Integration of Modeling Methods for Cyber-Physical Systems

Ivan Ruchkin

CMU-ISR-18-107
March 2019

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 NSF, AFRL, DARPA, NSA, DoD, ASD(R&E), or the SEI.

Keywords: cyber-physical system, embedded system, integration, model, model checking, logic, analysis, contract, architecture, view, 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* — analyses performed out-of-order and 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 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. This thesis proposes two abstractions: views (annotated component-and-connector models, inspired by software architecture) and behavioral properties (expressions in model-specific property languages, such as the linear temporal logic). Combining these abstractions lets engineers relate 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 the 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. This validation has shown that the approach can find safety-critical errors by specifying expressive integration properties and soundly checking them within practical constraints — all while being customizable to heterogeneous models, analyses, and domains.

Acknowledgments

First and foremost, I am grateful to my advisor David Garlan for his guidance, patience, and support. David's guidance comes in many forms, but most prominently in how he goes about excavating and conveying the key insight of a research study. David showed me a disciplined approach of understanding where the audience's mind is, where it needs to be, and how to smoothly move it from one place to the other. David's patience towards his PhD students is monumental, which works well for a student like me, who gets discouraged when the work does not progress fast enough (relative to some arbitrary preconceived expectation). David's support is rooted in his appreciation of the student's development as a researcher. Other attributes like expediency, costs, and external recognition are secondary to that development. Although at times I felt impatient, looking back now, I am happy that we did not compromise long-term investments for short-term convenience.

I am indebted to my thesis committee. André Platzer is my role model for rigor in research on cyber-physical systems, and an invaluable help in navigating the thicket of logic and automated reasoning. It is hard to find someone more dedicated to the depth and impact of inquiry. Every time André asked me "What does this mean?" I eventually arrived at two realizations: I did not fully understand the object in question, and he understood it more than I did. Bruce Krogh was very supportive of my pursuits, asked profound questions, and offered intriguing directions to explore. Conversations with Bruce made me see my work from new angles, nicely corresponding to the nature of the work itself. Dio de Niz was a great mentor early in my PhD, when I was the most confused and helpless. In addition to lending his expertise in real-time systems, Dio reinforced my belief in the value of connecting diverse technical domains. He also reminded me that, even if gruesome at times, research does not need to be somber — many memories are us laughing about the realities of producing knowledge. John Day brought a real-world perspective on model-based engineering, not letting me get too stuck in my academic bubble. It was reassuring to see that the problems I worked on were not a complete fabrication of curious minds and did affect other engineers. John stressed the importance of thinking carefully not only about *what* I was doing, but also *how* I was doing it — while sometimes I had no clue about either.

I also want to thank my other mentors. Throughout my stay at CMU, Bradley Schmerl assisted and supported me a lot, often through his exceptional ability to quickly understand a random technical caveat I brought up, summarize it clearly, and ask productive questions about it. Every time I went to Bradley's office, I would see an Einstein's quote: *"Imagination is more important than knowledge."* It gave me some consolation: although I was unsure about my imagination, I frequently found myself in a dire lack of knowledge. Bradley is also credited with making me see Australian white ibises for what they *really* are. Josh Sunshine offered ideas and feedback on the parts of my work that many readers would skip for the sake of their sanity. Conversations with Josh were refreshingly honest about software engineering research and academic careers. Josh also has a unique gift of bringing up the *exact* thing that I was hoping to avoid thinking about (and probably should

have thought about as soon as possible). Sagar Chaki is the single most cheerful and relaxed researcher I have ever met. Sometimes Sagar seemed so much at ease when facing difficult problems that it made me wonder if I am on a right career path, given my uncontrolled skepticism and doubt. Sagar's attitude was supportive and motivating, especially when reading what Reviewer 3 had to say about our papers.

A special gratitude goes to my two friends who made a big difference in how I approached this PhD. Before I met Ada Zhang, I knew that sitting down and working consistently on the most important (not necessarily urgent) research tasks was half the battle. I was also plagued by some unknown unknowns of American English pronunciation. After we met, Ada led our two-person team to deliver incredible victories against the *major* enemies of PhD students: avoidance and subsequent guilt. Had I been told about these victories earlier, I would have not believed it. Ashutosh Pandey, in addition to being a great friend and colleague, connected the PhD challenges to the concept of self (*"ego"*), which rewards inflation of self-importance and resists being humbled — even when it goes against our best interests. It was also insightful to observe closely how Ashutosh's PhD unfolded, without being a central actor in it. With these realizations, and the philosophy of having the right to do the work, but not to claim its outcomes, it got easier for me to face the daily struggle.

I want to acknowledge many collaborators and colleagues in ISR and CMU, including Vishal Dwivedi, Javier Camara, Selva Samuel, and many others. It was a pleasure to work with you. Ajinkya Bhave and Akshay Rajhans carried the flag of multi-model CPS research before me, making it easier to follow in their footsteps. I owe my meta-models of academia, disciplines, and research to Mary Shaw. Jing Xiang offered understanding and support no matter how good-to-have my issue seemed to be. I am also grateful to the students I supervised in summer programs and independent studies: without you, I would have graduated *even later*.

I would like to express my appreciation to the groups that I was fortunate to be part of. The ABLE group was a prime venue for debate and intellectual entertainment. The weekly SSSG was the main driving force behind my improvement as a speaker. The CPS group meetings and BRASS meetings gave me a lot of food for thought about how we develop complex technology. Other groups I would like to thank are the CMU PhD Support Group, the ISR Bakchodi Group, the ISR Pilsner Club, and the Oakland Toastmasters — all of which pushed me forward through various combinations of listening, speaking, and consuming beverages. A special thanks goes to my running friends at the Pittsburgh Pharaoh Hounds, who kept me in touch with the physical reality outside of CMU's walls.

Let me also thank my family and friends, mostly in Russia, who managed to not forget that I exist, despite my sporadic contact with them. My Mom supported my ambitions, but also accepted me as I was. When leaving the PhD program seemed my best option, she said, *"Then leave,"* which somehow led to me staying. My Dad checked up on me asking, *"How is your health and research?"* thus making me focus on what he, as a medical doctor and a professor, considered essential.

Last but not least, some very important people have to go unnamed in this section. They know who they are. Thank you for suffering with me.

Contents

List of Symbols **xi**

1 Introduction **1**
1.1 Thesis Statement and Claims 8
1.2 Thesis Organization 10

2 Background: Modeling Methods for Cyber-Physical Systems **13**
2.1 Models, Analyses, and Methods 13
2.2 Modeling Method Integration 17

3 Case Study Systems **21**
3.1 System 1: Energy-Aware Adaptation for Mobile Robot 21
3.2 System 2: Collision Avoidance for Mobile Robot 23
3.3 System 3: Thread and Battery Scheduling for Quadrotor 26
3.4 System 4: Reliable and Secure Sensing for Autonomous Vehicle 28

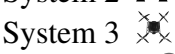
4 Approach to Modeling Method Integration **35**
4.1 Integration Abstractions 35
4.2 Integration Properties 37
4.3 Integration of Analyses 38
4.4 Integration Argument for Models 39
4.5 Integration Arguments for Analyses 42

5 Part I: Integration Property Language **45**
5.1 Motivating Integration Property 45
5.2 IPL Concepts and Preliminaries 46
5.2.1 IPL Design 46
5.2.2 Views and Behavioral Properties 47
5.3 IPL Syntax 51
5.3.1 Plugin Points for Behavioral Properties 53
5.3.2 LTL Plugin Syntax 54
5.3.3 PCTL Plugin Syntax 54
5.3.4 Syntactic Examples 55
5.4 IPL Semantics 56

5.4.1 Semantic Domains and Transfer . . . 56
5.4.2 Native semantics . . . 57
5.4.3 LTL Plugin Semantics . . . 58
5.4.4 PCTL Plugin Semantics . . . 59
5.5 IPL Verification Algorithm . . . 60
5.5.1 Formula Transformations . . . 61
5.5.2 Algorithm Steps . . . 62
5.5.3 Application to Running Example . . . 64
5.6 Theoretical Evaluation . . . 66
5.7 IPL Implementation . . . 69

6 Part II: Structural and Behavioral Integration Abstractions 71
6.1 Running Example: Hybrid Program, Hardware Model . . . 73
6.2 Structural Integration Abstractions: Views . . . 75
6.2.1 Internal Organization of Views . . . 76
6.2.2 Integration Viewpoints . . . 77
6.2.3 Conformance, Soundness, and Completeness of Views . . . 80
6.2.4 View Abstractions for Hybrid Programs . . . 83
6.2.5 Automating View Creation and Conformance . . . 86
6.2.6 Integration Argument with Views . . . 90
6.3 Behavioral Integration Abstractions: Properties . . . 95
6.3.1 Behavioral Languages and Queries . . . 95
6.3.2 Behavioral Property Abstractions for Hybrid Programs . . . 96
6.3.3 Integration Argument with Model Querying . . . 97
6.3.4 Shared Background between Abstractions . . . 100
6.4 Comparison of Integration Abstractions . . . 100

7 Part III: Analysis Execution Platform 103
7.1 Domain Signatures and Analysis Contexts . . . 103
7.2 Analysis Contracts . . . 106
7.3 Analysis Execution . . . 107
7.4 AEP Implementation . . . 109

8 Validation 111
8.1 Theoretical Evaluation of Soundness . . . 111
8.2 Validation of Part I: Integration Property Language . . . 114
8.2.1 Evaluation of IPL on System 1  . . . 114
8.2.2 Evaluation of IPL on System 3  . . . 130
8.2.3 Summary for Evaluation of IPL . . . 133
8.3 Validation of Part II: Integration Abstractions . . . 134
8.3.1 Evaluation of Integration Abstractions on System 1  . . . 134
8.3.2 Evaluation of Integration Abstractions on System 2  . . . 138
8.3.3 Evaluation of Integration Abstractions on System 3  . . . 144
8.3.4 Evaluation of Integration Abstractions on System 4  . . . 148

8.3.5 Summary for Evaluation of Integration Abstractions . . . 151
8.4 Validation of Part III: Analysis Execution Platform . . . 152
8.4.1 Evaluation of AEP on System 3 . . . 152
8.4.2 Evaluation of AEP on System 4 . . . 161
8.4.3 Summary for Evaluation of AEP . . . 165

9 Related Work **167**
9.1 Modeling Methods for CPS . . . 167
9.1.1 Discrete Modeling Methods . . . 167
9.1.2 Continuous and Hybrid Modeling Methods . . . 168
9.2 Foundations for the Integration Approach . . . 170
9.2.1 Software and Systems Architecture . . . 170
9.2.2 Logic and Verification . . . 171
9.2.3 Compositionality, Contracts, and Dependencies . . . 172
9.3 Existing Integration Approaches . . . 172
9.3.1 Structural Approaches . . . 173
9.3.2 Semantic Approaches . . . 175
9.3.3 Mixed Approaches . . . 176

10 Discussion **179**
10.1 Scope of Applicability . . . 179
10.2 Limitations . . . 181
10.3 Design Rationale . . . 184
10.4 Future Work . . . 187
10.4.1 Short-term Improvements . . . 187
10.4.2 Long-term Research Directions . . . 189

11 Conclusion **193**
11.1 Contributions . . . 194

Bibliography **195**

List of Symbols

Math and Logic

$\equiv$	Equivalence
$\lvert x \rvert$	Absolute value of x
$\mapsto$	Functional mapping
$\rightsquigarrow$	Syntactic transformation
$\phi\{x/y\}$	Substitution of x for y in ϕ
$\{x_1 \dots x_n\}$	Set of $x_1 \dots x_n$
$\langle x_1 \dots x_n \rangle$	Sequence from x_1 to x_n
$\mathcal{P}(X)$	Power set of X (set of all subsets of X)
$\mathbb{O}$	Some background set of values
$\mathbb{Z}, \mathbb{R}, \mathbb{B}$	Integers, reals, booleans
$\top, \bot$	True, false
$\wedge, \vee$	Conjunction, disjunction
$\rightarrow, \Leftrightarrow$	Implication, bi-implication
$\forall, \exists$	For all, exists
G, F	Globally, eventually
$\mathsf{U}, \mathsf{U}^{\leq \mathsf{b}}$	Until, bounded until
X	Next
$\models$	Models/satisfies

Models and Abstractions

$\mathcal{M}, \mathcal{M}^s, \mathcal{M}^b$	Model, structural model, behavioral model
$\mathbb{M}$	Set of models
$e^{\mathcal{M}}$	Model element
$\boldsymbol{e}^{\mathcal{M}}$	Tuple of model elements
$\mathbb{E}^{\mathcal{M}}$	Set of model elements
ω	Behavior/trace
Ω	Set of model behaviors
$\mathfrak{Q}$	Parametric structure of model behaviors
l	Behavioral property
$\mathbb{L}$	Behavioral language
Q	Behavioral query

$\mathcal{V}$	View
$\mathbb{V}$	Set of views
$e^{\mathcal{V}}$	View element
$\boldsymbol{e}^{\mathcal{V}}$	Tuple of view elements
$\mathbb{E}^{\mathcal{V}}$	Set of view elements
T	Typing function for view elements
$\mathbb{T}$	Set of view element types
P	View element property function
$\mathbb{P}$	Set of view element property functions
$\mathcal{VP}$	Viewpoint
mp	Matching predicate
$\mathbb{MP}$	Set of matching predicates
VA	Viewpoint algorithm
sound	View soundness predicate
$\mathrm{complete}$	View completeness predicate
$\mathrm{integprop}$	Ground-truth integration property predicate

Integration Property Language — Syntax

$\Sigma, \Sigma^{\mathcal{M}}, \Sigma^{\mathcal{V}}, \Sigma^{B}$	Syntactic signature, model signature, view signature, background signature
V	Set of variables
D	Variable's concrete domain
VAR	Quantified/free variable (syntax)
DOM	Quantification domain (syntax)
STVAR	State variable (syntax)
S	Set of state variables
CONST	Constant (syntax)
C	Set of constants
VFUNC	View function (syntax)
VF	Set of view function names
MFUNC	Model function (syntax)
MF	Set of model function names
BFUNC	Background function (syntax)
ELEMTYPE	Type of view elements (syntax)
PROP	View element property (syntax)
VALTYPE	Type of some background values (syntax)
MTERM	Term (syntax)
RTERM	Rigid term (syntax)
ATOM	Atomic formula (syntax)
RATOM	Rigid atomic formula (syntax)
MATOM	Model atomic formula (syntax)

TATOM	Temporal atom (LTL syntax)
PATHPROP	Path property (PCTL syntax)
RWDPATHPROP	Path property for rewards (PCTL syntax)
PPROP	Probabilistic property (PCTL syntax)
RWDPROP	Rewards property (PCTL syntax)
PQUERY	Probabilistic query (PCTL syntax)
RWDQUERY	Rewards query (PCTL syntax)
MDLINST	Model instantiation clause (syntax)
PN	Set of model instantiation parameter names
$\{\|p_1 = x_1 \ldots p_n = x_n\|\}$	Value assignment of model parameters
FORMULA	IPL formula (syntax)
ToPNF	Transformation to Prenex Normal Form (PNF)
RemQuant	Quantifier removal transformation
ConstAbst	Transformation of constant abstraction
FuncAbst	Transformation of functional abstraction

Integration Property Language — Semantics

$\Gamma, \Gamma^{\mathcal{V}}, \Gamma^{\mathcal{M}}, \Gamma^{B}$	Semantic structure, view structure, model structure, background structure
$\mathsf{D}, \mathsf{D}_{\mathcal{M}}, \mathsf{D}_{\mathcal{V}}$	Semantic domain, model's semantic domain, view's semantic domain
q	State in a model
μ	Assignment of variables
Θ	Set of possible variable assignments
MF	Set of possible model functions
PF	Set of functions from parameter names to parameter values
$I, I^{\mathcal{V}}, I^{B}$	Interpretation, view interpretation, background interpretation
$I^{\mathcal{M}}, I_q^{\mathcal{M}}, I_\omega^{\mathcal{M}}$	Model interpretation, on a state q, on sequence ω
F	Functional abstraction (function itself)
C	Constant abstraction (constant itself)
SV	Variable values satisfying the search formula
I^F	Interpretation of functional abstractions
I_{SV}^F	Interpretation of functional abstractions on SV
I^{CA}	Interpretation of constant abstractions
f, f^{PNF}	IPL formula, in PNF
$\hat{f}$	IPL formula without quantifiers
f^{FA}	IPL formula with functional abstractions
$\hat{f}^{FA}$	IPL formula with functional abstractions and without quantifiers
f^{CA}	IPL formula with constant abstractions
$\hat{f}^{CA}$	IPL formula with constant abstractions and without quantifiers

Analyses and Contracts

A	Analysis
$\mathbb{AN}$	Set of analyses
$\mathcal{C}$	Analysis contract
a, g	Analysis assumption, analysis guarantee
$\mathbb{A}, \mathbb{G}$	Set of assumptions, set of guarantees
i, o	Analysis input, analysis output
$\mathbb{I}, \mathbb{O}$	Set of inputs, set of outputs
Σ	Analysis domain signature
Υ	Analysis context
$\mathcal{O}$	Order of analyses
$\mathrm{depends}(\mathsf{A}_1, \mathsf{A}_2)$	Analysis A_1 depends on analysis A_2

Chapter 1

Introduction

An emerging class of systems, called *cyber-physical systems (CPS)*, rely on 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 [95]. Another example is that incorporating renewable energy sources into smart power grids could lead to large reductions in emissions and environmental damage [167, 260]. Similar effects are expected in other industries and domains [107, 144, 189, 248, 249, 253].

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 [232]. The models that are used to design CPS describe a broad range of structural and behavioral aspects that often include program execution, hardware design, and mechanical dynamics [32, 86, 253]. The main advantage of models over informal descriptions is that engineers can perform *analyses* over models [124, 201]. Broadly, analyses are any operations that interpret models and/or create new versions of models, ranging from manual safety inspections to algorithms that optimize model parameters (e.g., finding an optimal set of control gains). For example, a bin packing analysis [201] 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. Correctness is particularly needed in safety-critical contexts, which call for rigorous up-front verification and validation — as opposed to informal, post-factum, and ad hoc quality assurance. Reasoning about large systems is complicated by models using continuous, discrete, and probabilistic constructs to represent the physical and digital worlds [163]. Another factor that makes CPS engineering hard is the timing of various dynamics: computations, networking, and physical actions must be synchronized as intended by the system designers [162]. 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 [86, 139].

CPS modeling methods originate in heterogeneous disciplines, such as mechanical, control, and software engineering. This heterogeneity makes it hard to *integrate* (i.e., use related models and dependent analyses) 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 source code generated by the other. Generally, I observe that *model consistency* (i.e., the absence of contradictions or mismatches in the shared information and assumptions of models) is required for analyses to remain modular and have compatible downstream results. As illustrated in Figure 1.1, inconsistencies between models threaten our ability to combine the outputs of analyses, potentially leading 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 [9], in which a mismatch between imperial and metric units led to a trajectory miscalculation and subsequent disintegration of the spacecraft.

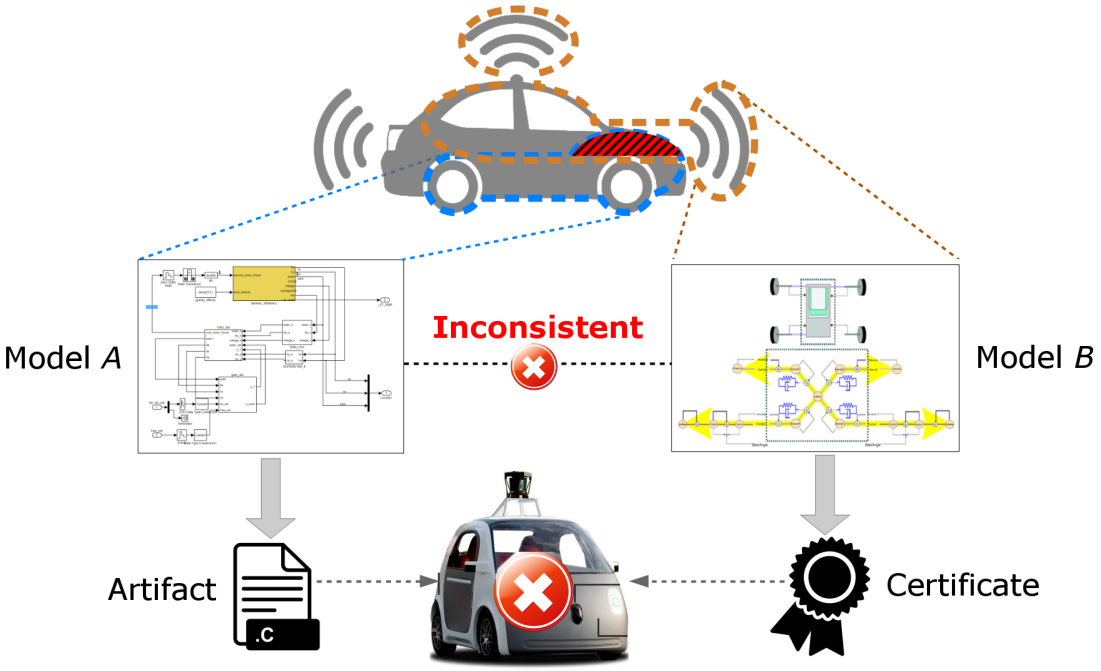


Figure 1.1: Inconsistent models A and B represent overlapping parts of the system in a conflicting way. Outputs of inconsistent models, when combined in a deployed system, may lead to failures.

Another challenge of combining CPS modeling methods is ensuring *correct analysis interactions*. One type of interaction is *data dependency*, where one analysis uses the model information that is produced by another analysis. The order of executing analyses should not violate their dependencies, otherwise outdated information may affect the design or conclusions about its qualities (e.g., safety). For example, accurate control simulation depends on how real-time scheduling analyses allocate computational tasks to processors [66]: this allocation determines the execution times of control tasks. Therefore, the simulation analysis should be performed after the scheduling analysis. If the simulation is performed on an earlier, outdated allocation, its outputs are not

necessarily valid for a newer allocation produced by running the allocation analysis after the simulation. An out-of-order execution is illustrated in Figure 1.2: if model A is changed, running analysis Q before analysis P would use an outdated version of model C, introducing an error into the system's implementation.

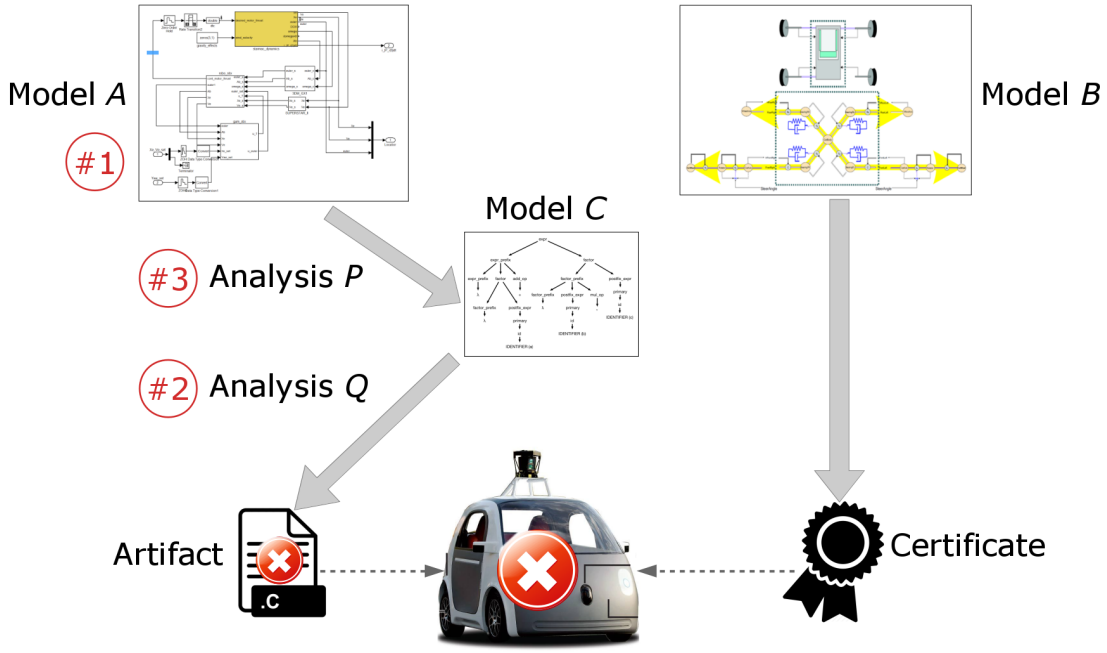


Figure 1.2: An out-of-order execution: change model A, run analysis Q, run analysis P. The correct order: change model A, run analysis P, run analysis Q.

Another type of analysis interactions, *context mismatch*, occurs when one analysis changes the *context* of another analysis (i.e., the models that this analysis interprets) in an unexpected way. For instance, suppose the control simulation is only applicable to allocations with a fixed execution time, but the scheduling analysis produced non-deterministic intervals for execution times. In this case, the simulation would not accurately represent the uncertainty of execution times, and may avoid the cases where control fails. Mismatch of analysis contexts is illustrated in Figure 1.3: analysis P changes model C, which is part of the context of analysis R, in an unexpected way, leading analysis R to produce a flawed certification. Generally, incorrect analysis interactions can lead to erroneous outputs of analyses: bugs in generated code or flaws in safety certifications. Such an interaction between electrical and mechanical aspects of the GM ignition switch led to multiple ignition failures, car crashes, and deaths [257]. 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, I introduce *modeling method integration (MMI)* — an approach for combining modeling methods without model inconsistencies or incorrect analysis interactions. The desired outcomes of MMI are defined

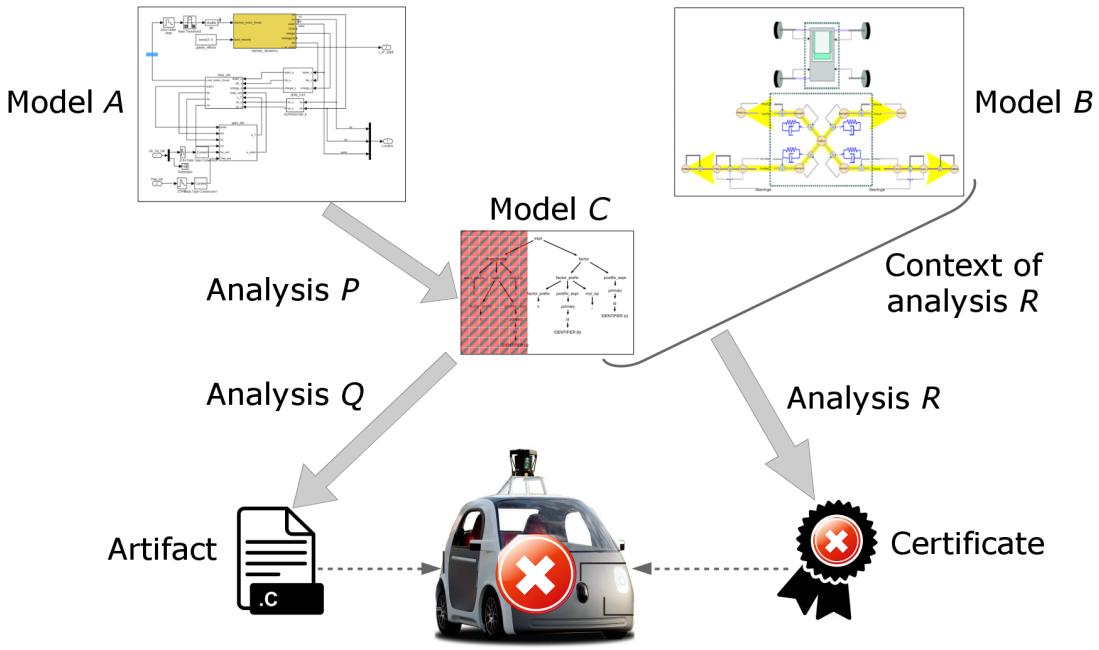


Figure 1.3: The context of analysis R includes models B and C. Analysis P introduces unexpected information into model C, leading analysis R to produce an erroneous certificate of safety.

for an *integration scenario* — an engineering context with a set of possible designs (defined in terms of models and analyses) and a system requirement, which may be violated in some of the designs due to the interactions between multiple models and analyses. In practice, MMI is typically performed informally and without guaranteeing safety-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 (i.e., a design declared erroneous/error-free is a priori guaranteed to be so), modular (i.e., the models/analyses remain independent and evolve separately), and practical (i.e., applicable to real-world CPS) way is the central problem addressed in this thesis.

Although partial MMI approaches exist, the body of CPS research does 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 [49], and thus rendering the approach inapplicable to systems of realistic size. For example, in hybrid systems, large numbers of continuous variables can make verification impractical [76, 89], and explicitly composing hybrid automata with source code can make verification impractical [14]. For timed automata, analyzing a combined model can take 60+ times longer than analyzing its parts individually [155]. Compositional and refinement approaches can improve scalability [169, 198], but still time out on

large inputs [39], and state space explosion remains a major obstacle in formal methods [202]. Another option is checking part of verification obligations at run time via monitors and enforcing safety via sandbox controllers [29, 192], but such approaches limit up-front design exploration. Finally, 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 [163].

