependency is a relation between two artifacts where one artifact needs to read or use the other for some task (e.g., to be compiled or executed). A dependency between decisions or operations is a relation where the subsequent decision/operation requires the preceding one to be made/completed. In CPS, dependencies can be represented in the Dependency Modeling Language (DML) [193], which separates the system's parameters into analytic (predicted characteristics of the design) and synthetic (design-time decisions) variables. With this specification, it is possible to understand the impact of a particular variable modification on the rest of the design, and to create consistency checks on variable values across models (similar to the NAOMI platform [59]). An important insight from this work is that tracking dependencies across disciplines and formalisms is beneficial to integration. Analysis dependencies expand on the above notions to incorporate the tools that make decisions/compute variables into the checking process. Thus, my work focuses on interaction between design tools, instead of design parts.

9.3 Integration Approaches

In this section I review other approaches to the problem of modeling method integration. Some of these approaches address only part of the problem or do not focus on integration, but can be used to discover or prevent integration issues.

For purely software systems, consistency of discrete models is a well-studied problem [62, 65, 184]. Since software models are relatively homogeneous, typically composing or relating these models is sufficient to expose inconsistencies. The composition can either be parallel (for state machines), in terms of inputs/outputs for components, or logical (conjunction/disjunction) for declarative descriptions, or refinement in general [1, 103, 219]. Typically, no special integration abstractions are necessary, and the relating of models can be done via direct references to model elements, matching rules, or by shared metamodels. This thesis target a broader scope of (cyber-physical) models that use heterogeneous formalisms, hence leading to more challenging integration problems [92, 120, 222].

A number of methods and tools have been envisioned to address integration problems in CPS. I organize the approaches for CPS integration along a spectrum, from structural approaches (focused on model syntax or structural decompositons e.g. into components) to semantic ones (reconciling models/analyses at the level of their meaning. e.g. the behaviors they enable) [207]. In the end of this section I review the approaches that combine the two perspectives. From the standpoint of classifications of integration problems [9, 10, 235], I focus on approaches to data, platform, and process integration — the categories of integration problems addressed in this thesis.

Several structural approaches extend the software consistency methods for CPS. One example is *model transformations* [147, 158]. When applied to heterogeneous semantics, these transformations are typically forced to either map models to translate from one model to another, or the translate all models into a unifying semantics. A recent example of this work, the two-hemisphere approach [182] enables transformations from process models and concept models to class models, thus generating an implementation that conforms to models by construction. This work binds to the specific types of models, allowing only limited behavioral reasoning or multi-model verification. Another idea is to use transformations with an abstract interpretation [116], which assumes a shared semantic basis of models in order to make models and their logical properties equivalently traslatable both ways (a restrictive assumption that this thesis does not make). Model transformation has been used to realign data schemas of models and enable their communication at run-time [63]. This communication is set up by generating model adapters and translating data exchanged between models. From the perspective of my approach, these works on model transformations describe useful view algorithms (applicable, for instance, to SysML and AADL views), which can simplify creation of views and their conformance to models.

Another category of structural integration approaches uses *ontologies* [230] and *metamodels* [227]. In integration, metamodels are used to represent relations between models by connecting the types of their elements [164]. When meta-models are shared, it leads to an aforementioned assumption of a single underlying model, which appears too restrictive given the model diversity in CPS. An outstanding instance of the metamodel approach is the ProMoBox framework [166, 167], which allows to integrate property languages given a shared basic metamodel. ProMoBox is similar to IPL in the intent to create a domain-specific language for multi-model properties. Moreover, the restriction to a metamodel enables automatic generation of property languages and traceability of verification results to domain concepts. However, this restriction limits the scope of ProMoBox to a single operational semantics, to which the models are mapped to check the properties. Also, automatic generation of languages restricts the temporal modalities ot a limited number of patterns. In contrast, my approach is based on a manually created integration language, which allows full-fledged behavioral plugins and expressive quantification. Thus, the metamodel approach has a potential to automate generation of models and languages, at the price of restrictive assumptions about the models and limited expressiveness.

A sub-category of metamodeling approaches focuses on coordinating *viewpoints* of different engineers in terms of their roles and responsibilities [188, 227]. The relationships between viewpoints are established via contracts. An example of such contracts which can be found in the work on integrating timing and control engineering [61]. The perspectives of engineers propagate to he level of tools, where compositions based on control flow, data flow, and exchanging execution traces using the Tool Integration Language (TIL) [227]. This approach has an advantage of resolving some integration issues at the meta level, without needing to check specific instances of models. Thus, my work is complementary because it relies on model instances, and has the potential to discover non-metamodel-related integration issues through verification.

A common approach to system integration splits the system's design into *components*, creates contracts for each component, and defines composition of components in terms of their contracts [22, 48, 183, 214]. The platform-based design approach, supported by the metroII environment [54], decomposes a system into layers of components, using their formalized interfaces (which play the role of integration abstractions) to arrive at the desired conclusions about the whole system. Thus, the problem of design and verification is split into checking correctness of black-box composition and checking conformance of an implementation to its interface. This componentization is useful to modularize the heterogeneous semantics of models without composing them directly. However, componentization may face difficulty with addressing cross-cutting interactions that do not manifest at the component interface. For example, security and energy are cross-cutting concerns, which are inconvenient to express and verify for each component. My approach uses the principle of contract-based componentization for the analyses, for which the cross-cutting interactions are described with statements over multiple models.

One of the most related structural approaches is the use of *architectural views* to check topological consistency of models (as graphs of elements). Similar to my approach, Acme Maps [24] constructs view representations of models, in the Acme architecture description augmented for CPS with explicit physical elements (e.g., efforts and flows). These representations can be checked for graph-based consistency (informally, whether elements in one view map to elements connected similarly in another view) with respect to a complete architectural model of the system (termed the base architecture). This thesis extends by providing another integration abstraction, generalizing the relation between views and models (so that it is not necessarily directly related to model structures), and building integration properties and analyses on top of views. In the view-based paradigm, the Sphinx environment [173] represents hybrid programs as UML class diagrams, with a UML metamodel directly tied to the syntax of HPs. The views for HPs that I develop in Subsection 6.2.3 are not directly tied to the syntax of HPs (the abstraction is based on actors and their interactions) and enable connectors with richer semantics (e.g., adding a delay or a measurement error).

Now I review the semantic side of the spectrum of integration approaches. One of the most characteristic behavioral approaches is to relate model behaviors directly [197, 198], evolving the prior work on simulation and bi-simulation relations [78, 89, 91]. This approach bypasses the issue of formalisms and syntactic differences, focusing only on the system behaviors that a model allows. The relations between behaviors thus serve as an integration abstractions. Despite its theoretical guarantees and generality, this approach has limited automation due to the potential complexity of relations between heterogeneous behaviors. Nevertheless, instances of this approach can be automated if tailored to specific models. For instance, transitions in Petri nets were related to transitions in queuing networks with the SYMTHESIS approach [110], and discrete transition systems were related to hybrid automata [13] using contract automata (which conceptually represent a behavior relation).

Another semantic approach is implemented in the OpenMETA toolchain for integration of domain-specific languages [223, 224]. This toolchain is organized in accordance with platform-based design [123] and tackles model integration on three levels: models, tools, and execution. At the model level, this approach uses CyPhyML – a component-based integration language [218] for describing semantic mappings (written as FORMULA [114] specifications) between various models, such as bond graphs and signal flows. Similar to views, OpenMETA uses architectural concepts like signals (i.e., connectors) and typed ports specified in the ESMoL ADL [192]. Model consistency in CyPhyML is defined as logical non-contradiction between semantic interfaces. These interfaces are fixed for a pair of formalisms, and thus allow less flexibility for custom integration properties than IPL. Unlike IPL, CyPhyML commits to continuous-trajectory semantics and supports model execution (a capability that IPL lacks), at the price of limiting the scope of models. At the tool level, OpenMETA transforms models and coordinates tool usage, but it does not support verification of contexts for tool/analyses.

Another category of semantic approaches is *heterogeneous simulation*, where the model behaviors are related through various proxies to construct a unified execution trace. The actor-based simulation platform Ptolemy II [146] implements rich simulations of heterogeneous models. Different models of computation, called *domains*, are split into two components: a *director* that determines the computational model, and *receivers* that manage data exchange for models. Models with different models of computation are integrated using a director's protocol. Although simulation is convenient and potentially more intuitive for engineers, it does not provide comprehensive coverage of the state space, and it only supports integration component-based models through a fixed data exchange interface. Thus, it is not possible to relate models with complex interactions beyond such interfaces. In contrast, my work provides flexible integration abstractions to tailor to an integration scenario. Similar downsides characterize other instances of heterogeneous simulation, such as the GEMOC studio [50] and combinations of VDM [26] with bond graphs [233] and 20-sim [75]. In contrast to these works, this thesis uses verification of multiple models (as opposed to their execution) as a way to discover inconsistencies.

Beyond relating behaviors described in a single logic is a set of approaches to *combinations of logics*, which is an ambitious direction of synthesizing property languages with a priori theoretical guarantees (as done with fibred semantics [82]). One example is hybridization that develops one logic's features on top of another logic, leading to potential reuse of modalities and operators [14]. These specifications can be useful specification exercise, but practical verification is difficult due to high complexity and limited automation. Even when designed in a modular way [130], combinations of logics merge their model structures, which may lead to tractability challenges in practice. My approach keeps models separately, allowing heterogeneous semantics to be changed independently.

Now I comment on the approaches that combine the structural and semantic perspectives. One of the common approaches is to explicitly assign behavior to a component in a component-based model [33]. Each component's behavior is described with a state machine, each state of which determines conditions and constraints on variables. Components' variables are "glued" by interaction elements that also contain constraints. The result of composing the behaviors can be fed into various analyses. For instance, the State Analysis [111] can determine a state trace that satisfies a given goal network – a form of temporal requirement specification for spacecraft activities. Other examples in this category include the Mechatronic UML [19] (translated to timed automata) and BIP [18] (translated to various automata). These models enable verification of coordination and timing properties, but do not aim to generally integrate other models. Another example is the Wright language for the Acme ADL [4], where composition of components is defined in terms of parallel composition of process algebras (which are assigned to each port). Compared to my approach, the above works assume a well-defined and fixed hierarchy between the structural and behavioral perspectives, which is too limiting in CPS integration scenarios.

Another integration approach is based on the Compass Modeling Language (CML) [237], which combines several formalisms, including SysMl, VDM, and CSP. This approach relies of multiple viewpoints and combines architectural abstractions with multiple behavioral formalism. The semantics is given in terms of the Unifying Theories of Programming (UTP), supporting execution of the models. This approach can be considered an instance of a successful a-priori integration of formalisms based on view abstractions. My approach, in contrast, enables aposteriori integration when the formalisms and models are not necessarily picked from several selected candidates.

Finally, some works focus directly on interactions between CPS analyses. For example, JANI [30] proposes a single format for Markov chain models and determines a protocol for analysis tools. This support is at the level of an implementation framework (e.g., in terms of start/stop method calls for an analysis), so my approach can complement it with high-level notions of dependency and context. Another framework focuses on re-analyzing updates to the system after deployment [104]. The updates are considered from multiple viewpoints, and reconciled by constraining the configuration space, potentially affecting other viewpoints. The analyses interact in terms of constraints on the design space. This solution assumed a fixed single model of the system's design space, but it goes beyond work preventing/detecting incorrect analysis interactions and offers and approach to fix them. This approach can be used in parallel with the analysis execution platform because the updates are typically one-off and unexpected, while the analyses are repeatable and known a priori.

To summarize, several key differences separate this thesis from the existing integration approaches. First, I use a combination of structural and behavioral features, but without a predefined relation between them. Second, I focus on verification using multiple models, with heavy reliance on their reasoning engines (as opposed to simulation or simple analyses). Finally, my approach does not require a shared underlying semantics or meta-model from heterogeneous modeling methods. Also, many of the above approaches are complementary with mine, and can be used in parallel or synergistically.

Chapter 10

Discussion

This chapter discusses the broader interpretations and implications of this thesis, beyond the technical descriptions in Chapters 5 to 7. I split this chapter into four parts. First, I summarize the scope of applicability of the proposed approach to modeling method integration. Second, I summarize the limitations of the approach, including the concerns of its practicality. Then, I discuss the key design decisions behind the approach. Finally, I describe the directions of future work enabled by this thesis.

10.1 Scope

I describe the scope of applicability of the integration approach from four perspectives:

- The types of models that the approach can integrate.
- The types of analyses that the approach can integrate.
- The types of integration properties that the approach can check for the above models/analyses.
- The types of domains and systems to which the approach can be applied.

I start by considering the scope in terms models. The structural abstractions (views) apply to a relatively wide range of models by representing hierarchical key-value information in these models, independent of the syntax or the type of dynamics. However, the views make three assumptions about the models. One is that the model elements represented in views are finite, leading to a finite view. Another assumption is that these models are complete in terms of the said elements; i.e., that the system does not contain elements that the model is intended to capture, but does not. A model can be complete with respect to one view/viewpoint, and incomplete with respect to another. This completeness is required because views do not have a mechanism to indicate that more elements may be available (I return to the discussion of completeness later in this chapter). The final assumption of views is that the model is unambiguous regarding the elements and properties of a view. For instance, if a model allows several values for the energy of tasks, it cannot serve as a basis for a view with tasks that have fixed energy values cannot be used for it.

The requirements for a behavioral model in my approach are more restrictive than for models abstracted with views. First, the model has to interpret a model property language, with a sound algorithm to check/query the statements. The algorithm should terminate, at least in practical conditions if not theoretically. This requirement is satisfied mainly by formal models. In addition, similar to the case of views, the model has to be unambiguous and complete with respect to the statements in the property language: the model should contain the amount of information to answer each query, and not contain ambiguous answers. Thus, simulation models without model checking capabilities are not in the scope of behavioral models for my approach: simulation generates and checks individual traces (e.g., for falsification), but does not reason over all possible traces in the model.