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) [25], behavior relations (mapping traces of states between models) [226], CyPhyML/OpenMETA (connecting models through logical interfaces) [251], Ptolemy II (simulation of heterogeneous computations) [165]. 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, they are not general or expressive enough to address the broad variety of modeling methods used for CPS); finally, some assume top-down development from requirements to implementation using fixed modeling methods like Hybrid Event-B [15] and iCyPhy [208] (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 [26], static constraints on view parameters [227], and directly relating behaviors from multiple models [225]. 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 processors (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 preconceived definitions of consistency (such as structural consistency of views [26]), 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 [25]. Another issue is that once committed to a given abstraction (e.g., EAST-ADL for timed automata [178]), the framework is difficult to extend for other, potentially more convenient abstractions. Finally, many integration abstractions, such as logical interfaces [246] and architectural views [25], 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 1.4. Models and analyses are the inputs of the approach, and its goal is to check consistency between the models and prevent incorrect interactions between 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 a model-specific logic to 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: engineers can tailor integration properties to describe the particular notion of consistency that is relevant for the models and the requirement of the scenario. To control analysis interactions, model-based analyses are executed by the *Analysis Execution Platform (AEP)* that ensures that the analyses read and write information to the models in a correct order, and that the execution context of each analysis is appropriate. The execution of the analyses is based on *contracts* — specifications describing how analyses interact with models, and consisting of the inputs, outputs, assumptions, and guarantees of each analysis.

More specifically, this thesis makes three central contributions 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 representations of models for the purposes of integration.

III. An analysis execution platform that provides an environment for execution of model-based analyses. Using analysis contracts, the platform guarantees satisfaction of analysis dependencies and execution only in appropriate contexts.

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 approaches to consistency checking. To co-constrain the structure and behavior of heterogeneous models, I have developed IPL — a customizable formal specification language based on the first-order logic. This language allows one to plug in expressions in arbitrary model-specific (e.g., modal) logics as sub-formulas. I have also developed a verification algorithm that combines satisfiability solving

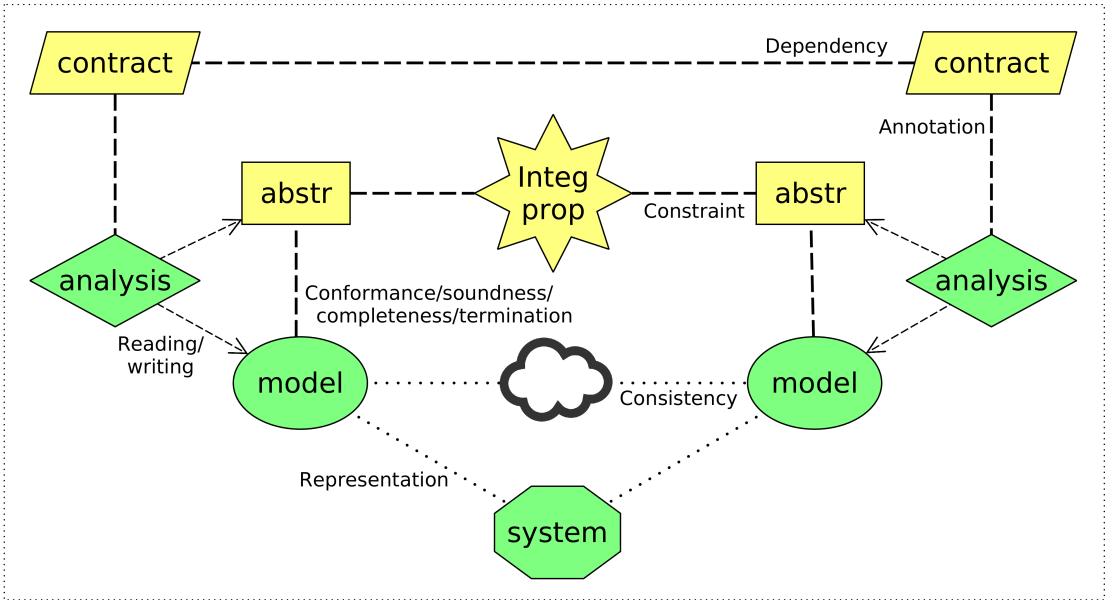


Figure 1.4: The proposed approach to CPS modeling method integration.

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 complementary integration abstractions: views and behavioral properties. The first abstraction, views, serve as a common language to represent structures contained in 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 (view soundness and completeness) that are sufficient for correct integration. The second abstraction, behavioral properties, are statements in model-specific languages that are plugged into IPL. The models are then queried to interpret these statements and return their values. Similar to the case of views, I formalize the conditions sufficient for correct integration with behavioral properties (query soundness and termination).

Part III of this thesis is an execution platform for 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, specified in terms of view types, determine the correct order of analysis execution, guaranteeing that no stale information is consumed and no newer information is overwritten. By checking the assumptions and guarantees, the platform ensures that analyses are run only in contexts where they produce correct results.

This thesis aims to perform integration of reasoning over models — not develop individual models or single-model reasoning from scratch. Thus, my approach assumes that the existing models are syntactically well-formed, and that their analyses (including reasoning, such as model checking) are performed correctly with respect to the model semantics. 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.

I have validated my approach in 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 task scheduling and dynamic battery cell charge/discharge scheduling, and an autonomous vehicle with redundant sensors. Each case study exercises different parts of the approach to provide evidence for the research claims described in the following section. To enable this validation, the specification languages and algorithms were implemented in two architectural design environments: *AcmeStudio* [243] (for views to take advantage of customized architectural styles of the *Acme* architectural description language [98]) and *OSATE2* [79] (to apply the execution platform to the analyses available for the *Architecture Analysis and Design Language — AADL* [80]).

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 the following four qualities of integration outcomes:

- *Expressiveness:* an expressive MMI approach captures properties that depend on both the structure of models and their behavior (described in more detail in Chapter 2). As a result, such an approach is suitable for detection of inconsistencies that manifest as mismatches between structure and behavior. Therefore, 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, in a general sense, produces only the outputs that can be trusted. For model consistency (defined formally as integration properties), the approach reports that a set of models is consistent (inconsistent) if and only if these models are indeed consistent (inconsistent) according to the semantics of the integration properties. The approach does not have to produce an answer for every integration property on every set of models (completeness). For analysis interactions, the approach executes a set of analyses if and only if this execution would respect their input-output dependencies (specified by the analysis contracts) and invoke every analysis only in an appropriate context (specified by the analysis contracts as well). The approach does not have to find an execution for every possible set of analyses: it is acceptable to abort executions that do not satisfy the above conditions.
- *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 explicit equational models (e.g., algebraic and differential equations and inequalities). 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 consistency of models in terms of structure and behavior (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 between analyses and models, 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 the research claims below.

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

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

Claim 2. The *soundness* of 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. If sound verification of multi-model consistency determines whether an integration property holds on a set of models, this determination always agrees with the semantics of the integration property on these models. Sound integration of multiple analyses executes them only in the order of their dependencies and within an appropriate context.

Claim 3. The *practical applicability* of MMI is enabled by using flexible abstractions to handle corner cases and delegate 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 CPS engineering 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. The *customizability* of MMI is enabled by syntactically embedding model-specific behavioral property languages into integration properties (Part I) and tailoring views to heterogeneous models from diverse 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., *Computation Tree Logic* — CTL [48] — for a model 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 MMI. To exemplify the challenges of MMI, Chapter 3 describes the four systems that were used for validation in this thesis. These systems are used as illustrations throughout the subsequent chapters, and also serve as the contexts for the validation case studies described later, in Chapter 8:

- *System 1:* energy-aware adaptation for a mobile robot
- *System 2:* collision avoidance for a mobile robot
- *System 3:* scheduling of real-time tasks and dynamic batteries 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 Chapters 5 to 7 elaborate on the three technical parts of the thesis. 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 is wrapped up with a discussion of limitations, design rationale, and future directions in Chapter 10, and concluded in Chapter 11.

To assist the reader's navigation through this thesis, Table 1.1 indicates how the technical parts of the approach relate to the claims and the case study systems. Each claim was validated on at least two systems, chosen based on the evaluation opportunities in each system.[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 (Part II) that supported Part III.

Approach part	Claim	Case study system			
		#1	# 2	# 3	# 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: A mapping between technical parts, claims, and case study systems. A check mark indicates that a claim (row) for a technical part of the thesis (row) was sufficiently evaluated on a system (column). An absence of a check mark indicates that the system was not appropriate for evaluating the claim.

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. In the end, 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 [65, 122, 164]. A *model* is a formal representation of a system or its part [232]. 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 [115]. The meaning of the model is known as the model's *semantics*, often given denotationally (as a mapping from a model to a mathematical structure, such as a set of behaviors) or operationally (as a set of rules for executing an abstract machine). 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 model concepts of state, computation, and time [86].

Models are typically specified using *formal languages* [32] — collections of sentences defined by a formal *syntax* that is based on explicit rules for generating those sentences. 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 [89] 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 and inconsistencies, as discussed below.

Integration of heterogeneous models requires creating special representations, or *abstractions* (similar to the concepts of wrappings [22], model aspects [250], integration adapters [68], and semantic interfaces [252] in related work). For the purposes of integration, I interpret models as collection of abstract parts, called *model elements*. To make my integration approach customizable, I make only minimal assumptions about the nature, characteristics, and relations of these elements. Model elements may be syntactic (i.e., constructs of the language in which the model is specified) or semantic (i.e., constructs of the interpretation or meaning of the model). The examples of elements 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 that is written in Linear Temporal Logic (LTL) [218] is used as an abstraction, the property exposes the behavioral elements (traces) that are specified by it. If, on the other hand, the abstraction focuses on the syntactic aspects of a model, the model and the abstraction are called *structural*. For instance, system modes in an LHA (e.g., inactive, active, and safe) encoded in a model may be represented as a set of discrete entities in a structural 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.

The types of models and their contents are summarized in Figure 2.1. Structural models contain static model elements, which are treated as abstract entities and can take a variety of forms in practice. Behavioral models encode behaviors, which are potentially infinite sequences of states. The specific forms of behavior may also differ across models. The details of abstractions for these models are presented in Chapter 6.

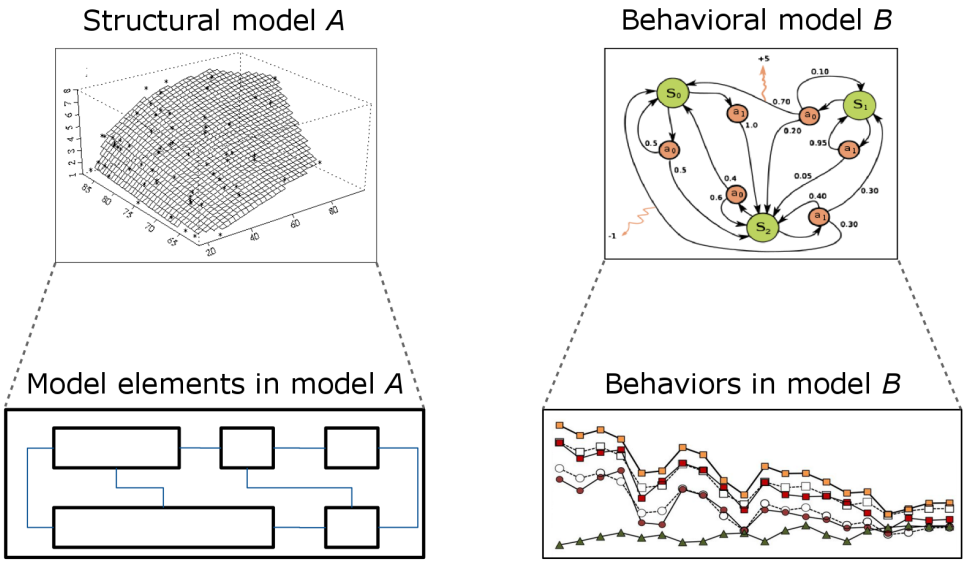


Figure 2.1: Structural models contain static model elements. Behavioral models encode behaviors of dynamic systems.

A model is useful when 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 refining them. For instance, a bin-packing analysis [201] 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 A_1 changes the same (type of) model elements that analysis A_2 consumes as inputs, I say that A_2 *is dependent* on 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 component properties (e.g., the frequency of a CPU — *Central Processing Unit*) and types of components (e.g., CPUs or threads) and 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 [38]. 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 [222, 228]. These methods rely on modeling notations that differ in their level of abstraction, computational model, notion of time, and so on [228]. For example, a synchronous dataflow program [161] describes discrete, ordered, and time-unaware computations. On the other hand, an *ordinary differential equation (ODE)* [262] 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, such as efficiency and security, can be explored and satisfied; (ii) higher degree of safety assurance by modeling the system's assumptions from multiple perspectives; (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 [58] or SCADE Suite [73] for control design, and SPICE2 for circuit design [200]. These models are widely used for control design and tuning via simulation in many industries including automotive, industrial, and aerospace engineering [134, 172].
- Discrete state-based modeling using state machines, process algebras, statecharts, labeled transition systems, and timed automata. The notations and tools include Promela with Spin [120], FSP with LTSA [174], Matlab and Stateflow [58], MontiArcAutomaton [230], Harel statecharts [113], and UPPAAL [159]. 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 [52].
- Hybrid system modeling with linear hybrid automata [7] or hybrid programs [213, 214, 215]. These models 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 [134]. Hybrid programs have been used to formally verify in aerospace [133, 171] and automotive domains [170, 191, 196]. Hybrid models are associated with a set of analyses based on reachability (via overapproximation) [42, 89, 147] and falsification (via underapproximation) [8, 70, 216].
- Differential equation modeling, often simplified to ODEs [262]. A commonly used notation that encapsulates ODEs is a lumped element model: a set of discrete entities that 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 [125]).
- System architecture modeling with languages such as the *Architecture Analysis and Design Language (AADL)* [80], the *Systems Modeling Language (SysML)* [63], the *Unified Modeling Language* (*UML*) [51], and Acme [97]. 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 [81] [96]. One of the extensions that combines architecture and hybrid modeling is the Sphinx toolset [194] that specifies hybrid programs in terms of UML class and activity diagrams, and enables collaborative proof engineering.

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 [222]. For example, car engine control [56] and infusion pump control [180] rely on the same core principles of control theory,

[1]This list is incomplete, but it represents the state-of-practice in CPS engineering. For more detail, see Section 9.1.

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 [162]. In contrast, combining the results of different methods remains an outstanding challenge [66, 162, 228].

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 [135]. Below I discuss two classes of integration issues: *model inconsistencies* and *incorrect analysis interactions*.

Consistency of models means that the models are related to each other in a way that is intended by the creators of these models. Thus, consistency can be understood as a subset of all possible model relations. Informally, consistent models describe a cohesive design of the same system without flaws or contradictions. For example, models that represent power consumption in a robot are consistent if their estimates of required energy agree for the same tasks of the robot. Consistency is threatened by potentially conflicting information across multiple models. Models of the same CPS are usually not fully 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 different forms: a mathematical function, a hardware chip, a collection of signal blocks, or a piece of source code.

Models are *inconsistent* if the relation between them is not from the intended set of consistent relations. An *inconsistency* is, then, a contradiction or a design flaw in how the models are related (see Figure 1.1 in Chapter 1). These contradictions and flaws are defined relative to the requirements of integration scenarios: an error in one design may be a non-issue in another. For example, an electrical model of a battery (the number, location, and connections of battery cells) needs to 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: bounded inconsistency (e.g., in terms of average differences in battery cell positions between the two

models) would not interfere with the correct operation of the battery.

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 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 downstream in the development. For example, a safety certification from one analysis may need to apply to the code generated by another analysis, and if the models are inconsistent, the safety certification may not apply to the generated code. What would it take to create analyses that do not require consistency of their models? An analysis not reliant on consistency would have to check all related models to ensure their compatibility with its outputs. This checking would be impractical in complex systems, and therefore almost all analyses implicitly rely on consistency of their models.

The consistency conditions across CPS models is often 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 the integration abstractions that are discussed in Chapter 6.

The second class of integration issues, *incorrect analysis interactions*, 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. For instance, 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 second analysis is run first and approves the set of sensors, running the first analysis later may invalidate the conclusion and potentially deliver an unsecure design. Such out-of-order executions are illustrated in Figure 1.2 in Chapter 1.

Finally, analyses may be run in a *mismatched context*, in which the expectations of an analysis are not satisfied by the models it reads. This situation is illustrated in Figure 1.3 in Chapter 1. Mismatch of execution contexts 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 the aforementioned 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 [257]. 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 [257]. 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 their 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 Summary

This chapter introduced important notions of CPS engineering: model, abstraction, analysis, and modeling method — along with a representative set of CPS modeling methods. When multiple models and analyses are involved in engineering, three problems may lead to errors and failures. First, models may be inconsistent. Second, analysis execution may disregard data dependencies. Third, analyses may be executed outside of their expected context.

Chapter 3

Case Study Systems

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

- *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 System 1: Energy-Aware Adaptation for 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 target location through a physical environment using a 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) [221], 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 minimizing the mission time and power consumption. For instance, if an obstacle blocks the chosen path, the robot needs to re-plan, potentially changing its configuration to reduce power consumption or re-charging along the way.

This system is a CPS of realistic complexity, so I chose it to investigate the applicability of my approach. Furthermore, this robot was engineered using 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 results of MMI.

The robot uses multiple models to adapt, including the environment, mission, and the robot's architecture models. The validation study (described further in Section 8.2) focused on two

models with a complex relationship: a power prediction model and a planning model, shown in Figure 3.1. The power prediction model ($\mathcal{M}_{\text{power}}$), or just power model, 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.

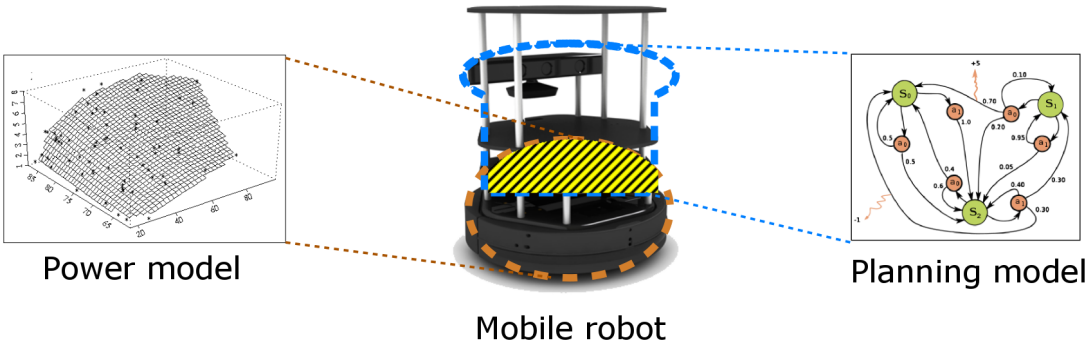


Figure 3.1: The power and planning models of a mobile robot have overlapping concerns.

The planning model ($\mathcal{M}_{\text{plan}}$) computes a planned path to the target location. The model represents a map and the robot's non-deterministic movements, with their duration and energy requirements, in a *Markov Decision Process (MDP)* [121]. The potential paths are evaluated with a utility function that weighs the relative priorities of the mission's duration and energy consumption. The model's state evolves temporally and includes the robot's current 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 at run time, the robot runs the PRISM probabilistic model checker [154] to produces a plan, which resolves the non-determinism of the action choices in each state. These choices are then fed to the robot's motion control. Although the energy-related coefficients in $\mathcal{M}_{\text{plan}}$ are derived from $\mathcal{M}_{\text{power}}$, these two models are is not equivalent to each other due to various modeling choices, optimizations, and compromises. For example, $\mathcal{M}_{\text{plan}}$ 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: $\mathcal{M}_{\text{power}}$ continuously monitors the robot's execution and alerts $\mathcal{M}_{\text{plan}}$ that the robot may run out of energy before completing the mission. Thus, $\mathcal{M}_{\text{plan}}$ only needs to be used 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 $\mathcal{M}_{\text{plan}}$ has overly conservative energy estimates compared to $\mathcal{M}_{\text{power}}$, 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 ($\mathcal{M}_{\text{map}}$) 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

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

locations. The map model serves as a foundation for $\mathcal{M}_{\text{plan}}$: the states and transitions in $\mathcal{M}_{\text{plan}}$ are created for a specific map. $\mathcal{M}_{\text{power}}$ does not directly relate to map, although, as discussed in Section 8.3, creating integration abstractions for $\mathcal{M}_{\text{power}}$ requires a known map.

In this scenario, modeling method integration should assist the *power safety argument* ("the robot never runs out of power"). To be robust to model inaccuracies, this argument asserts that if the robot's better level is always above some small amount of energy (say, $\overline{err}_{\text{total}}$) in the battery, then the inaccuracies of the models will be less than $\overline{err}_{\text{total}}$, and thus the robot will not run out of power. Finding this constant requires estimating the 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:

$$\overline{err}_{\text{total}} = f(\overline{err}_{\text{pow}}, \overline{err}_{\text{mdp}}, \overline{err}_{\text{cons}}),$$

where:

- $\overline{err}_{\text{pow}}$ is the error of energy expense estimation by $\mathcal{M}_{\text{power}}$, related to experimental noise and imperfect fit of the regression function.
- $\overline{err}_{\text{mdp}}$ is the error of approximation in the description of $\mathcal{M}_{\text{plan}}$, as well as floating point operations when finding an optimal policy in the MDP.
- $\overline{err}_{\text{cons}}$ is the error of consistency between $\mathcal{M}_{\text{power}}$ and $\mathcal{M}_{\text{plan}}$, which may arise due to heterogeneity, errors, and optimizations in modeling. Often, this error is ignored and assumed to be 0. However, as the experiments in 8.2 demonstrate, that assumption is not realistic for these models.

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

The challenge of integration here is to verify that the value of $\overline{err}_{\text{cons}}$ is within a certain bound, thus limiting the inconsistency between the two models. Multiple sources of inconsistency between $\mathcal{M}_{\text{power}}$ and $\mathcal{M}_{\text{plan}}$ (i.e., high values of $\overline{err}_{\text{cons}}$) are possible: mismatch in the maps (e.g., the distances between locations not matching $\mathcal{M}_{\text{map}}$), 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 8.2 presents a study of model integration for this system, and Section 8.3 provides more detail about the abstractions used in that study.

3.2 System 2: Collision Avoidance for 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 [31, 173, 190]. 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 [87]) do not achieve flawless practical safety; lately, accidents

continue to happen due to the increasing scale of deployment of such systems [54, 105]. One of the challenges is the 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 of integration abstractions, consider an autonomous wheeled robot moving in a 2D space with other obstacles [193, 196]. 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: local planning and motion control.

The collision safety requirement can be operationalized in three ways (ordered from the most aggressive to the most 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 [173].
- *Absolute safety:* collisions must not occur under any circumstances [31].

A trade-off between these requirements is that stricter notions of safety may lead to unnecessarily conservative control. For example, a robot would not attempt to enter a crowded area to satisfy the absolute safety requirement, but could make progress in small steps under the passive 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 follow? 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? Different answers to these questions have different effects on the robot's decisions and the guarantees that can be obtained from the models. Therefore, an engineer needs to explore a large modeling space when developing collision avoidance systems. Table 3.1 summarizes the high-level concerns that underlie the modeling of robotic collision avoidance.

The dynamics of the collision avoidance protocol was modeled (by different authors and independently of this thesis, as part of the *Robix* study) in a combination of hybrid programs and Differential Dynamic Logic (d$\mathcal{L}$) [214]. This modeling method enables the development of formal proofs for 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.

Concern	Variations
Tactic	Avoiding obstacles, passing intersection, arriving at goal.
Physical space	Unconstrained, constrained box, intersection.
Property type	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.

Due to their unique syntax and semantics, hybrid programs are difficult to relate to other models, especially non-hybrid ones. Without such a relation, it is difficult to guarantee consistency of hybrid programs and the other models, so the formal guarantees of hybrid programs may not transfer to, for instance, the implementations generated from the other models. In this context, as a first step towards consistency checking, the challenge is to design integration abstractions for hybrid programs that support the four qualities of integration (described in Section 1.1) in the following way:

- *Expressiveness:* the abstractions should expose the dimensions of variability between the hybrid programs (see Table 3.1) and allow reasoning about hybrid programs in $\mathsf{d}\mathcal{L}$.
- *Soundness:* the abstractions should preserve the soundness of $\mathsf{d}\mathcal{L}$ reasoning (achieved with a formal mapping between the abstractions and hybrid programs).
- *Customizability:* the abstractions should be tailorable to specific programs, possibly representing their common parts in a reusable way (achieved via a common customizable representation).
- *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 is comprised of model variants from an independent robotic collision case study [194, 195], 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, validation on these models is appropriate for this thesis.

The preliminaries on hybrid programs are presented in Section 6.1. The integration abstractions for hybrid programs are presented in Subsections 6.2.4 and 6.3.2. Section 8.3 discusses a validation study of integration abstractions for hybrid programs.

3.3 System 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 [143].

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 ($Thrds$), which are characterized by periods, deadlines, execution times, and thread security classes ($SecCls$), of which I consider three: $\mathbf{normal}$, $\mathbf{secret}$, and $\mathbf{topsecret}$.

The CPUs of the quadrotor ($CPUs$) are characterized by thread scheduling policies (from set $SchedPols$) and execution frequencies (CPUFreq) that affect power consumption through frequency scaling [123]. Each CPU dynamically executes the threads that were bound to it at design time (with a CPUBind function). Each thread arrives to the execution queue periodically with a fixed period (Per) and has to finish its execution 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 ($\mathbf{rms}$), earliest deadline first ($\mathbf{edf}$) [168], and deadline monotonic scheduling ($\mathbf{dms}$) [12]. 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 ($Batts$) that are mounted on the quadrotor. Each battery consists of a rectangular array of cells that, in combination, maintain a fixed voltage (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* ($ConnSchedPols$) [142, 143], and I consider three policies for this system: 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 ($\mathbf{GPwRR}$).

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 [141]. 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. A relatively large number of thermal neighbors indicates 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:

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

- *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 use minimal frequency, 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* [43].

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 preemption rules encoded in a relation $\mathrm{canprmt}$.
- 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 different 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 [39]. This checking is only applicable when the system uses 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's algorithm that checks whether overheating in one cell would lead to 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 ($\mathbb{AN}$, illustrated in Figure 3.2):

- Bin packing [61]: assigns threads to 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 [39]: 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, if bin packing is executed before secure thread allocation, the assignment

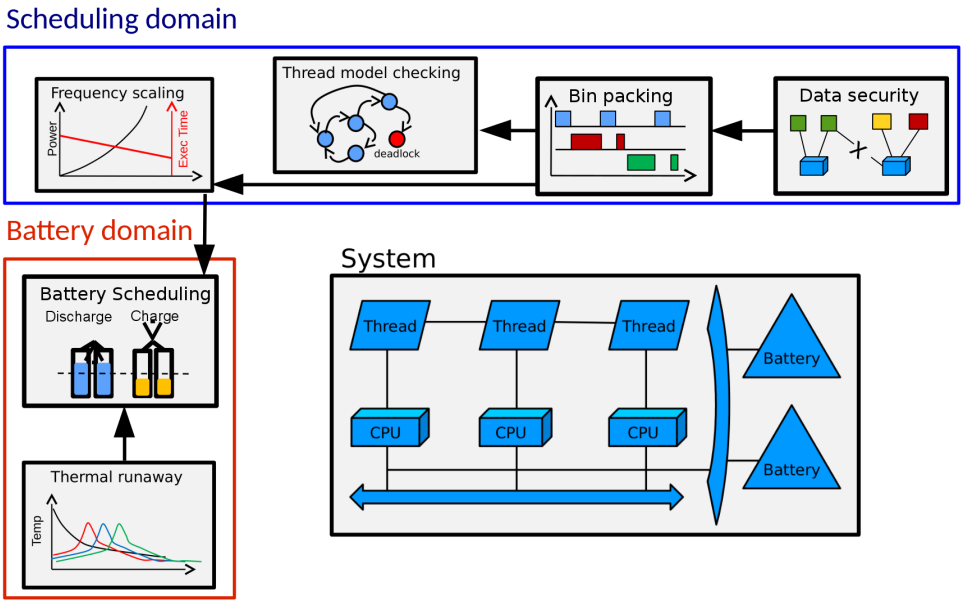


Figure 3.2: The analyses for System 3, and a system design example. Arrows for analyses indicate dependencies. The system consists of components with different types, and lines indicate interactions between components.

of threads to CPUs may violate the secure co-location constraints. Similarly, a system may miss deadlines if the frequency of CPU with the **edf** policy is determined by a frequency scaling algorithm that assumes the **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 8.4 demonstrates systematic integration of these analyses, ensuring satisfaction of their dependencies and assumptions. The assumptions are formalized using IPL in Subsection 8.2.2. An evaluation of the integration abstractions supporting this study can be found in Subsection 8.3.3.

3.4 System 4: Reliable and Secure Sensing for Autonomous Vehicle

This validation system consists of 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 uses information from several sensors that estimate velocity and position relative to the leading car.

The car systems use velocity and distance sensors for braking. There are two distance sensors using different technologies to measure distance: (i) a Lidar for laser ranging, and (ii) a car-to-car

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

Table 3.2: Sensor type, technology and placement.

(C2C) communication network[3] to exchange position information. The Lidar is located inside the car, and the network can 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 [179], 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 component trustworthiness would be complementary and are out of scope of this scenario.

To describe potential security threats, consider three adversary profiles:

- A *powerful adversary* 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 [150]. By forging CAN packets, the attacker can cause various system failures. However, full internal network access is not always a realistic assumption for a moving vehicle.
- An *external adversary* can successfully attack external sensors via man-in-the-middle attacks on physical channels, such as infrared [256] or short-range wireless [40].
- An *internal adversary* has access to part of the car's internal network and can compromise 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. I 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

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

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.

such information from third parties. I assume that the adversary does not have the computational capabilities to break strong cryptographic security measures, e.g., encryption. An adversary can attack sensors in any order, and I do not make any limiting assumptions on the duration of attacks. The adversary profile is recorded in the variable $atkm$.

In this context, consider three analyses:

- *Failure Modes and Effects Analysis* (*FMEA*, A_{fmea}).
- Sensor trustworthiness analysis ($\mathsf{A}_{\text{trust}}$).
- Control safety analysis (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* [62], 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

Sensor	Placement	Powerful Adv.	External Adv.	Internal Adv.
Lidar	Internal	✗	✓	✗
C2C	External	✗	✗	✓
Speedometer	Internal	✗	✓	✗
GPS	External	✗	✗	✓

Table 3.4: Sensor trustworthiness for the three adversary models. ✓ indicates that the sensor is trustworthy. ✗ indicates that the sensor may have been compromised.

model (formulated in terms of possible actions on components). Note that, unlike FMEA, 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 target the existing design-time and run-time analyses [179, 188]. 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 this 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. [75] 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): a reliability model, a trustworthiness model, and a control model. The reliability model ($\mathcal{M}_{\text{fmea}}$) 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 ($\mathcal{M}_{\text{trust}}$) determines whether sensors can be compromised by the selected attacker, and serves as the basis for the sensor trustworthiness analysis. The control model ($\mathcal{M}_{\text{ctrl}}$) 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,

Variable	Sensor	Trustworthiness	Available in mode			
			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.

secure, and safe design. Potentially incorrect interactions between analyses present a challenge: unsatisfied assumptions of the analyses can lead to vulnerabilities, which can be exploited by an adversary. In this scenario, the 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, due the lack of mutual knowledge between A_{fmea} and A_{trust}. The first way may occur at design-time, when the most error-prone sensors are also the ones that are not trustworthy. FMEA may try to replicate untrustworthy sensors to increase reliability, thus decreasing the ratio 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 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 A_{ctrl} doesn't hold in all likely failure modes.

Table 3.5 illustrates an external adversary using the unsatisfied assumption about failure modes to cause system failures in two out of four modes. In the nominal mode, both distance and velocity sensors have the trustworthiness ratio of 50%. In fail mode 1, distance sensing is compromised because the only distance sensor C2C is untrustworthy. Fail mode 2 has the required ratio of trustworthy sensors. Fail mode 3 violates the assumption because the only available velocity sensor (GPS) is compromised. Thus, an external attacker may be harmless in the nominal mode, but is still capable of exploiting vulnerabilities in 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 8.4.2, while the integration abstractions for it are described in Subsection 8.3.4.

Chapter Summary

This chapter introduced four contexts that will be used for illustration and validation of this thesis. The contexts cover multiple dimensions of CPS engineering: different application domains, quality attributes, modeling methods, and integration issues. The domains involve robotics, automotive, and aerospace. The quality attributes include security, reliability, real-time schedulability, and various aspects of safety. The modeling methods range from deterministic discrete security models to hybrid programs for robot motion, to probabilistic automata of energy-aware trajectory planning. The integration issues are rooted in mismatches between structural and behavioral models, and complex interactions of diverse analyses.

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, a set of models with associated analyses), 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 the 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 logical assertion over several models. Integration properties express conditions of model consistency or matching analysis contexts. Unlike most existing integration approaches, these properties target specific integration scenarios, allowing engineers to tailor the formal definition of consistency to their needs.

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

1. *Create integration abstractions:* construct intermediate representations (views and behavioral properties) that provide the basis for multi-model integration properties. Integration abstractions are shown as rectangles above the models in Figure 4.1.
2. *Specify and verify integration properties:* express the intended relationships among the models and check if the models indeed relate in the specified ways. Integration properties are shown as a star between the integration abstractions in Figure 4.1.
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. The execution is facilitated by *analysis contracts*, shown as parallelograms above the analyses in Figure 4.1.

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 *integration abstractions*:

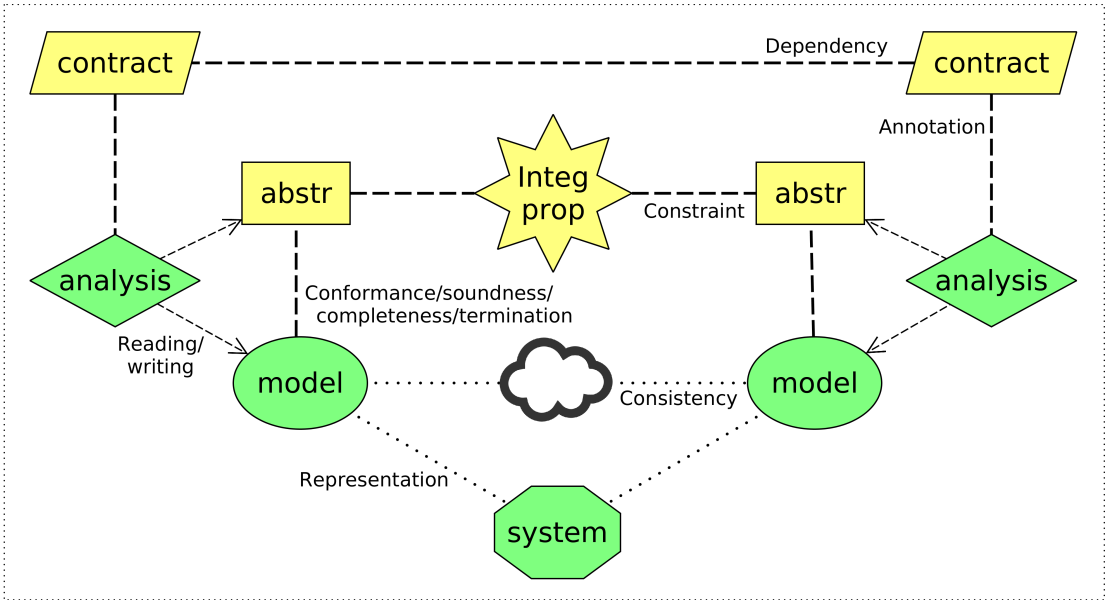


Figure 4.1: The proposed approach to integration of CPS modeling methods. Green elements represent the existing artifacts. Yellow elements represent the contributions of the approach.

- *Views* — hierarchical component models annotated with properties. These models are inspired by architectural descriptions [51].
- *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 [236]. In my approach, views can be instantiated in two architecture description languages (ADLs) — Acme [98] and AADL [80] — 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 [25]. 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 (or, query) a model to interpret an expression, which evaluates to a concrete value only in the context of the model. As a response to the query, the model returns this value of the expression. For example, a model of a mobile robot can be used to evaluate an assertion that the robot eventually reaches the goal, and the model would return a boolean value. In my approach, behavioral properties are specified in languages that are rooted in two logics: linear temporal logic (LTL) [218], and probabilistic computation tree logic (PCTL) [112]. Aside from these two languages 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 the integration property depends on few discrete elements of the model. In contrast, behavioral properties are used to (indirectly) access the model's 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 to be used for integration, views are required to *sound* (i.e., every view element is based on some model elements) and *complete* (i.e., every relevant[1] model element is accounted for by view elements). Behavioral properties are computable functions of model elements, so they are required to be *sound* (i.e., the returned value is correct with respect to the semantics of the behavioral language and the model) and to *terminate* (i.e., the computation of the query ends within a finite time).

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 necessary for successful integration as described at the end of Section 2.2): model consistency (i.e., models are consistent if their integration properties hold) and appropriateness of analysis contexts (i.e., a context is appropriate if its integration properties hold). 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 a *Satisfiability Modulo Theories (SMT* [204]*)* 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 using 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

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

result should be correct with respect to the semantics of IPL). IPL verification is not necessarily *complete:* IPL is not guaranteed to return an answer for every expression within its syntax. Completeness is limited by the use of first-order logic (quantified variables and uninterpreted functions) in IPL, as discussed in Chapter 5.

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

4.3 Integration of Analyses

Each analysis interacts with one or several models by reading and changing their elements (e.g., setting an optimal set of control gains), as well as producing other, non-model artifacts (e.g., generating source code). The third and 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 four cases (which are 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 needs 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 outputs are compatible with the analyses that consume them. A guarantee can be use used to constrain the analysis that is expected to alter the contexts of other analyses in an inappropriate way.
4. 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 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, facilitated by the Analysis Execution Platform (AEP).

4.4 Integration Argument for Models

This section provides a template of an argument for integration of models according to the described approach. This argument identifies the obligations of integration abstractions and IPL necessary to show that an integration property holds. This argument addresses the first (consistency of models) and the third (context checking for analyses) requirements from Section 2.2. The second requirement (data dependencies for analyses) is not related to integration of models, and is addressed by analysis execution in Chapter 7.

Many concepts and symbols used in this section are defined in later chapters. Here I explain the intuitive meaning behind these concepts and provide forward references to their definitions and explanations.

Consider two models:

- A structural model $\mathcal{M}^s$, which contains a finite set of *model elements* $\mathbb{E}^{\mathcal{M}}$. These elements are abstract entities with multiple attributes (name-value pairs). For more details, see Definition 3 in Section 5.2.
- A behavioral model $\mathcal{M}^b$, which contains a *parameterized behavioral description* $\mathbb{\Omega}$, which, given concrete values of its parameters, determines a potentially infinite *set of behaviors* Ω. Behaviors are a special kind of model elements, which are accessed indirectly through *behavioral properties* l in a *behavioral language* $\mathbb{L}$. This language is a collection of formulas (each denoted as l), which can be used to write property expressions about $\mathcal{M}^b$. These properties are evaluated on $\mathcal{M}^b$, yielding a value of a standard type (boolean, integer, or real). For more details, see Definitions 4 and 5 in Section 5.2 and Definition 25 in Subsection 6.3.1.

The integration property between these two models is represented as a ground-truth predicate over the model elements and behaviors:

$$\mathrm{integprop}(\mathbb{E}^{\mathcal{M}}, \mathbb{\Omega}).$$

This predicate is satisfied if and only if models $\mathcal{M}^s$ and $\mathcal{M}^b$ are consistent (in terms of the engineer's intent). This predicate is not directly checkable. My approach aims to check $\mathrm{integprop}$

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

using views, behavioral properties, and IPL specifications. In the coming paragraphs, I identify the responsibilities of views for $\mathcal{M}^s$, behavioral properties for $\mathcal{M}^b$, and how IPL connects them to integprop. A summary of the integration argument is shown in Figure 4.2, with the relevant entities and their qualities.

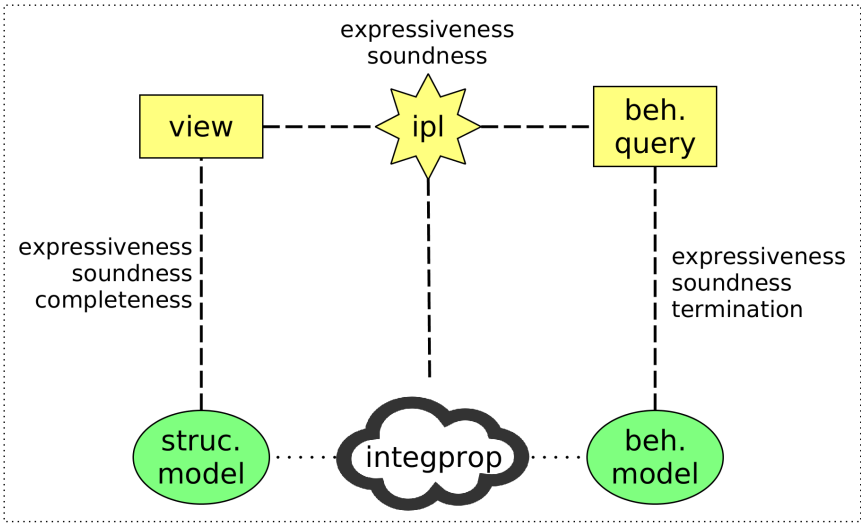


Figure 4.2: Parts of the integration argument and their key properties.

The integration argument proceeds in four steps:

1. Ω is replaced with one or several behavioral properties in integprop.
2. $\mathbb{E}^{\mathcal{M}}$ contained in $\mathcal{M}^s$ are represented using view elements $\mathbb{E}^{\mathcal{V}}$, which can be checked using IPL formulas.
3. An IPL formula is written over view elements and behavioral properties. The formula, together with certain model-view constraints, implies the original predicate integprop.
4. The IPL formula is checked, and its satisfaction implies the satisfaction of the original predicate.

To represent Ω in $\mathcal{M}^s$, I use some behavioral property language $\mathbb{L}$ that is compatible with $\mathcal{M}^s$ (see Definition 25 in Subsection 6.3.1). Given a set of traces $\Omega \in \text{range}(\Omega)$ and values μ of free variables used in the property, the language provides its *semantic interpretation* $[\![\cdot]\!]^{\mu}_{\Omega}$ that gives meaning to any $l \in \mathbb{L}$, mapping it to a boolean, real, or integer value.

$\mathbb{L}$ is required to be expressive enough to represent the behavior constraints of integprop on Ω using a finite number (say, m) of behavioral properties $l_1 \dots l_m$ (see Definition 28 in Subsection 6.3.3). These properties should be evaluated over some given variable values $\mu_1 \dots \mu_m$ and trace sets $\Omega_1 \dots \Omega_m$, which can be produced by providing parameter values to Ω of model $\mathcal{M}^b$ (see Definition 4 in Section 5.2). In integprop, these parameter values are determined by $\mathbb{E}^{\mathcal{M}}$.

Thus, due to the assumed expressiveness of $\mathbb{L}$, there exists another predicate integprop$'$ over $l_1 \dots l_m$ that is logically equivalent to integprop and uses only these behavioral properties to interact with Ω and $\mathcal{M}^b$:

$$\text{integprop}(\mathbb{E}^{\mathcal{M}}, \Omega) \iff \text{integprop}'(\mathbb{E}^{\mathcal{M}}, [\![l_1]\!]^{\mu_1}_{\Omega_1} \dots [\![l_m]\!]^{\mu_m}_{\Omega_m}). \tag{4.1}$$

The second step is to represent $\mathbb{E}^{\mathcal{M}}$ contained in $\mathcal{M}^s$ with a view $\mathcal{V}$, which contains a finite set of view elements $\mathbb{E}^{\mathcal{V}}$ (see Definitions 1 and 2 in Section 5.2). View elements have multiple name-value attributes (called element properties) similarly to $\mathbb{E}^{\mathcal{M}}$, but unlike model elements, they are created in a unified format that can be read by IPL.

The view is constructed in a way that imposes a certain pre-defined relationship between model elements and view elements. This relationship is expressed using a matching predicate mp (see Definition 13 in Subsection 6.2.2) over fixed-dimensional tuples of view elements $\boldsymbol{e}^{\mathcal{V}} : (\mathbb{E}^{\mathcal{V}})^k$ and model elements $\boldsymbol{e}^{\mathcal{M}} : (\mathbb{E}^{\mathcal{M}})^l$, for some natural numbers k and l. The matching predicate is selected to imply $\mathrm{integprop}$ when conjoined with an IPL formula, as per the third step below.

The relationship between tuples $\boldsymbol{e}^{\mathcal{V}}$ and $\boldsymbol{e}^{\mathcal{M}}$ is constrained by requiring that the view is sound and complete with respect to mp (only one of these restrictions may be needed, depending on the quantifiers in $\mathrm{integprop}$, as further detailed in Subsection 6.2.6 and Section 8.1). Soundness intuitively means that every view tuple $\boldsymbol{e}^{\mathcal{V}}$ satisfies mp with some model tuple $\boldsymbol{e}^{\mathcal{M}}$. Completeness means that every model tuple $\boldsymbol{e}^{\mathcal{M}}$ satisfies mp with some view tuple $\boldsymbol{e}^{\mathcal{V}}$. These notions are formalized in Definitions 17 and 18 in Subsection 6.2.3.

The third step is writing an IPL formula (denoted here as a predicate ipl) over view elements and behavioral properties. The goal is to find such a formula that, together with mp, implies $\mathrm{integprop}'$. It is assumed that the view language and IPL are expressive enough to create such an IPL formula and a view. Thus, the IPL formula and mp are chosen so that the following implication holds:

$$\mathrm{ipl}(\mathbb{E}^{\mathcal{V}}, [\![l_1]\!]^{\mu_1}_{\Omega_1} \ldots [\![l_m]\!]^{\mu_m}_{\Omega_m}) \wedge \mathrm{mp}(\mathbb{E}^{\mathcal{V}}, \mathbb{E}^{\mathcal{M}}) \rightarrow \mathrm{integprop}'(\mathbb{E}^{\mathcal{M}}, [\![l_1]\!]^{\mu_1}_{\Omega_1} \ldots [\![l_m]\!]^{\mu_m}_{\Omega_m}).$$

Notice that the behavioral properties are, implicitly, arguments of $\mathbb{E}^{\mathcal{M}}$ in $\mathrm{integprop}'$ and $\mathbb{E}^{\mathcal{V}}$ in ipl: the parameter values that determine $\Omega_1 \ldots \Omega_m$ come from $\mathbb{E}^{\mathcal{M}}$ in the $\mathrm{integprop}'$ predicate, and from $\mathbb{E}^{\mathcal{V}}$ in the $\mathrm{integprop}$ predicate. There exists a way to infer $\mathrm{integprop}'$ based on ipl and mp, and its exact preconditions are stated in Theorem 2 in Subsection 6.2.6).

Using the equivalence Equation (4.1), the above implication can be rewritten as follows:

$$\mathrm{ipl}(\mathbb{E}^{\mathcal{V}}, [\![l_1]\!]^{\mu_1}_{\Omega_1} \ldots [\![l_m]\!]^{\mu_m}_{\Omega_m}) \wedge \mathrm{mp}(\mathbb{E}^{\mathcal{V}}, \mathbb{E}^{\mathcal{M}}) \rightarrow \mathrm{integprop}(\mathbb{E}^{\mathcal{M}}, \mathbb{\Omega}). \tag{4.2}$$

The fourth and final step is checking whether $\mathrm{ipl}(\mathbb{E}^{\mathcal{V}}, l_1 \ldots l_m)$ holds for $\mathcal{M}^s$ and the constructed $\mathcal{V}$. The IPL verification algorithm (Section 5.5) performs this check. The value of formulas in $\mathbb{L}$ is computed through a query interface Q, which returns the value of a sentence l given a trace set Ω. These queries are required to be sound (i.e., return only values according to the semantics of $\mathbb{L}$) and terminate within a finite time. See Definitions 29 and 30 in Subsection 6.3.3. The soundness condition is be written as follows:

$$\mathsf{Q}(l, \Omega, \mu) = [\![l([\![\mathrm{RATOM}_1]\!]_\mu \ldots [\![\mathrm{RATOM}_k]\!]_\mu)]\!]^{\mu}_{\Omega}, \tag{4.3}$$

where $\mathrm{RATOM}_1 \ldots \mathrm{RATOM}_k$ are the rigid atoms that are subformulas of l, and μ is a valuation of free variables over which l was written.

Thus, IPL verification relies on sound queries and checks the satisfaction of ipl soundly (which is detailed in Section 5.6). In Equation (4.2), the conjunction over mp holds by construction of views, so if ipl holds, then $\mathrm{integprop}(\mathbb{E}^{\mathcal{M}}, \mathbb{\Omega})$ holds by implication.

The above argument extends to multiple structural and behavioral models. Structural models contain their respective model element sets $\mathbb{E}_1^{\mathcal{M}} \ldots \mathbb{E}_p^{\mathcal{M}}$, which are abstracted by a collection of views $\mathcal{V}_1 \ldots \mathcal{V}_p$. IPL places no restrictions on using elements from multiple views. Behavioral properties can be queried on different parameterized models without changes to the above argument. As long as the behavioral languages are expressive enough to represent all behavioral constraints in $\mathrm{integprop}$ through behavioral properties, it is possible to replace $\mathrm{integprop}$ with $\mathrm{integprop}'$, as done in Equation (4.1). Thus, the integration argument applies when $\mathrm{integprop}$ is a predicate over multiple structural and behavioral models.

The assumptions of the above argument are detailed in the following sections:

- Checking the satisfaction of IPL formulas is performed by the algorithm described in Section 5.5 and proved sound in Section 5.6.
- Soundness, completeness, and expressiveness of views, which connect assertions about $\mathbb{E}^{\mathcal{M}}$ and $\mathbb{E}^{\mathcal{V}}$, are discussed in Subsections 6.2.3 and 6.2.6.
- Soundness, termination, and expressiveness of behavioral queries, which let IPL constrain behaviors using behavioral properties, are discussed in Subsection 6.3.2.

A more detailed formalization of this argument can be found in Section 8.1, which relies on the details of each assumption above.

4.5 Integration Arguments for Analyses

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

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

An engineer is required to specify the assumptions (a) and guarantees (g) of an analysis, which are recorded as IPL statements over the models that the analysis reads and writes (see Definition 34 in Section 7.2). These statements are part of a contract ($\mathcal{C}$) for A — a collection of specifications that are enforced when A is executed. The assumptions are checked before A is executed, and guarantees — after.

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:

$$\big[\mathbb{M} \models \mathsf{a}\big] \wedge \big[\mathbb{M}' \models \mathsf{g}\big]. \tag{4.4}$$

Below, I describe four application cases for analysis contracts:

1. Checking appropriate context.
2. Preserving model consistency.
3. Guaranteeing assumptions.
4. Checking implementation.

In the first case (checking appropriate context), the assumptions specify the (previously informal) conditions of an analysis matching its intended context. The guarantees specify the expectations from the post factum context. Thus, the checking is performed according to Equation (4.4). If this specification is complete (i.e., covers all the expected conditions), it is guaranteed that the analysis will be run in its expected context. 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 be 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:

$$\big[\mathsf{a} \equiv \mathrm{integprop}(\mathbb{M})\big] \wedge \big[\mathsf{g} \equiv \mathrm{integprop}(\mathbb{M}')\big].$$

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 assumptions and guarantees to complement the pre-analysis consistency and imply the post-analysis consistency:

$$\mathrm{integprop}(\mathbb{M}) \wedge \big[\mathbb{M} \models \mathsf{a}\big] \wedge \big[\mathbb{M}' \models \mathsf{g}\big] \rightarrow \mathrm{integprop}(\mathbb{M}').$$

In the third case (guaranteeing assumptions), a guarantee of one analysis combines the typical assumptions ($\mathsf{a}_1 \ldots \mathsf{a}_n$) of several other analyses. This way, the guarantee can be checked once and right after the undesired changes may have been introduced, instead of delaying the checking and distributing it to multiple other contracts.

$$\big[\mathsf{g} \equiv \mathsf{a}_1 \wedge \cdots \wedge \mathsf{a}_n\big] \wedge \big[\mathbb{M}' \models \mathsf{g}\big].$$

For the fourth case (checking implementation), the contract is written based on the internal business logic of the analysis (i.e., the contract is 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):

$$\big[\mathbb{M} \models \mathsf{a}\big] \rightarrow \big[\mathbb{M}' \models \mathsf{g}\big].$$

The four above cases are not mutually exclusive: all of them 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 trade-off of this freedom is that analysis contracts require models to be evaluated and, hence, do not permit abstract reasoning about analyses separately from models.

Chapter Summary

This chapter described the three parts of the approach: the Integration Property Language, two integration abstractions (views and behavioral properties), and the Analysis Execution Platform. The following three chapters are devoted to each part of the approach. Further, this chapter introduced a formal argument for integration. Integration of models relies on checking elements of the models by their respective abstractions and soundly checking properties over the abstractions. Integration of analyses relies on contracts for each analysis, which can be defined in multiple ways depending on the integration scenario.

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). 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. The creation of these abstractions is covered in Part II (Chapter 6). In addition to integration of models, 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 and preliminaries. Afterwards, I describe the syntax of IPL in Section 5.3, its semantics in Section 5.4, and the verification algorithm in Section 5.5. Validation studies of IPL can be found in Section 8.2.

5.1 Motivating Integration Property

In the context of an energy-aware mobile robot, consider a potential inconsistency between $\mathcal{M}_{\text{power}}$ and $\mathcal{M}_{\text{plan}}$ that threatens the soundness of the power safety argument described in Section 3.1. These models may have inconsistent energy estimates for the robot's turning actions due to their difference in representing these turns: $\mathcal{M}_{\text{plan}}$ models turns implicitly, combining them with forward motions into single actions to reduce the state space and planning time. In $\mathcal{M}_{\text{power}}$ however, turns are explicit tasks, separate from forward motion. The potential inconsistency between turning energies between $\mathcal{M}_{\text{power}}$ and $\mathcal{M}_{\text{plan}}$ 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 $\overline{err}_{\text{cons}}$.[1]

This integration property is difficult to verify for two reasons. First, the abstractions are different: $\mathcal{M}_{\text{plan}}$ describes states and transitions (with turns embedded in them), whereas $\mathcal{M}_{\text{power}}$ describes a stateless relation between energy and time. Second, there is no single means to express such integration properties formally: PCTL (Probabilistic Computation Tree Logic [154]) is a

[1] As detailed later, I 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).

property language for $\mathcal{M}_{\text{plan}}$, but $\mathcal{M}_{\text{power}}$ does not come with a reasoning engine. Finally, even if the aforementioned obstacles are overcome, the models are often developed by different teams, so these models need to stay separate and co-evolve.

To verify this property, I take the approach described in Chapter 4. PCTL-based behavioral properties of $\mathcal{M}_{\text{plan}}$ will be used to reason about the probabilistic and temporal aspects of $\mathcal{M}_{\text{plan}}$ traces, such as checking whether, given a mission and an initial state, the robot will eventually reach its goal according to $\mathcal{M}_{\text{plan}}$. A view $\mathcal{V}_{\text{power}}$ will be used for reasoning about the static and stateless elements of $\mathcal{M}_{\text{power}}$, such as the energy required for a motion task. $\mathcal{V}_{\text{power}}$ acts as a *task library*, containing all atomic tasks (going straight and rotating, for 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 $\mathcal{V}_{\text{power}}$ 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 $\mathcal{M}_{\text{power}}$ and $\mathcal{M}_{\text{plan}}$)**.** For any three tasks from $\mathcal{M}_{\text{power}}$ that form sequence $\langle$go straight, rotate, go straight$\rangle$, do not self-intersect, and have sufficient energy, any execution in $\mathcal{M}_{\text{plan}}$ that goes through that sequence in the same order, if initialized appropriately, does not lead to the robot running out of power (allowing for error $\overline{err}_{\text{cons}}$ in battery charge).

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

5.2 IPL Concepts and Preliminaries

In this section, after providing a description of general design principles behind IPL, I formalize the central concepts that IPL is based on: views, models, and model property languages.

5.2.1 IPL Design

For applicability to real-world model integration, 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 customizable 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 re-engineering 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.* IPL combines reasoning over static aspects in first-order logic with "deep dives" into behavioral models to retrieve only the necessary values. Tractability is preserved 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, I create 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 formal notation of the model's properties for verification. This way, IPL does not make extra assumptions on models beyond the plugin points, hence enhancing modularity.

5.2.2 Views and Behavioral Properties

This section provides formal definitions for the abstractions that underlie IPL. For more detail and rationale on these abstractions, see Chapter 6. I start with views, which enable integration of static elements of models.

Definition 1 (Architectural View)**.** An *architectural view* $\mathcal{V}$ is a 4-tuple ($\mathbb{E}^{\mathcal{V}}$, $\mathbb{T}$, T, $\mathbb{P}$):

- *View elements* ($\mathbb{E}^{\mathcal{V}}$) are abstract entities[2] characterized by the functions below. I consider only finite sets of view elements.
- *Element types* ($\mathbb{T}$) are labels of architectural elements that determine some properties and semantics of these elements. I consider only finite sets of element types.
- *Typing function* Each element can have multiple types, determined by the function $\mathsf{T} : \mathbb{E}^{\mathcal{V}} \rightarrow \mathcal{P}(\mathbb{T})$, where $\mathcal{P}()$ means a power set.

[2]Typical architectural elements include components, connectors, and ports. My approach considers a single set of elements per view, differentiating them through types.

- *Element properties* ($\mathbb{P}$) are functions from architectural elements to arbitrary domains of values. Each property is represented with a function $\mathsf{P} : \mathbb{E}^{\mathcal{V}} \rightarrow \mathbb{O}$. I consider a finite number of properties in each view.

An element type can be associated with some properties: the elements of that type are characterized by these properties. For example, components of type CPU have an integer-valued property "clock frequency." This property maps components of that type to integer values. In general, an element type T is associated with an element property P when $\mathbf{dom}\ \mathsf{P} = \{e \mid e \in \mathbb{E}^{\mathcal{V}} \wedge \mathsf{T}(e) = T\}$.

IPL uses views for modeling static, behavior-free projections of models. For example, a 2D map encoded in an XML ($\mathcal{M}_{\text{map}}$) is a model from which locations can be exposed in a view ($\mathcal{V}_{\text{map}}$) as a set of interconnected components ($\overline{Locs}$). Each component is a location, and connectors indicate direct reachability between them. I 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* $\mathcal{V}$ is a triplet ($\Sigma^{\mathcal{V}}$, $\Gamma^{\mathcal{V}}$, $I^{\mathcal{V}}$):

- A *view signature* ($\Sigma^{\mathcal{V}}$) is a tuple of three finite sets of symbols: element types ($\mathbb{T}$), element properties ($\mathbb{P}$), and view functions (VF). Note that view signatures do not contain any interpretation of these symbols.
- A *view structure* ($\Gamma^{\mathcal{V}}$) is a tuple consisting of (i) a finite set of architectural elements ($\mathbb{E}^{\mathcal{V}}$) in this view, (ii) a set of arbitrary value domains (infinite sets, each of which is denoted as $\mathbb{O}$), and (iii) a potentially infinite set of functions that map $\mathbb{E}^{\mathcal{V}}$ to various $\mathbb{O}$. View structures provide entities on which the meaning of symbols from signatures is defined.
- A *view interpretation* ($I^{\mathcal{V}}$) is a function from the symbols in $\Sigma^{\mathcal{V}}$ to the entities in $\Gamma^{\mathcal{V}}$, $I^{\mathcal{V}} : \Sigma^{\mathcal{V}} \rightarrow \Gamma^{\mathcal{V}}$. Each type is mapped to a subset of $\mathbb{E}^{\mathcal{V}}$, and each property is mapped to a function from $\Gamma^{\mathcal{V}}$. Note that view interpretations give static meaning (independent of state or time) to the symbols in view signatures.