The scope is rather broad in terms of analyses: any analyses that have inputs/outputs described in terms of view elements (i.e., the analyses with unambiguous underlying models) can be integrated with my approach. The only limitation for sets of analyses is that their dependencies should not form dependency cycles; however, this restriction does not put any individual analysis outside of the scope. The analysis contexts can be checked if they can be specified using the integration properties described below.

The approach focuses on mixed structural-behavioral integration properties, in which structural (rigid) elements bind multiple isolated behavioral expressions. As a trivial case, the approach can check purely structural (i.e., written over views) or behavioral (i.e., written in property languages for behavioral models) properties. The approach cannot directly specify properties that mix behavioral languages for different models, since this capability would likely require model composition or assumptions on how their behaviors relate.

My approach does not place specific restrictions on the domains and systems, as long as they can be described as CPS or CPS-related. Given the evidence in Chapter 8, the approach can be applied to a variety of domains, provided that the above assumptions on models, analyses, and integration properties hold. The benefits of the approach are likely to be experienced for systems with multiple dependent models/analyses and cross-cutting systemic properties like timing, energy, and security.

10.2 Limitations

The limitations describe the potential shortcomings of the approach. Some limitations lead to the aforementioned scope restrictions, while some apply to the models/analyses/systems *within the scope*. Thus, these limitations can be interpreted as threats to external validity, affecting the transfer of the approach from the case study systems to other systems that are in the scope (as defined in Section 10.1).

One assumption of the approach is the *presence* of models, in a syntactically complete and interpretable form. The approach does not produce system models. Model-free reasoning is not currently supported. For instance, the types of view/model elements are not sufficient to determine consistency (without the views/models themselves). The approach does not provide general guarantees of analysis applicability, without a specific model to which the analysis would be applied. The only tool in this thesis that can be used without instances of models/views is dependency analysis in the analysis execution platform (see Equation (7.5) in Section 7.2). Nevertheless, my approach can be complemented with model-free methods that would reason abstractly about integration properties. For instance, if it can be a priori derived that, given certain assumptions, any model of some system would satisfy an integration property, then only the assumptions need to be checked by my approach, instead of the integration properties.

The approach also assumes that queries/checks of behavioral models can be *trusted*. This assumption relies on several characteristics of these queries and models. One required characteristic is soundness, introduced in Subsection 6.3.2 and discussed throughout the thesis. Another characteristic is completeness with respect to the elements referenced in a query (described above in Section 10.1). These characteristics are necessary so that an engineer can write an integration property that is semantically equivalent to the intended consistency relation (represented with $\mathrm{semequiv}$ in the description of the integration argument in Section 4.4). Trustworthiness is only required from behavioral checking, but not necessarily from all analyses: correctness of an analysis can be checked with a contract (see the third case in Section 4.3).

Views used in the approach are expected to *conform* to their respective models in sound and complete way, according to the chosen viewpoint. This assumption is needed for the views to adequately represent the elements of interest in models. Syntactic incompleteness (e.g., underspecification of property values) is permitted in views to an extent that does not make the view unsound or incomplete with respect to its viewpoint. If these assumptions are violated, integration checks may produce false positives (e.g., erroneous alerts for an existentially quantified property when a view is unsound) and false negatives (e.g., erroneously satisfaction of a universally quantified properties when a view is incomplete).

Analysis execution is limited to sets of analyses without input/output *dependency cycles*. These cycles cannot be resolved by the current execution platform, leaving the execution of these analyses to engineers. As mentioned above, some cycles may be caused by the coarse granularity of types in views (in terms of which the dependencies are specified), so some of the cycles may be modeled away. Also, a number of ways have been considered to resolve these loops [210]. Nevertheless, some analyses have tight mutual dependences or draw on competing approaches to system design, and therefore may be not automatically executable within the same dependency graph.

Expressiveness of the approach has been shown sufficient for the integration scenarios available in the four case studies. However, expressiveness is limited in multiple ways, representing a threat to construct validity (i.e., whether IPL specifications represent meaningful integration properties). In some cases, the constructs of views may be too coarse-grained to specify an intended integration property. For instance, it is difficult to use a view to capture the dynamics of a scheduling policy, beyond naming it (e.g., rate-monotonic scheduling). Another potential consequence of limited view expressiveness is dependency cycles of analyses (described below as a separate limitation as well). These cycles may appear due to the views not distinguishing the necessary types; for instance, if both CPUs and RAMs are typed as hardware devices in a view, the CPU frequency scaling analysis may have a circular dependency with a memory allocation analysis. Nevertheless, it is often possible to refine views to represent the model elements at the required level of granularity.

Another expressiveness limitation is that a behavioral property language may be based on a limited modal logic. One example is expressing a relation of values from multiple states, which is not possible in many modal logics like LTL. For instance, it is not possible to express that the next state increments the current state's energy by a given amount. To enable such statements, an extension to a language may be needed, such as a nullary "memory" function that refers to a value in another state [15]. The tools available in practice may not support such extensions, limiting the expressiveness of checkable properties. In this case, the hope is that if a property language are adequate for the system (otherwise, why would its model be used in the first place?), then this language would be adequate for the system's integration properties.

Finally, the IPL syntax is limiting to expressiveness. As noted above, IPL cannot describe properties of models with intertwined behaviors, where a property cannot be separated into subformulas evaluated by individual models. Another limitation of the IPL syntax is absence of first-order logical theories that are decidable (in combination with other theories). For instance, the assumption of minimal sensor trust for $\mathcal{A}_{ctrl}$ (see Subsection 8.3.2) is not expressible SMT due to the lack of operators for set cardinality or counting array elements. However, as theories become available (e.g., for lists, arrays, and strings) can be incorporated into IPL, independently of the behavioral models.

The validation case studies have shown that the *performance* (in terms of time and memory required for verification) is adequate to realistic systems. However, scalability of the approach is likely to be a challenge for larger models, as CPS grow in scale and complexity. These issues arise primarily from the reliance of the approach on formal verification tools (intrinsically, IPL has a remarkably low overhead, as shown in Subsection 8.1.2). The performance of these tools depends on carefully designed abstractions and constraints, which in the case of my approach are concentrated in the integration abstractions and properties. Therefore, their judicious use can mitigate the scalability issues. As the performance of SMT solvers and model checkers improves in the future, they can replace the existing versions in the IPL implementation, leading to improved IPL performance.

10.2.1 Practical Concerns

This section addresses the prominent practical concerns.

The *complexity* of the proposed specifications (IPL formulas and analysis contracts) threatens their application in other CPS: what if an average engineer does not possess the required expertise in modeling, logic, and verification? What if no engineer has the deep understanding of several models required to specify integration properties? What if these specifications are too complex to write down without introducing more errors?

Essential complexity is inevitable in modeling method integration. In addition to the complexity of each modeling method, there is extra complexity in how these methods may relate to each other, and what the effects of that relation might be. I simplify the complex task of integration by decomposing it into several simpler tasks and solving some of them. The first task is to design an integration schema, which describes how the models, analyses, as well as abstractions and specifications for them, fit together. This thesis provides one such schema, with some parts of it implemented. I created languages and representations to capture the constructs of integration (in the form of integration properties, views, and analysis contracts). I also provide algorithms (with implementations) to perform integration based on these constructs.

Another task is to customize integration abstractions for the domain. These customizations include finding the types and properties for views and preparing the behavioral abstractions (choosing state variables and initialization parameters, creating an IPL plugin). These tasks require an understanding of the integration approach, and may be difficult to perform for some engineers. However, it is possible to reuse the results of these tasks for models and systems in the domain by, for instance, building up a library of implemented viewpoints. Thus, the investment into integration abstractions can be amortized over multiple systems.

The remaining integration task is to formulate integration scenarios, and specify/check the integration properties in each scenario. Formulating the scenarios requires a substantial understanding of the models and the meaning of their consistency. This formulation can perhaps be accomplished as a collaborative effort between teams with different expertise, similar to how an API is negotiated between a service client and a service provider. The specification and verification of integration properties does not necessarily require deep understanding of both models — merely of the integration abstractions — and can be accomplished by an engineer who is not aware of the formal aspects of integration. Therefore, my research simplifies a complex task of model integration by defining a schema for integration, providing of its parts, and enabling reuse of its other parts.

A similar decomposition of tasks applies to integrating analyses. The execution platform can be reused for any analysis contracts. Contracts can be specified within certain domains (i.e., with respect to a set of views) and reused, modifying the contracts only when a new domain is considered. Once a domain is described (as a domain signature, see Definition 24 in Section 7.1), writing a contract requires understanding only one analysis — not necessarily all of the analyses in the domain. Hence, some tasks require a broad understanding, but their results can be reused, and other tasks require narrow understanding that is generally available.

Another concern for practical adoption is the *return on investment:* is the additional modeling effort behind my approach justified by its benefits? In some cases, it is possible for the costs to outweigh the benefits. For instance, in projects that do not use many dependent analyses or heterogeneous models with overlapping referents, the investment in creating integration abstractions may not pay off. However, I argue that for safety-critical and mission-critical systems that use multiple overlapping/dependent models/analyses, the investment is likely to pay off: violations of integration properties threaten safety arguments, and the costs of failures are prohibitively high in such systems. The costs of my approach can be further reduced by targeted application of integration abstractions (the guidelines are described in Section 6.4) and reusing the abstractions and their automated generation (as discussed above).

10.3 Design Rationale

Now I turn to a discussion of the design decisions in the approach and their justifications.

One of the central themes in the design of the approach, and particularly IPL, is the *relativity* of integration and consistency to the engineering needs. For example, some models may need to be perfectly consistent, whereas others can tolerate a bounded amount of inconsistency. Similarly, one project may impose narrower bounds than others. The conditions of consistency are determined by engineers, as opposed to imposing a fixed a priori notion of consistency (e.g., a graph homomorphism), thus enabling the necessary tolerance of inconsistency and uncertainty [40]. The notion of relativity also affects execution of analyses: assumptions and guarantees can be strengthened or relaxed depending on the need to guarantee varying degrees of correctness for analysis execution.

The relativity of integration is supported by the decision to not rely on the “base architecture” of the system — a complete architectural model of all elements that constitute a system’s design [23]. Prior work relied on the base architecture to check that view elements in different views are connected in a compatible pattern. This thesis does not rely on the base architecture, relaxing the assumption that a full model of the system is needed. As a result, consistency can be checked “locally“ — between any pair of models that has abstractions and integration properties — and models can evolve independently, without coordinating changes to a complete architecture of the system. Nevertheless, my approach is compatible with prior work, and if a base architecture is provided, structural consistency can be checked using the views created for IPL.

My approach does not attempt to unify semantics of models directly, thus avoiding classic composition methods (i.e., those that create a combined model from several constituents). This decision allows me to integrate models with heterogeneous semantics that do not allow straightforward composition, and avoid the problem of composing heterogeneous compositions of models. Moreover, the need to commit to some composition semantics may put additional constraints on models and analyses, impeding their independent co-evolution. Instead, the thesis uses logical connections over abstractions of models, without restricting the semantics or evolutions of models. Composition of the abstractions is allowed due to their unified semantics: views can be composed to create larger views, and statements over views and models are combined in IPL formulas. This way, models can be used for integration and remain independent at the same time.

One of the central design decisions in IPL is that the rigid layer (comprised of views, which are reasoned about by an SMT solver) serves as a “glue” for behavioral models and their properties. In other words, behavioral properties are treated as plugins into rigid statements (which are interpreted by an SMT solver) — and not vice versa. Thus, the syntax of IPL puts rigid statements on a higher level than flexible statements (as shown in Figure 5.1 in Section 5.4. As a result, the verification algorithm performs inter-model reasoning at the view level, “diving” into behavioral models as needed. This design was chosen for three reasons:

- Behavioral models represent a variety of dynamics (discrete, continuous, probabilistic). To use these models as a “glue,” I would need to unify their semantics or compose their dynamics, which is a complex task that is avoided in my approach (as noted earlier).
- Models often focus a system’s specific part or aspect, providing detailed information about it. To reason about the system as a whole using one model, this model would need to be substantially extended and customized to fit the other models. This task goes against the principle of preserving independence of models , articulated above. Unlike models, views are designed to represent a “global” perspective on the system, and are easier to relate/compose due to their homogeneity and relative simplicity.
- First-order logic is instrumental for reasoning across multiple models: quantifiers enable general properties (without references to specific elements of the system — only their types), and uninterpreted functions can be used to reason about partially-known behavioral semantics. Instead of combining these first-order aspects with each model’s logic (which would require customized reasoning engines), I use an out-of-the-box SMT solver to reason over views, which have a uniform way to be translated to SMT specifications.

The integration abstractions in this thesis are designed to enable application and reuse of prior work. Views link CPS models and architectural representations, thus enabling application of architectural analyses in schedulability, synchronization, and design synthesis [72, 101, 203, 238]. Behavioral properties can be instantiated for existing logics and property languages, enabling application of model checking and theorem proving. Analyses can be extended with existing model transformations and optimizations. Thus, the approach is design to adding new modeling methods without up-front planning, in contrast with existing top-down approaches that prescribe specific formalisms and analyses.

This work has treated viewpoints (i.e., algorithms to produce views for models, see Definition 11 in Subsection 6.2.1) as independent algorithms. Their independent treatment allows additions or changes to a viewpoint without propagation to other viewpoints. Nonetheless, dependencies between viewpoints are compatible with the approach: constraints are placed on the views, not on the process of their creation or update. An example of a viewpoint dependency is when a view is created using information in another view (along with a model). Combination of viewpoints can enable reuse and reduction of manual effort in creating and updating views

Analysis contracts express input/output dependencies between analyses in terms of view types only, without references to model elements. In practice, analyses change models, and these changes propagate to views, changing their elements (the types of which are specified in the contracts). Behavioral properties do not have a suitable granularity for dependency specifications: the valuation of constraints and queries may change due to any model changes, and considering dependencies at that level would lead to dependency cycles. Specifying dependencies in terms of more granular elements (e.g., specific states or segments of traces) is feasible, but would require a unified semantics for the models, which, as discussed earlier, is not the direction taken in this thesis.

One might argue that embedding one model (or its parts) into another is sufficient for integration. I consider embedding a form of model composition, where the model with embeddings is formed from two or more other models. Some integration issues can be discovered and prevented through embedding, such as the contradictions between the embedded information contradicts and its local context. However, embedding does not address the full scope of issues that my approach targets. For instance, in the power-aware robot case study (Section 3.1), the energy values from M_{power} are embedded in M_{plan} as effects of transition, yet this does not reconcile these models in terms of their treatment of turns.

One might also argue that run-time checking of models can be used instead of design-time model integration. While runtime verification can be a useful complement (as discussed below in Section 10.4), it does not prevent the failures that cannot be addressed by online responses. For example, while executing a mission, a robot with a power-related integration issue may detect that the mission is unsafe and it may run out of power. However, without charging stations nearby, the robot does not have a way to resolve this issue. Instead, design-time integration would give the engineers an opportunity to fix the issue in the models.

This thesis does not prescribe any specific engineering process, focusing instead on providing tools for modeling method integration. These tools can be incorporated in different processes and stages of engineering, from up-front system design to post-factum application (like in the validation case studies). Some of the tools can be used independently, provided that they are based on the prescribed integration abstractions. IPL can be used without analysis contracts or the execution platform, but it requires at least views or behavioral properties. The analysis execution platform can be used without IPL, but requires views.

In my experience from the case studies, one of the most challenging tasks in the approach is finding a potential integration issue, and framing it as an integration scenario. Describing a scenario requires a set of models/analyses and a property that could be violated due to an interaction between these models/analyses.

To find integration scenarios for models, one can start with specific instances of models that appear related or redundant. Often this relation or redundancy occurs when several models represent overlapping functionality. In CPS, this functionality could include control, planning, scheduling, sensing, and communication. Integration properties are often found when requirements relate multiple quality aspects with potential conflicts, such as safety and efficiency. Differences in treatment of these qualities by models could lead to violations, and an integration property would check for such violations. For instance, time efficiency and energy economy may conflict in a robot, possibly leading to their respective models having an inconsistency.

Finding integration scenarios for analyses is relatively simpler than for models: identifying the analyses and their inputs/outputs often highlights the dependencies and potential context mismatches. Sometimes it is difficult to draw boundaries between individual analyses: multiple operations may be tightly connected in a workflow. To focus on potential integration issues with analyses, one can look for hints that fully automated execution does not apply: whenever an analysis needs to be reversed or human assistance required, context may be not appropriate a priori. In addition, frequent communication between teams beyond the expected process may indicate dependencies between their analyses.

10.4 Future Work

The future work enabled by my approach can be grouped in two categories: short-term improvements that enhance and develop the original approach, and long-term ideas that significantly modify the approach or extend it in new directions.

10.4.1 Short-term Improvements

Multiple short-term improvements can reducing the manual effort required to use the approach. Some of these improvements focus on view creation. Composition and integration of viewpoints [163], as mentioned above, can automate and reduce the effort for view creation, similarly to how it was previously done in non-CPS settings [62, 204]. For instance, a manually created view that indicates the actors in a model can be sufficient to automatically construct a timing and an energy view for these actors. If multiple views are created automatically, view creation and update can be seen as a set of meta-analyses and encoded in the execution platform to further automate the process. View-related operations can be arbitrarily complex and automated, producing a set of sound and complete views that are fed into the IPL verification. From another perspective, compositions of views and behavioral properties can be developed, like it was done for $\mathsf{d}\mathcal{L}$ view formulas, which are written for HP views (Section 6.3). Another example is automatically generating behavioral languages given a view and a viewpoint. Such compositions could allow engineers to interact primarily with integration abstractions, reducing the restore the conformance of these abstractions to their respective models.

Another way to reduce manual effort is to create IPL macros that replicate the same statement for multiple variables. For example, in the current syntax of IPL, one often has to repeat a similar rigid constraint for several quantified variables. For instance, the constraint may be that the robot needs to face in the direction of its task (suppose described by a predicate $P(v_i)$), repeated for every variable representing a task of a robot (v_i). These repetitive constraints could be replaced by a macro FOR$(i, 1..3, P(v_i), \wedge)$, where i is the counter, v_i is a quantified variable, P is some rigid expression of interest, and $\wedge$ is the connecting operator. This macro would be transformed into $P(v_i) \wedge P(v_2) \wedge P(v_3)$, which is a well-formed IPL formula. Note that a macros is a syntactic shortcut to finite and well-formed IPL formula — not a loop construct or an instance of second-order quantification. These macros would make IPL formulas more compact and readable, and likely reduce the effort to write and debug IPL specifications.

Another short-term direction is to improve performance and scalability of checking integration properties. SMT can be used through an incremental API, where additional constraints are added to an in-memory satisfaction problem, as opposed to re-generating a new problem in a standardized syntax, as it is currently done. Views can also be translated into more efficient SMT specifications. In particular, view elements can be represented with uninterpreted constants instead of integer IDs used in the current version. Furthermore, SMT can be generated selectively for the elements and types used in an integration property, reducing the size of SMT specifications and their checking time. Checking behavioral properties can be sped up by parallelizing the checks (there is no dependencies between them) or using parametric model checking.

A different approach to improve verification performance is to develop deductive symbolic reasoning for IPL specifications. A proof calculus would allow checking formulas based on their assumptions without using models. As a result, only simplified assumptions may be checked on specific models, and some integration properties could be guaranteed a priori, regardless of the specific design. Developing deductive model-free reasoning may require behavioral plugins to include proof rules, to enable equivalent transformations of flexible subformulas.

Applications of the approach in several CPS domains look particularly promising. One domain is at the intersection of robust control, security, and privacy. Does detection of anomalies and response to them as attacks [175] violate privacy restrictions? Do privacy-enhancing mechanisms increase sensor noise beyond the robustness of attack detection [187]? Another interesting application domain is medical CPS: what are the design-time conditions for device models to be consistent with patient models [37]? Finally, the approach can be extended to behavioral models beyond model checking, in particular simulation, optimization, and game-theoretic models models. Simulation could be used to falsify properties in signal temporal logic [155]. Game-theoretic can be used to check equilibria and constraints on the synthesized solutions. Optimization can be incorporated with subformulas that describe of convex problems and handing them off to a separate solver [217].

10.4.2 Long-term Research Directions

A significant extension of the execution platform is to automatically resolve circular dependencies between analyses. To resolve the loops dependencies, one could search for *fixpoints* — system designs that are invariant to application of all the analyses that form a cycle. A preliminary exploration has shown that several approaches to fixpoint search are feasible: analysis iteration, constraint solving, and genetic search [210]. This extension makes it possible to have unrelated analyses converge on an acceptable system design out-of-the-box, without the modifications to the analyses to enable their cooperation.

The analysis platform can be extended for deductive reasoning about the system's properties, as shown in Figure 10.1. Analyses would use properties that currently hold to discharge their assumptions, and contribute properties to the current knowledge base. Changes to models would invalidate some that would be either re-verified or discarded. The status of each assumption is tracked: whether it follows from the current set of facts, or needs to be proven. The guarantees may be checked (in case they represent heuristics), added (trusted to be true), or removed from the facts. Specialized analyses would derive new facts from the current properties, without changing the models. With this approach, the amount of re-verification can be reduced dramatically because the proven properties are recorded and used for new tasks. Analysis contracts can also be refined into specific contracts for individual tools, which adhere to the general contract, but introduce additional facts depending on their specifics.

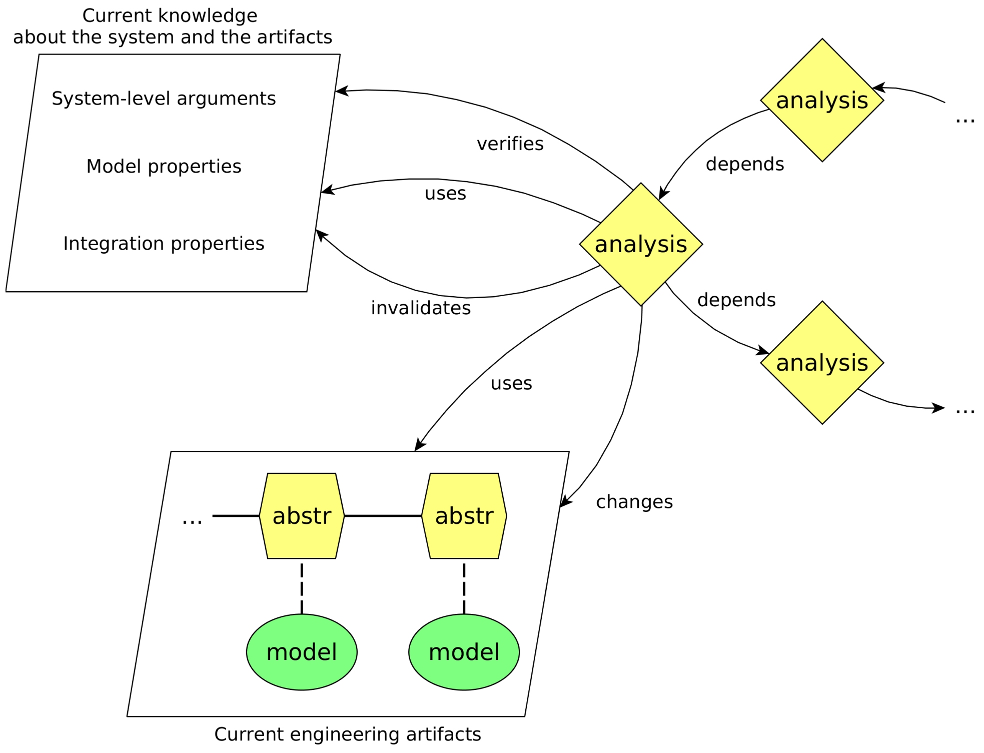


Figure 10.1: A schema of knowledge management for analyses. Arrows indicate operations.

This thesis proposed a new specification approach, but the problem of specifying complex multi-model properties will remain open [206]. Fundamentally, expressiveness of IPL, particularly its rigid part, can be extended to enable higher-order logics. Higher-order logics would enable comprehensive properties that might currently need to be manually split into several. The main obstacle to using these logics directly is absence of decidable procedures to check their satisfiability. This obstacle could be side-stepped by using model-free symbolic reasoning (which has been discussed above) and reducing higher-order formulas to multiple first-order formulas.

Alternatively, in the case of second-order logics, quantified functions with bounded domains and ranges could be decidable by existing SMT solvers. Integrating second-order verification into IPL would require substantial changes to the IPL's syntax, semantics, and implementation.

One can extend the integration approach to handle incomplete models, enabling integration at an early stage in modeling. One could replace values in views and models with constraints, which would be combined with IPL verification, thus performing design space exploration in parallel with integration. Another idea is to allow some models or views to have an open-world assumption, and filter out integration properties that would be a priori unsound given those assumptions. Finally, interactive integration checking can request additional information on view elements (their types and properties) and model elements (state variables and traces), thus implementing a manual abstraction refinement procedure.

A different approach to address incompleteness is to transfer IPL verification (or its part) to run time. For instance, instead of assuming that a power model returns accurate energy values for tasks, a robot could observe and approximate the values, periodically checking assumptions that imply an integration property. While this approach may provide weaker guarantees than design-time exploration, it may allow checking properties with higher precision and avoid combinatorial explosion. To monitor a full integration property, a middleware may be generated to aggregate information from multiple models, which would need to be updated at run time. An interesting case of online checking is learning, where models change in response to the collected data, potentially requiring re-writing of integration properties.

An ambitious research direction is automatic fixing for integration issues. The first step would be to localize the part(s) of models that need to change when an integration property fails. This step is perhaps the most difficult as well: while a counter-example is available, it does not necessarily suggest any prioritization of model elements that were used in the property. If the first step is completed, the second step would be to generate a set of potential fixes that resolve the inconsistency. This can be done through a variety of formal and stochastic search methods. Finally, a single fix needs to be applied, requiring either an evaluation metric for model fixes or human judgment.

Finally, human factors related to modeling method integration could be investigated from two perspectives: integration problems and integration tools. For the former, what factors lead to integration issues: engineers' background, team communication, dependencies between models, structure of models or formalisms? For the latter, what are the helpful user interfaces to support creation of integration abstractions and properties, and how to present integration results to an engineer proficient with some models, but not others? Answering these questions would lead to improved adoption and outcomes of modeling method integration.

Chapter 11

Conclusion

This final chapter summarizes the thesis and its contributions. In this dissertation, I addressed the problem of modeling method integration, which manifests as inconsistencies between models and incorrect interactions between analyses. I particularly focus on inconsistencies that involve erroneous relations between structures and behaviors in models. Incorrect interactions occur due to incorrect invocation order (with respect to their input-output dependencies) and execution in the context of models that the analysis was not designed to use.

To address the above problems, the thesis proposes as approach that relies on two key entities: integration abstractions and integration properties. The former are representations of models suitable for integration. In this thesis two abstractions are used: component-and-connector views (to represent static structures of models) and behavioral properties (queryable expressions in model-specific property languages). These abstractions are used to specify integration properties — assertions of relations between structures and behaviors of multiple models.

On top of the abstractions and integration properties, the approach builds the solutions to the aforementioned problems. Inconsistencies in models are discovered by verifying integration properties with a cooperative use of SMT solving and model checking. To prevent incorrect interactions of analyses, their invocation is managed by the analysis execution platform, which requires each analysis to be accompanied by an analysis contract. The inputs and outputs, specified in terms of elements of views, determine a correct execution order of analyses. The effects of an analysis take place only if its assumptions and guarantees (specified and checked as integration properties) hold on the models that the analysis uses, hence preventing context mismatch.

The approach was validated from the theoretical and empirical standpoints. The verification was proved to be sound and terminate under certain assumptions. The empirical validation consisted of four integration case studies for different systems: energy-aware planning in a mobile robot, collision avoidance in a mobile robot, thread/battery scheduling in a quadrotor, and reliable/secure sensing in an autonomous vehicle. The approach was successfully customized to these systems, which involve multiple domains and modeling methods. These case studies validation demonstrated that the approach is expressive enough to capture complex and relevant relations between models. The approach was also shown reasonably scalable and flexible for practical application.