I use formal views to define the syntax and semantics of IPL. For example, suppose the signature of a view contains a symbol for the component type of batteries ($Batts$) and a property that defines a battery's maximum charge (the number of watt-hours of charge that the battery can hold at its fully charged state), which is modeled as a function $\overline{maxbat} : Batts \mapsto \mathbb{Z}$, where $\mathbb{Z}$ is the set of integers. A view structure contains $\mathbb{E}^{\mathcal{V}}$ with two batteries and two CPUs: $\mathbb{E}^{\mathcal{V}} = \{b_1, b_2, c_1, c_2\}$. Then, $I^{\mathcal{V}}$ would map the symbol $Batts$ to the two batteries: $I^{\mathcal{V}}(Batts) = \{b_1, b_2\}$, and the symbol $\overline{maxbat}$ to a function that returns the maximum charge for each of these two batteries, e.g., $I^{\mathcal{V}}(\overline{maxbat}) = \{b_1 \rightarrow 20, b_2 \rightarrow 30\}$.

An isomorphic relationship between the two definitions of views is established by converting architectural models to SMT specifications, as done in my prior work [235, 240]. In short, that relationship is established as follows. The elements from the architectural view are added to $\mathbb{E}^{\mathcal{V}}$ in the structure $\Gamma^{\mathcal{V}}$ of the formal view. For each property, its range is added to $\Gamma^{\mathcal{V}}$, and all possible functions between the range and the domain are also added to $\Gamma^{\mathcal{V}}$. The view signature $\Sigma^{\mathcal{V}}$ is constructed by adding all types and names of properties to it as labels. The view interpretation is constructed according to the meaning of $\mathbb{T}$ and $\mathbb{P}$ in the architectural view.

These two definitions of views 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.[3]

Now I define two kinds of models: structural and behavioral. The former contain static structures, and the latter contain dynamic behaviors. The ultimate goal is to check that the structures in some models are appropriately related to behaviors in other models, as per the consistency predicate integprop (introduced in Section 4.4).

Definition 3 (Structural Model)**.** A *structural model* $\mathcal{M}$ is a finite set of model elements $\mathbb{E}^{\mathcal{M}}$, which are abstract entities that partition the model. Model elements are multidimensional, having multiple potential attributes, which can be represented as functions.

A model element is an abstract part of a model. Examples include a location in a map, a line of code, or a block in a signal-flow diagram. Thus, a map with a set of locations is a structural model. Model elements can often be distinguished by their syntactic boundaries, although they are not necessarily bound to the syntax.

Depending on the integration scenario, it is up to engineers to define model elements that are related to other models and may be potentially contradictory. I do not make up-front assumptions about model elements, except their finiteness. Therefore, practically any model can be seen as a structural model.

Structural models are integrated through views: a view's elements, types, and properties represent the model elements. Sometimes there is no one-to-one mapping between view elements and model elements. For instance, if a view represents actions of a robot that are possible in a map model, then these tasks are associated with pairs of locations. Further details on how views relate to structural models can be found in Section 6.2.

Definition 4 (Behavioral Model)**.** A *behavioral model* $\mathcal{M}$ is a triplet ($\Sigma^{\mathcal{M}}$, $\Gamma^{\mathcal{M}}$, $I^{\mathcal{M}}$):

- A *model signature* ($\Sigma^{\mathcal{M}}$) is a finite set of labels that represent state variables (S), modal functions (MF), and model parameters (PN). A model signature does not contain any interpretations of these symbols.
- A *model structure* ($\Gamma^{\mathcal{M}}$) is a tuple of the following:
  - A *set of trace sets* $\Qoppa$ (each trace set is denoted as Ω), where a trace (ω) is a potentially infinite sequence of states (a state is denoted as q). Behavioral models are parametric, and each set of parameter values leads to a different set of traces. A trace set is selected from $\Qoppa$ by model interpretations given parameter values, as described below. States are abstract entities that determine the values of S and MF, as described below in the model interpretation. The trace sets and $\Qoppa$ are potentially infinite. Thus, $\Qoppa$ parameterizes selection of a trace set Ω, indirectly defines a set of states. The details of the specific trace formalism may differ [13, 110], and the approach is customizable to different trace formalisms. It is assumed that Ω contains sufficient information to

[3]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 seem 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.

interpret S and MF, as described below.

- A finite *set of arbitrary value domains* ($\mathbb{O}$), similar to $\Gamma^{\mathcal{V}}$ in Definition 2. These domains contain values for state variables, modal functions, and model parameters.
- A potentially infinite *set of parameter name-value functions* PF, which map parameter names PN to some values $\mathbb{O}$. Each function in PF represents a set of values for model parameters. This set of parameter values, in turn, determines a concrete set of traces in $\mathbb{\Omega}$.
- A potentially infinite *set of modal functions* MF over arbitrary domains, mapping $\mathbb{O}$ to $\mathbb{O}'$. Each symbol in MF is mapped to MF by a state q, hence $q : MF \rightarrow \mathbb{O}$.

- A *model interpretation* ($I^{\mathcal{M}}$) is a combination of three functions: (i) a mapping from PF to $\mathbb{\Omega}$, determining how values of parameters map to trace sets, (ii) a mapping from S and state q to values $\mathbb{O}$, determining the value of each state variable in each state, and (iii) a mapping from MF and state q to MF, determining the interpretation of each modal function in each state. In the following text, all three mappings are referred to as $I^{\mathcal{M}}$, and one of them is picked depending on the argument. Note that this interpretation determines the values depending on a state, and is therefore called *modal*.

An example of a behavioral model is $\mathcal{M}_{\text{plan}}$. Its signature contains the current battery charge ($\underline{bat}$) as a state variable that changes dynamically. That is, in a particular state, $\underline{bat}$ is mapped to a concrete integer value. Model parameters include the initial charge of the battery (also an integer). Given a charge value, $I^{\mathcal{M}}$ determines a set of traces, which are generated by all possible permutations of the robot's actions. In the case of $\mathcal{M}_{\text{plan}}$, the traces are given by a Markov chain. For a state in the chain, $I^{\mathcal{M}}$ maps the $\underline{bat}$ symbol to an integer value (battery charge in that state).

There is a similarity between structural and behavioral models: the trace sets in $\mathbb{\Omega}$, as well as individual traces ω, can be considered a special, potentially infinite kind of model elements. This thesis makes more detailed assumptions about behavioral models, in order to take advantage of specialized languages for model properties, as discussed below.

Behavioral models are analyzed with the help of property languages, which express properties of these models. These properties are augmented with free variables to enable integration with views via IPL statements, as demonstrated in the next section. Note that property languages differ from model languages in which models are specified.

Definition 5 (Behavioral Property Language)**.** For a model signature $\Sigma^{\mathcal{M}}$ and a set of free variables V, a *behavioral property language* is a set of sentences over the symbols in $\Sigma^{\mathcal{M}}$, V, expressions[4] over V, and language-specific operators[5] that can be evaluated, given an assignment of values to V and a model $\mathcal{M}$ for $\Gamma^{\mathcal{M}}$.

For $\mathcal{M}_{\text{plan}}$, the property language is the input language of the PRISM model checker [154]. Based on PCTL, this language is used for expressing constraints and querying probabilities over executions of $\mathcal{M}_{\text{plan}}$. 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

[4]These expressions are free of model symbols and have evaluations that are determined by variable values, such as addition of two variables. These expressions are defined later in the IPL syntax as rigid atoms (RATOM).

[5]These operators can include standard logical operators and any other operators and modalities that can be given meaning for a model with $\Sigma^{\mathcal{M}}$.

with symbols from $\Sigma^{\mathcal{M}}$ (e.g., $\underline{bat}$ is a state variable signifying the battery charge in that state).

Views and models can share commonly used symbols, which may include, for instance, constants (e.g., integers, reals, or boolean $\top$ and $\bot$), operations over them (e.g., addition), and predicates (e.g., equality). These symbols are described by the background theories (e.g., the theory of equality or linear real arithmetic), provided in a background signature Σ^B. Views and models agree on the interpretation of these symbols, which is provided by a shared background interpretation I^B. Formally, IPL allows only theories that are decidable [152] and form decidable combinations [203], but in practice it is acceptable to use undecidable combinations for which available heuristics resolve relevant IPL statements.

For the rest of this chapter, I consider a given set of behavioral models ($\mathbb{M}$) and a set of views ($\mathbb{V}$). IPL formulas are written in a context of views and models. As explained in Chapter 6, each view is 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 over a signature (Σ) and interpreted over a structure (Γ) using an interpretation, as per the definition below.

Definition 6 (IPL Signature, Structure, and Interpretation)**.** Given a set of models ($\mathbb{M}$), views ($\mathbb{V}$), and background theories, the *IPL signature* Σ is a union of symbols in the signatures $\Sigma^{\mathcal{V}}$, $\Sigma^{\mathcal{M}}$, and Σ^B. The *IPL structure* is a combination of $\Gamma^{\mathcal{V}}$, $\Gamma^{\mathcal{M}}$, and Γ^B. The IPL interpretation Imaps symbols in Σ to Γ using $I^{\mathcal{V}}$, $I^{\mathcal{M}}$, and I^B, as defined by the IPL semantics in Section 5.4.

Before proceeding, I reiterate four important assumptions that underlie the above definitions:

1. For verification purposes, the signatures and structures of views are fully determined, and are up-to-date with the models that these views represent.
2. Views can be translated into finite SMT specifications, as assertions about architectural elements, their types, and properties.
3. Once provided its instantiation parameters, any model can check/query any statement in its property language
4. Models and views share and agree on the background interpretation I^B.

All of these assumptions are satisfied in all the systems and models studied in this thesis (described in Chapter 3), and the implications of their non-satisfaction are discussed in Chapter 10.

5.3 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. To this end, I introduce the rigid/flexible separation: *flexible* terms (denoted with underlines, like $\underline{loc}$) are interpreted by $I^{\mathcal{M}}$, and *rigid* terms (denoted with overlines, like $\overline{Tasks}$) are interpreted by $I^{\mathcal{V}}$. 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 formula is verified (as described below in Section 5.5), its native parts are evaluated in a general way, 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, while $\underline{Q}(\overline{R}(x))$ is passed to a behavioral model for checking.

The native and plugin parts of the syntax are combined according to Figure 5.1. I define each syntax element (box) on top of symbols in the IPL signature (Σ) and a set of variables under quantifiers (V). I build two types of subformulas: rigid atomic formulas (RATOM) from rigid terms (RTERM), and flexible atomic formulas (MATOM). The design 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 are deconstructed into simpler ones that are evaluated by either models or views, but not both at once (which would require giving views specific behavioral semantics). The production rules for rigid atoms and terms are given below.

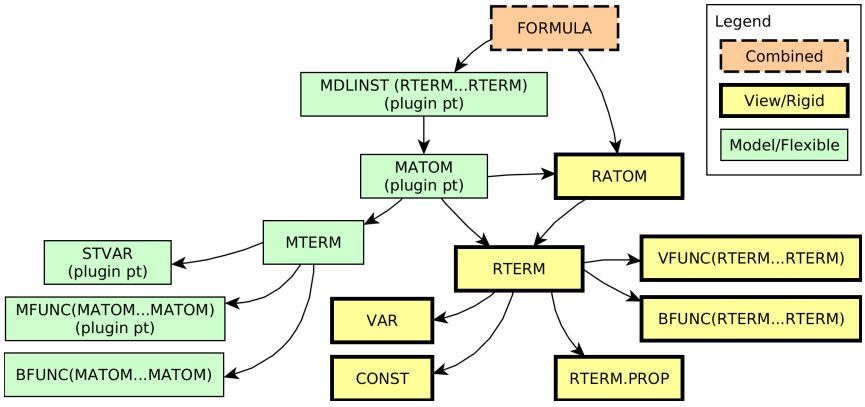


Figure 5.1: IPL abstract syntax. Boxes are syntax elements, and arrows are syntactic expansions.

Definition 7 (Rigid term)**.** *Rigid terms* of the language are defined as follows:

$$\begin{aligned}\text{RTERM} ::= \ & \text{VAR} \mid \text{CONST} \mid \text{RTERM.PROP} \mid \text{BFUNC}(\text{RTERM}_1 \ldots \text{RTERM}_n) \mid \\ & \text{VFUNC}(\text{RTERM}_1 \ldots \text{RTERM}_n).\end{aligned}$$

A *rigid term* RTERM is either a variable VAR from V, a constant CONST from Σ, an architectural element type $e^{\mathcal{V}}$ from $\Sigma^{\mathcal{V}}$, a property PROP[6] of a rigid term RTERM from $\Sigma^{\mathcal{V}}$, a background function BFUNC, or a view function VFUNC.

Definition 8 (Rigid atom)**.** *Rigid atoms* are logical formulas over rigid terms:

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

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

[6]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.

5.3.1 Plugin Points for Behavioral Properties

To integrate multiple behavioral property languages 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* (MTERM), 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. Flexible terms also incorporate background functions, over the syntax element MATOM, which is defined below. Thus, flexible terms offer three alternatives:

$$\text{MTERM} ::= \text{STVAR} \mid \text{MFUNC}(\text{MATOM}_1 \ldots \text{MATOM}_n) \mid \text{BFUNC}(\text{MATOM}_1 \ldots \text{MATOM}_n).$$

The third plugin point is *model atom* (MATOM), e.g., the expression $P_{max=?}$ of PCTL. 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 below. Each parameter name (from $PN = \{p_1 \ldots p_n\}$, given in $\Sigma^{\mathcal{M}}$) is bound to a rigid term.

Definition 9 (Model Instantiation Clause)**.** *Model instantiation clauses* wrap MATOM by binding model parameters $p_1 \ldots p_n$ to rigid terms $\text{RTERM}_1 \ldots \text{RTERM}_n$:

$$\text{MDLINST} ::= \text{MATOM}\{\!|p_1 = \text{RTERM}_1 \ldots p_n = \text{RTERM}_n|\!\}.$$

The values of RTERM_i are passed as parameters to the behavioral model. For models without parameters ($PN = \emptyset$), an empty instantiation clause $\{\!|\,|\!\}$ needs to be provided.

Finally, quantification binds the rigid and flexible syntax defined so far, via shared variables from V. IPL supports three kinds of quantification domains (DOM). First, a domain can be a type of architectural elements (ELEMTYPE, drawn from $\mathbb{T}$ in $\Sigma^{\mathcal{V}}$), and in this case the variable iterates through all elements of that type in $\mathbb{E}^{\mathcal{V}}$. Second, a domain can be a set of background values (VALTYPE, one of sets $\mathbb{O}$ in Σ^{B}), and the variable iterates over all values in that set. Finally, a domain can be a result of a rigid term (e.g., an intersection of two sets).

$$\text{DOM} ::= \text{ELEMTYPE} \mid \text{VALTYPE} \mid \text{RTERM}.$$

Definition 10 (IPL Formula)**.** *IPL formulas* are logical formulas with first-order quantification over an instantiated model formula or a rigid atom.

$$\text{FORMULA} ::= \forall \text{VAR} : \text{DOM} \cdot \text{FORMULA} \mid \text{FORMULA} \wedge \text{FORMULA} \mid$$
$$\neg\text{FORMULA} \mid \text{MDLINST} \mid \text{RATOM}.$$

To demonstrate the customizability of IPL, I provide two extensions of the grammar: first with Linear Temporal Logic (LTL) [218], and second with Probabilistic Computational Tree Logic (PCTL) [154]. Plugins for these logics expand MATOM in different ways.

5.3.2 LTL Plugin Syntax

Linear Temporal Logic is a logic to express temporal constraints on traces [218]. This plugins uses the two usual temporal modalities (*Until* U and *Next* X), and the other modalities (*Globally* G and *Eventually* F) 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_{lts} (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 ::= $\underline{v}$, where $v \in S$.
- State function: MFUNC ::= $\underline{g}(t_1 \dots t_n)$, where $\underline{g} \in MF$ and $t_1 \dots t_n \in$ MATOM.
- Background function: BFUNC ::= $g(t_1 \dots t_n)$, where $\underline{g} \in MF$ and $t_1 \dots t_n \in$ MATOM.
- Until: TATOM_u ::= TATOM U TATOM.
- Next: TATOM_x ::= X TATOM.
- Conjunction: TATOM_a := TATOM $\wedge$ TATOM.
- Negation: TATOM_n := $\neg$TATOM.
- A wrapper replaces the MATOM plugin point:

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

5.3.3 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)* [154]. 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* ($\mathsf{U}^{\leq \mathsf{k}}$ with a time bound k).
- 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.

$$\begin{aligned}
\text{PATHPROP} ::= {} & \text{RATOM} \mid \text{MTERM} \mid \text{PATHPROP} \wedge \text{PATHPROP} \mid \neg\text{PATHPROP} \mid \\
& \text{PATHPROP}\ \mathsf{U}^{\leq \mathsf{k}}\ \text{PATHPROP} \mid \mathsf{X}\ \text{PATHPROP}, \\
\text{RWDPATHPROP} ::= {} & \text{RATOM} \mid \text{MTERM} \mid \text{RWDPATHPROP} \wedge \text{RWDPATHPROP} \mid \\
& \text{RWDPATHPROP} \vee \text{RWDPATHPROP} \mid \mathsf{X}\ \text{RWDPATHPROP} \mid \\
& \text{RWDPATHPROP}\ \mathsf{U}^{\leq \mathsf{k}}\ \text{RWDPATHPROP} \mid \mathsf{C}^{\leq \mathsf{k}} \mid \mathsf{I}^{=t} \mid \mathsf{S},
\end{aligned}$$

$$\begin{aligned}
\text{PPROP} &::= P_{o\sim p}[\text{PATHPROP}],\ \text{PQUERY} ::= P_{o=?}[\text{PATHPROP}] \\
\text{RWDPROP} &::= R^r_{o\sim v}[\text{RWDPATHPROP}],\ \text{RWDQUERY} ::= R^r_{o=?}[\text{RWDPATHPROP}], \\
\text{MATOM} &::= \text{PPROP} \mid \text{PQUERY} \mid \text{RWDPROP} \mid \text{RWDQUERY},
\end{aligned}$$

where $p \in [0,1], \sim \in \{<, \leq, >, \geq\}, o \in \{max, min, \emptyset\}, t \in \mathbb{N}, \mathrm{k} \in \mathbb{N} \cup \{\inf\}$, $v \in \mathbb{R}$, 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.

5.3.4 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)	$\forall x : \overline{X} \cdot \overline{P}(x)$	Quantification over a view element type	Y
(3)	$(\mathsf{G}\ \underline{y} = 10)\{\!\|\!\}$	Model instance over a modality	Y
(4)	$\forall x : \overline{X} \cdot (\mathsf{G}\ \underline{Q}(x, \underline{y}))\{\!\|\!\}$	Quantification over a model instance	Y
(5)	$\exists x : \overline{X} \cdot \overline{P}(x) \rightarrow (\mathsf{F}\ \underline{Q}(x, \underline{y}))\{\!\|\!\}$	Quantification over an atom with a model instance	Y
(6)	$(\mathsf{G}\ (\exists x : \overline{X} \cdot \overline{P}(x, \underline{y})))\{\!\|\!\}$	Model instance over a quantified formula	N
(7)	$(\mathsf{F}\ (\underline{y} = \underline{z}))\{\!\|\!\}$	Mixed models in one term: $\mathcal{M}_1$ owns $\underline{y}$ and $\mathcal{M}_2$ owns $\underline{z}$	N

Table 5.1: Examples of acceptable and unacceptable IPL syntax. The symbols have belong to the following signature sets: $\overline{F}, \overline{P} \in VF, \underline{Q} \in MF, \overline{X} \in \mathbb{T}, x \in V, y \in S_1 \in \Sigma_1^{\mathcal{M}}, z \in S_2\ \in \Sigma_2^{\mathcal{M}}$.

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

Example (6) is unacceptable because it goes against the modularity principle (Subsection 5.2.1). 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 the design principles of 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 directly, 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 $\mathcal{V}_{\text{power}}$ (with task attributes $start$, end, and expected $energy$) and a PCTL query for $\mathcal{M}_{\text{plan}}$. Other integration properties for the energy-aware mobile robot can be found in Subsection 8.2.1.

Property 2. For any three sequential $\mathcal{M}_{\text{power}}$ tasks $\langle$go straight, rotate, go straight$\rangle$ that do not self-intersect and have sufficient energy, any execution in $\mathcal{M}_{\text{plan}}$ 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 $\overline{err}_{\text{cons}}$).

"For any three tasks from $\mathcal{M}_{power}$ 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 $\mathcal{M}_{plan}$ 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 $\overline{err}_{cons}$)."

$$\{\![\underline{initloc} = t_1.start, \underline{goal} = t_3.end, \underline{initbat} = \Sigma_{i=1}^{3} t_i.energy + \overline{err}_{\text{cons}}]\!\} = 1.$$

5.4 IPL Semantics

Now I give meaning to the IPL syntax over the IPL signature Σ in terms of the IPL structure Γ (consisting of the model and view structures, see Definition 6) by reducing parts of any IPL formula to either the model part of Γ (interpreted by $I^{\mathcal{M}}$) or the view part of Γ (interpreted by $I^{\mathcal{V}}$) — but not both for a given subformula. Ultimately, each formula is mapped to either true ($\top$) or false ($\bot$) in the context of a specific Γ.

5.4.1 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 I define two domains: the *model domain* ($\mathsf{D}_{\mathcal{M}}$) and the *view domain* ($\mathsf{D}_{\mathcal{V}}$). $\mathsf{D}_{\mathcal{M}}$ is associated with $I^{\mathcal{M}}$, and $\mathsf{D}_{\mathcal{V}}$ — with $I^{\mathcal{V}}$.

Definition 11 (Belonging to semantic domain)**.** Syntactic element s *belongs to a semantic domain* D if there exists an interpretation I such that $I(s) \in \mathsf{D}$.

$\mathsf{D}_{\mathcal{M}}$ and $\mathsf{D}_{\mathcal{V}}$ 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 interpretations and some set in the other domain exists. "By value" means mapping to a constant in the other domain. For instance, quantified variables (VAR, with domains of view types and rigid expressions) are transferred to

View domain $\mathsf{D}_{\mathcal{V}}$	**Is transferable**	**Model domain** $\mathsf{D}_{\mathcal{M}}$
VAR	Yes, by value in an assignment μ	
PROP	Yes, by value on a specific element	
VFUNC	Yes, by value, if all arguments are transferable. Otherwise, no.	
RTERM	Yes, by value	
ELEMTYPE	No (only by individual element reference through VAR)	
$\forall x : X \cdot f$	No	
	No	STVAR
	No	MFUNC
	No	MATOM
	Yes, by value	MDLINST
Shared: constants and BFUNC from background theories, interpretation I^B.		

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

model domains as individual constant values. "By reference" means mapping to a unique integer ID. For example, to reference elements in $\mathbb{E}^{\mathcal{V}}$ in a model, unique integer IDs are generated for each element). Notice that view domain elements are mostly transferable to the model domain (except quantification). To support modularity, models can transfer only values of MDLINST.

5.4.2 Native semantics

For each symbol of IPL, I provide the meaning given its context, which can be comprised of the following:

- Mapping μ of variable names to any set of values in Γ.
- A full IPL structure Γ, comprised of views ($\mathbb{V}$), models ($\mathbb{M}$), and background theories (Γ^B). The following parts of the structure are used as contexts:
  - Views $\mathbb{V}$, with interpretation $I^{\mathcal{V}}$.
  - A model $\mathcal{M}$ from $\mathbb{M}$, comprised of a set of trace sets (Ω) and interpretation $I^{\mathcal{M}}$. The following parts of the model are used as contexts:
    - A set of traces Ω.
    - A trace, which is a potentially infinite sequence of states starting from zero $\omega = \langle q\_1, q_1, \ldots \rangle$.
    - An individual state (q) of $\mathcal{M}$, with a state-specific interpretation $I^{\mathcal{M}}$.

For each semantic rule below, I keep track of the context, denoted in the subscript of $[\![\ ]\!]$ and on the left of $\models$. Starting from the bottom of Figure 5.1 with rigid terms (RTERM), I gradually

simplify the context so that only one interpretation can be used.

$[\![\text{CONST}]\!]_\Gamma = I^B(\text{CONST})$;
$[\![\text{VAR}]\!]_\mu = \mu(\text{VAR})$;
$[\![\text{VFUNC}(r_1 \dots r_n)]\!]_{\mathbb{V},\mu} = I^{\mathcal{V}}(\text{VFUNC})([\![r_1]\!]_{\mathbb{V},\mu} \dots [\![r_n]\!]_{\mathbb{V},\mu})$, where $r_1 \dots r_n \in$ RTERM;
$[\![\text{BFUNC}(r_1 \dots r_n)]\!]_{\Gamma,\mu} = I^B(\text{BFUNC})([\![r_1]\!]_{\Gamma,\mu} \dots [\![r_n]\!]_{\Gamma,\mu})$, where $r_1 \dots r_n \in$ RTERM;
$[\![\text{ELEMTYPE}]\!]_{\mathbb{V},\mu} = I^{\mathcal{V}}(\text{ELEMTYPE}) = \{e \mid e \in \mathbb{E}^{\mathcal{V}} \wedge \mathsf{T}(e) = \text{ELEMTYPE}\}$, where e is an element;
$[\![\text{VALTYPE}]\!]_\Gamma = I^B(\text{VALTYPE}) = \{o \mid o \in \text{VALTYPE}\}$, where o is a value from a background set;
$[\![\text{DOM}]\!]_{\Gamma,\mu} = S$, where S is an interpretation of ELEMTYPE, VALTYPE, or a set-valued RTERM;
$[\![\text{RTERM.PROP}]\!]_{\mathbb{V},\mu} = I^{\mathcal{V}}(\text{PROP})([\![\text{RTERM}]\!]_{\mathbb{V},\mu})$;
$[\![\text{STVAR}]\!]_{\mathcal{M},q} = I^{\mathcal{M}}(\text{STVAR}, q)$, where $I^{\mathcal{M}}$ is used in the second sense of Definition 4;
$[\![\text{MFUNC}(t_1, \dots t_n)]\!]_{\Gamma,q,\mu} = I^{\mathcal{M}}(\text{MFUNC})([\![t_1]\!]_{\Gamma,q,\mu} \dots [\![t_n]\!]_{\Gamma,q,\mu})$,
where $t_1 \dots t_n \in$ MTERM, and $I^{\mathcal{M}}$ is used in the third sense of Definition 4;
$[\![\text{MATOM}\{|p_1 = r_1 \dots p_n = r_n|\}]\!]_{\mathbb{V},\mathcal{M},\mu} = [\![\text{MATOM}]\!]_{\mathbb{V},\Omega,\mu}$, where $\Omega = I^{\mathcal{M}}(\Omega, \{p_1 \mapsto r_1 \dots$
$p_n \mapsto r_n\})$ (here $I^{\mathcal{M}}$ is used in the first sense of Definition 4), and Ω is part of $\Gamma^{\mathcal{M}}$ in $\mathcal{M}$);
$\Gamma, \mu \models$ RTERM *iff* $[\![\text{RTERM}]\!]_{\Gamma,\mu} = \top$, where RTERM is of boolean type;
$\Gamma, \mu \models \neg$FORMULA *iff* $\Gamma, \mu \not\models$ FORMULA;
$\Gamma, \mu \models$ FORMULA$_1 \wedge$ FORMULA$_2$ *iff* $\Gamma, \mu \models$ FORMULA$_1$ and $\Gamma, \omega, \mu \models$ FORMULA$_2$;
$\Gamma, \mu \models \forall x :$ DOM $\cdot$ FORMULA *iff* $\Gamma, \mu' \models$ FORMULA with $\mu' = \mu \cup \{x \mapsto v\}$
for all values v in $[\![\text{DOM}]\!]_{\Gamma,\mu}$.

5.4.3 LTL Plugin Semantics

The model of LTL sentences is a state transition system (a state set, an action set, a transition function, an initial state, and a state interpretation $I_q^{\mathcal{M}}$ to determine valid propositions in state q) [50]. A transition system defines a set of execution traces, which corresponds to Ω in $\Gamma^{\mathcal{M}}$. As a model in the IPL sense (Definition 4), I use a set of transition systems M_{lts} characterized by a set of parameters PN. Assigning values to parameters determines an individual transition system.

Below I provide the semantics for the LTL plugin. The notation $\omega^{i,j}$ means a substring of ω from element i to element j inclusively. I evaluate TATOM and FORMULA on a sequence of states (ω). Logical operations and quantifiers are evaluated natively, as defined above.

$\Gamma, q, \mu \models$ MTERM *iff* $[\![\text{MTERM}]\!]_{\Gamma,q,\mu} = \top$, where MTERM is of boolean type.
$\Gamma, \omega, \mu \models f$ *iff* $\Gamma, q, \mu \models f$, where $q = f \in$ MTERM and $\omega^{0,0}$;
$\Gamma, \omega, \mu \models \mathsf{X}$ TATOM *iff* $\Gamma, \omega^{1,\infty}, \mu \models$ TATOM;
$\Gamma, \omega, \mu \models$ TATOM$_1$ U TATOM$_2$ *iff*
$\exists i : \mathbb{N}_0 \cdot (\Gamma, \omega^{i,\infty}, \mu \models \text{TATOM}_2 \wedge \forall j : \mathbb{N}_0^{[0,i)} \cdot \Gamma, \omega^{j,\infty}, \mu \models \text{TATOM}_1)$.