11.1 Contributions

This thesis makes the following contributions to the *theory of modeling and verification of cyber-physical systems:*

1. A description of views and behavioral properties as integration abstractions, enabling integration of structural and behavioral elements of models. This description includes the definitions and sufficient conditions for integration arguments (Chapter 6).
2. A definition of the Integration Property Language, comprised of the syntax and semantics. The syntax enables combination of structural and behavioral aspects (Section 5.4), whereas the semantics evaluates each sentence in a way that (Section 5.5).
3. A verification algorithm for IPL statements. The algorithm checks the validity of IPL statements on a set of models and views (Section 5.6).
4. A proof of the soundness of the verification algorithm and its termination conditions (Subsection 8.1.1).
5. A specification schema of analysis contracts, which includes the syntax and semantics for each part of a contract (Section 7.2). The contracts describe the input-output dependencies of analyses, as well as their assumptions and guarantees on the modeling context.
6. An algorithm to execute analyses given their dependencies (Section 7.3). The algorithm is guaranteed to find a correct order for any set of analyses without dependency cycles, and not apply any analyses that do not match their context.

This thesis makes the following contributions to the *practice of engineering cyber-physical systems:*

1. An implementation of the IPL editor and verifier based on the Eclipse/OSATE environment [213].
2. A generator of SMT specifications from AADL views (part of ACTIVE) [211].
3. An implementation of the analysis execution platform based on the Eclipse/OSATE environment (part of ACTIVE) [211].
4. A generator of hybrid programs from HP views based on the AcmeStudio environment [212].
5. Guidelines for practical application of the integration abstractions. These guidelines consist of a comparison of the circumstances when each abstractions would be convenient or difficult to use (Section 6.4), and a description of techniques for automating model-view conformance (Subsection 6.2.4).

Bibliography

[1] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010. ISBN 978-0-521-89556-9. 9.1.1, 9.3

[2] Ehsan Ahmad, Brian R. Larson, Stephen C. Barrett, Naijun Zhan, and Yunwei Dong. Hybrid Annex: An AADL Extension for Continuous Behavior and Cyber-physical Interaction Modeling. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 29–38, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3217-0. doi: 10.1145/2663171.2663178. URL `http://doi.acm.org/10.1145/2663171.2663178`. 9.2

[3] Robert Allen and David Garlan. Formalizing Architectural Connection. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 71–80, Sorrento, Italy, May 1994. Awarded ICSE2004 Best Paper (ICSE-10). 6.2.1

[4] Robert Allen, Remi Douence, and David Garlan. Specifying and Analyzing Dynamic Software Architectures. In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, volume 1382 of *Lecture Notes in Computer Science*, Lisbon, Portugal, March 1998. Springer. An expanded version of the paper Specifying Dynamism in Software Architectures, which appeared in the Proceedings of the Workshop on Foundations of Component-Based Software Engineering, September 1997. 9.3

[5] Rajeev Alur. *Principles of Cyber-Physical Systems*. The MIT Press, Cambridge, Massachusetts, April 2015. ISBN 978-0-262-02911-7. 9.1.1

[6] Rajeev Alur and Thomas A. Henzinger. Logics and models of real time: A survey. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, number 600 in Lecture Notes in Computer Science, pages 74–106. Springer Berlin Heidelberg, January 1992. ISBN 978-3-540-55564-3 978-3-540-47218-6. URL `http://link.springer.com/chapter/10.1007/BFb0031988`. 4.1

[7] Rajeev Alur, Thomas A. Henzinger, and Howard Wong-toi. Symbolic Analysis of Hybrid Systems. In *Proc. of the IEEE CDC*, 1997. 2.1, 9.1.2

[8] Arthur Stephenson, Lia LaPiana, Daniel Mulville, Peter Rutledge, Frank Bauer, David Folta, Greg Dukeman, Robert Sackheim, and Peter Norvig. Mars Climate Orbiter Mishap Investigation Board Phase I Report. Technical report, NASA, November 1999. 1

[9] Fredrik Asplund and Martin Torngren. The Discourse on Tool Integration Beyond Technology, A Literature Survey. *Journal of Systems and Software*, 106, 2015. ISSN 1873-1228.

doi: 10.1016/j.jss.2015.04.082. 9.3

[10] Fredrik Asplund, Matthias Biehl, Jad El-Khoury, and Martin Trngren. Tool Integration beyond Wasserman. In Camille Salinesi and Oscar Pastor, editors, *Advanced Information Systems Engineering Workshops*, number 83 in Lecture Notes in Business Information Processing, pages 270–281. Springer Berlin Heidelberg, January 2011. ISBN 978-3-642-22055-5 978-3-642-22056-2. URL `http://link.springer.com/chapter/10.1007/978-3-642-22056-2_29`. 9.3

[11] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline-Monotonic Approach. *IFAC Proceedings Volumes*, 24(2):127–132, May 1991. ISSN 1474-6670. doi: 10.1016/S1474-6670(17)51283-5. URL `http://www.sciencedirect.com/science/article/pii/S1474667017512835`. 3.3

[12] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008. ISBN 978-0-262-02649-9. 3, 9.2

[13] Stanley Bak and Sagar Chaki. Verifying Cyber-Physical Systems by Combining Software Model Checking with Hybrid Systems Reachability. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*, 2016. 1, 9.3

[14] Luis S. Barbosa, Manuel A. Martins, Alexandre Madeira, and Renato Neves. Reuse and Integration of Specification Logics: The Hybridisation Perspective. In Thouraya Bouabana-Tebibel and Stuart H. Rubin, editors, *Theoretical Information Reuse and Integration*, Advances in Intelligent Systems and Computing, pages 1–30. Springer International Publishing, 2016. ISBN 978-3-319-31311-5. 9.3

[15] Jeffrey Barnes. *Software Architecture Evolution*. PhD thesis, Carnegie Mellon University, 2013. 10.2

[16] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 171–177. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-22110-1. 5.7

[17] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. 5.7, 9.2

[18] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling Heterogeneous Real-time Components in BIP. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, SEFM '06, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2678-0. doi: 10.1109/SEFM.2006.27. URL `http://dx.doi.org/10.1109/SEFM.2006.27`. 6.2.3, 9.1.2, 9.3

[19] Steffen Becker, Stefan Dziwok, Christopher Gerking, Christian Heinzemann, Wilhelm Schfer, Matthias Meyer, and Uwe Pohlmann. The MechatronicUML Method: Model-driven Software Engineering of Self-adaptive Mechatronic Systems. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 614–615, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2768-8. doi: 10.1145/2591062.2591142. URL `http://doi.acm.org/10.1145/2591062.2591142`. 9.3

[20] Kirstie L. Bellman and Christopher Landauer. Towards an Integration Science: The Influence of Richard Bellman on Our Research. *Journal of Mathematical Analysis and Applications*, 249(1):3–31, September 2000. ISSN 0022-247X. doi: 10.1006/jmaa.2000.6949. URL `http://www.sciencedirect.com/science/article/pii/S0022247X0096949X`. 2.1

[21] A. Benveniste, T. Bourke, B. Caillaud, J. Colao, C. Pasteur, and M. Pouzet. Building a Hybrid Systems Modeler on Synchronous Languages Principles. *Proceedings of the IEEE*, 106(9):1568–1592, September 2018. ISSN 0018-9219. doi: 10.1109/JPROC.2018.2858016. 9.1.2

[22] L. Benvenuti, A. Ferrari, L. Mangeruca, E. Mazzi, R. Passerone, and C. Sofronis. A contract-based formalism for the specification of heterogeneous systems. In *Forum on Specification, Verification and Design Languages, 2008. FDL 2008*, pages 142–147, September 2008. doi: 10.1109/FDL.2008.4641436. 9.3

[23] Ajinkya Bhave. *Multi-View Consistency in Architectures for Cyber-Physical Systems*. PhD thesis, Carnegie Mellon University, December 2011. 1, B., 4.1, 6.2, 2, 6.2.2, 6.2.2, 1, 10.3

[24] Ajinkya Bhave, Bruce Krogh, David Garlan, and Bradley Schmerl. View Consistency in Architectures for Cyber-Physical Systems. In *IEEE/ACM International Conference on Cyber-Physical Systems (ICCPS)*, 2011. doi: 10.1109/ICCPS.2011.17. A., 1, 6.2, 9.2, 9.3

[25] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003. doi: 10.1016/S0065-2458(03)58003-2. URL `https://doi.org/10.1016/S0065-2458(03)58003-2`. 6.3.3

[26] Dines Bjrner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*. Springer-Verlag, Berlin, Heidelberg, 1978. ISBN 978-3-540-08766-3. 9.3

[27] Egon Borger, Erich Gradel, and Yuri Gurevich. *The Classical Decision Problem*. Springer, October 2001. ISBN 978-3-540-42324-9. 9.2

[28] Sara Bouraine, Thierry Fraichard, and Hassen Salhi. Provably safe navigation for mobile robots with limited field-of-views in dynamic environments. *Autonomous Robots*, 32(3): 267–283, April 2012. ISSN 0929-5593, 1573-7527. doi: 10.1007/s10514-011-9258-8. URL `https://hal.inria.fr/hal-00733913/`. 3.2

[29] David Broman, Edward A. Lee, Stavros Tripakis, and Martin Torngren. Viewpoints, Formalisms, Languages, and Tools for Cyber-Physical Systems. In *Proc. of the 6th International Workshop on Multi-Paradigm Modeling*, October 2012. 1, 2.1, 6.2.1

[30] Carlos E. Budde, Christian Dehnert, Ernst Moritz Hahn, Arnd Hartmanns, Sebastian Junges, and Andrea Turrini. JANI: Quantitative Model and Tool Interaction. In *Proceedings, Part II, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 10206*, pages 151–168, Berlin, Heidelberg, 2017. Springer-Verlag. ISBN 978-3-662-54579-9. doi: 10.1007/978-3-662-54580-5_9. URL `https://doi.org/10.1007/978-3-662-54580-5_9`. 9.3

[31] Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and

Frantisek Plasil. DEECO: An Ensemble-based Component System. In *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '13, pages 81–90, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2122-8. doi: 10.1145/2465449.2465462. URL `http://doi.acm.org/10.1145/2465449.2465462`. 6.2.3, 9.2

[32] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. Strengthening Architectures of Smart CPS by Modeling Them As Runtime Product-lines. In *Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '14, pages 91–96, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2577-6. doi: 10.1145/2602458.2602478. URL `http://doi.acm.org/10.1145/2602458.2602478`. 9.2

[33] Jean-Francois Castet, Matthew L. Rozek, Michel D. Ingham, Nicolas F. Rouquette, Seung H. Chung, J. Steven Jenkins, David A. Wagner, and Daniel L. Dvorak. Ontology and Modeling Patterns for State-Based Behavior Representation. In *AIAA Infotech @ Aerospace*. American Institute of Aeronautics and Astronautics, 2015. URL `http://arc.aiaa.org/doi/abs/10.2514/6.2015-1115`. 9.3

[34] Sagar Chaki and James R. Edmondson. Model-Driven Verifying Compilation of Synchronous Distributed Applications. In *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings*, pages 201–217, 2014. doi: 10.1007/978-3-319-11653-2_13. URL `http://dx.doi.org/10.1007/978-3-319-11653-2_13`. 2.1

[35] Sagar Chaki, Arie Gurfinkel, Soonho Kong, and Ofer Strichman. Compositional Sequentialization of Periodic Programs. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science, pages 536–554. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-35873-9. 1, 3.3, 8.3.1

[36] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *Proc. of the 20th USENIX Conference on Security*, pages 6–22, Berkeley, CA, USA, 2011. 3.4

[37] Sanjian Chen, Oleg Sokolsky, James Weimer, and Insup Lee. Data-driven Adaptive Safety Monitoring using Virtual Subjects in Medical Cyber-Physical Systems: A Glucose Control Case Study. *Journal of Computer Science and Engineering*, pages 75–84, September 2016. doi: 10.5626/JCSE.2016.10.3.75. URL `http://repository.upenn.edu/cis_papers/826`. 10.4.1

[38] Xin Chen, Erika brahm, and Sriram Sankaranarayanan. Flow*: An Analyzer for Non-linear Hybrid Systems. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 258–263. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-39799-8. 9.1.2

[39] Yufei Chen and James W. Evans. Thermal Analysis of Lithium-Ion Batteries. *Journal of The Electrochemical Society*, 143(9):2708–2712, September 1996. ISSN 0013-4651, 1945-7111. doi: 10.1149/1.1837095. URL `http://jes.ecsdl.org/content/143/9/2708`.

3.3

[40] Antonio Cicchetti. Consistency and Uncertainty in the Development of Cyber-Physical Systems. In *Proceedings of the 4th Workshop of the MPM4CPS COST Action*, Gdansk,Poland, 2016. 10.3

[41] Antonio Cicchetti, Federico Ciccozzi, and Thomas Leveque. A hybrid approach for multi-view modeling. *Electronic Communications of the EASST*, 50(0), July 2012. ISSN 1863-2122. doi: 10.14279/tuj.eceasst.50.738. URL `https://journal.ub.tu-berlin.de/eceasst/article/view/738`. 9.2

[42] Alessandro Cimatti, Marco Roveri, Angelo Susi, and Stefano Tonetta. Formalizing requirements with object models and temporal constraints. *Software & Systems Modeling*, 10(2): 147–160, August 2009. ISSN 1619-1366, 1619-1374. 9.2

[43] David W. St Clair. *Controller Tuning and Control Loop Performance*. Straight-Line Control Co., Newark, second edition, January 1990. ISBN 978-0-9669703-0-2. 9.1.2

[44] Edmund M. Clarke, William Klieber, Milo Novek, and Paolo Zuliani. Model Checking and the State Explosion Problem. In Bertrand Meyer and Martin Nordio, editors, *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, Lecture Notes in Computer Science, pages 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-35746-6. doi: 10.1007/978-3-642-35746-6_1. URL `https://doi.org/10.1007/978-3-642-35746-6_1`. 1

[45] Edward M. Clarke, E. Allen Emerson, and Aravinda Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, April 1986. ISSN 0164-0925. doi: 10.1145/5397.5399. URL `http://doi.acm.org/10.1145/5397.5399`. 5.5.2, 9.2

[46] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2 edition, October 2010. ISBN 0-321-55268-7. 2.1, 4.1, 6.2, 6.2.1

[47] Darren Cofer. Formal Methods in the Aerospace Industry: Follow the Money. In Toshiaki Aoki and Kenji Taguchi, editors, *Formal Methods and Software Engineering*, number 7635 in Lecture Notes in Computer Science, pages 2–3. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-34280-6 978-3-642-34281-3. URL `http://link.springer.com/chapter/10.1007/978-3-642-34281-3_2`. 2.1

[48] Darren D. Cofer, Andrew Gacek, Steven P. Miller, Michael W. Whalen, Brian LaValley, and Lui Sha. Compositional Verification of Architectural Models. In Alwyn E. Goodloe and Suzette Person, editors, *Proceedings of the 4th NASA Formal Methods Symposium (NFM 2012)*, volume 7226, pages 126–140, Berlin, Heidelberg, April 2012. Springer-Verlag. 9.3

[49] David Coldewey. Uber in fatal crash detected pedestrian but had emergency braking disabled, May 2018. URL `http://social.techcrunch.com/2018/05/24/uber-in-fatal-crash-detected-pedestrian-but-had-emergency-braking-disabled/`. 3.2

[50] B. Combemale, J. Deantoni, B. Baudry, R.B. France, J.-M. Jezequel, and J. Gray. Globalizing Modeling Languages. *Computer*, 47(6):68–71, June 2014. ISSN 0018-9162. doi: 10.1109/MC.2014.147. 9.3

[51] Jeffrey A. Cook, Jing Sun, Julia H. Buckland, Ilya V. Kolmanovsky, Huei Peng, and Jessy W. Grizzle. Automotive Powertrain Control A Survey. *Asian Journal of Control*, 8(3):237–260, September 2006. ISSN 1934-6093. doi: 10.1111/j.1934-6093.2006.tb00275.x. URL `http://onlinelibrary.wiley.com/doi/10.1111/j.1934-6093.2006.tb00275.x/abstract`. 2.1

[52] Arnaud Da Costa, Franois Laroussinie, and Nicolas Markey. Quantified CTL: Expressiveness and Model Checking. In *CONCUR 2012 Concurrency Theory*, Lecture Notes in Computer Science, pages 177–192. Springer, Berlin, Heidelberg, September 2012. ISBN 978-3-642-32939-5 978-3-642-32940-1. doi: 10.1007/978-3-642-32940-1_14. URL `https://link.springer.com/chapter/10.1007/978-3-642-32940-1_14`. 9.2

[53] James Dabney and Thomas L Harman. *Mastering SIMULINK 2*. Prentice Hall, Upper Saddle River, N.J., 1998. ISBN 0-13-243767-8 978-0-13-243767-7. 2.1, 9.1.2

[54] Abhijit Davare, Douglas Densmore, Liangpeng Guo, Roberto Passerone, Alberto L. Sangiovanni-Vincentelli, Alena Simalatsar, and Qi Zhu. metroII: A Design Environment for Cyber-physical Systems. *ACM Trans. Embed. Comput. Syst.*, 12(1s):49:1–49:31, 2013. ISSN 1539-9087. doi: 10.1145/2435227.2435245. 9.3

[55] Leonardo De Moura and Nikolaj Bjrner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78799-2 978-3-540-78799-0. URL `http://dl.acm.org/citation.cfm?id=1792734.1792766`. 5.7, 8.1.2, 9.2

[56] Dionisio De Niz and Raj Rajkumar. Partitioning bin-packing algorithms for distributed real-time systems. *International Journal of Embedded Systems*, 2(3-4):196–208, January 2006. ISSN 1741-1068. doi: 10.1504/IJES.2006.014855. URL `https://www.inderscienceonline.com/doi/abs/10.1504/IJES.2006.014855`. 3.3

[57] J. Delange and P. Feiler. Architecture Fault Modeling with the AADL Error-Model Annex. In *40th Conference on Software Engineering and Advanced Applications*, pages 361–368, 2014. doi: 10.1109/SEAA.2014.20. 3.4

[58] Lenny Delligatti. *SysML Distilled: A Brief Guide to the Systems Modeling Language*. Addison-Wesley Professional, Upper Saddle River, NJ, 1 edition, November 2013. ISBN 978-0-321-92786-6. 2.1, 9.1.1

[59] Trip Denton and Edward Jones. NAOMI — An Experimental Platform for Multimodeling. In *Proc, of Model Driven Engineering Languages and Systems (MoDELS)*, pages 143–157, 2008. doi: 10.1007/978-3-540-87875-9_10. 9.2

[60] P. Derler, E. A. Lee, and A. S. Vincentelli. Modeling Cyber-Physical Systems. *Proceedings of the IEEE*, 100(1):13–28, January 2012. ISSN 0018-9219, 1558-2256. doi: 10.1109/JPROC.2011.2160929. URL `https://chess.eecs.berkeley.edu/pubs/843.html`. 2.1

[61] Patricia Derler, Edward A. Lee, Stavros Tripakis, and Martin Torngren. Cyber-physical System Design Contracts. In *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*, ICCPS '13, pages 109–118, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1996-6. doi: 10.1145/2502524.2502540. 1, 2.2, 9.3

[62] Remco Matthijs Dijkman. *Consistency in multi-viewpoint architectural design*. PhD thesis, Telematica Instituut, Enschede, The Netherlands, 2006. 9.2, 9.3, 10.4.1

[63] Vladimir Dimitrieski. Model-Driven Technical Space Integration Based on a Mapping Approach. , March 2018. URL `http://nardus.mpn.gov.rs/handle/123456789/9309`. 2.1, 9.3

[64] Cristian Dimitrovici, Udo Hummert, and Laure Petrucci. Semantics, composition and net properties of algebraic high-level nets. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1991*, number 524 in Lecture Notes in Computer Science, pages 93–117. Springer Berlin Heidelberg, June 1990. ISBN 978-3-540-54398-5 978-3-540-47600-9. doi: 10.1007/BFb0019971. URL `http://link.springer.com/chapter/10.1007/BFb0019971`. 9.1.1

[65] Alexander Franz Egyed. *Heterogeneous View Integration and its Automation*. PhD thesis, University of Southern California, 2000. 9.3

[66] Esterel Technologies. SCADE Suite, 2015. URL `http://www.esterel-technologies.com/products/scade-suite/`. esterel-technologies.com/products/scade-suite. 2.1, 9.1.2

[67] Andy Evans, Stuart Kent, and Bran Selic. *UML 2000 - The Unified Modeling Language. Advancing the Standard: Third International Conference York, UK, October 2-6, 2000 Proceedings*. Springer, New York, 2000 edition, October 2000. ISBN 978-3-540-41133-8. 2.1, 9.1.1

[68] H. Fawzi, P. Tabuada, and S. Diggavi. Secure Estimation and Control for Cyber-Physical Systems Under Adversarial Attacks. *IEEE Transactions on Automatic Control*, 59(6): 1454–1467, June 2014. ISSN 0018-9286. doi: 10.1109/TAC.2014.2303233. 3.4, 8.3.2

[69] Peter Feiler and Aaron Greenhouse. OSATE Plugin Guide. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2006. 5.7, 8.1.2

[70] Peter Feiler, Lutz Wrage, Julien Delange, and Joe Siebel. OSATE2, 2015. URL `https://github.com/osate`. github.com/osate. 1, 7.4, 8.1.2

[71] Peter H. Feiler and David P. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley, Upper Saddle River, NJ, 1 edition, 2012. ISBN 978-0-321-88894-5. 2.1, 4.1, 5.7, 9.2

[72] Peter H. Feiler, David P. Gluch, John J. Hudak, and Bruce A. Lewis. Embedded System Architecture Analysis Using SAE AADL. Technical report, June 2004. 2.1, 10.3

[73] Peter H. Feiler, David P. Gluch, and John J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, 2006. URL `http://www.aadl.info/aadl/currentsite/currentusers/notation.html`. 6.1

[74] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. *International Journal of Software Engineering and Knowledge Engineering*, 2, 1992. 6.2.1

[75] J. Fitzgerald, K. Pierce, and C. Gamble. A rigorous approach to the design of resilient cyber-physical systems through co-simulation. In *2012 IEEE/IFIP 42nd International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 1–6, 2012. doi: 10.1109/DSNW.2012.6264663. 9.3

[76] J. Fitzgerald, C. Gamble, P. G. Larsen, K. Pierce, and J. Woodcock. Cyber-Physical Systems Design: Formal Foundations, Methods and Integrated Tool Chains. In *2015 IEEE/ACM 3rd FME Workshop on Formal Methods in Software Engineering*, pages 40–46, May 2015. doi: 10.1109/FormaliSE.2015.14. 1, 2.1

[77] Luke Fletcher, Seth Teller, Edwin Olson, David Moore, Yoshiaki Kuwata, Jonathan How, John Leonard, Isaac Miller, Mark Campbell, Dan Huttenlocher, Aaron Nathan, and Frank-Robert Kline. The MIT Cornell Collision and Why It Happened. In Martin Buehler, Karl Iagnemma, and Sanjiv Singh, editors, *The DARPA Urban Challenge*, number 56 in Springer Tracts in Advanced Robotics, pages 509–548. Springer Berlin Heidelberg, January 2009. ISBN 978-3-642-03990-4 978-3-642-03991-1. URL `http://link.springer.com/chapter/10.1007/978-3-642-03991-1_12`. 3.2

[78] G. Frehse, Zhi Han, and B. Krogh. Assume-guarantee reasoning for hybrid I/O-automata by over-approximation of continuous interaction. In *2004 43rd IEEE Conference on Decision and Control (CDC) (IEEE Cat. No.04CH37601)*, volume 1, pages 479–484 Vol.1, December 2004. doi: 10.1109/CDC.2004.1428676. 9.3

[79] Goran Frehse, Colas Le Guernic, Alexandre Donz, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. SpaceEx: Scalable Verification of Hybrid Systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 379–395. Springer Berlin Heidelberg, January 2011. ISBN 978-3-642-22109-5 978-3-642-22110-1. URL `http://link.springer.com/chapter/10.1007/978-3-642-22110-1_30`. 1, 2.1, 9.1.2

[80] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. John Wiley & Sons, January 2004. ISBN 978-0-471-47163-9. Google-Books-ID: IzqY8Abz1rAC. 9.1.2

[81] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, and Andre Platzer. KeYmaera X: An Axiomatic Tactical Theorem Prover for Hybrid Systems. In *Proc. of CADE-25*, volume 9195, Berlin, Germany, 2015. doi: 10.1007/978-3-319-21401-6_36. 8.2.2

[82] D. M. Gabbay. Fibred Semantics and the Weaving of Logics Part 1: Modal and Intuitionistic Logics. *The Journal of Symbolic Logic*, 61(4):1057–1120, 1996. ISSN 0022-4812. doi: 10.2307/2275807. URL `http://www.jstor.org/stable/2275807`. 9.3

[83] Paul Gao, Russel Hensley, and Andreas Zielke. A road map to the future for the auto industry. *McKinsey Quarterly*, October 2014. 1

[84] David Garlan and Bradley Schmerl. Architecture-driven Modelling and Analysis. In

Proceedings of the 11th Australian Workshop on Safety Related Programmable Systems (SCS'06), volume 69 of *Conferences in Research and Practice in Information Technology*, Melbourne, Australia, 2006. 2.1

[85] David Garlan, Robert Monroe, and David Wile. Acme: an architecture description interchange language. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '97, pages 7–22. IBM Press, 1997. URL `http://dl.acm.org/citation.cfm?id=782010.782017`. 2.1, 6.2.1

[86] David Garlan, Robert Monroe, and David Wile. Acme: Architectural Description of Component-Based Systems. *Foundations of component-based systems*, pages 47–67, January 2000. URL `http://repository.cmu.edu/compsci/693`. 4.1, 9.2

[87] Silvio Ghilardi, Enrica Nicolini, Silvio Ranise, and Daniele Zucchelli. Combination Methods for Satisfiability and Model-Checking of Infinite-State Systems. pages 362–378. Springer Berlin Heidelberg, July 2007. doi: 10.1007/978-3-540-73595-3_25. URL `http://link.springer.com/chapter/10.1007/978-3-540-73595-3_25`. 9.2

[88] Holger Giese and Robert Wagner. Incremental Model Synchronization with Triple Graph Grammars. In *Proc. of MoDELS*, 2006. ISBN 3-540-45772-0 978-3-540-45772-5. doi: 10.1007/11880240_38. 9.2

[89] Antoine Girard and George J. Pappas. Approximate Bisimulation: A Bridge Between Computer Science and Control Theory. *European Journal of Control*, 17(56): 568–578, 2011. ISSN 0947-3580. doi: 10.3166/ejc.17.568-578. URL `http://www.sciencedirect.com/science/article/pii/S0947358011709773`. 9.3

[90] Fausto Giunchiglia, Paolo Pecchiari, and Carolyn Talcott. Reasoning Theories. In Frans Baader and Klaus U. Schulz, editors, *Frontiers of Combining Systems: First International Workshop, Munich, March 1996*, Applied Logic Series, pages 157–174. Springer Netherlands, Dordrecht, 1996. ISBN 978-94-009-0349-4. doi: 10.1007/978-94-009-0349-4_8. URL `https://doi.org/10.1007/978-94-009-0349-4_8`. 9.2

[91] Cludio Gomes, Casper Thule, Julien Deantoni, Peter Gorm Larsen, and Hans Vangheluwe. Co-simulation: The Past, Future, and Open Challenges. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Distributed Systems*, Lecture Notes in Computer Science, pages 504–520. Springer International Publishing, 2018. ISBN 978-3-030-03424-5. 9.3

[92] Susanne Graf, Sophie Quinton, Alain Girault, and Gregor Gssler. Building Correct Cyber-Physical Systems: Why we need a Multiview Contract Theory. volume 11119, pages 19–31. Springer, September 2018. doi: 10.1007/978-3-030-00244-2_2. URL `https://hal.inria.fr/hal-01891146/document`. 9.2, 9.3