The meaning of TATOM is given by iterating over all traces in the trace set (Ω, chosen from Ω

by parameter values in MDLINST) without any up-front variable mappings:

$$\Gamma \models \text{TATOM} \ \textit{iff} \ \forall \omega : \Omega \cdot \Gamma, \omega, \mu \models \text{TATOM}.$$

5.4.4 PCTL Plugin Semantics

To evaluate a PCTL formula, I use an MDP as a behavioral model in Γ (or a DTMC, if the model does not have non-deterministic transitions) [154]. Structure Γ contains parameterized sets of MDPs (M_{mdp}) or DTMCs (M_{dtmc}).

An MDP is characterized by the following:

- A finite state set S.
- An initial state $q_0 \in S$.
- A finite action set A.
- A probability transition function $P : S \times A \times S \to [0, 1]$, which indicates the probability that taking the action in the first state leads to the second state.
- Reward structures $r_i : S \times A \times S \to \mathbb{R}_{\geq 0}, i \in \mathbb{N}^{[1,n]}$, indicating the immediate reward for transitioning from the first state to the second using the action.
- A discount factor $\gamma \in [0, 1]$ that offsets rewards that are further away in the future.
- A state interpretation $I_q^{\mathcal{M}}$ 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 ($\mathsf{U}^{\leq \mathsf{k}}$):

$$\Gamma, \omega, \mu \models f_1 \ \mathsf{U}^{\leq \mathsf{k}} \ f_2 \ \textit{iff} \ \exists i : \mathbb{N}_0^{[0,k]} \cdot (\Gamma, \omega^{i,\infty}, \mu \models f_2 \wedge \forall j : \mathbb{N}_0^{[0,i)} \cdot \Gamma, \omega^{j,\infty}, \mu \models f_1), \text{ where}$$
f_1 and f_2 are both either PATHPROP or RWDPATHPROP.

A solution to an MDP is a policy $\pi : S \to A$ (belongs to a set of all potential policies Π). With a given π, the transition probability P induces a probability measure Pr_q^π over paths $\text{Paths}(q)$ starting in state q. In turn Pr_q^π induces a probability function over formulas that determines the probability of taking a path that satisfies formula f from state q: $\text{Prob}^\pi(q, f) = Pr_q^\pi\{\omega \in \text{Paths}(q) \mid \omega \models f\}$.

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

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

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:

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

where $X_f : \text{Paths}^\pi(q) \to \mathbb{R}_{\geq 0}$ is a random reward variable for paths that satisfy f (defined canonically for co-safe LTL and special formulas [154]), and Exp^π is its expectation with respect to Pr_q^π; other variables mean the same as above.

The semantics of IPL and two plugins has now been fully defined. Consider the following example as an illustration. Suppose a view $\mathcal{V}$ captures high-level characteristics of bouncing balls (type $\overline{B}$): the height from which a ball is dropped (a property $\overline{H}$ that ranges over positive real numbers) and its coefficient of restitution (a real $[0, 1]$-valued property $\overline{R}$). The view contains a finite number of elements that represent instances of such balls. Two behavioral models are available to model the behavior of a bouncing ball. The first model, $\mathcal{M}_1$, is an LTS, with a state variable $\underline{bn}_1$ that counts the number of bounces so far. The second model, $\mathcal{M}_2$, is a DTMC, a state variable $\underline{bn}_2$ that also counts the number of bounces so far. Unlike $\mathcal{M}_1$ with deterministic dynamics, $\mathcal{M}_2$ assigns a probability to the ball stopping without further bouncing. Both models have the same set of parameters: the initial height h from which a ball is dropped, and the ball's coefficient of restitution r.

The following formula expresses that for each ball in $\mathcal{V}$, the models should agree (with a 95% chance for $\mathcal{M}_2$) on whether the ball will eventually bounce at least 10 times. This property can be used to check that the models have consistent treatment of zeno effect.

$$\forall b : \overline{T} \cdot (\mathsf{F}\ \underline{bn}_1 \geq 10)\{\!|h = b.H, r = b.R|\!\} \Leftrightarrow (P_{\geq 0.95}[\mathsf{F}\ \underline{bn}_2 \geq 10)\{\!|h = b.H, r = b.R|\!\}. \quad (5.2)$$

Notice that the first-order logic is used in the outer layer of the formula, for a quantified variable b over the domain of T. The modal logics are localized to their respective subformulas, under MDLINST. The specified parameters allow selecting a particular trace set Ω from each model's $\mathbb{\Omega}$. Each modal subformula would be evaluated separately, by the interpretation of its model. Thus, quantified variables are used to bind multiple model-specific expressions in one IPL formula. With respect to the running example, the above semantics leads to an interpretation of Equation (5.1) that satisfies the intent of Property 1 (Page 46).

5.5 IPL Verification Algorithm

Suppose an engineer needs to verify an integration property FORMULA written for a signature Σ on a corresponding Γ (see Definition 6). That is, the engineer needs to check whether f is a sentence in the *IPL theory* for Γ. This section describes the steps — first abstractly, and then in detail — and provides an illustrative example afterwards, in Subsection 5.5.3. For convenience of reading this section, the common symbols are summarized on Page xiii.

Problem 1 (IPL Formula Satisfaction on a Model)**.** Given $f \in$ FORMULA in signature Σ and a corresponding structure Γ, decide whether $\Gamma, \emptyset \models f$, where $\emptyset$ indicates no a priori assignments of variable values.

Intuitively, the goal of IPL verification is to determine whether an IPL formula holds for a given set of views and behavioral models. The process is assisted by an SMT solver, and in preparation all the views are converted to SMT specifications as collections of facts about the existence of view elements, their types, and their properties.

The algorithm consists of the initial processing and two computationally-intensive steps. A visual summary of the algorithm steps is shown in Figure 5.2. Initially, the IPL formula is

processed for SMT interaction via formula transformations. Each version of the formula is shown on the right side of the figure. To make the formula SMT-readable, all behavioral subformulas (which do not have any a priori SMT interpretation) are replaced with uninterpreted functions.

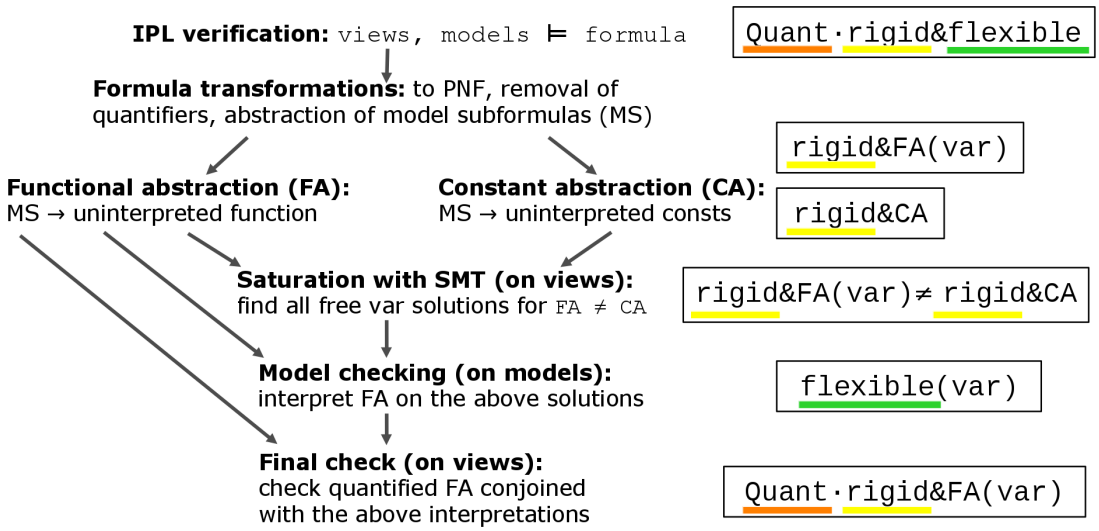


Figure 5.2: A summary of the verification algorithm. The formula involved in each step is schematically shown to the right. The meaning of colors is the same as in Figure 5.1: yellow is for rigid/native subformulas, green — for flexible/model-specific, and orange — for mixed/quantified.

The first computationally-intensive step of the algorithm is *saturation*. It determines what behavioral information is needed from each model. This task is accomplished by using SMT to find values of free variables (quantified in the original formula) for which the removed behavioral plugins could affect the satisfaction of the formula.

The second computationally-intensive step is *model querying*. The necessary information is extracted from each behavioral model by evaluating a corresponding subformula on the variable values found during saturation. The received behavioral information is used to construct interpretations of the uninterpreted functions that replace the behavioral subformulas. Once all the functions are augmented with the necessary values, the original quantified formula with uninterpreted functions is checked by SMT to determine whether the formula holds.

5.5.1 Formula Transformations

To formalize the algorithm to solve Problem 1, I introduce several *syntactic transformations* (also known as rewrite rules) of IPL formulas below. Here, $A \rightsquigarrow B$ defines each transformation, meaning that every subformula matching A is replaced a formula B. Also, $A\{x/y\}$ means that in formula A all occurrences of x are substituted by y.

I start with the transformation of a formula to its *prenex normal form* (ToPNF), formalized as the following rewrite system. The quantifiers (Q) are propagated to the outer layers of the formula, and inverted ($\overline{Q}$) when passing through negation (i.e., $\forall$ is replaced with $\exists$, and vice versa). This

transformation is executed until all quantifiers are outside of the formula's operators.

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

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

Next, I formalize a transformation that *removes quantifiers* from a formula (RemQuant), replacing the occurrences of quantified variables (x) with corresponding free variables ($\hat{x}$). This transformation is executed on the same formula until no quantifiers remain in the formula.

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

The transformation of *constant abstraction* (ConstAbst) replaces a top-level model-specific subformula (which consists of MATOM and MDLINST) with an uninterpreted constant (C). C has the same type as MATOM. The subsequent steps of the algorithm develop an interpretation for C by querying a model, as described below in Subsection 5.5.2.

$$\mathsf{ConstAbst} \equiv\ (\text{MATOM})\ \text{MDLINST} \rightsquigarrow C.$$

Finally, the transformation of *functional abstraction* (FuncAbst) replaces a top-level model-specific subformula (which consists of MATOM and MDLINST) with an uninterpreted function F, with arguments $x_1 \ldots x_n$ that list all free variables that occur in MATOM (as terms) and MDLINST (as parameter values). F has the same type as MATOM. Similarly to the uninterpreted constants before, an interpretation for F is built later in the algorithm by querying a model.

$$\mathsf{FuncAbst} \equiv\ (\text{MATOM})\ \text{MDLINST} \rightsquigarrow F(x_1 \ldots x_n).$$

5.5.2 Algorithm Steps

The verification steps for IPL formulas are presented in Algorithm 1. For simplicity, the algorithm is presented for one model $\mathcal{M}$, but can be trivially extended to handle multiple models by associating each MDLINST with its model. The algorithm is agnostic to the specifics of the plugin syntax, as long as each model can interpret its property language in finite time.

The first step is equivalently transforming the input formula $f \in \text{FORMULA}$ to its prenex normal form (PNF, i.e., all quantifiers are placed at the beginning of the formula) with $\mathsf{ToPNF}(f)$. The resulting formula has all quantifiers $Q_1 \ldots Q_n$ (any of which can be $\forall$ or $\exists$ without restrictions) for variables $x_1 \ldots x_n$ in the front. The variables vary over domains $D_i \in \text{DOM}$. Below, $\boldsymbol{x}$ means $x_1 \ldots x_n$, and $\hat{f}$ is the remainder of the formula f without the quantifiers, written as a predicate over all quantified variables and m model instances.

$$f^{PNF} \equiv \mathsf{ToPNF}(f) = Q_1 x_1 : D_1 \ldots Q_n x_n : D_n \cdot \hat{f}(\boldsymbol{x}, \text{MDLINST}_1(\boldsymbol{x}) \ldots \text{MDLINST}_m(\boldsymbol{x})).$$

The next step is to replace occurrences of instance terms MDLINST_i with abstract terms. The goal is to remove the syntax that cannot be interpreted in a model-agnostic way, replacing it with uninterpreted terms, building their interpretation in later steps. At the start, the interpretation of

these abstractions is yet unknown to the SMT solver, which at the outset only has access to view specifications. Thus, I replace MDLINST_i with two kinds of abstractions:[7]

1. *Functional abstraction (FA)*. The FA transformation replaces MDLINST_i with uninterpreted functions F_i. The arguments of these functions are the free variables that are present in the syntactic sub-tree of MDLINST_i.

$$f^{FA} \equiv \mathsf{FuncAbst}(f^{PNF}) = Q_1 x_1 : D_1 \dots Q_n x_n : D_n \cdot \hat{f}(\boldsymbol{x}, F_1(\boldsymbol{x}) \dots F_m(\boldsymbol{x})).$$

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

$$f^{CA} \equiv \mathsf{ConstAbst}(f) = Q_1 x_1 : D_1 \dots Q_n x_n : D_n \cdot \hat{f}(\boldsymbol{x}, C_1 \dots C_m).$$

Algorithm 1 IPL verification algorithm

procedure VERIFY($f, \mathcal{M}$)
  $f^{PNF} \leftarrow \mathsf{ToPNF}(f)$ ▷ Put the formula into the prenex normal form
  $f^{FA} \leftarrow \mathsf{FuncAbst}(f^{PNF})$ ▷ Replace model instances with functional abstractions
  $f^{CA} \leftarrow \mathsf{ConstAbst}(f^{PNF})$ ▷ Replace model instances with constant abstractions
  $\hat{f}^{FA} \leftarrow \mathsf{RemQuant}(f^{FA})$ ▷ Remove quantifiers in the formula with FA
  $\hat{f}^{CA} \leftarrow \mathsf{RemQuant}(f^{CA})$ ▷ Remove quantifiers in the formula with CA
  $SV \leftarrow$ all μ s.t. $\exists I \cdot I, \mu \models \hat{f}^{FA} \neq \hat{f}^{CA}$ ▷ *Saturation:* find all[8] variable values that satisfy non-matching abstractions
  $I^F_{SV}(F_i(\mu)) \leftarrow [\![\text{MDLINST}_i]\!]_{\mathcal{M},\mu}$ for each $\mu \in SV$ ▷ *Model querying:* run model instances to interpret functional abstractions on the above values
  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
  else return $\top$ ▷ Otherwise, return true

Next, the algorithm removes all quantifiers $Q_1 \dots Q_n$, replacing all bound quantified variables ($\boldsymbol{x}$) with their free counterparts ($\hat{\boldsymbol{x}}$). Below, $\hat{f}^{FA}$ and $\hat{f}^{CA}$ are quantifier-free versions of f^{FA} and f^{CA} respectively.

$$\hat{f}^{FA} \equiv \mathsf{RemQuant}(f^{FA}) = \hat{f}(\hat{\boldsymbol{x}}, F_1(\hat{\boldsymbol{x}}) \dots F_m(\hat{\boldsymbol{x}}))$$
$$\hat{f}^{CA} \equiv \mathsf{RemQuant}(f^{CA}) = \hat{f}(\hat{\boldsymbol{x}}, C_1 \dots C_m).$$

We look for interpretations (I^F_{SV}) of model instances that affect satisfaction of f. I^F_{SV} are characterized by valuations (μ) of free variables (i.e., vectors of n values from domains $D_1 \dots D_n$) 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: $I^F(F_i(\mu)) = [\![\text{MDLINST}_i]\!]_{\mathcal{M},\mu}$ for any $\mu \in D_1 \times \dots D_n, i \in [1, m]$.

[7]The word "abstraction" here is used in the logical sense, for replacing a detailed formula with an abstract one. This term is not related to integration abstractions.

[8]For the algorithm to terminate, each quantification domain ($D_1 \dots D_n$) needs to be interpreted to a finite set.

Instead of constructing full I^F (which requires exhaustive model checking), the algorithm determine I^F_{SV} by searching for such μ that make FA and CA not equal to each other. In other words, these are such valuations that it is possible to interpret the two abstractions in a way that makes one of the two formulas ($\hat{f}^{FA}$, $\hat{f}^{CA}$) satisfied, and the other one — unsatisfied (or, in short, not equal in their boolean value). Thus, the algorithm constructs a set SV that contains all μ satisfying the *search formula* for f: $\exists I \cdot I, \mu \models \hat{f}^{FA} \neq \hat{f}^{CA}$.

In the process of *saturation*, the algorithm enumerates all the μ in the following two-step process. First, the algorithm solves the search formula for some solution μ', and adds μ' to SV. Second, the algorithm adds a *blocking clause* for that solution ($\mu \neq \mu'$) to the search formula, in order to find its other solutions. These two steps alternate until the process terminates. With a finite number of solutions for the search formula (which becomes one of the termination conditions in Section 5.6), this process will terminate, thus saturating SV. 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 the variable assignments (SV) are determined, the algorithm constructs an interpretation of flexible subformulas I^F_{SV} (a subset of I^F) by *model querying* — executing a behavioral query as follows. For each flexible subformula (MDLINST_i), the algorithm substitutes its free variables for values in μ and uses the respective behavioral model ($\mathcal{M}$) to interpret that subformula:

$$I^F_{SV}(F_i)(\mu) = [\![\text{MDLINST}_i]\!]_{\mathcal{M},\mu} \text{ for all } \mu \in SV \text{ and all } i \in [1, m]. \tag{5.3}$$

Finally, the algorithm performs a final check by checking satisfiability of the negation of f^{FA}, in combination with the interpretations I^F_{SV} obtained by querying. f is satisfied *iff* the check fails to find an interpretation that agrees with I^F_{SV} and satisfies $\neg f^{FA}$.

5.5.3 Application to Running Example

Now, I illustrate the application of the IPL verification algorithm to the running example, Equation (5.1). The formula is already in its PNF:

$$\begin{aligned}
&\forall t_1, t_2, t_3 : \overline{Tasks} \cdot t_1.type = t_3.type = \text{STR} \wedge t_2.type = \text{ROT} \wedge \\
&t_1.end = t_2.start = t_3.start \wedge t_1.start \neq t_3.end \wedge \Sigma_{i=1}^{3} t_i.energy \leq \overline{maxbat} \rightarrow \\
&\quad P_{max=?}[(\underline{loc} = t_1.start \ \mathsf{U} \ (\underline{loc} = t_2.start \ \mathsf{U} \ \underline{loc} = t_3.end)) \wedge (\mathsf{F} \ \underline{loc} = t_2.start)] \\
&\qquad \{\!|\underline{initloc} = t_1.start, \underline{goal} = t_3.end, \underline{initbat} = \Sigma_{i=1}^{3} t_i.energy + \overline{err}_{\text{cons}}|\!\} = 1.
\end{aligned} \tag{5.4}$$

Then MDLINST is abstracted away in two different ways, once via FuncAbst with a real-valued function $\text{FA}(t_1, t_2, t_3)$, and once via ConstAbst with a real constant CA:

$$\begin{aligned}
&\forall t_1, t_2, t_3 : \overline{Tasks} \cdot t_1.type = t_3.type = \text{STR} \wedge t_2.type = \text{ROT} \wedge \\
&t_1.end = t_2.start = t_3.start \ \wedge t_1.start \neq t_3.end \wedge \Sigma_{i=1}^{3} t_i.energy \leq \overline{maxbat} \rightarrow \\
&\quad \text{FA}(t_1, t_2, t_3) = 1;
\end{aligned} \tag{5.5}$$

$$\begin{aligned}
&\forall t_1, t_2, t_3 : \overline{Tasks} \cdot t_1.type = t_3.type = \text{STR} \wedge t_2.type = \text{ROT} \wedge \\
&t_1.end = t_2.start = t_3.start \ \wedge t_1.start \neq t_3.end \wedge \Sigma_{i=1}^{3} t_i.energy \leq \overline{maxbat} \rightarrow
\end{aligned} \tag{5.6}$$

$$\text{CA} = 1.$$

The next step is to remove quantifiers and replace t_1, t_2, t_3 with their free counterparts $\hat{t}_1, \hat{t}_2, \hat{t}_3$. An application of RemQuant to the formulas above yields the following results:

$$\begin{aligned}
&\hat{t}_1.type = \hat{t}_3.type = \text{STR} \wedge \hat{t}_2.type = \text{ROT} \wedge \\
&\hat{t}_1.end = \hat{t}_2.start = \hat{t}_3.start \ \wedge \hat{t}_1.start \neq \hat{t}_3.end \wedge \Sigma_{i=1}^{3} \hat{t}_i.energy \leq \overline{maxbat} \rightarrow \\
&\quad \text{FA}(\hat{t}_1, \hat{t}_2, \hat{t}_3) = 1;
\end{aligned} \tag{5.7}$$

$$\begin{aligned}
&\forall \hat{t}_1, t_2, t_3 : \overline{Tasks} \cdot \hat{t}_1.type = \hat{t}_3.type = \text{STR} \wedge \hat{t}_2.type = \text{ROT} \wedge \\
&\hat{t}_1.end = \hat{t}_2.start = \hat{t}_3.start \ \wedge \hat{t}_1.start \neq \hat{t}_3.end \wedge \Sigma_{i=1}^{3} \hat{t}_i.energy \leq \overline{maxbat} \rightarrow \\
&\quad \text{CA} = 1.
\end{aligned} \tag{5.8}$$

Following the abstraction, the saturation process populates SV with all tuples of values for $\hat{t}_1, \hat{t}_2, \hat{t}_3$ (i.e., triplets of tasks) that satisfy the following search formula:

$$\begin{aligned}
&\hat{t}_1.type = \hat{t}_3.type = \text{STR} \wedge \hat{t}_2.type = \text{ROT} \wedge \\
&\hat{t}_1.end = \hat{t}_2.start = \hat{t}_3.start \ \wedge \hat{t}_1.start \neq \hat{t}_3.end \wedge \Sigma_{i=1}^{3} \hat{t}_i.energy \leq \overline{maxbat} \rightarrow \\
&\quad \text{FA}(\hat{t}_1, \hat{t}_2, \hat{t}_3) = 1 \\
&\neq \\
&\hat{t}_1.type = \hat{t}_3.type = \text{STR} \wedge \hat{t}_2.type = \text{ROT} \wedge \\
&\hat{t}_1.end = \hat{t}_2.start = \hat{t}_3.start \ \wedge \hat{t}_1.start \neq \hat{t}_3.end \wedge \Sigma_{i=1}^{3} \hat{t}_i.energy \leq \overline{maxbat} \rightarrow \\
&\quad \text{CA} = 1.
\end{aligned} \tag{5.9}$$

Each tuple $(T_1, T_2, T_3) \in satvals$ represents a concrete mission. For each tuple/mission, $\mathcal{M}_{\text{plan}}$ is instantiated according to the MDLINST of the original formula (Equation (5.4)): the initial location is set to $T_1.start$, the goal location is set to $T_3.end$, and the initial battery is set to $T_1.energy + T_2.energy + T_3.energy + \overline{err}_{\text{cons}}$. Next, the following PCTL formula is evaluated on the instantiated model (hence performing the model-querying step), determining the probability of the robot arriving at the goal for each mission:

$$P_{max=?}[(\underline{loc} = T_1.start \ \mathsf{U} \ (\underline{loc} = T_2.start \ \mathsf{U} \ \underline{loc} = T_3.end)) \wedge (\mathsf{F} \ \underline{loc} = T_2.start)]. \tag{5.10}$$

Each output of querying q (a real number from 0 to 1) is added to the interpretation of FA, specifically recording an SMT assertion: $\text{FA}(T_1, T_2, T_3) = q$. Finally, the obtained interpretations will be conjoined with the negation of Equation (5.5) for the ultimate satisfiability check:

$$\begin{aligned}
&\neg \ (\forall t_1, t_2, t_3 : \overline{Tasks} \cdot t_1.type = t_3.type = \text{STR} \wedge t_2.type = \text{ROT} \wedge \\
&t_1.end = t_2.start = t_3.start \ \wedge t_1.start \neq t_3.end \wedge \Sigma_{i=1}^{3} t_i.energy \leq \overline{maxbat} \rightarrow \\
&\quad \text{FA}(t_1, t_2, t_3) = 1).
\end{aligned} \tag{5.11}$$

Receiving UNSAT from Equation (5.11) means that the original formula (Equation (5.1)) is satisfied, while receiving SAT means that it is not satisfied.

The IPL verification algorithm has been formally analyzed for soundness, with its soundness theorem and its proof presented below. The implications for the soundness of the integration approach are presented in Chapter 8.

5.6 Theoretical Evaluation

I start with an abstract, domain-independent analysis of soundness of the IPL verification algorithm (Algorithm 1) presented in the previous section. To remind the reader, the goal of the IPL algorithm is to solve the IPL satisfaction problem (Problem 1): given an IPL formula in a signature Σ and a corresponding structure Γ (see Definition 6), determine whether the formula is satisfied on Γ.

To avoid false positives and false negatives, IPL verification should produce sound results. To this end, I formally prove that whenever an IPL is verified by Algorithm 1, the result is correct with respect to the IPL semantics — regardless of the details of behavioral plugins, as long as behavioral queries deliver correct results. To be valuable in engineering, the verification algorithm should terminate on practical problems, hence I also describe the termination conditions.

I start with a theorem that links interpretations of flexible clauses (MDLINST, see Definition 9) in an IPL formula to the satisfaction of that formula on a full interpretation (that is, the semantic satisfaction of that formula). These interpretations are obtained in Algorithm 1 during model querying over the values of free variables (SV) that satisfy the search formula (see Subsection 5.5.2). Thus, the theorem connects the two determinations of IPL formula satisfaction: the algorithmic output (an interpretation I^F_{SV} based on queries of MDLINST on SV) and the semantic truth (an interpretation I^F based on the meaning of flexible clauses on any values of free variables). The soundness of the verification algorithm, as well as its termination conditions, follow directly from this theorem in Corollary 2.

Several remarks need to be made before proceeding:

- Recall that the functional abstraction (f^{FA}) of an IPL formula (f) is the result of replacing flexible clauses (MDLINST) with uninterpreted functions in f. The quantifiers and variables are the same in f and f^{FA}. The verification algorithm uses f^{FA} for a final satisfaction check in the last step.
- The theorem assumes that behavioral queries are sound (see Definition 29) and terminate in a finite time.
- Below, logical connectives are used both as IPL syntactic elements and as meta-operators over the statements about formula interpretations.

Theorem 1 (Agreement of Semantic and Algorithmic Interpretations)**.** Non-existence of flexible interpretations (I) that agree with model interpretations I^F_{SV} on SV *and* satisfy the negation of the functional abstraction ($\neg f^{FA}$) is necessary and sufficient for satisfaction of the functional abstraction (f^{FA}) on the model's full interpretation (I^F):

$$\left(\nexists I \cdot (I^F_{SV} \subseteq I) \wedge (I \models \neg f^{FA})\right) \textit{ iff } I^F \models f^{FA}.$$

Proof. Sufficiency 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 a full interpretation I^F leads to $I^F_{SV} \subseteq I^F \rightarrow I^F \models f^{FA}$. The premise of this implication holds by construction: I^F_{SV} are obtained by querying the values of MDLINST from the model, and since the queries are assumed to be sound, the returned values of I^F_{SV} agree with the full interpretation I^F. Thus, modus ponens yields $I^F \models f^{FA}$.

Necessity relies on constructing a variable valuation μ that leads to a contradiction. For necessity, it is required to show the following:

$$\text{If } \exists I \cdot I^F_{SV} \subseteq I \wedge I \models \neg f^{FA}, \text{ then } I^F \not\models f^{FA}. \tag{5.12}$$

Assume, for contradiction, that $I^F \models f^{FA}$. Instantiation of $\exists I$ in the antecedent above leads to some interpretation I' that agrees with I^F_{SV} (and, therefore, with I^F on SV due to query soundness) and satisfies $\neg f^{FA}$:

$$I^F \models f^{FA}, \tag{5.13}$$

$$I' \models \neg f^{FA}. \tag{5.14}$$

Using the two satisfactions above, I will construct a variable assignment μ, starting from $\emptyset$, by unwrapping quantifiers in f^{FA}. By f^{FA}_i I mean a formula that results from removing[9] the first i quantifiers and replacing their variables with free variables. Removal of quantifiers is not an equivalent transformation, but under certain conditions it results in a logical consequence, as done below. When the first i quantifiers are removed from f^{FA}, their quantified variables $x_1 \dots x_i$ are replaced with free variables $\hat{x}_1 \dots \hat{x}_i$:

$$f^{FA}_i(\hat{x_1} \dots \hat{x_i}) \equiv Q_{i+1}x_{i+1} : D_{i+1} \cdot \dots Q_n x_n : D_n \cdot f^{FA}(\hat{x}_1 \dots \hat{x}_i, x_{i+1} \dots x_n).$$

Now I describe how exactly the quantifiers in f^{FA} are removed. Consider the two cases of the outermost quantifier Q_1 in f^{FA}. $f^{FA}_1(\hat{x}_1)$ is the result of removing Q_1 from f^{FA}, and $\hat{x}_1$ is a free variable in $f^{FA}_1(\hat{x}_1)$.

If $Q_1 \equiv \exists$, then I instantiate the existential quantifier in $I^F \models f^{FA}$ (Equation (5.13)) with some value v_1, leading to $I^F \models f^{FA}_1(v_1)$. Then, I push the negation through the existential quantifier in $I' \models \neg f^{FA}$ (Equation (5.14)), leading to $I' \models \forall x_1 : D_1 \cdot \neg f^{FA}_1$. Instantiating the universal quantifier with v_1, I get $I' \models \neg f^{FA}_1(v_1)$.