[93] Jason Green. Tesla says crashed vehicle had been on autopilot prior to accident. *Reuters*, March 2018. URL `https://www.reuters.com/article/us-tesla-crash/tesla-says-crashed-vehicle-had-been-on-autopilot-prior-to-accident-idUSKBN1H7023`. 3.2

[94] High Confidence Software and Systems Coordinating Group. High-Confidence Medical Devices: Cyber-Physical Systems for 21st Century Health Care. Technical report, Networking and Information Technology Research and Development Program, 2007. 1

[95] Esther Guerra, Juan de Lara, Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schnbck, and Wieland Schwinger. Automated verification of model transformations based on visual contracts. *Automated Software Engineering*, 20(1): 5–46, March 2013. ISSN 0928-8910, 1573-7535. doi: 10.1007/s10515-012-0102-y. URL `http://link.springer.com/article/10.1007/s10515-012-0102-y`. 9.2

[96] Arie Gurfinkel, Sharon Shoham, and Yuri Meshman. SMT-based Verification of Parameterized Systems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 338–348, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4218-6. doi: 10.1145/2950290.2950330. URL `http://doi.acm.org/10.1145/2950290.2950330`. 3

[97] Grgoire Hamon and John Rushby. An Operational Semantics for Stateflow. In Michel Wermelinger and Tiziana Margaria-Steffen, editors, *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, pages 229–243. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-24721-0. 9.1.2

[98] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, September 1994. ISSN 0934-5043, 1433-299X. doi: 10.1007/BF01211866. URL `http://link.springer.com/article/10.1007/BF01211866`. 4.1, 8.3.2, 9.2

[99] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. The MIT Press, Cambridge, Mass., September 2000. ISBN 978-0-262-52766-8. 9.1.1

[100] T.A. Henzinger. The theory of hybrid automata. In *, Eleventh Annual IEEE Symposium on Logic in Computer Science, 1996. LICS '96. Proceedings*, pages 278–292, July 1996. doi: 10.1109/LICS.1996.561342. 2.1

[101] S. J. I. Herzig, K. Berx, K. Gadeyne, M. Witters, and C. J. J. Paredis. Computational design synthesis for conceptual design of robotic assembly cells. In *2016 IEEE International Symposium on Systems Engineering (ISSE)*, pages 1–8, October 2016. doi: 10.1109/SysEng.2016.7753183. 10.3

[102] Rich Hilliard. Views and Viewpoints in Software Systems Architecture. In *Proc. of the First Working IFIP Conference on Software Architectu*, pages 22–24, San Antonio, TX, 1999. 6.2.1, 9.2

[103] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, August 1978. ISSN 0001-0782. doi: 10.1145/359576.359585. URL `http://doi.acm.org/10.1145/359576.359585`. 9.1.1, 9.3

[104] Snke Holthusen, Sophie Quinton, Ina Schaefer, Johannes Schlatow, and Martin Wegner. Using Multi-Viewpoint Contracts for Negotiation of Embedded Software Updates. *Electronic Proceedings in Theoretical Computer Science*, 208:31–45, May 2016. ISSN 2075-2180. doi: 10.4204/EPTCS.208.3. URL `http://arxiv.org/abs/1606.00504`. arXiv: 1606.00504. 9.3

[105] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997. ISSN 0098-5589. doi: 10.1109/32.588521. URL `http://dx.doi.org/10.1109/32.588521`. 2.1, 8.3.1, 9.1.1

[106] Ronald A. Howard. *Dynamic Programming and Markov Processes*. Technology Press of the Massachusetts Institute of Technology, 1960. 3.1

[107] Fei Hu, Yu Lu, Athanasios V. Vasilakos, Qi Hao, Rui Ma, Yogendra Patil, Ting Zhang, Jiang Lu, Xin Li, and Neal N. Xiong. Robust CyberPhysical Systems: Concept, models, and implementation. *Future Generation Computer Systems*, 56:449–475, March 2016. ISSN 0167-739X. doi: 10.1016/j.future.2015.06.006. URL `http://www.sciencedirect.com/science/article/pii/S0167739X15002071`. 2.1

[108] J. Hugues and G. Brau. Analysis as a First-Class Citizen: An Application to Architecture Description Languages. In *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 214–221, June 2014. doi: 10.1109/ISORC.2014.60. 1

[109] T. Huria, M. Ceraolo, J. Gazzarri, and R. Jackey. High fidelity electrical model with thermal dependence for characterization and simulation of high power lithium battery cells. In *Electric Vehicle Conference (IEVC), 2012 IEEE International*, pages 1–8, March 2012. doi: 10.1109/IEVC.2012.6183271. 2.1

[110] M. Iacono and M. Gribaudo. Element Based Semantics in Multi Formalism Performance Models. In *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 413–416, August 2010. doi: 10.1109/MASCOTS.2010.54. 9.3

[111] Michel D. Ingham, Robert D. Rasmussen, Matthew B. Bennett, and Alex C. Moncada. Engineering Complex Embedded Systems with State Analysis and the Mission Data System. *Journal of Aerospace Computing, Information, and Communication*, 2(12):507–536, 2005. doi: 10.2514/1.15265. URL `http://dx.doi.org/10.2514/1.15265`. 9.3

[112] ISO/IEC/IEEE. 42010:2011 - Systems and software engineering – Architecture description. Technical report, International Standards Organization (ISO), 2011. URL `http://www.iso.org/iso/catalogue_detail.htm?csnumber=50508`. 6.2.1, 9.2

[113] Daniel Jackson. *Software abstractions: logic, language, and analysis*. MIT Press, Cambridge, Mass., 2012. ISBN 978-0-262-01715-2 0-262-01715-6. 9.1.1, 9.2

[114] Ethan K. Jackson, Tihamr Levendovszky, and Daniel Balasubramanian. Reasoning About Metamodeling with Formal Specifications and Automatic Proofs. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*, MODELS'11, pages 653–667, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-24484-1. URL `http://dl.acm.org/citation.cfm?id=2050655.2050722`. 9.3

[115] Bart Jacobs and Erik Poll. A Logic for the Java Modeling Language JML. In *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*, FASE '01, pages 284–299, London, UK, UK, 2001. Springer-Verlag. ISBN 978-3-540-41863-4. URL `http://dl.acm.org/citation.cfm?id=645369.651284`. 9.1.1

[116] N. Jarus, S. S. Sarvestani, and A. R. Hurson. Models, metamodels, and model transformation for cyber-physical systems. In *2016 Seventh International Green and Sustainable Computing Conference (IGSC)*, pages 1–8, November 2016. doi: 10.1109/IGCC.2016.7892611. 9.3

[117] Xiaoqing Jin, Jyotirmoy V. Deshmukh, James Kapinski, Koichi Ueda, and Ken Butts. Powertrain Control Verification Benchmark. In *Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control*, HSCC '14, pages 253–262, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2732-9. doi: 10.1145/2562059.2562140. URL `http://doi.acm.org/10.1145/2562059.2562140`. 2.1

[118] Stephen B. Johnson and John C. Day. Theoretical Foundations for the Discipline of Systems Engineering. In *54th AIAA Aerospace Sciences Meeting*, San Diego, CA, 2016. American Institute of Aeronautics and Astronautics. URL `http://arc.aiaa.org/doi/abs/10.2514/6.2016-0212`. 2.2

[119] J. Kapinski, J. V. Deshmukh, X. Jin, H. Ito, and K. Butts. Simulation-Based Approaches for Verification of Embedded Control Systems: An Overview of Traditional and Advanced Modeling, Testing, and Verification Techniques. *IEEE Control Systems*, 36(6):45–64, December 2016. ISSN 1066-033X. doi: 10.1109/MCS.2016.2602089. 9.1.2

[120] Gabor Karsai and Janos Sztipanovits. Model-Integrated Development of Cyber-Physical Systems. In Uwe Brinkschulte, Tony Givargis, and Stefano Russo, editors, *Software Technologies for Embedded and Ubiquitous Systems*, volume 5287 of *Lecture Notes in Computer Science*, pages 46–54. Springer Berlin / Heidelberg, 2008. ISBN 978-3-540-87784-4. URL `http://www.springerlink.com/content/6xp8567xt2565721/abstract/`. 9.3

[121] Joost-Pieter Katoen, Maneesh Khattri, and Ivan S. Zapreev. A Markov Reward Model Checker. In *Proc. of the 2nd International Conference on the Quantitative Evaluation of Systems*, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2427-3. doi: 10.1109/QEST.2005.2. URL `http://dx.doi.org/10.1109/QEST.2005.2`. 8.3.2

[122] Oliver Kautz, Alexander Roth, and Bernhard Rumpe. Achievements, Failures, and the Future of Model-Based Software Engineering. In Volker Gruhn and Rdiger Striemer, editors, *The Essence of Software Engineering*, pages 221–236. Springer International Publishing, Cham, 2018. ISBN 978-3-319-73897-0. doi: 10.1007/978-3-319-73897-0_13. URL `https://doi.org/10.1007/978-3-319-73897-0_13`. 1

[123] K. Keutzer, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli. System-level Design: Orthogonalization of Concerns and Platform-based Design. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 19(12):1523–1543, November 2006. ISSN 0278-0070. doi: 10.1109/43.898830. URL `http://dx.doi.org/10.1109/43.898830`. 9.3

[124] Gi-Heon Kim and Ahmad Pesaran. Analysis of Heat Dissipation in Li-Ion Cells & Modules for Modeling of Thermal Runaway. In *Proc. of 3rd International Symposium on Large Lithium Ion Battery Technology and Application*, Long Beach, CA, 2007. 3.3, 8.3.1, 8.3.1

[125] Hahnsang Kim and K.G. Shin. On Dynamic Reconfiguration of a Large-Scale Battery System. In *15th IEEE Real-Time and Embedded Technology and Applications Symposium,*

2009. RTAS 2009, pages 87–96, April 2009. doi: 10.1109/RTAS.2009.13. 3.3, 8.3.1

[126] Hahnsang Kim and K.G. Shin. Scheduling of Battery Charge, Discharge, and Rest. In *30th IEEE Real-Time Systems Symposium, 2009, RTSS 2009*, pages 13–22, December 2009. doi: 10.1109/RTSS.2009.38. 3.3, 8.3.1

[127] Kyoung-Dae Kim and P.R. Kumar. Cyber-Physical Systems: A Perspective at the Centennial. *Proceedings of the IEEE*, 100:1287–1308, 2012. ISSN 0018-9219. doi: 10.1109/JPROC.2012.2189792. 1

[128] Joachim Klein and Christel Baier. On-the-Fly Stuttering in the Construction of Deterministic -Automata. In Jan Holub and Jan rek, editors, *Implementation and Application of Automata*, Lecture Notes in Computer Science, pages 51–61. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-76336-9. 8.2.1

[129] Zuzana Komarkova and Jan Kretinsky. Rabinizer 3: Safraless Translation of LTL to Small Deterministic Automata. In *Automated Technology for Verification and Analysis*, Lecture Notes in Computer Science, pages 235–241. Springer, Cham, November 2014. ISBN 978-3-319-11935-9 978-3-319-11936-6. doi: 10.1007/978-3-319-11936-6_17. URL `https://link.springer.com/chapter/10.1007/978-3-319-11936-6_17`. 8.1.2, 8.2.1

[130] Savas Konur, Michael Fisher, and Sven Schewe. Combined model checking for temporal, probabilistic, and real-time logics. *Theoretical Computer Science*, 503:61–88, September 2013. ISSN 0304-3975. doi: 10.1016/j.tcs.2013.07.012. URL `http://www.sciencedirect.com/science/article/pii/S030439751300515X`. 9.3

[131] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental Security Analysis of a Modern Automobile. In *2010 IEEE Symposium on Security and Privacy (SP)*, pages 447–462, May 2010. doi: 10.1109/SP.2010.34. 3.4

[132] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View*. Springer, 2008. 5.3

[133] Philippe Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12:42–50, 1995. ISSN 0740-7459. doi: http://doi.ieeecomputersociety.org/10.1109/52.469759. 6.2

[134] Marta Kwiatkowska, Gethin Norman, and David Parker. Stochastic Model Checking. In Marco Bernardo and Jane Hillston, editors, *Formal Methods for Performance Evaluation*, number 4486 in Lecture Notes in Computer Science, pages 220–270. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-72482-7 978-3-540-72522-0. URL `http://link.springer.com/chapter/10.1007/978-3-540-72522-0_6`. 3.1, 5.1, 5.4.1, 5.4.3, 5.5.3, 6.3, 8.1.2, 8.3.2, 9.2

[135] Alexander Knigs. *Model Integration and Transformation A Triple Graph Grammar-based QVT Implementation*. phd, TU Darmstadt, November 2008. URL `http://tuprints.ulb.tu-darmstadt.de/1194/`. 9.2

[136] Kai Lampka, Simon Perathoner, and Lothar Thiele. Component-based system de-

sign: analytic real-time interfaces for state-based component implementations. *International Journal on Software Tools for Technology Transfer*, 15(3):155–170, June 2013. ISSN 1433-2779, 1433-2787. doi: 10.1007/s10009-012-0257-7. URL `http://link.springer.com/article/10.1007/s10009-012-0257-7`. 1

[137] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, Boston, 1 edition edition, July 2002. ISBN 978-0-321-14306-8. 9.1.1

[138] Kevin Lano. *The B Language and Method: A Guide to Practical Formal Development*. Formal Approaches to Computing and Information Technology (FACIT). Springer-Verlag, London, 1996. ISBN 978-3-540-76033-7. URL `//www.springer.com/us/book/9783540760337`. 9.1.1

[139] Juan de Lara and Hans Vangheluwe. AToM3: A Tool for Multi-formalism and Meta-modelling. In *Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, FASE '02, pages 174–188, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43353-8. URL `http://dl.acm.org/citation.cfm?id=645370.651300`. 9.2

[140] Kim G. Larsen, Paul Pettersson, and Wang Yi. Model-Checking for Real-Time Systems. In *Proc. of Fundamentals of Computation Theory*, Lecture Notes in Computer Science, pages 62–88, August 1995. 2.1, 9.1.2

[141] Peter Gorm Larsen, Kenneth Lausdahl, Nick Battle, John Fitzgerald, Sune Wolff, Shin Sahara, Marcel Verhoef, Peter W. V. Tran-Jrgensen, and Tomohiro Oda. VDM-10 Language Manual. Technical Report TR-001, The Overture Initiative, www.overturetool.org, April 2013. 9.1.1

[142] E.A. Lee and D.G. Messerschmitt. Pipeline interleaved programmable DSP's: Synchronous data flow programming. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 35(9):1334–1345, September 1987. ISSN 0096-3518. doi: 10.1109/TASSP.1987.1165275. 2.1

[143] Edward A. Lee. Cyber Physical Systems: Design Challenges. In *Proceedings of the 11th Symposium on Object Oriented Real-Time Distributed Computing*, pages 363–369, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3132-8. doi: 10.1109/ISORC.2008.25. 1, 2.2

[144] Edward A. Lee. CPS Foundations. In *Proceedings of the 47th Design Automation Conference*, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0002-5. doi: 10.1145/1837274.1837462. 1, 9.1.1

[145] Edward A. Lee. The Past, Present and Future of Cyber-Physical Systems: A Focus on Models. *Sensors (Basel, Switzerland)*, 15(3):4837–4869, 2015. ISSN 1424-8220. doi: 10.3390/s150304837. 2.1

[146] Edward A. Lee, Stephen Neuendorffer, and Gang Zhou. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. ISBN 1-304-42106-6. 1, 6.2.3, 9.3

[147] Lichen Lichen. Model Integration and Model Transformation Approach for Multi-Paradigm

Cyber Physical System Development. In Henry Selvaraj, Dawid Zydek, and Grzegorz Chmaj, editors, *Progress in Systems Engineering*, Advances in Intelligent Systems and Computing, pages 629–635. Springer International Publishing, 2015. ISBN 978-3-319-08422-0. 9.3

[148] M. Liserre, T. Sauter, and J. Y. Hung. Future Energy Systems: Integrating Renewable Energy Sources into the Smart Power Grid Through Industrial Electronics. *IEEE Industrial Electronics Magazine*, 4(1):18–37, March 2010. ISSN 1932-4529. doi: 10.1109/MIE.2010.935861. 1

[149] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, 20(1):46–61, January 1973. ISSN 0004-5411. doi: 10.1145/321738.321743. URL `http://doi.acm.org/10.1145/321738.321743`. 3.3

[150] Sarah M. Loos, Andre Platzer, and Ligia Nistor. Adaptive Cruise Control: Hybrid, Distributed, and Now Formally Verified. In *FM 2011: Formal Methods*, number 6664 in Lecture Notes in Computer Science, pages 42–56. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-21436-3 978-3-642-21437-0. URL `http://link.springer.com/chapter/10.1007/978-3-642-21437-0_6`. 2.1

[151] Sarah M. Loos, David Renshaw, and Andre Platzer. Formal Verification of Distributed Aircraft Controllers. In *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control*, HSCC '13, pages 125–130, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1567-8. doi: 10.1145/2461328.2461350. URL `http://doi.acm.org/10.1145/2461328.2461350`. 2.1

[152] Sergey E. Lyshevski. *Engineering and Scientific Computations Using MATLAB*. Wiley-Interscience, Hoboken, 1st edition, June 2003. ISBN 978-0-471-46200-2. 2.1

[153] Kristijan Macek, Dizan Alejandro Vasquez Govea, Thierry Fraichard, and Roland Siegwart. Towards Safe Vehicle Navigation in Dynamic Urban Scenarios. *Automatika*, November 2009. URL `https://hal.inria.fr/inria-00447452/document`. This paper describes the deliberative part of a navigation architecture designed for safe vehicle navigation in dynamic urban environments. It comprises two key modules working together in a hierarchical fashion: (a) the Route Planner whose purpose is to compute a valid itinerary towards the a given goal. An itinerary comprises a geometric path augmented with additional information based on the structure of the environment considered and traffic regulations, and (b) the Partial Motion Planner whose purpose is to ensure the proper following of the itinerary while dealing with the moving objects present in the environment (eg other vehicles, pedestrians). In the architecture proposed, a special attention is paid to the motion safety issue, ie the ability to avoid collisions. Different safety levels are explored and their operational conditions are explicitly spelled out (something which is usually not done). 3.2

[154] Jeff Magee and Jeff Kramer. *Concurrency: State Models & Java Programs*. Wiley, April 1999. ISBN 0-471-98710-7. 2.1, 9.1.1

[155] Oded Maler and Dejan Nickovic. Monitoring Temporal Properties of Continuous Signals. In *Proc. of Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant*

Systems (FORMATS), pages 152–166, 2004. doi: 10.1007/978-3-540-30206-3_12. URL `http://link.springer.com/chapter/10.1007/978-3-540-30206-3_12`. 10.4.1

[156] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992. 9.2

[157] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 444–454, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2237-9. doi: 10.1145/2491411.2491414. URL `http://doi.acm.org/10.1145/2491411.2491414`. 6.2

[158] Raluca Marinescu. *Model-driven Analysis and Verification of Automotive Embedded Systems*. PhD thesis, Maladaren University, 2016. B., 9.1.2, 9.3

[159] Felix Gomez Marmol and Gregorio Martinez Perez. Towards pre-standardization of trust and reputation models for distributed and heterogeneous systems. *Computer Standards & Interfaces*, 32(4):185–196, June 2010. ISSN 0920-5489. doi: 10.1016/j.csi.2010.01.003. 3.4, 3.4

[160] Paolo Masci, Anaheed Ayoub, Paul Curzon, Insup Lee, Oleg Sokolsky, and Harold Thimbleby. Model-Based Development of the Generic PCA Infusion Pump User Interface Prototype in PVS. In *Computer Safety, Reliability, and Security*, number 8153 in Lecture Notes in Computer Science, pages 228–240. Springer Berlin Heidelberg, January 2013. ISBN 978-3-642-40792-5 978-3-642-40793-2. URL `http://link.springer.com/chapter/10.1007/978-3-642-40793-2_21`. 2.1

[161] Ethan McGee, Mike Kabbani, and Nicholas Guzzardo. Collision Detection AADL, 2013. URL `https://github.com/mikekab/collision_detection_aadl`. 8.2.4

[162] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. In *Proceedings of the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC '97/FSE-5, pages 60–76, New York, NY, USA, 1997. Springer-Verlag New York, Inc. ISBN 3-540-63531-9. doi: http://dx.doi.org/10.1145/267895.267903. URL `http://dx.doi.org/10.1145/267895.267903`. 6.2.5

[163] Johannes Meier and Andreas Winter. Towards Metamodel Integration Using Reference Metamodels. In *Proceedings of the 4th Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling (VAO)*, Karlsruhe, Germany, 2016. 10.4.1

[164] Johannes Meier and Andreas Winter. Model Consistency ensured by Metamodel Integration. In *Proceedings of the 6th International Workshop on The Globalization of Modeling Languages (GEMOC)*, Copenhagen, Denmark, 2018. 9.3

[165] Bertrand Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, October 1992. ISSN 0018-9162. doi: 10.1109/2.161279. URL `http://dx.doi.org/10.1109/2.161279`. 9.2

[166] B. Meyers, H. Vangheluwe, J. Denil, and R. Salay. A Framework for Temporal Verification Support in Domain-Specific Modelling. *IEEE Transactions on Software Engineering*, pages 1–1, 2018. ISSN 0098-5589. doi: 10.1109/TSE.2018.2859946. 9.3

[167] Bart Meyers, Romuald Deshayes, Levi Lucio, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. ProMoBox: A Framework for Generating Domain-Specific Property Languages. In *Software Language Engineering*, pages 1–20. Springer, Cham, September 2014. doi: 10.1007/978-3-319-11245-9_1. URL `http://link.springer.com/chapter/10.1007/978-3-319-11245-9_1`. 9.3

[168] Chenglin Miao, Liusheng Huang, Weijie Guo, and Hongli Xu. A Trustworthiness Evaluation Method for Wireless Sensor Nodes Based on D-S Evidence Theory. In Kui Ren, Xue Liu, Weifa Liang, Ming Xu, Xiaohua Jia, and Kai Xing, editors, *Wireless Algorithms, Systems, and Applications*, number 7992 in Lecture Notes in Computer Science, pages 163–174. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-39700-4 978-3-642-39701-1. URL `http://link.springer.com/chapter/10.1007/978-3-642-39701-1_14`. 3.4

[169] David Millward. Smart traffic lights to stop speeders. *The Telegraph*, May 2011. URL `http://www.telegraph.co.uk/motoring/news/8521769/Smart-traffic-lights-to-stop-speeders.html`. 1

[170] James A. Misener. Cooperative Intersection Collision Avoidance System (CICAS): Signalized Left Turn Assist and Traffic Signal Adaptation. *PATH Research Report*, March 2010. ISSN 1055-1425. URL `http://trid.trb.org/view.aspx?id=919887`. 3.2

[171] S. Mitsch, S.M. Loos, and A. Platzer. Towards Formal Verification of Freeway Traffic Control. In *2012 IEEE/ACM Third International Conference on Cyber-Physical Systems (ICCPS)*, pages 171–180, April 2012. doi: 10.1109/ICCPS.2012.25. 2.1

[172] Stefan Mitsch, Khalil Ghorbal, and Andr Platzer. On Provably Safe Obstacle Avoidance for Autonomous Robotic Ground Vehicles. In *Proc. of Robotics: Science and Systems*, 2013. 3.2, 6.2.3, 8.2.2

[173] Stefan Mitsch, Grant Olney Passmore, and Andre Platzer. Collaborative Verification-Driven Engineering of Hybrid Systems. *Mathematics in Computer Science*, 8(1):71–97, 2014. doi: 10.1007/s11786-014-0176-y. 2.1, 3.2, 8.2.2, 8.2.2, 9.3

[174] Stefan Mitsch, Jan-David Quesel, and Andre Platzer. Refactoring, Refinement, and Reasoning. In Cliff Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods*, Lecture Notes in Computer Science, pages 481–496. Springer International Publishing, January 2014. ISBN 978-3-319-06409-3 978-3-319-06410-9. URL `http://link.springer.com/chapter/10.1007/978-3-319-06410-9_33`. 3.2, 8.2.2

[175] Yilin Mo, T.H.-H. Kim, K. Brancik, D. Dickinson, Heejo Lee, A. Perrig, and B. Sinopoli. Cyber-Physical Security of a Smart Grid Infrastructure. *Proceedings of the IEEE*, 100(1): 195–209, January 2012. ISSN 0018-9219. doi: 10.1109/JPROC.2011.2161428. 10.4.1

[176] J. Muskens, R. J. Bril, and M. R. V. Chaudron. Generalizing Consistency Checking between Software Views. In *5th Working IEEE/IFIP Conference on Software Architecture, 2005.*

WICSA 2005, pages 169 –180, 2005. doi: 10.1109/WICSA.2005.37. 9.2

[177] Laurence W. Nagel. *SPICE2: A Computer Program to Simulate Semiconductor Circuits*. PhD thesis, EECS Department, University of California, Berkeley, 1975. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/1975/9602.html`. 2.1

[178] Min-Young Nam, Dionisio de Niz, Lutz Wrage, and Lui Sha. Resource allocation contracts for open analytic runtime models. In *Proc. of EMSOFT*, 2011. ISBN 978-1-4503-0714-7. doi: 10.1145/2038642.2038647. 1, 2.1, 9.2

[179] Faranak Nejati, Abdul Azim Abd Ghani, Ng Keng Yap, and Azmi Jaafar. Handling state space explosion in verification of component-based systems: A review. *arXiv:1709.10379 [cs, math]*, July 2017. URL `http://arxiv.org/abs/1709.10379`. arXiv: 1709.10379. 1

[180] Greg Nelson and Derek C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, October 1979. ISSN 0164-0925. doi: 10.1145/357073.357079. URL `http://doi.acm.org/10.1145/357073.357079`. 5.3, 9.2

[181] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006. ISSN 0004-5411. doi: 10.1145/1217856.1217859. 9.2

[182] Oksana Nikiforova, Nisrine El Marzouki, Konstantins Gusarovs, Hans Vangheluwe, Tomas Bures, Rima Al-Ali, Mauro Iacono, Priscill Orue Esquivel, and Florin Leon. The Two-Hemisphere Modelling Approach to the Composition of Cyber-Physical Systems. pages 286–293, October 2018. ISBN 978-989-758-262-2. URL `http://www.scitepress.org/PublicationsDetail.aspx?ID=Wn2BiHZYdOs=&t=1`. 9.3

[183] G. Nitsche, K. Gruttner, and W. Nebel. Power contracts: A formal way towards power-closure. In *23rd International Workshop on Power and Timing Modeling, Optimization and Simulation*, pages 59–66, 2013. doi: 10.1109/PATMOS.2013.6662156. 9.3

[184] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification. *IEEE Trans. Softw. Eng.*, 20(10):760–773, October 1994. ISSN 0098-5589. doi: 10.1109/32.328995. URL `http://dx.doi.org/10.1109/32.328995`. 9.3

[185] Francesco Oliviero, Lorenzo Peluso, and Simon Pietro Romano. REFACING: An autonomic approach to network security based on multidimensional trustworthiness. *Computer Networks*, 52:2745–2763, 2008. ISSN 1389-1286. doi: 10.1016/j.comnet.2008.04.022. 8.2.4

[186] Richard F. Paige, Phillip J. Brooke, and Jonathan S. Ostroff. Metamodel-based model conformance and multi-view consistency checking. *ACM TOSEM*, 16, 2007. 9.2

[187] M. Pajic, J. Weimer, N. Bezzo, P. Tabuada, O. Sokolsky, Insup Lee, and G.J. Pappas. Robustness of attack-resilient state estimators. In *International Conference on Cyber-Physical Systems*, pages 163–174, 2014. doi: 10.1109/ICCPS.2014.6843720. 10.4.1

[188] M. Persson, M. Torngren, A. Qamar, J. Westman, M. Biehl, S. Tripakis, H. Vangheluwe, and J. Denil. A characterization of integrated multi-view modeling in the context of embedded and cyber-physical systems. In *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 1–10, September 2013. doi: 10.1109/EMSOFT.2013.6658588. 9.3