If $Q_1 \equiv \forall$, then I push the negation over the universal quantifier in $I' \models \neg f^{FA}$ (Equation (5.14)) and instantiate the resulting existential quantifier with some value v_1. Then, $I' \models \neg f^{FA}_1(v_1)$. By instantiating the outer universal quantifier in $I^F \models f^{FA}$ (Equation (5.13)) with v_1, I get $I^F \models f^{FA}_1(v_1)$.

Thus, regardless of the outer quantifier in f^{FA}, the following holds for a known value v_1:

$$I^F, (\hat{x}_1 \mapsto v_1) \models f^{FA}_1(\hat{x}_1),$$
$$I', (\hat{x}_1 \mapsto v_1) \models \neg f^{FA}_1(\hat{x}_1).$$

Next, I add $(x_1 \mapsto v_1)$ to μ and repeat the above process for the remaining $n-1$ quantifiers, reducing the quantified formula f^{FA} to its quantifier-free version $\hat{f}^{FA}$ (which is the same as f^{FA}_n). The variable assignment μ contains values for free variables $\hat{x}_1 \dots \hat{x}_n$. Thus, two assertions hold after the process of removing the quantifiers:

$$I', \mu \models \neg \hat{f}^{FA}(\hat{x}_1 \dots \hat{x}_n), \tag{5.15}$$

[9]Note that the described process of instantiation and removal of quantifiers always targets the outermost quantifier(s). Its goal is to obtain a contradictory consequence — not perform Skolemization or Herbrandization.

$$I^F, \mu \models \hat{f}^{FA}(\hat{x}_1 \ldots \hat{x}_n). \tag{5.16}$$

Now I will show that $\mu \in SV$. Consider the constant abstraction $\hat{f}^{CA}$ that corresponds to $\hat{f}^{FA}$, that is $\hat{f}^{CA} = \mathsf{RemQuant}(\mathsf{ConstAbst}(f))$. Let I^{CA} be some interpretation of the constant abstractions in $\hat{f}^{CA}$. Notice that I^{CA}, I', and μ are disjoint interpretations: I^{CA} is for constant symbols, I'uses function symbols, and μ is for free variable symbols (which may serve as arguments for I'). The same applies to I^{CA}, I^F, and μ.

By the principle of excluded middle, $\hat{f}^{CA}$ is either satisfied by I^{CA} or not. If $I^{CA}, \mu \models \hat{f}^{CA}$, then by combining it with Equation (5.15) I get $I', I^{CA}, \mu \models \hat{f}^{FA} \not\Leftrightarrow \hat{f}^{CA}$. Notice that t If $I^{CA}, \mu \models \neg\hat{f}^{CA}$, then by combining it with Equation (5.16) I get $I^F, I^{CA}, \mu \models \hat{f}^{FA} \not\Leftrightarrow \hat{f}^{CA}$. These statements are the definition of μ in the algorithm, and therefore $\mu \in SV$.

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 (5.15) and Equation (5.16)) should agree on the satisfaction of $\hat{f}^{FA}$. Since the semantics of IPL is unambiguous, the above leads to a contradiction.

Therefore, $I^F \models f^{FA}$. □

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

Corollary 1 (Relation of Final Check and Initial Formula)**.** Satisfaction of formula f is equivalent to unsatisfiability $I^F_{SV} \models \neg f^{FA}$.

$$\mathcal{M} \models f \textit{ iff } \nexists I \cdot I^F_{SV} \subseteq I \wedge I \models \neg f^{FA}.$$

Proof. By construction of f^{FA} in the algorithm, $\mathcal{M} \models f$ 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 (Soundness of IPL verification)**.** Algorithm 1 is sound for solving Problem 1. The algorithm terminates if (i) satisfiability checking is decidable, (ii) behavioral checking with $\mathcal{M}$ 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 satisfaction 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 $\mathcal{M}$ is decidable (premise (ii) above). □

Corollary 2 is the central piece of evidence for 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 decided to hold or not on a given model.

5.7 IPL Implementation

A design and verification environment for IPL was implemented based on the Xtext language framework (`https://www.eclipse.org/Xtext`) in the Eclipse IDE. The sources of this IPL implementation are available online (`https://github.com/bisc/IPL`) and have also been archived [241]. 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 5.3 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, I 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, I used the Architecture Analysis and Description Language (AADL, version 2.1) [80]. 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 5.2: the views are statically defined and can have custom properties with fixed values. IPL's implementation relies on OSATE2 (version 2.3.0) [78] — 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. This translator is part of the aforementioned Eclipse-based IPL environment. The SMT solvers are interfaced done through the SMT-LIB v2.6 [19] syntax to abstract away from specific implementations. The back-end solvers are Z3 (version 4.5.0) [60] and CVC4 (version 4.1.5) [18].

Chapter Summary

This chapter described the first part of the integration approach — the Integration Property Language. The language was defined via extensible syntax and semantics, which allows plugging in various behavioral property languages. IPL statements can be checked over models by using an algorithm, which relies on SMT solving and model checking. This algorithm was proved to be sound, and implemented as a plugin for the Eclipse development environment. Given some assumptions on integration abstractions, IPL verification can detect model inconsistencies.

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. As part of these interactions, the tools need to access the models through a standardized interface, and this interface should be customizable for different integration scenarios. Thus, integration abstractions serve as "interfaces" through which the IPL verification and the analysis execution platform access the model elements.

Integration abstractions are crucial in integration arguments: the claims about models' consistency or dependencies of analyses depend on what proxies of models are used by an integration mechanism. The final part of the integration argument is formulated at the levels of IPL (in Part I) and analysis execution (in Part III). 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 views that reflect the static elements of a model. View elements can be verified directly in IPL. The behavioral abstractions rely on specifying and checking behavioral properties of models in appropriate property languages. The behavioral properties This chapter defines these abstractions, explains how they are constructed, and how their *characteristics* (described below in Subsections 6.2.6 and 6.3.3) fit into the model integration argument (presented earlier 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 of abstractions support the checking that an analysis is executed in an appropriate context, which presented in Part III, Chapter 7.

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

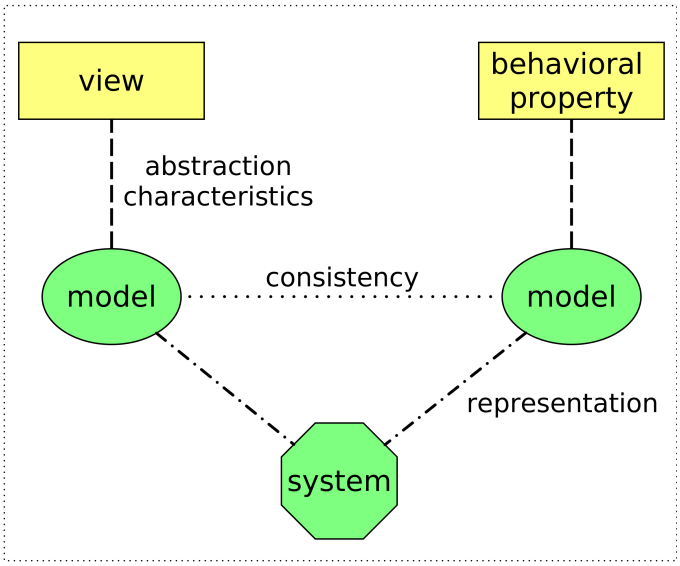


Figure 6.1: The role of integration abstractions.

integration abstraction:

- *The aspect of modeling* involved in the integration scenario. The aspects/qualities may include timing, energy, convergence, approximation error, refinement, or other aspects in terms of which the models can be related.
- *The formalism of models.* Differences between formalisms may affect the choice of abstraction because related information may be encoded in different ways, altering the convenient ways of representing and relating 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 objects to be manipulated by the integration. For example, verification typically manipulates models with non-determinism, and abstractions for verification should 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, formal notations for the models, 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.

6.1 Running Example: Hybrid Program, 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) [82]. The model includes sensors, actuators, processing units, batteries, and connections between them — electrical and wireless.
- A *hybrid program* [214] (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 [77] is given in Figure 6.2. The model shows speed control component that receives data from a speed sensor and an interface with the higher-level planning. The speed controller outputs speed commands to the throttle actuator. The data is communicated over a shared bus, connected a CPU (*RT_MHz350*) and a shared memory unit. This model captures how the data flows between hardware elements, indicating the ports and the types of exchanged data.

Hybrid programs are built using operators in Table 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 variable values as specified by differential equations within an evolution domain (part of the state-space where this evolution is allowed to continue), 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.

Consider a mobile robot with position x, velocity v, and acceleration a. This robot needs to reach the goal location 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

[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.

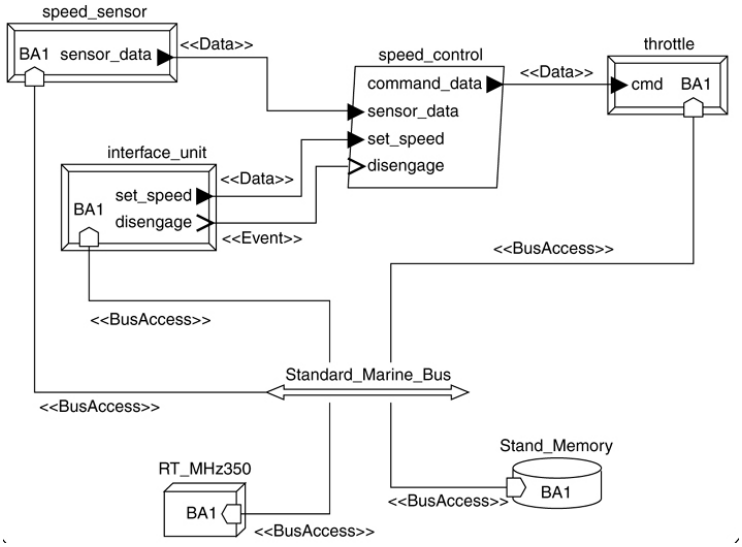


Figure 6.2: An AADL hardware model for a speed control subsystem.

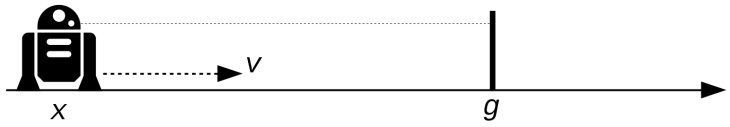


Figure 6.3: An illustration of the HP α_{robot} and its property in Equation (6.3).

kinematic movement ($v' = a, x' = v$) in a non-deterministic loop. The following hybrid program models possible motion of the robot:

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

Differential dynamic logic ($\mathsf{d}\mathcal{L}$) [214] 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 (illustrated in Figure 6.3) 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 ($x \geq g$):

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

Integration of hybrid programs and hardware models proceeds in two steps. The first step is to relate hybrid programs and AADL hardware models. For instance, one can construct a mapping

Statement	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.

from variables in the hybrid program to the components in the hardware model that assign values to those variables. The second step is to check properties of how the two models are related. For example, one can check whether there exists a sensor in the hardware model for every input used to control the system in a hybrid program.

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 Abstractions: Views

First, I introduce important concepts for architectural views. Then, I formalize the intended relationship between models and views — model-view conformance. Finally, I describe the activities that achieve and maintain conformance, and relate the properties of views to the integration argument laid out in Section 4.4 and detailed in Section 8.1.

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 [51, 153, 177]. 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 [25]. 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 that determine the organization of a broad range of CPS models [26].
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 [25].
3. Architectural views can check constraints over parameter values across several models [227].

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 [26].

Instead, consistency and completeness of views are defined in relation to model elements. Moreover, IPL gives views a logical basis, enabling custom logical properties and extending the pre-conceived property of structural consistency. Below I formulate the fundamentals of views for their use in IPL integration.

6.2.1 Internal Organization of Views

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. Given an integration scenario, I denote the set of models related to it 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 5.2 for IPL. The formal definition of views (Definition 2, as a triplet of a signature of symbols, a structure of view elements, and an interpretation mapping the symbols to the structure) also applies, but it focuses on the distinction between specification and verification, and is therefore less convenient for this chapter.

Definition 12 (View)**.** An (architectural) *view* $\mathcal{V}$ is a tuple $(\mathbb{E}^{\mathcal{V}}, \mathbb{T}, \mathsf{T}, \mathbb{P})$, where $\mathbb{E}^{\mathcal{V}}$ is a finite set of architectural elements, $\mathbb{T}$ is a finite set of architectural types, $\mathsf{T} : \mathbb{E}^{\mathcal{V}} \rightarrow \mathcal{P}(\mathbb{T})$ is a typing function that maps every architectural element to multiple types, and $\mathbb{P}$ is a set of property functions $p : \mathbb{E}^{\mathcal{V}} \rightarrow \mathbb{O}$ that map architectural elements to a value from some set of values ($\mathbb{O}$). See Definition 1 in Section 5.2 for more detail.

First, I 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 6.4, follows the classic works of software architecture [3, 97, 244]. A view contains architectural elements that include components (CMS, e.g., a controller or a sensor), connectors (CNS, e.g., data exchange between components), ports/roles ($Prts$ and Rls, i.e., interfaces through which components and connectors, respectively, are attached).

Each view element has a finite number of types, with four primitive types for CMS, CNS,

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

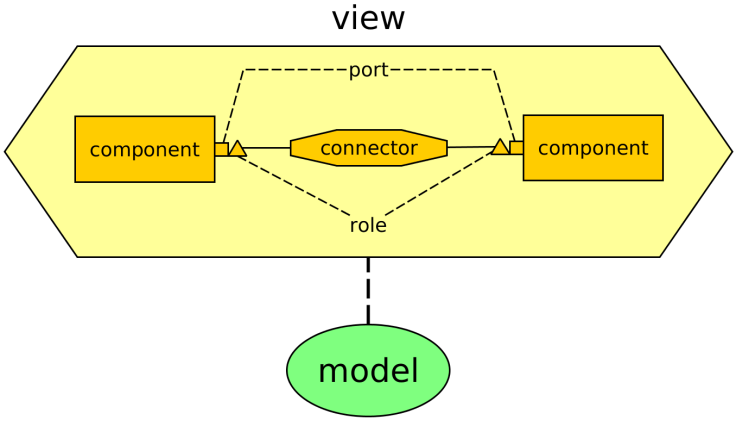


Figure 6.4: Elements of architectural views.

$Prts$, and Rls. Organized in hierarchies of inheritance (or, extension), architectural types are used by system engineers to distinguish elements of different nature. For instance, cyber types (e.g., a software thread) represent digitally interacting entities, while physical types (e.g., a motor) represent objects present in 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 $\mathbb{E}^{\mathcal{V}}$:

$$CMS \cup CNS \cup Prts \cup Rls \equiv \mathbb{E}^{\mathcal{V}}, \tag{6.4}$$

Sets CMS, CNS, $Prts$, Rls are pairwise disjoint.

In this thesis, view elements use the standard containment hierarchy of architectural languages: 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 view elements with concrete values. 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 $\mathbb{T}$): $\mathsf{P} : \mathbb{E}^{\mathcal{V}} \mapsto \mathbb{O}$. Syntactically, I denote the value of a property PROP of an element e as e.PROP. Properties are often declared for architectural types, thus the property becomes applicable to all instances of that type. The property values, however, 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.

6.2.2 Integration Viewpoints

Now I turn to using views as abstractions of models. Recalling Definition 3 in Section 5.2, a model contains model elements $\mathbb{E}^{\mathcal{M}}$, information about which needs to be represented in views.

To relate models and views, I introduce the concept of an *integration viewpoint*. This concept is inspired by the ISO standard 42010 [128] for architectural description, which uses viewpoints

as descriptions of each view's (or person's) perspective. A similar interpretation of viewpoints has appeared in seminal works in software architecture and cyber-physical systems [32, 51, 83, 117].

In the context of model integration, a viewpoint consists of several entities that relate the elements in models and views. Intuitively, integration viewpoints serve two functions:

1. Define the aspects of the model that need to be extracted as view elements. For example, a timing viewpoint would describe what information is related to timing in models: delays, deadlines, execution times, and so on.
2. Create views from models. For example, a timing viewpoint would create a timing view from a model with timing-related information.

Multiple viewpoints are often necessary to represent realistic systems. In this thesis I treat viewpoints separately from each other, without considering their combinations or interactions.

From a model's perspective, a viewpoint describes and uses certain parts or objects in that model, which are called *model elements* ($\mathbb{E}^{\mathcal{M}}$). These elements vary widely depending on the formal language of the model: they can be blocks, modules, statements, or certain values in the model. Scoping the relevant model elements is also a responsibility of a viewpoint. The models that are seen as collections of model elements are called structural.

View creation with viewpoints consists of two steps. The first step of creating a view is to indicate which model elements are relevant and how they should related to the to-be-created view elements. The second step is to execute an algorithm that produces a view from a model. In the rest of this section I describe these two steps.

I start with describing how viewpoints relate model and view elements, which is accomplished using matching predicates. A matching predicate describes a relation between tuples of model and view elements. The sizes of model and view tuples are a pair of integers that are called the *dimensions* of a matching predicate. In simple cases, both tuples can be of size one, and such matching predicates describe one-to-one relations between model and view elements. In more sophisticated cases, matching predicates can have larger dimensions as needed. For instance, a matching predicate can relate a pair of locations to a task that moves a robot between them. Below, I handle the general case of matching predicates with arbitrary fixed dimensions.

Definition 13 (Viewpoint Matching Predicate)**.** Given a view $\mathcal{V}$ with elements $\mathbb{E}^{\mathcal{V}}$, a structural model $\mathcal{M}$ with elements $\mathbb{E}^{\mathcal{M}}$, and a pair of natural numbers k and l, a *viewpoint matching predicate* mp is a predicate over a tuple of view elements $\boldsymbol{e}^{\mathcal{V}} \equiv (e_1^{\mathcal{V}} \dots e_k^{\mathcal{V}}), e_i^{\mathcal{V}} \in \mathbb{E}^{\mathcal{V}}, i \in [1, k]$ and a tuple of model elements $\boldsymbol{e}^{\mathcal{M}} \equiv (e_1^{\mathcal{M}} \dots e_l^{\mathcal{M}}), e_i^{\mathcal{M}} \in \mathbb{E}^{\mathcal{M}}, i \in [1, l]$:

$$\mathrm{mp} : (\mathbb{E}^{\mathcal{V}})^n \times (\mathbb{E}^{\mathcal{M}})^m \to \mathbb{B},$$

where $\mathbb{B}$ is the set of boolean values (truth $\top$ and falsehood $\bot$).

Given a view $\mathcal{V}$ and a model $\mathcal{M}$, a matching predicate mp indicates the *intention* of how model elements $\mathbb{E}^{\mathcal{M}}$ and view elements $\mathbb{E}^{\mathcal{V}}$ should be related, $\mathrm{mp} : \mathcal{P}\left(\mathbb{E}^{\mathcal{M}}\right) \times \mathcal{P}\left(\to\right)\mathbb{B}$. Therefore, the goal of mp is to abstractly prescribe how model elements should be related to view elements, thus playing the role of a specification for view construction. To express a matching predicate in a closed-form logical formula, one can use view- and model-related constructs. On the view side, matching predicates can be expressed using properties and types of view elements. On the model side, matching predicates can be expressed using model-specific functions and predicates.

Here is an illustrative example of matching between elements in models and views. Suppose a map model containing physical locations is represented with a view that shows all possible motion tasks of a robot on that map. Each pair of locations in the model relates to a pair of motion tasks in the map: a task to move from one location to another, and another task for the opposite direction. In this case, a $(2, 1)$-dimensional matching predicate can be defined over a pair of locations (model elements) (l_1, l_2) and a motion task (view element) t as follows:

$$\mathrm{mp}((t), (l_1, l_2)) \equiv \mathrm{adjacent}(l_1, l_2) \wedge t.start = l_1 \wedge t.end = l_2. \tag{6.5}$$

As Equation (6.5) indicates, not every pair of locations should be related to a task: only the adjacent locations can be traversed via a single task. Similarly, not every task should be related to a pair of location with mp, but only the tasks that connect adjacent locations.

A view can have multiple matching predicates, so describe different relationships between its elements and model elements. In the example of map locations and tasks, one may want to describe a $(3, 2)$-dimensional predicate over triplets of locations and corresponding pairs of tasks:

$$\begin{aligned}\mathrm{mp}((l_1, l_2, l_3), (t_{1\to2}, t_{2\to3})) \equiv\ & \mathrm{adjacent}(l_1, l_2) \wedge \\ & \mathrm{adjacent}(l_2, l_3) \wedge t_{1\to2}.start = l_1 \wedge t_{1\to2}.end = l_2 \wedge \\ & t_{2\to3}.start = l_2 \wedge t_{2\to3}.end = l_3.\end{aligned} \tag{6.6}$$

This thesis considers views with a finite number n of matching predicates, which constitute a set $\mathbb{MP} = \{\mathrm{mp}_1 \ldots \mathrm{mp}_n\}$.

The second step for a viewpoint is executing an algorithm to generate a view from a model. This viewpoint algorithm creates view elements and annotates them with types and properties. In practice, creation of views can be performed manually (according to well-defined guidelines) or automated by implementing the algorithm.

Definition 14 (Viewpoint Algorithm)**.** Given a set of structural models $\mathbb{M}$ and a set of their possible views $\mathbb{V}$ that share a view signature $\Sigma^{\mathcal{V}}$, a *viewpoint algorithm* (VA) is a function that represents an algorithm of deriving a view from a given model:

$$\mathsf{VA} : \mathbb{M} \to \mathbb{V}.$$

For any model $\mathcal{M} \in \mathbb{M}$, a viewpoint algorithm creates a view $\mathcal{V} = \mathsf{VA}(\mathcal{M})$ given a vocabulary of types and properties specified in $\Sigma^{\mathcal{V}}$. The viewpoint algorithm decides which model and view elements are matched, to satisfy matching predicates, as described in the next subsection. Definition 14 makes viewpoints unambiguous: when applied to a given model, a viewpoint produces only one[4] view. An example of a potentially ambiguous viewpoint would be an underspecified manual method of mapping Simulink blocks to view components [25]. Representing viewpoints as functions does not limit the generality of the approach: if a model requires several views, they can be created with different viewpoints.

As an example, if $\mathcal{M}$ with map locations has a set of two adjacent locations $\mathbb{E}^{\mathcal{M}} = \{l_1, l_2\}$, then a view $\mathcal{V} = \mathsf{VA}(\mathcal{M})$ with tasks $\mathbb{E}^{\mathcal{V}} = \{t_{1\to2}, t_{2\to1}\}$ will be created. The properties of $\mathbb{E}^{\mathcal{V}}$ are assigned in a way that satisfies Equation (6.5).

[4]View can and should differ between viewpoints.

A finite set of matching predicates and a viewpoint algorithm together comprise an integration viewpoint. Thus, a viewpoint contains both a specification and an implementation for creation of views from models.

Definition 15 (Integration Viewpoint)**.** Given a set of structural models $\mathbb{M}$ and a set of views $\mathbb{V}$ with the same view signature $\Sigma^{\mathcal{V}}$, an *integration viewpoint* is a tuple $(\mathbb{MP}, \mathsf{VA})$, where $\mathbb{MP}$ is a set of n viewpoint matching predicates $\{\mathrm{mp}_1 \ldots \mathrm{mp}_n\}$, and VA is a viewpoint algorithm that maps $\mathbb{M}$ to $\mathbb{V}$.

6.2.3 Conformance, Soundness, and Completeness of Views

A basic relation between a model and a view is *conformance*, which holds when the view was created from the model by some viewpoint. Whenever views or models change, it is important to restore conformance, as discussed below in Subsection 6.2.5.

Definition 16 (Model-View Conformance)**.** A view $\mathcal{V}$ *conforms* to a model $\mathcal{M}$ with respect to a viewpoint $\mathcal{VP} = (\mathbb{MP}, \mathsf{VA})$ if $\mathcal{V} = \mathsf{VA}(\mathcal{M})$.

However, conformance is insufficient for reliable reasoning about models using views. In particular, without additional requirements, valid IPL statements over views do not necessarily translate into guarantees about model elements. Two potential issues may occur. First, some of the view elements might not match model elements (in terms of matching predicates, $\mathbb{MP}$), leading to false-positive universally-quantified IPL formulas. Such views are called *incomplete*. Second, a view might contain extraneous elements that are not matching any model elements, leading to false-positive existentially-quantified IPL formulas. Such views are called *unsound*.

To rule out unsound and incomplete views, additional constraints on views are necessary. In order to distinguish views based on how well they represent model elements, I introduce two characteristics: *soundness* and *completeness*. The intuitive intent behind these characteristics is shown in Figure 6.5. These notions are expressed without explicitly prescribing how exactly model elements are related to view elements: since relations between these model/view elements can be complex and differ between viewpoints, assuming a particular form would be limiting to the customizability of the approach. Hence, soundness and completeness of views is described in terms of matching predicates.

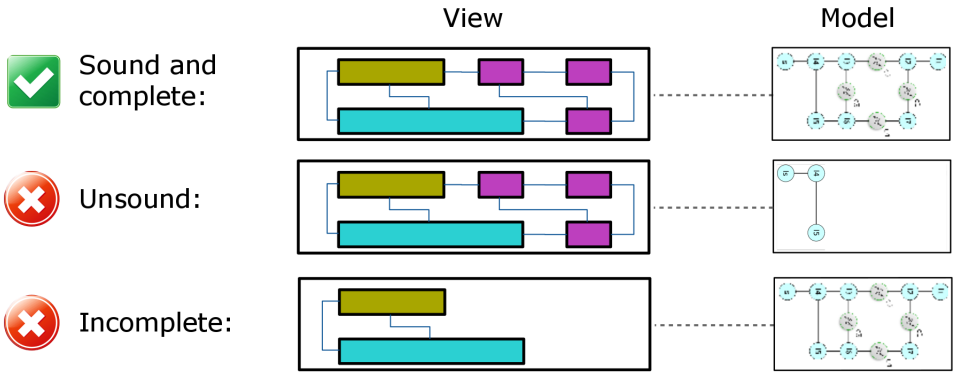


Figure 6.5: The visual intuition behind soundness and completeness of views.

Intuitively, a view is *sound* with respect to a matching predicate mp if any tuple of view elements is matched via mp to some tuple of model elements. A sound view is meant to guarantee that every element that is assumed to be in the model (based on a view) is indeed there. In other words, every element of a sound view is an adequate proxy for some model elements. For instance, every CPU in a sound view maps to an actual CPU in a hardware model. This mapping is established using mp, in tuples of elements of sizes according to the dimensions of mp.

Definition 17 (View Soundness)**.** Given a matching predicate mp with dimensions (k, l), a view $\mathcal{V}$ (containing elements $\mathbb{E}^{\mathcal{V}}$) of a model $\mathcal{M}$ (containing elements $\mathbb{E}^{\mathcal{M}}$) is *sound* with respect to mp if any k-tuple $\boldsymbol{e}^{\mathcal{V}}$ of view elements matches (via mp) some l-tuple $\boldsymbol{e}^{\mathcal{M}}$ of model elements:

$$\mathrm{sound}(\mathcal{V}, \mathcal{M}, \mathrm{mp}) \equiv \ \forall \boldsymbol{e}^{\mathcal{V}} : (\mathbb{E}^{\mathcal{V}})^k \ \exists \boldsymbol{e}^{\mathcal{M}} : (\mathbb{E}^{\mathcal{M}})^l \cdot \mathrm{mp}(\boldsymbol{e}^{\mathcal{V}}, \boldsymbol{e}^{\mathcal{M}}).$$

If a view is sound, it is known that every view element is accounted for by at least one model element. In the map and task example, soundness of a task view with respect to the predicate in Equation (6.5) would mean that every task in the view is matchable to a pair of locations in the map model. Thus, soundness guarantees that only tasks corresponding to actual adjacent map locations are part of the view.

Now, I introduce completeness of views. Intuitively, a view is complete with respect to some matching predicate mp if every tuple of model elements is matched via mp a to tuple of view elements. For instance, a view for a hardware model is complete if it represents all CPUs from the model. The definition below formalizes this intuition.

Definition 18 (View Completeness)**.** Given a matching predicate mp with dimensions (k, l), a view $\mathcal{V}$ (containing elements $\mathbb{E}^{\mathcal{V}}$) of a model $\mathcal{M}$ (containing elements $\mathbb{E}^{\mathcal{M}}$) is *complete* with respect to mp if any l-tuple $\boldsymbol{e}^{\mathcal{M}}$ of model elements matches via mp to an k-tuple $\boldsymbol{e}^{\mathcal{V}}$ of view elements:

$$\mathrm{complete}(\mathcal{V}, \mathcal{M}, \mathrm{mp}) \equiv \ \forall \boldsymbol{e}^{\mathcal{M}} : (\mathbb{E}^{\mathcal{M}})^l \ \exists \boldsymbol{e}^{\mathcal{V}} : (\mathbb{E}^{\mathcal{V}})^k \cdot \mathrm{mp}(\boldsymbol{e}^{\mathcal{V}}, \boldsymbol{e}^{\mathcal{M}}).$$

The definition of view completeness demands that every model tuple is matched to a tuple in a view. For example, to have complete views with tasks that account for every pair of adjacent locations, the following matching predicate can be used together with the definition of completeness:

$$\mathrm{mp}((t), (l_1, l_2)) \equiv \ \mathrm{adjacent}(l_1, l_2) \ \rightarrow t.start = l_1 \wedge t.end = l_2. \tag{6.7}$$

It is possible to put additional constraints on matching between model and view elements. For example, every mapping can be made unique: only one tuple can be allowed to map to a given tuple. Such a constraint would be necessary if view functions in IPL (VFUNC) include counting of elements (hence, multiple copies of the same elements should be prohibited). If IPL uses only properties and types, such constraints are not needed. Another possible change is allowing view tuples of multiple dimensions. This change may be needed in complex views with differently-sized tuples, but it is not necessary in the simple case of a fixed tuple size. In the general case, however, the above notions of soundness and completeness are sufficient for reasoning about models using views, as demonstrated in Subsection 6.2.6.

Similar concepts of view soundness and completeness have been studied in prior work [25], but their meaning differs from the above. 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 the base architecture is that views can be completely agnostic of the models that 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 Section 4.4). If a model is precisely and comprehensively represented by a view, then any operation on a view is equivalent to the same operation on the model. 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 view requirements are related to the overall integration argument later in this chapter, in Subsection 6.2.6.

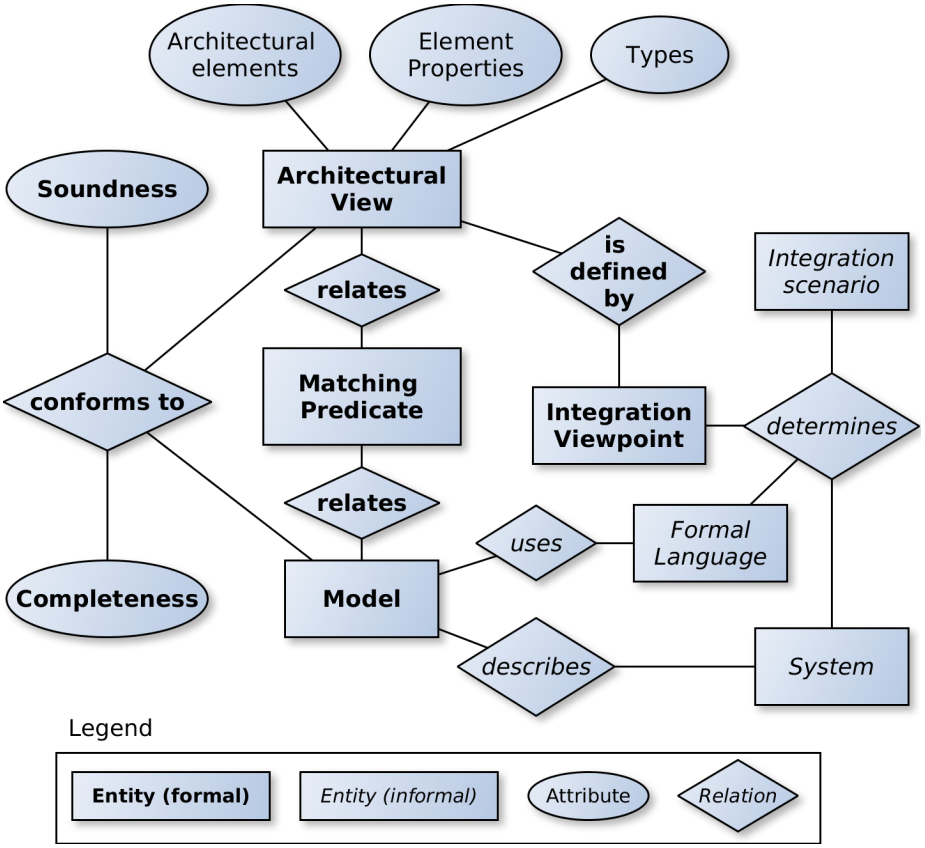


Figure 6.6: View abstraction concepts. Bold means formalized, italicized means informal.

The concepts of a view and a viewpoint are summarized in an entity-relationship diagram in Figure 6.6. In short, the integration viewpoint is determined by the integration approach, the formalism/formal language of the model, 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.

Next, I illustrate how non-trivial views are defined on the example of hybrid programs. Later in this section, I discuss the strategies for automatic creation and updating of views, and conclude this section with the model-view part of the integration argument.

6.2.4 View Abstractions for Hybrid Programs

To illustrate using views as abstractions, I developed a viewpoint $\mathcal{VP}^{\text{hp}}$ for hybrid programs. The goal of this viewpoint is to create an HP view ($\mathcal{V}^{\text{hp}}$) that represents the primary agents of a hybrid programs and their interactions, along with relevant properties (such as a control algorithm and a set of physical equations).

The first step in relating hybrid programs and views is defining elements of models and views. In this context, a model elements is a hybrid programs (α) drawn from a set of all possible hybrid programs ($\mathbb{HP}$) over a given set of variables ($Vars$). These programs which are arbitrarily nested formulas of the HP syntax presented in Table 6.1.

View elements in $\mathcal{V}^{\text{hp}}$ are organized in three tiers:

1. *HP actors:* typed components that modularize hybrid programs into several independent subprograms. These components can be seen as relations to multiple sub-programs that are stored in their properties.
2. *HP connectors:* connectors between components that represent operations between the HP actors. These connectors can be seen as relations between HP actors, or between sub-programs that characterize the actors.
3. *HP composers:* operations that compose multiple actors into a hybrid program. These composers can be seen as relations between tuples of HP actors and a hybrid program.

A viewpoint is defined with matching predicates for each of the above tiers. The algorithm for $\mathcal{VP}^{\text{hp}}$ is sketched in this section, and its further discussion can be found below, in Subsection 6.2.5. Below I formalize the three tiers of HP view elements.

Definition 19 (Hybrid Program Actor)**.** A *hybrid program actor* (hpa) is a component instance that is characterized by a tuple:

$$hpa \equiv (State, Prts, ctrl, phys).$$

The state of hpa, or actor for short, is a combination of variables and constraints: $State \equiv (Vars, Constr)$, where $Vars \subseteq V$ is set of a typed[5] variables drawn from a fixed set V, and $Constr \equiv \{\varphi_i\}$ is a set of state constraint formulas, defined by Equation (6.2), over variables in $Vars$. For example, for a robot that moves in a one dimension (along a line), $State \equiv (\{x, v, a, o\}, \{o \in \{-1, 1\}\})$.

A port is an external interface of an actor – a variable that is used in interaction between actors. Ports contain variables of an actor that are externally exposed. It is not required that $Vars \cap Prts = \emptyset$: a port p may expose a state variable ($p \in Vars$) or define its own ($p \notin Vars$). For example, if a robot is sensing an obstacle's x coordinate, I 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.

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

An actor's control is a hybrid program: $ctrl \in \mathbb{HP}$. It is defined by HP operators over variables in $Vars$ and $Prts$, and describes computations executed by the actor. An actor's physics is also a hybrid program, but constrained to be a set of differential equations with an evolution domain constraint: $phys \equiv \{x_i' = \theta_i \& F\}$ over variables in $Vars$ and $Prts$. The goal of separating the physical dynamics from the control is that the former is repeated in many model variants, while the latter may be more specific the variant's combination of concerns. A separate collection of physical dynamics would enable their independent analysis and reuse.

Definition 20 (Hybrid Program Connector)**.** Given a set of actors $\{hpa_i\}$, ***HP connector*** hpc is a connector instance that is characterized by a tuple:

$$(Rls, Rtp, \mathsf{Trf}).$$

Roles $Rls \equiv \{r_i\}$ 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: $Rtp \equiv Rls \to \bigcup hpa_i.Prts$. The transformation function Trf captures the connector's effect on the attached actors so that the connector can be reused in multiple model variants with different actors. Unlike Rls and Rtp that define *what a connector is*, Trf defines *how the connector operates*. Formally, Trf maps a set of actors and their attachments to a set of new actors, which were changed by the transformation (Trf) to reflect its effects:

$$\mathsf{Trf} : (hpa_i)^n \times Rls \times Rtp \to \{hpa_i\}.$$

Consider a simple *Immediate Precise Sensing Connector (IPSC)*, which senses the precise value of a variable and returns the result immediately. It has two roles: $Rls = \{\text{sense}, \text{sensed}\}$. 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 generates a_1' from a_1 and a_2' from a_2.[6] IPSC replaces the readings of variable p_1 in a_1 with readings of p_2 in $a_1.ctrl$:

$$IPSC.\mathsf{Trf}((a_1, a_2), Rls, Rtp) \equiv (a_1', a_2') \text{ s.t.} \tag{6.8}$$

$$\begin{aligned}
&a_1'.State = a_1.State, && a_2'.State = a_2.State,\\
&a_1'.Prts = a_1.Prts \setminus \{p_1\}, && a_2'.Prts = a_2.Prts \setminus \{p_2\},\\
&a_1'.ctrl = a_1.ctrl\{p_1/p_2\}, && a_2'.ctrl = a_2.ctrl,\\
&a_1'.phys = a_1.phys, && a_2'.phys = a_2.phys.
\end{aligned}$$

To enable automated generation of HPs from views, the gap between HP actors and hybrid programs needs to be bridged. To this end, I compose the actors until there is a single mega-actor, which then generates a HP. There are, however, several ways to compose actors. Therefore, I encapsulate a mechanism of composition in a *composer* (similar to component glue [20], director [165], and coordinator [35] in related work):

Definition 21 (Hybrid Program Composer)**.** A *hybrid program composer* cpr is a pair $(\mathsf{Compose}, \mathsf{ToHP})$, where $\mathsf{Compose}$ is a function that maps a tuple of several actors into one actor: $(hpa_i)^n \to \{hpa_i\}$, and ToHP is a function that maps hpa to a hybrid program.

[6] I use $\alpha\{a/b\}$ to mean substitution of a for b in HP α.

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 general, Compose can be arbitrarily complex. In this thesis, I focus on the *sequential composer* $SeqC$ that is implicitly used throughout all collision avoidance models in the original materials for my study [193]. This composer orders the executions of actors in a given sequence:

$$SeqC.\mathsf{Compose}(a_1, \ldots, a_n : hpa) \equiv a' \text{ s.t.}$$

$$\begin{aligned} a'.State &= a_1.State \cup \cdots \cup a_n.State, \\ a'.Prts &= a_1.Prts \cup \cdots \cup a_n.Prts, \\ a'.ctrl &= a_1.ctrl; \ldots; a_n.ctrl, \\ a'.phys &= \{a_1.phys, \ldots, a_n.phys\}. \end{aligned} \tag{6.9}$$

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

$$SeqC.\mathsf{ToHP}(a) \equiv (a.ctrl; a.phys)^* \tag{6.10}$$

HP actors, connectors, and a composer constitute a hybrid program view:

Definition 22 (Hybrid Program View)**.** A *HP view* $\mathcal{V}^{\text{hp}}$ is a tuple $(\{hpa_i\}, \{hpc_i\}, cpr)$.

Using the above concepts, an HP view conforms to a hybrid programs if that view's actors and connectors, when composed by the view's cpr, synthesize a program that is equivalent to the given one. This intuition is formalized below using several mapping predicates.

The matching predicate for HP actors (used in view soundness) requires that an actor only uses the variables and ports from its subprograms and that its parts map to certain hybrid sub-programs $\alpha_1, \alpha_2, \alpha_3$:

$$\begin{aligned} &\text{mp}_{hpa}(hpa, \alpha_1, \alpha_2, \alpha_3) \equiv \\ &\quad hpa.State.Vars \cup hpa.Prts = \text{vars}(\alpha_1 \cup \alpha_2 \cup \alpha_3) \wedge \\ &\quad hpa.State.Constr = \alpha_1 \ \wedge hpa.ctrl = \alpha_2 \ \wedge hpa.phys = \alpha_3, \end{aligned}$$

where vars is a set of all variables present in a hybrid program.

To define view completeness with respect to HP actors, one would need to relax the conjunctions in the above definition to become disjunctions. The reason for that is that different sub-programs can belong to different actors, hence the above predicate would not be satisfied. When satisfied, that completeness definition would indicate that a view contains all the information of the program, in addition to the knowledge of how this information is distributed among view elements.

The matching predicate for HP connectors matches between views with connectors and views without connectors (after applying the connector transformations), based on the definition of the Trf function:

$$\text{mp}_{hpc}((hpa_1 \ldots hpa_n), hpc, hpa) \equiv hpc.\mathsf{Trf}(hpa_1 \ldots hpa_n) = hpa.$$

Finally, the matching predicate for the HP composer in the view uses both composer's function to match to a given hybrid program α:

$$\mathrm{mp}_{cpr}(cpr, (hpa_1 \ldots hpa_n), \alpha) \equiv cpr.\mathsf{ToHP}(cpr.\mathsf{Compose}(hpa_1 \ldots hpa_n)) = \alpha.$$

These matching predicates can be used together to generate a hybrid program if $\mathcal{V}^{\text{hp}}$ is given, leading to a constructive definition of conformance for HP views:

Definition 23 (Conformance for HP Views)**.** An HP view $\mathcal{V}^{\text{hp}}$ *conforms* to a hybrid program α (i.e., $\mathcal{V}^{\text{hp}} = \mathcal{VP}(\alpha)$), if

$$cpr.\mathsf{ToHP}(cpr.\mathsf{Compose}(\mathsf{TC}(\{hpa_i\}, \{hpc_i\}))) = \alpha.$$

Notice that the above definition creates a model from a given view, whereas typically a viewpoint algorithm creates a view from the model (see Definition 14). Thus, $\mathcal{VP}^{\text{hp}}$ specifies the inverse algorithm, $(\mathsf{VA}^{\text{hp}})^{-1}$, in Definition 23. The viewpoint itself is defined with the direct model-to-view algorithm:

Definition 24 (HP Viewpoint)**.** The *HP viewpoint* ($\mathcal{VP}^{\text{hp}}$) is a combination of the matching predicates for HP actors, connectors, and composers with an algorithm inversed to the one in Definition 23:

$$\mathcal{VP}^{\text{hp}} \equiv (\mathsf{VA}^{\text{hp}}, (\mathrm{mp}_{hpa}, \mathrm{mp}_{hpc}, \mathrm{mp}_{cpr})).$$

While the algorithm for view-to-model transformation for hybrid programs is well-defined, it may be difficult to invert automatically without additional assumptions. The next subsection explores the practical aspects of creating and maintaining conforming views.

6.2.5 Automating View Creation and Conformance

Generally, defining and maintaining views for 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 elements. 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 change as well, so that the 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 12. Many such models are (trivially) sound and complete views of themselves. 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 use in IPL verification. Thus, the matching predicates are either constantly true, or over a homogeneous of a view and a model, making it easy to implement a viewpoint algorithm that is guaranteed to produce a sound and complete view.

The second special case is when, for every element type in $\mathbb{T}$ (see Definition 12), a viewpoint implements an incremental transformation using a model and a partially constructed view. The availability of a partial view reduces the non-determinism regarding how matching predicates should be satisfied to a level that can be handled automatically. 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, making it straightforward to write and satisfy matching predicates. This is usually the case 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 [25]. 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.

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, an integration viewpoint $\mathcal{VP}$ needs to be defined. It requires a set of matching predicates 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). These predicates are specific to each viewpoint, allowing viewpoints to utilize the same parts of the same model differently. Suppose that precise (although not necessarily machine-readable) descriptions of these predicates exist, and it is known for each predicate if the view needs to be sound, complete, or both.

When implementing viewpoint algorithms, two options[7] 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 I discuss these options and exemplify Option (i) with views for hybrid programs. Examples that illustrate Option (ii) are reviewed in Chapter 8.

As an example, consider hybrid program CollAvoid (Equation (6.11)) 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 (RobotCtrl and ObstCtrl) and their continuous physics (RobotPhys and ObstPhys) execute in an indefinite loop. During robot's turn, its controller (Equation (6.12)) 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 because it contains complex hybrid dynamics and a logical statement over these dynamics, and is not explicitly separated into components or connectors.

$$\begin{aligned}\textsc{CollAvoid} \equiv |x_r - x_o| > \Delta \rightarrow [\ (&\textsc{RobotCtrl}; \textsc{ObstCtrl};\\ &(\textsc{RobotPhys}, \textsc{ObstPhys})\ )* \ ](x_r \neq x_o),\end{aligned} \tag{6.11}$$

$$\text{where } \textsc{RobotCtrl} \equiv (a := -b \ \cup \ (?|x_r - x_o| > v^2/b;\ a := *)). \tag{6.12}$$

A view conforming to the above model is shown in Figure 6.7. It factors the control and physics of 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 CollAvoid. Thus, the view represents a hybrid program (without inherent component structure) in terms of the vocabulary of HP actors (components) and sensing between them (connectors).

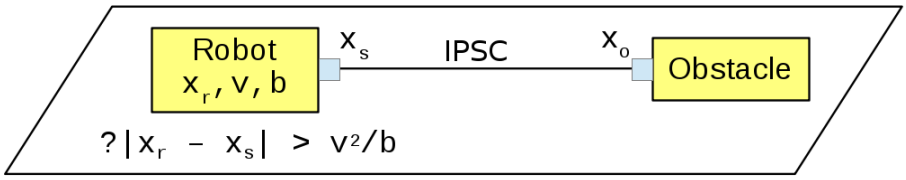


Figure 6.7: A view for the *CollAvoid* program.

An appropriate model (Equation (6.11)) 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 with other models and views: a view could be a coarse abstraction and, while conforming to a model, miss some details required for synthesizing

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

a complete model. In such cases, this approach should be to generate a template of a model, which will need to be completed manually.

Generally, the process of synthesizing a model from a view consists of 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. Option (i) follows these steps:

1. Create an empty model in the formal language prescribed by the viewpoint.
2. Instantiate all model elements determined by the view's components, according to the matching predicates. For instance, in Simulink, this step may include instantiating the controller and plant subsystems.
3. Instantiate all model elements or code determined by the view's connectors, according to the matching predicates. This may require altering the existing model elements, depending on the model language.
4. Set the direct properties (those corresponding one-to-one between model and view elements) in the model according to the view.
5. Verify that the emergent properties (those with many-to-many relations between model and view elements) 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 [25]). This step includes determination of properties required by the elements, and leads to formalizing matching predicates.
2. Determine the instances of components (with ports) that populate the view based on the model elements ($\mathbb{E}^{\mathcal{M}}$).
3. Determine connector instances between components (as well as specific roles, if necessary). The connectors may encode relations between model elements (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 elements (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 elements).

Co-evolution

Co-evolution of views and models means maintaining conformance whenever a view or a model changes. Therefore, co-evolution is necessary in two scenarios. Consider the first scenario: a model $\mathcal{M}$ changes (becomes $\mathcal{M}'$), and a previously conforming view $\mathcal{V}$ needs to be updated to a view $\mathcal{V}'$ that conforms to $\mathcal{M}'$. Assume that the viewpoint $\mathcal{VP}$ is fixed and implemented.

I envision two strategies to construct $\mathcal{V}'$. The first one is generating a fresh view $\mathcal{V}' = \mathcal{VP}(\mathcal{M}')$ by applying the viewpoint algorithm, thus abandoning $\mathcal{V}$ 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 potential options for implementing this strategy via bootstrapping are described above.

The second strategy (for the first scenario) applies when view generation is computationally intensive or not fully automatic (e.g., may require subjective judgment of an expert to determine how to satisfy the matching predicates). The strategy is to reuse parts of $\mathcal{V}$ to simplify the generation of $\mathcal{V}'$, and it proceeds in two steps. The first step is to identify and localize the model elements that *differ* between $\mathcal{M}$ and $\mathcal{M}'$. For instance, in an HP for a robot and an obstacle, if only the robot's control changes in $\mathcal{M}'$, the physical dynamics and the obstacle's control can remain the same in $\mathcal{V}'$ as in $\mathcal{V}$. The second step is to generate elements of $\mathcal{V}'$ that correspond to only updated elements of $\mathcal{M}'$. In the third and final step, these new view elements are merged with the unchanged elements of $\mathcal{V}$, thus forming $\mathcal{V}'$. 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 $\mathcal{V}$ changes (becomes $\mathcal{V}'$), and the previous $\mathcal{M}$ needs to be updated to $\mathcal{M}'$ such that $\mathcal{VP}(\mathcal{M}') = \mathcal{V}'$. This is a more challenging scenario because $\mathcal{V}'$ 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 6.2.5), 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 $\mathcal{V}$ and $\mathcal{V}'$. The second step is to generate a template for $\mathcal{M}'$ from $\mathcal{V}'$, filling in the syntax from model elements in $\mathcal{M}$ that correspond to the identical elements of $\mathcal{V}$ and $\mathcal{V}'$. This approach is not guaranteed to produce a fully-specified $\mathcal{M}'$, may limit its practicality. A fully-specified $\mathcal{M}'$ could be created for the updates from $\mathcal{V}$ to $\mathcal{V}'$ that preserve the mapping between the view and model elements.

Changes of the viewpoint are not examples of co-evolution. Instead, when the viewpoint changes, one needs to follow the bootstrapping process, as described earlier in this subsection.

6.2.6 Integration Argument with Views

Now I examine views as integration abstractions from the standpoint of the integration argument laid out in Section 4.4. Views are used by IPL as proxies of structural models: constraints on view elements indirectly constrain model elements through matching predicates and soundness/completeness of views. The outputs of IPL verification are, therefore, dependent on how accurately view elements represent the underlying model elements.

In this section, I assume that the other links of the integration argument work as intended:

- The IPL verification produces sound outputs (see Theorem 1 in Section 5.6).
- Behavioral queries produce sound outputs and always terminate (see Subsection 6.3.3 for the integration argument with behavioral properties).

In the context of connecting views and models, the integration argument is concerned with three characteristics of views:

- *Expressiveness:* the ability to (i) encode static elements ($\mathbb{E}^{\mathcal{M}}$) of various models in view

elements ($\mathbb{E}^{\mathcal{V}}$), matching them with mp, and (ii) encode first-order logical constraints on $\mathbb{E}^{\mathcal{V}}$, implying the constraints on $\mathbb{E}^{\mathcal{M}}$ placed by the $\mathrm{integprop}$ predicate.

- *Soundness:* according to Definition 17, sound views do not contain elements that do not correspond to any model elements.
- *Completeness:* according to Definition 18, complete views do not fail to represent any model elements.

The challenge of creating expressive, sound, and complete views lies in the complexity of the mapping between view and model elements. For simple matching predicates, 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 precisely 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.

Expressiveness of views is determined by three factors: the language of views themselves, the native rigid sub-language of IPL, and the custom types used in a particular view. Within my approach, the language of views is drawn from the family of architecture description languages (e.g., Acme or AADL), allowing for hierarchical modeling of static finite name-value data [182]. The annotations are key-based finite values of common data types (like integers or strings) or lists/sets of values. The IPL syntax generally allows for first-order constraints on any information related to $\mathbb{E}^{\mathcal{V}}$ in general architecture languages (existence, types, property values), see Section 5.3 for details. The view element types determine convenient distinctions between groups of view elements, supporting their mapping to $\mathbb{E}^{\mathcal{M}}$. View element types are typically specific to a particular integration scenario. For instance, the actor subtypes *robot* and *obstacle* are only used for the set of models concerned with obstacle avoidance for mobile robots. The view types do not limit expressiveness of views in general, since any desired grouping of view elements can be constructed in some view.

The expressiveness of views is intentionally limited (e.g., to finite sets of view elements), so that the saturation procedure in the IPL verification (see Section 5.5) can be performed efficiently. Another reason for constraining expressiveness of views is to enable dependency resolution for analyses based on view element types (presented later in Section 7.3). Thus, views are a relatively inexpressive abstraction compared to behavioral properties.

In case the view language is not expressive enough in some scenario, this shortcoming becomes apparent when the view cannot capture the information from the model elements in a way intended by mp. When the rigid IPL constraints on views are not expressive enough, it would be manifested as inability to constrain the view elements in a way that the ground-truth property $\mathrm{integprop}$ constrains model elements.

Reasoning about Models with Sound and Complete Views

Soundness and completeness of views play a role in logically connecting structural models to IPL properties through view elements. Below I demonstrate this connection in a theorem that allows to draw conclusions about model elements while checking IPL statements about view elements. For the purposes of this subsection, I consider behaviors as constants and do not explicitly show

them as parameters of the ground-truth integration property ($\mathrm{integprop}$) and IPL formulas. Thus, while $\mathrm{integprop}$ and IPL formulas depend on behaviors, here I treat $\mathrm{integprop}$ and IPL formulas as predicates over view elements only.

Suppose a structural model $\mathcal{M}$ has a view $\mathcal{V} = \mathsf{VA}(\mathcal{M})$ with respect to a viewpoint $\mathcal{VP} = (\{\mathrm{mp}_1 \dots \mathrm{mp}_n\}, \mathsf{VA})$. A tuple $\boldsymbol{e}^{\mathcal{V}} = (e_1^{\mathcal{V}} \dots e_k^{\mathcal{V}})$ is a vector of view element variables from some subsets of the view's elements $\mathbb{E}^{\mathcal{V}}$, and the size of $\boldsymbol{e}^{\mathcal{V}}$ is $\left|\boldsymbol{e}^{\mathcal{V}}\right| = k$. Similarly, $\boldsymbol{e}^{\mathcal{M}} = (e_1^{\mathcal{M}} \dots e_l^{\mathcal{M}})$ is a tuple of model element variables from some subsets of the model's elements $\mathbb{E}^{\mathcal{M}}$, with size $\left|\boldsymbol{e}^{\mathcal{M}}\right| = l$. The quantification domains of these variables are elided below for brevity.

The goal is to demonstrate some ground-truth integration property $\mathrm{integprop}(\mathbb{E}^{\mathcal{M}})$, which also depends on another model's behaviors that are assumed to be fixed in this section. This integration property can be written as a quantified expression (with some quantifiers $Q_1 \dots Q_n$ over vectors $\boldsymbol{e}_1^{\mathcal{M}} \dots \boldsymbol{e}_n^{\mathcal{M}}$)[8] over the model's variables over some predicate ip:

$$Q_1 \boldsymbol{e}_1^{\mathcal{M}} \dots Q_n \boldsymbol{e}_n^{\mathcal{M}} \cdot \mathrm{ip}(\boldsymbol{e}_1^{\mathcal{M}} \dots \boldsymbol{e}_n^{\mathcal{M}}). \tag{6.13}$$

This is a statement that needs to be shown to hold, given certain conditions outlined below and formalized in a theorem. Such statements are not directly checkable over arbitrary models, hence the need for abstraction via views.

Similarly, consider an IPL formula $\textsc{formula}(\mathbb{E}^{\mathcal{V}})$, which also depends on another model's behaviors that are assumed to be fixed. This IPL formula can be written as a quantified formula over some predicate ipl:

$$Q_1 \boldsymbol{e}_1^{\mathcal{V}} \dots Q_n \boldsymbol{e}_n^{\mathcal{V}} \cdot \mathrm{ipl}(\boldsymbol{e}_1^{\mathcal{V}} \dots \boldsymbol{e}_n^{\mathcal{V}}). \tag{6.14}$$

This is a rewritten general form of IPL statements, which can be verified as described in Chapter 5. In the following, the above statement is assumed to hold.

Finally, consider a proof obligation that connects ip, ipl, and $\{\mathrm{mp}_1 \dots \mathrm{mp}_n\}$:

$$\forall \boldsymbol{e}_1^{\mathcal{V}} \dots \boldsymbol{e}_n^{\mathcal{V}}, \boldsymbol{e}_1^{\mathcal{M}} \dots \boldsymbol{e}_n^{\mathcal{M}} \cdot \mathrm{ipl}(\boldsymbol{e}_1^{\mathcal{V}} \dots \boldsymbol{e}_n^{\mathcal{V}}) \wedge \bigwedge_{i=1..n} \mathrm{mp}_i(\boldsymbol{e}_i^{\mathcal{V}}, \boldsymbol{e}_i^{\mathcal{M}}) \rightarrow \mathrm{ip}(\boldsymbol{e}_1^{\mathcal{M}} \dots \boldsymbol{e}_n^{\mathcal{M}}). \tag{6.15}$$

This proof obligation requires that a conjunction of the IPL formula and matching predicates leads to the integration property. While this statement is given in a general form above, in some cases it can be simplified due to the same quantifiers for sequential tuples and the same matching predicates. In practice, such proof obligations are often straightforwardly shown using propositional proof rules, without the need for verifying them on models.

Additional assumptions for inferring Equation (6.13) link the quantifiers and soundness and completeness of view $\mathcal{V}$ with respect to the matching predicates. As the theorem's proof will show, to reason about a universal quantifier with index i, $\mathcal{V}$ is required to be complete with respect to the matching predicate mp_i. To reason about an existential quantifier with index i, $\mathcal{V}$ is required to be sound with respect to the matching predicate mp_i.

Theorem 2 (Sufficiency for Model-View Reasoning)**.** In order to conclude that $\mathrm{integprop}(\mathbb{E}^{\mathcal{M}})$ (Equation (6.13) holds for some structural model $\mathcal{M}$, view $\mathcal{V}$, and some set of matching predicates $\{\mathrm{mp}_1 \dots \mathrm{mp}_n\}$, it is sufficient to establish the following five conditions:

[8] A quantifier Q over a vector $\boldsymbol{e} = (e_1 \dots e_n)$ is equivalent to writing $Qe_1 \dots Qe_n$.

- The IPL formula holds, $\text{FORMULA}(\mathbb{E}^{\mathcal{V}})$ (Equation (6.14)).[9].
- The model-view proof obligation holds (Equation (6.15)).
- If mp_i has dimensions (k, l), then $\left|\boldsymbol{e}_i^{\mathcal{V}}\right| = k$ and $\left|\boldsymbol{e}_i^{\mathcal{M}}\right| = l$ for $i = 1..n$.
- If $Q_i \equiv \forall$, then view $\mathcal{V}$ is complete with respect to mp_i.
- If $Q_i \equiv \exists$, then view $\mathcal{V}$ is sound with respect to mp_i.