[189] Andre Platzer. Differential Dynamic Logic for Hybrid Systems. *Journal of Automated Reasoning*, 41(2):143–189, August 2008. ISSN 0168-7433, 1573-0670. doi: 10.1007/s10817-008-9103-8. 2.1, 3.2, 6.1, 6.1, 9.1.2

[190] Andre Platzer. *Logical foundations of cyber-physical systems*. Springer Berlin Heidelberg, New York, NY, 2018. ISBN 978-3-319-63587-3. 9.1.2

[191] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, 1977*, pages 46–57, October 1977. doi: 10.1109/SFCS.1977.32. 2.1, 4.1, 5.4.1, 5.4.2

[192] Joseph Porter, Gbor Karsai, Pter Vlgyesi, Harmon Nine, Peter Humke, Graham Hemingway, Ryan Thibodeaux, and Jnos Sztipanovits. Towards Model-Based Integration of Tools and Techniques for Embedded Control System Design, Verification, and Implementation. In Michel R. V. Chaudron, editor, *Models in Software Engineering*, number 5421 in Lecture Notes in Computer Science, pages 20–34. Springer Berlin Heidelberg, January 2009. ISBN 978-3-642-01647-9 978-3-642-01648-6. URL `http://link.springer.com/chapter/10.1007/978-3-642-01648-6_3`. 9.3

[193] Ahsan Qamar. *Model and Dependency Management in Mechatronic Design*. PhD thesis, KTH Sweden, Stockholm, Sweden, 2013. 9.2

[194] Morgan Quigley, Brian Gerkey, and William D. Smart. *Programming Robots with ROS: A Practical Introduction to the Robot Operating System*. O'Reilly Media, 1 edition, November 2015. 3.1

[195] Radhakisan Baheti and Helen Gill. Cyber-Physical Systems. Technical report, National Science Foundation, 2011. 2.1

[196] A. Radjenovic and R.F. Paige. The Role of Dependency Links in Ensuring Architectural View Consistency. In *Seventh Working IEEE/IFIP Conference on Software Architecture, 2008. WICSA 2008*, pages 199 –208, February 2008. doi: 10.1109/WICSA.2008.30. 9.2

[197] Akshay Rajhans. *Multi-Model Heterogeneous Verification of Cyber-Physical Systems*. PhD thesis, Carnegie Mellon University, 2013. 9.3

[198] Akshay Rajhans and Bruce H. Krogh. Heterogeneous verification of cyber-physical systems using behavior relations. In *Proc. of the 15th ACM conference on Hybrid Systems:Computation and Control (HSCC)*, pages 35–44, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1220-2. A., 9.3

[199] Akshay Rajhans and Bruce H. Krogh. Compositional Heterogeneous Abstraction. In *Proc. of HSCC*, pages 253–262. ACM, 2013. ISBN 978-1-4503-1567-8. doi: 10.1145/2461328.2461368. 1

[200] Akshay Rajhans, Ajinkya Bhave, Sarah Loos, Bruce Krogh, Andre Platzer, and David

Garlan. Using Parameters in Architectural Views to Support Heterogeneous Design and Verification. In *Proc. of the 50th IEEE Conference on Decision and Control and European Control Conference (CDC)*, 2011. A., 3, 9.2

[201] R. Rajkumar, Insup Lee, Lui Sha, and J. Stankovic. Cyber-physical systems: The next computing revolution. In *2010 47th ACM/IEEE Design Automation Conference (DAC)*, pages 731–736, 2010. 2.1, 2.2

[202] Andreas Rentschler. *Model Transformation Languages with Modular Information Hiding*. PhD thesis, Karlsruhe Institute of Technology, Karlsruhe, Germany, 2015. 9.2

[203] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton. Technical report, RWTH Aachen University, 2014. 2.1, 10.3

[204] J.R. Romero and A. Vallecillo. Well-formed Rules for Viewpoint Correspondences Specification. In *Enterprise Distributed Object Computing Conference Workshops, 2008 12th*, pages 441 –443, September 2008. doi: 10.1109/EDOCW.2008.63. 10.4.1

[205] Jeff Rothenberg, Lawrence E. Widman, Kenneth A. Loparo, and Norman R. Nielsen. The Nature of Modeling. In *in Artificial Intelligence, Simulation and Modeling*, pages 75–92. John Wiley & Sons, 1989. 1, 2.1

[206] Kristin Yvonne Rozier. Specification: The Biggest Bottleneck in Formal Methods and Autonomy. In Sandrine Blazy and Marsha Chechik, editors, *Verified Software. Theories, Tools, and Experiments*, Lecture Notes in Computer Science, pages 8–26. Springer International Publishing, 2016. ISBN 978-3-319-48869-1. 10.4.2

[207] Ivan Ruchkin. Integration Beyond Components and Models: Research Challenges and Directions. In *Proc. of the Third Workshop on Architecture Centric Virtual Integration (ACVI)*, pages 8–11, Venice, Italy, 2016. doi: 10.1109/ACVI.2016.8. 9.3

[208] Ivan Ruchkin, Dionisio de Niz, Sagar Chaki, and David Garlan. Contract-based Integration of Cyber-physical Analyses. In *Proc. of the Intl. Conf. on Embedded Software (EMSOFT)*, pages 23:1–23:10, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3052-7. doi: 10.1145/2656045.2656052. 5.3

[209] Ivan Ruchkin, Bradley Schmerl, and David Garlan. Architectural Abstractions for Hybrid Programs. In *Proc. of CBSE*, 2015. ISBN 978-1-4503-3471-6. doi: 10.1145/2737166.2737167. 4.1, 9.1.2

[210] Ivan Ruchkin, Bradley R. Schmerl, and David Garlan. Analytic Dependency Loops in Architectural Models of Cyber-Physical Systems. In *ACES-MB at MODELS 2015*, Ottawa, Canada, 2015. 10.2, 10.4.2

[211] Ivan Ruchkin, Dionisio de Niz, Sagar Chaki, and David Garlan. Supplementary Materials for a Case Study of Analysis Contracts with the ACTIVE tool, 2018. 7.4, 9, 2, 3

[212] Ivan Ruchkin, Bradley Schmerl, and David Garlan. Supplementary Materials for a Case Study of Architectural Abstractions for Hybrid Programs, 2018. 8.2.2, 4

[213] Ivan Ruchkin, Joshua Sunshine, Grant Iraci, Bradley Schmerl, and David Garlan. Supplementary Materials for a Case Study of a Power-aware Mobile Robot with the Integration

Property Language, 2018. 5.7, 3, 8.1.2, 1

[214] Alberto Sangiovanni-Vincentelli, Werner Damm, and Roberto Passerone. Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems. *European Journal of Control*, 18(3):217–238, 2012. ISSN 0947-3580. doi: 10.3166/ejc.18.217-238. 9.3

[215] Bradley Schmerl and David Garlan. AcmeStudio: Supporting Style-Centered Architecture Development. In *Proc. of ICSE*, 2004. ISBN 0-7695-2163-0. 1, 8.2.2

[216] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, April 1996. ISBN 0-13-182957-2. 6.2.1, 9.2

[217] Yasser Shoukry, Pierluigi Nuzzo, Alberto L. Sangiovanni-Vincentelli, Sanjit A. Seshia, George J. Pappas, and Paulo Tabuada. SMC: Satisfiability Modulo Convex Optimization. In *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control*, HSCC '17, pages 19–28, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4590-3. doi: 10.1145/3049797.3049819. URL `http://doi.acm.org/10.1145/3049797.3049819`. 10.4.1

[218] Gabor Simko, David Lindecker, Tihamer Levendovszky, Sandeep Neema, and Janos Sztipanovits. Specification of Cyber-Physical Components with Formal Semantics Integration and Composition. In *Model-Driven Engineering Languages and Systems*, pages 471–487. Springer Berlin Heidelberg, January 2013. ISBN 978-3-642-41532-6 978-3-642-41533-3. B., 9.3

[219] Graeme Smith. *The Object-Z Specification Language*. Springer, 2000. URL `http://www.springer.com/computer/ai/book/978-0-7923-8684-1`. 9.1.1, 9.3

[220] SPARC. Robotics 2020 Multi-Annual Roadmap For Robotics in Europe, 2015. 1

[221] Steering Committee for Foundations For Innovation in Cyber-Physical Systems. Strategic R&D Opportunities for 21st Century Cyber-Physical Systems. Technical report, National Institute of Standards and Technology, January 2013. 1

[222] J. Sztipanovits, X. Koutsoukos, G. Karsai, N. Kottenstette, P. Antsaklis, V. Gupta, B. Goodwine, J. Baras, and Shige Wang. Toward a Science of Cyber-Physical System Integration. *Proc. of the IEEE*, 2011. ISSN 0018-9219. 2.1, 9.3

[223] J. Sztipanovits, T Bapty, S. Neema, L Howard, and E Jackson. OpenMETA: A Model and Component-Based Design Tool Chain for Cyber-Physical Systems. In *From Programs to Systems - The Systems Perspective in Computing*, Grenoble, France, 2014. ISBN 978-3-642-54847-5. 1, 9.3

[224] J. Sztipanovits, T. Bapty, X. Koutsoukos, Z. Lattmann, S. Neema, and E. Jackson. Model and Tool Integration Platforms for CyberPhysical System Design. *Proceedings of the IEEE*, 106(9):1501–1526, September 2018. ISSN 0018-9219. doi: 10.1109/JPROC.2018.2838530. 2.1, 9.3

[225] Janos Sztipanovits. Cyber Physical Systems Convergence of Physical and Information Sciences. *it - Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, 54(6):257–265, 2012. doi: 10.1524/itit.2012.0688. URL `http://www.degruyter.com/view/j/itit.2012.54.issue-6/itit.2012.0688/`

itit.2012.0688.xml. 1

[226] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 107–119, New York, NY, USA, 1999. ACM. ISBN 1-58113-074-0. doi: 10.1145/302405.302457. URL http://doi.acm.org/10.1145/302405.302457. 9.2

[227] Martin Torngren, Ahsan Qamar, Matthias Biehl, Frederic Loiret, and Jad El-khoury. Integrating viewpoints in the development of mechatronic products. *Mechatronics*, 2013. ISSN 0957-4158. 9.3

[228] A.S. Uluagac, V. Subramanian, and R. Beyah. Sensory channel threats to Cyber Physical Systems: A wake-up call. In *Conference on Communications and Network Security*, pages 301–309, 2014. doi: 10.1109/CNS.2014.6997498. 3.4

[229] Anton Valukas. Report to Board of Directors of General Motors Company Regarding Ignition Switch Recalls. Technical report, Jenner & Block, 2014. 1, 2.2

[230] Ken Vanherpen, Joachim Denil, Istvan David, Paul De Meulenaere, Pieter J. Mosterman, Martin Torngren, Ahsan Qamar, and Hans Vangheluwe. Ontological reasoning for consistency in the design of cyber-physical systems. pages 1–8. IEEE, April 2016. ISBN 978-1-5090-1156-8. 9.3

[231] Dniel Varr, Gbor Bergmann, bel Hegeds, kos Horvth, Istvn Rth, and Zoltn Ujhelyi. Road to a Reactive and Incremental Model Transformation Platform: Three Generations of the VIATRA Framework. *Softw. Syst. Model.*, 15(3):609–629, July 2016. ISSN 1619-1366. doi: 10.1007/s10270-016-0530-4. URL http://dx.doi.org/10.1007/s10270-016-0530-4. 9.2

[232] G. K. Venayagamoorthy. Potentials and promises of computational intelligence for smart grids. In *2009 IEEE Power Energy Society General Meeting*, pages 1–6, July 2009. doi: 10.1109/PES.2009.5275179. 1

[233] Marcel Verhoef, Peter Visser, Jozef Hooman, and Jan Broenink. Co-simulation of Distributed Embedded Real-Time Control Systems. In Jim Davies and Jeremy Gibbons, editors, *Integrated Formal Methods*, Lecture Notes in Computer Science, pages 639–658. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-73210-5. 9.3

[234] Gerhard Wanner, Ernst Hairer, and Syvert Nrsett. *Solving Ordinary Differential Equations I - Nonstiff Problems*. Springer Series in Computational Mathematics. 2nd edition, 1993. URL http://www.springer.com/mathematics/analysis/book/978-3-540-56670-0. 2.1, 9.1.2

[235] Anthony I. Wasserman. Tool Integration in Software Engineering Environments. In *Proceedings of the International Workshop on Environments on Software Engineering Environments*, pages 137–149, New York, NY, USA, 1990. Springer-Verlag New York, Inc. ISBN 978-3-540-53452-5. URL http://dl.acm.org/citation.cfm?id=111335.111346. 9.3

[236] E.M. Wolff, U. Topcu, and R.M. Murray. Automaton-guided controller synthesis for

nonlinear systems with temporal logic. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4332–4339, November 2013. doi: 10.1109/IROS.2013.6696978. 9.1.1

[237] J. Woodcock, A. Cavalcanti, J. Fitzgerald, P. Larsen, A. Miyazawa, and S. Perry. Features of CML: A formal modelling language for Systems of Systems. In *2012 7th International Conference on System of Systems Engineering (SoSE)*, pages 1–6, July 2012. doi: 10.1109/SYSoSE.2012.6384144. 9.1.1, 9.3

[238] Zhibin Yang, Kai Hu, Dianfu Ma, Lei Pi, and J.-P. Bodeveix. Formal semantics and verification of AADL modes in Timed Abstract State Machine. In *International Conference on Progress in Informatics and Computing*, volume 2, pages 1098–1103, 2010. doi: 10.1109/PIC.2010.5687996. 10.3

[239] Robert K. Yin. *Case Study Research: Design and Methods*. Sage Publications, Inc, 4th edition, October 2008. ISBN 1-4129-6099-1. 8.1.2

[240] Olaf Zimmermann, Jana Koehler, Frank Leymann, Ronny Polley, and Nelly Schuster. Managing architectural decision models with dependency relations, integrity constraints, and production rules. *J. Syst. Softw.*, 82(8):1249–1267, August 2009. ISSN 0164-1212. doi: 10.1016/j.jss.2009.01.039. 9.2