Proof. Assume, for contradiction, that Equation (6.13) does not hold:

$$\neg Q_1 \boldsymbol{e}_1^{\mathcal{M}} \dots Q_n \boldsymbol{e}_n^{\mathcal{M}} \cdot \text{ip}(\boldsymbol{e}_1^{\mathcal{M}} \dots \boldsymbol{e}_n^{\mathcal{M}}). \tag{6.16}$$

Now I construct a pair of vectors that lead to a contradiction: a vector of model elements $\hat{\boldsymbol{e}}^{\mathcal{M}}$ and a vector of view elements $\hat{\boldsymbol{e}}^{\mathcal{V}}$. They will be constructed by unwrapping the quantifiers in Equations (6.16) and (6.14).

Consider two alternative cases of Q_1:

- If $Q_1 \equiv \forall$, then advance the negation in Equation (6.16):

  $$\exists \boldsymbol{e}_1^{\mathcal{M}} \cdot \neg Q_2 \boldsymbol{e}_2^{\mathcal{M}} \dots Q_n \boldsymbol{e}_n^{\mathcal{M}} \cdot \text{ip}(\boldsymbol{e}_1^{\mathcal{M}} \dots \boldsymbol{e}_n^{\mathcal{M}}). \tag{6.17}$$

  In the above equation, instantiate $\boldsymbol{e}_1^{\mathcal{M}}$ to some value $\hat{\boldsymbol{e}}_1^{\mathcal{M}}$. Since $Q_1 \equiv \forall$, the theorem's premise requires that view $\mathcal{V}$ is complete with respect to mp_1 (see Definition 18). Notice that the size of $\hat{\boldsymbol{e}}_1^{\mathcal{M}}$ is the same as the second dimension of mp_1. Therefore, due to completeness, there exists some view element vector $\hat{\boldsymbol{e}}_1^{\mathcal{V}}$ (of the size equal to the first dimension of mp_1) such that the following holds:

  $$\text{mp}_1(\hat{\boldsymbol{e}}_1^{\mathcal{V}}, \hat{\boldsymbol{e}}_1^{\mathcal{M}}).$$

  Now, using Equation (6.14), instantiate $\forall \boldsymbol{e}_1^{\mathcal{V}}$ to $\hat{\boldsymbol{e}}_1^{\mathcal{V}}$, leading to the following:

  $$Q_2 \boldsymbol{e}_2^{\mathcal{V}} \dots Q_n \boldsymbol{e}_n^{\mathcal{V}} \cdot \text{ipl}(\hat{\boldsymbol{e}}_1^{\mathcal{V}}, \boldsymbol{e}_2^{\mathcal{V}} \dots \boldsymbol{e}_n^{\mathcal{V}}).$$

- If $Q_1 \equiv \exists$, then using Equation (6.14), instantiate $\exists \boldsymbol{e}_1^{\mathcal{V}}$ to some value $\hat{\boldsymbol{e}}_1^{\mathcal{V}}$ for which the following holds:

  $$Q_2 \boldsymbol{e}_2^{\mathcal{V}} \dots Q_n \boldsymbol{e}_n^{\mathcal{V}} \cdot \text{ipl}(\hat{\boldsymbol{e}}_1^{\mathcal{V}}, \boldsymbol{e}_2^{\mathcal{V}} \dots \boldsymbol{e}_n^{\mathcal{V}}).$$

  Since $Q_1 \equiv \exists$, according to the theorem's premise, view $\mathcal{V}$ is sound with respect to mp_1. Notice that the size of $\hat{\boldsymbol{e}}_1^{\mathcal{V}}$ equals to the first dimension of mp_1. Thus, soundness means that there exists such a model element vector $\hat{\boldsymbol{e}}_1^{\mathcal{M}}$ that the following is true:

  $$\text{mp}_1(\hat{\boldsymbol{e}}_1^{\mathcal{V}}, \hat{\boldsymbol{e}}_1^{\mathcal{M}}).$$

  In Equation (6.16), advance the negation to obtain the following:

  $$\forall \boldsymbol{e}_1^{\mathcal{M}} \cdot \neg Q_2 \boldsymbol{e}_2^{\mathcal{M}} \dots Q_n \boldsymbol{e}_n^{\mathcal{M}} \cdot \text{ip}(\boldsymbol{e}_1^{\mathcal{M}} \dots \boldsymbol{e}_n^{\mathcal{M}}).$$

  In the above equation, instantiate $\forall \boldsymbol{e}_1^{\mathcal{M}}$ to $\hat{\boldsymbol{e}}_1^{\mathcal{M}}$, leading to the following:

  $$\neg Q_2 \boldsymbol{e}_2^{\mathcal{M}} \dots Q_n \boldsymbol{e}_n^{\mathcal{M}} \cdot \text{ip}(\hat{\boldsymbol{e}}_1^{\mathcal{M}}, \boldsymbol{e}_2^{\mathcal{M}} \dots \boldsymbol{e}_n^{\mathcal{M}}).$$

[9]Notice that the same quantifiers $Q_1 \dots Q_n$ are used in Equation (6.13)

As the result of the first step, regardless of the quantifier, it is possible to unwrap it and obtain a pair of vectors $\hat{\boldsymbol{e}}_1^{\mathcal{V}}$ and $\hat{\boldsymbol{e}}_1^{\mathcal{M}}$ that satisfy the matching predicate and the equations above, which are the same in both cases.

The above process is repeated for every outer quantifier. After the i-th step, I have obtained vectors $\hat{\boldsymbol{e}}_1^{\mathcal{V}} \dots \hat{\boldsymbol{e}}_i^{\mathcal{V}}$ and $\hat{\boldsymbol{e}}_1^{\mathcal{M}} \dots \hat{\boldsymbol{e}}_i^{\mathcal{M}}$ that satisfy the following:

$$\bigwedge_{j=1..i} \mathrm{mp}_j(\hat{\boldsymbol{e}}_j^{\mathcal{V}}, \hat{\boldsymbol{e}}_j^{\mathcal{M}}),$$
$$Q_{i+1}\boldsymbol{e}_{i+1}^{\mathcal{V}} \dots Q_n\boldsymbol{e}_n^{\mathcal{V}} \cdot \mathrm{ipl}(\hat{\boldsymbol{e}}_1^{\mathcal{V}} \dots \hat{\boldsymbol{e}}_i^{\mathcal{V}}, \boldsymbol{e}_{i+1}^{\mathcal{V}} \dots \boldsymbol{e}_n^{\mathcal{V}}),$$
$$\neg Q_{i+1}\boldsymbol{e}_{i+1}^{\mathcal{M}} \dots Q_n\boldsymbol{e}_n^{\mathcal{M}} \cdot \mathrm{ip}(\hat{\boldsymbol{e}}_1^{\mathcal{M}} \dots \hat{\boldsymbol{e}}_i^{\mathcal{M}}, \boldsymbol{e}_{i+1}^{\mathcal{M}} \dots \boldsymbol{e}_n^{\mathcal{M}}).$$

After completing all the steps and unwrapping all n quantifiers, the vectors $\hat{\boldsymbol{e}}^{\mathcal{M}} = (\hat{\boldsymbol{e}}_1^{\mathcal{M}} \dots \hat{\boldsymbol{e}}_n^{\mathcal{M}})$ and $\hat{\boldsymbol{e}}^{\mathcal{V}} = (\hat{\boldsymbol{e}}_1^{\mathcal{V}} \dots \hat{\boldsymbol{e}}_n^{\mathcal{V}})$ satisfy the following assertions:

$$\bigwedge_{i=1..n} \mathrm{mp}_i(\hat{\boldsymbol{e}}_i^{\mathcal{V}}, \hat{\boldsymbol{e}}_i^{\mathcal{M}}), \tag{6.18}$$
$$\mathrm{ipl}(\hat{\boldsymbol{e}}_1^{\mathcal{V}} \dots \hat{\boldsymbol{e}}_n^{\mathcal{V}}), \tag{6.19}$$
$$\neg\, \mathrm{ip}(\hat{\boldsymbol{e}}_1^{\mathcal{M}} \dots \hat{\boldsymbol{e}}_n^{\mathcal{M}}). \tag{6.20}$$

Now, returning to Equation (6.15), I instantiate all the quantified variables with values from $\hat{\boldsymbol{e}}^{\mathcal{V}}$ and $\hat{\boldsymbol{e}}^{\mathcal{M}}$. Since the antecedents of that implication are satisfied (Equations (6.18) and (6.19)), modus ponens leads to the following:

$$\mathrm{ip}(\hat{\boldsymbol{e}}_1^{\mathcal{M}} \dots \hat{\boldsymbol{e}}_n^{\mathcal{M}}).$$

The above assertion contradicts Equation (6.20), leading to the conclusion that the predicate ip holds for the quantifiers $Q_1 \dots Q_n$:

$$Q_1\boldsymbol{e}_1^{\mathcal{M}} \dots Q_n\boldsymbol{e}_n^{\mathcal{M}} \cdot \mathrm{ip}(\boldsymbol{e}_1^{\mathcal{M}} \dots \boldsymbol{e}_n^{\mathcal{M}}).$$

□

A stronger version of this theorem would include necessity, making the ipl satisfied if and only if ip is satisfied. This is possible when the proof obligation in Equation (6.15) is an equivalence instead of an implication. Whether this equivalence applies depends on the matching predicates and how closely ipl mimics ip.

This theorem directly extends to multiple models and views: each matching predicate can relate an arbitrary pair of models and views, as long as the quantification domains of the respective variables are restricted to the elements of those models and views.

To summarize, the sufficient conditions described above support the model-view link of the integration argument (Section 4.4). To apply this theorem in an integration scenario, the integration property, the IPL formula, and the matching predicates need to be chosen to support the proof obligation (Equation (6.15)). Also, for each matching predicate, the view needs to be either sound or complete, depending on the quantifiers used in the formula. Finally, IPL verification needs to indicate that the IPL formula holds.

6.3 Behavioral Integration Abstractions: 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 [154] 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 describe the requirements on using these languages 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 27), although developing approaches to such composition is outside of the scope of this thesis.

6.3.1 Behavioral Languages and Queries

In the formalization below, I build on the definitions of a behavioral model (Definition 4) in the IPL preliminaries (Section 5.2). I start with defining a behavioral property language — a way to express model-specific properties, which can be syntactically plugged into IPL (explained earlier in Subsection 5.3.1).

Definition 25 (Behavioral Property Language)**.** Given a model signature $\Sigma^{\mathcal{M}}$ and a set of free variables V, a *behavioral property language* $\mathbb{L}$ for a given model is a (potentially infinite) set of formulas over the model's symbols ($\Sigma^{\mathcal{M}}$), rigid atoms[10] $\text{RATOM}_1 \ldots \text{RATOM}_k$ over V, and language-specific operators. These formulas are evaluated according to the language's semantics ($[\![]\!]^{\mu}_{\Omega}$), given an assignment $\mu \in \Theta$ of variables in V and a trace set Ω of a model $\mathcal{M}$ for $\Sigma^{\mathcal{M}}$. A *behavioral property* l is a sentence in $\mathbb{L}$. The semantic evaluation maps each sentence to some known domain $\mathbb{O}$ (boolean, integer, or real):

$$l \in \mathbb{L},$$
$$[\![l]\!]^{\mu}_{\Omega} : \mathbb{L} \times \Theta \to \mathbb{O}.$$

In the running example of this chapter (introduced in Section 6.1), $\mathsf{d}\mathcal{L}$ is an example of a behavioral language ($\mathbb{L}$), and an example of a behavioral property (l) is the property in Equation (6.3) that states that the robot eventually reaches its goal: $l_{robot} \equiv x < g \to \langle \alpha_{robot} \rangle (x \geq g)$. Here, x and g are free variables and terms of this behavioral property. Therefore, $\Theta \equiv \mathbb{R}^2$, where $\mathbb{R}$ is the set of reals. While such statements can be analyzed without knowing the particular values of x and g, in this thesis the variable values need to be provided for evaluating this formula. For the hardware model, logical constraints on the hardware architecture can be considered an instance of $\mathbb{L}$.

Definition 26 (Behavioral Query)**.** Given a model $\mathcal{M}$, a behavioral language $\mathbb{L}$, and a set of possible values Θ of free variables V, a *behavioral query* Q is a function that computes the value

[10]Rigid atoms are part of the IPL syntax that can be embedded in behavioral properties. The atoms are defined in Definition 8 in Section 5.3. They are evaluated according to the semantics of IPL, given in Section 5.4.

of a formula $l \in \mathbb{L}$ on a trace set Ω (a member of set $\mathbb{\Omega}$ in $\mathcal{M}$) for some valuation $\mu \in \Theta$. The value of a formula belongs to some known domain $\mathbb{O}$. The goal of Q is to evaluate the sentence according to the semantics of $\mathbb{L}$, but errors and timeouts may occur.

$$\mathsf{Q} : \mathbb{L} \times \mathbb{\Omega} \times \Theta \to \mathbb{O}.$$

Behavioral queries that return boolean values can represent verification of models — a common set of behavioral queries, including model checking and theorem proving. For instance, determining whether a robot gets to a goal, as specified by l_{robot}, given its program α_{robot}, is an example of a query. The traces are, in this case, the set of trajectories admitted by α_{robot}, and the initial position and goal are given concrete values $\hat{x}$ and $\hat{g}$:

$$\mathsf{Q}(l_{robot}, \alpha_{robot}, (\hat{x}, \hat{g})) \equiv \big(\alpha_{robot} \models \hat{x} < \hat{g} \to \langle \alpha_{robot} \rangle (\hat{x} \geq \hat{g})\big).$$

If the above holds, $\top$ is returned (otherwise, $\bot$).

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

The concepts from this section are summarized in an entity-relationship diagram in Figure 6.8. Behavioral properties and queries are connected to the overall integration approach below, in Subsection 6.3.3.

6.3.2 Behavioral Property Abstractions for Hybrid Programs

The language of differential dynamic logic ($\mathsf{d}\mathcal{L}$) serves as a property language for hybrid programs. Each such formula is either valid or invalid, enabling validity queries with a boolean output. To further raise the abstraction level, one 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 27. A *$\mathsf{d}\mathcal{L}$ view formula* over HP views $\mathcal{V}_1^{\text{hp}}, \ldots, \mathcal{V}_n^{\text{hp}}$ is a $\mathsf{d}\mathcal{L}$ formula over variables from these views $V_1 \cup \cdots \cup V_n$, parametric terms $Constr_1 \ldots Constr_n$, and hybrid programs $\mathcal{V}_1^{\text{hp}} \ldots \mathcal{V}_n^{\text{hp}}$.

Given several HP views, one can express a property in a formula that combines these views, their state variables (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 $\mathsf{Compose}(\mathsf{TC}(\mathcal{V}_i^{\text{hp}})).State.Constr$ and replace HP view references $\mathcal{V}_i^{\text{hp}}$ with their synthesized hybrid program code according to Definition 23. The view formulas are equivalent to $\mathsf{d}\mathcal{L}$ in expressiveness, but provide a more abstract way to specify behavioral properties over multiple programs. Just like $\mathsf{d}\mathcal{L}$, queries over this language have a binary output ("valid" and "not valid").

Unlike HPs, hardware models in the running example do not come with a "native" language for properties. However, it is possible to use a hypothetical language similar to the Object

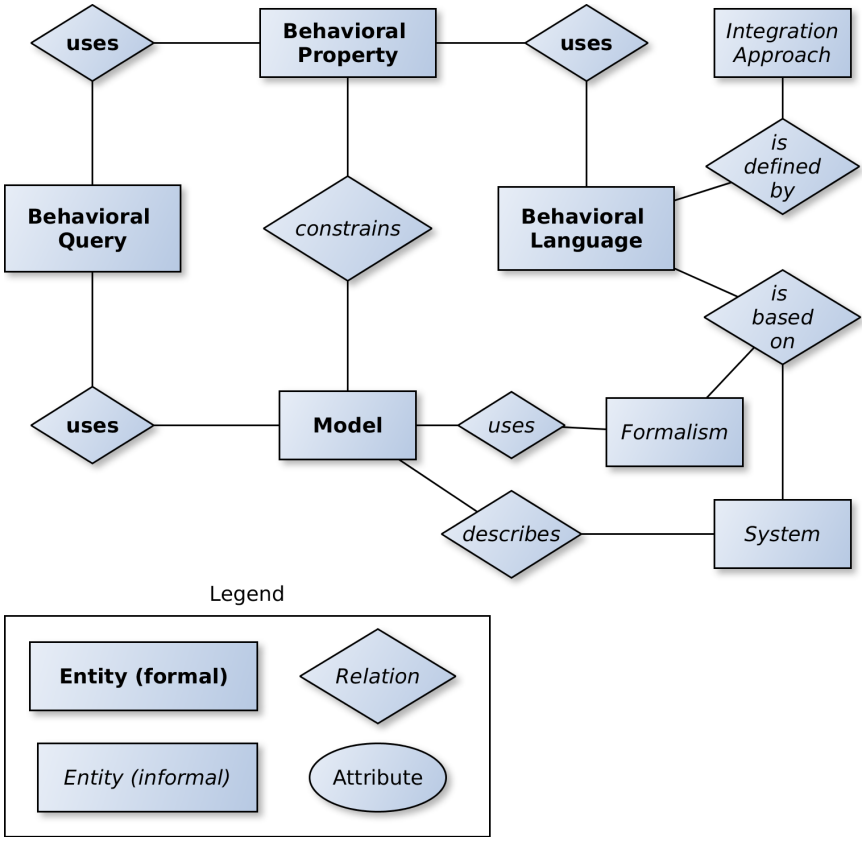


Figure 6.8: Behavioral property abstraction concepts. Bold is formalized, italicized is informal.

Constraint Language (OCL) for UML [108]. The language expresses propositions over hardware models, 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 $cpu_1.frequency \geq cpu_2.frequency$. This language, however, does not have a notion of behavior or modal state in it, and thus can be treated as a set of constraints over views.

6.3.3 Integration Argument with Model Querying

Now I describe how a query mechanism is used as a behavioral abstraction of models. Its role is to link the IPL verification process and behaviors in models. A behavioral abstraction of a model $\mathcal{M}$ is a combination of a behavioral language $\mathbb{L}$ and an implementation of a query mechanism Q that evaluates sentences on $\mathcal{M}$ according to the semantics of $\mathbb{L}$. The query mechanism receives requests in the model querying step of the IPL verification algorithm (Section 5.5), processes them by interacting with the behavioral model $\mathcal{M}$, and returns values to the algorithm.

Queries receive three arguments. One argument is a set of values μ for free variables V. These values are drawn from some set Θ. In the overall approach, these values are obtained in the saturation step of the IPL verification algorithm. For instance, in Property 2 (Subsection 5.3.4) a robot goes through three sequential tasks, which are represented with quantified variables t_1, t_2, t_3.

These variables are bound to concrete tasks elements from $\mathcal{V}_{\text{power}}$. The variable values μ determine the values of the other two query arguments.

Another argument of behavioral query is a formula from $\mathbb{L}$ with rigid atoms that contain free variables. These atoms are bound to concrete values using the IPL semantics, based on the valuation μ of V. The resulting formula does not have any free variables, and can be evaluated by the language-specific semantics.

The final argument of a behavioral query is a set of traces. This set is selected from the parametric structure Ω of $\mathcal{M}$. To select a set, IPL passes the values of model parameters, which are decided based on the rigid terms in the MDLINST clause and variable assignments μ. The set of model parameters is a member of the set PF, which contains all name-value functions for model parameters. The values of model parameters are mapped to a concrete trace set Ω by the model interpretation $I^{\mathcal{M}}$ (used in the first sense of Definition 4).

In the overall integration argument (explained in Section 4.4), behavioral properties are used to access traces Ω in the ground-truth integration property $\text{integprop}(\mathbb{E}^{\mathcal{M}}, \Omega)$, where $\mathbb{E}^{\mathcal{M}}$ are elements in structural models, which are accessed through views. Below I consider $\mathbb{E}^{\mathcal{M}}$ a fixed set, thus integprop can be considered as a predicate over Ω only. I also assume that the other links of the integration argument (laid out in Section 4.4 and detailed in Section 8.1) work as intended:

- The IPL verification produces sound outputs (see Theorem 1 in Section 5.6).
- Assertions over view elements reliably lead to assertions over model elements in structural models (see Subsection 6.2.6).

Model integration is affected by three characteristics of behavioral queries and languages: behavioral property expressiveness, query soundness, and query termination. Below I define these characteristics and the requirements placed on them by the integration approach.

Definition 28 (Expressiveness of Behavioral Properties)**.** *Expressiveness of behavioral properties* is the capacity of a behavioral language to re-state a given predicate (integprop) over behaviors using a finite number of behavioral properties. Specifically, the predicate integprop over a set of traces Ω of some model $\mathcal{M}$ should be equivalent to some predicate $\text{integprop}'$ over behavioral properties $l_1 \dots l_m \in \mathbb{L}$ that are evaluated over variable values $\mu_1 \dots \mu_m \in \Theta$ and trace sets $\Omega_1 \dots \Omega_m$. Here, each Ω_i is the result of mapping of some parameter name-value function to the parametric trace structure Ω of $\mathcal{M}$.

$$\text{integprop}(\Omega) \iff \text{integprop}'([\![l_1]\!]^{\mu_1}_{\Omega_1} \dots [\![l_m]\!]^{\mu_m}_{\Omega_m}). \tag{6.21}$$

For instance, if integprop requires a robot to finish its mission, then $\mathbb{L}$ should support querying temporal relations of the robot's state variables. Another way to understand expressiveness is the ability to write two queries that differ only in a narrow way (e.g., whether the robot reached the finish with an empty battery or not) and return different results on the same trace set.

Definition 29 (Soundness of Behavioral Queries)**.** A behavioral query Q is *sound* if it returns values that match the semantic evaluation of the behavioral property in the language's semantics. The rigid atoms $\text{RATOM}_1 \dots \text{RATOM}_k$ evaluated according to the IPL semantics. This is a requirement on the computation that performs the query of a sentence $l \in \mathbb{L}$ over some trace set Ω and free variable values μ:

$$\mathsf{Q}(l, \Omega, \mu) = [\![l([\![\text{RATOM}_1]\!]_\mu \dots [\![\text{RATOM}_k]\!]_\mu)]\!]^{\mu}_{\Omega}.$$

For instance, if a query $\mathsf{Q}(\mathcal{M}_{hw}, "mass(*)")$ returns a number that is not equal to the total mass in $\mathcal{M}_{hw}$, this query would be unsound. Soundness of queries is a specialization of the general conformance concept from Figure 6.1: queries in a behavioral language "conform" to the model if they return the correct answers according ot its semantics.

Definition 30 (Termination of Behavioral Queries)**.** A behavioral query Q *terminates* if it returns a value from the property's value domain $\mathbb{O}$ in a finite amount of time. This requirement constrains the computation that performs Q: if it does not terminate or returns "unknown," this property fails, preventing the IPL verification process from concluding with a definite outcome.

These three requirements are challenging to satisfy at once because they are often conflicting. It is difficult to create a decision procedure for a highly expressive language — let alone an efficient procedure. Sound reasoning requires eliminating the possibility of an incorrect result, which may lead to an infinite computation. On the other hand, guaranteed termination may lead to constraints or approximations that may lead to incorrect results.

Expressiveness of behavioral properties is scenario-specific because it is relative to the predicate $\mathrm{integprop}$. My approach generally assumes that it is possible to find a finite number of behavioral properties $l_1 \ldots l_m$ that, once evaluated on specific trace sets, would extract behavior information that is sufficient to determine whether $\mathrm{integprop}(\Omega)$ holds. In cases when $\mathbb{L}$ is not expressive enough (e.g., classical LTL cannot perform counting), this shortcoming should become apparent at the stage of writing IPL specifications, the meaning of which would not match the meaning of the $\mathrm{integprop}$ predicate.

Once $\mathbb{L}$ is embedded into IPL, all flexible IPL subformulas (with MDLINST as the outer construct) can be translated into $\mathbb{L}$ by providing the values of free variables. This capability follows directly from embedding $\mathbb{L}$ as a plugin of IPL: the values of rigid sub-expressions of l can be computed based on free variable values and transferred to the domain of $\mathbb{L}$. Therefore, it is guaranteed that behavioral queries will be well-formed. The above holds for the IPL plugins of LTL and PCTL (see Subsections 5.3.2 and 5.3.3).

Unlike expressiveness, soundness and termination can often (but not always) be evaluated theoretically and a priori. The most desirable outcome is that any queries in $\mathbb{L}$ for $\mathcal{M}$ are guaranteed to terminate and be sound, which leads to the query mechanism working sufficiently well for the integration argument in Section 4.4. However, it might not always be the case; for instance , model checking within a bounded state-space [27] is guaranteed to terminate, but cannot guarantee soundness when an unbounded state-space is considered.

If a priori evaluation for the full $\mathbb{L}$ is not possible, it may be possible to consider individual queries for a given integration property. Specifically, these are the queries of behavioral properties $l_1 \ldots l_m$ in Equation (6.21). If every query from that finite set used by $\mathrm{integprop}'$ terminates and returns a sound result, this behavioral abstraction is sufficient the overall integration argument.

In the running example, query soundness for hybrid programs is supported by correct theorem proving: if a proof of a theorem exists (e.g., that a robot reaches the goal Equation (6.3)), 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. Thus, differential dynamic logic ($\mathsf{d}\mathcal{L}$) is an expressive and sound behavioral abstraction for hybrid programs. While the proof theory for hybrid systems is complete [213], in practice there is no guarantee that a proof would be found, so automated theorem proving might not satisfy

the termination condition. The OCL-like language for the hardware model enables abstraction through customized queries and predicate logic formulas (limited expressiveness), which can be implemented to be sound with guarantees of termination.

To summarize, this section showed that behavioral properties and queries provide a black-box interface to a model's behaviors, without placing assumptions on the model's syntactic structure. Three integration-relevant conditions of queries are expressiveness, soundness, and termination.

6.3.4 Shared Background between Abstractions

One of the assumptions in IPL is that the view interpretations ($I^{\mathcal{V}}$) and model interpretations ($I^{\mathcal{M}}$) do not contradict each other over the shared symbols, thus forming a single consistent background interpretation (I^{B}). This condition is necessary for successful domain transfer — exchange of values between views and behavioral properties as part of the IPL verification (Subsection 5.4.1). 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 belong to common sets, such as integers or reals, and the domain transfer happens trivially. In some cases, however, the exchanged values carry the meaning of identifiers or references to elements in models or views. For example, a behavioral query may return an identifier of the most energy-consuming task of a robot. In such cases, it is necessary to ensure that the interpretation of references in the view agrees with that in the behavioral property. Even though the references may be represented as integers, their meaning is components. An example of such a case can be found in the case study of System 3 (Subsections 8.2.2 and 8.3.3): thread IDs are exchanged between views (which represent threads with components) and models (which represent threads with 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 22)	$\mathsf{d}\mathcal{L}$ formulas (Equation (6.2)), $\mathsf{d}\mathcal{L}$ view formulas (Definition 27)
Hardware model (Section 6.1)	Hardware architecture view (Definition 12)	OCL-like constraint formulas (example before Subsection 6.3.3)

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

In practice, when faced with integration scenarios, engineers need to choose 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 types of elements in views are used to specify dependencies between analyses (as further explained in the next chapter); for instance, an analysis might take the threads, which is a type of view elements, as an input.

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 are *convenient abstractions* in the following circumstances:

- The model relies on 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.
- The desired relations between model and view elements can be described by simple matching predicates with little non-determinism.
- The view has relatively few elements (on the order of dozens), which would simplify the saturation process in the IPL verification.
- The viewpoint has an automated implementation, to reduce the effort of creating views.
- The viewpoint can be applied unambiguously and automatically to generate views or models in a negligible time.
- Multiple analyses have overlapping dependencies over one or several models.

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

- The model relies on a formalism that is difficult to map to the components and connectors.
- The model's infinite or implicit elements (e.g., all possible trajectories of a robot on a given map) need to be exposed for integration.
- The desired relations between model and view elements are described by complex and highly non-deterministic matching predicates.
- The view has relatively many elements (on the order of hundreds and more).
- The viewpoint needs to be designed and implemented from scratch.
- The viewpoint's application is a manual, ambiguous, and time-consuming process.

Behavioral properties are *convenient abstractions* 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 an expressive 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 *difficult to use* in the following circumstances:

- The model elements need to be directly 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 or expressive 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 presented views and behavioral properties as integration abstractions of models. Each of the abstractions has three characteristics that affect its usefulness for integration with IPL: expressiveness, soundness, and completeness for views, and expressiveness, soundness, and termination for behavioral queries. Without these characteristics, the correctness of integration is threatened, as further addressed in Section 8.1. 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 Summary

This chapter describes the second part of the integration approach. This part focused on two abstractions: view for structural models, and behavioral properties for behavioral models. These abstractions were formalized and exemplified in this chapter. Further, this chapter provided a formal link between the models and abstractions, thus enabling inconsistency checking via IPL verification. For views, assertions over view elements imply certain related assertions over model elements. For behavioral abstractions, sound queries of models can evaluate behavioral properties as functions of model elements, in a way that agrees with the model semantics.

Chapter 7

Part III: Analysis Execution Platform

This chapter presents the third and final technical part of this thesis — the *Analysis Execution Platform* (AEP). This platform supports the third step of the approach proposed in Chapter 4 by executing multiple analyses in a correct way. To this end, the platform manages execution of analyses, making sure that their dependencies are respected and the execution contexts are appropriate. 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 to specify what execution context is appropriate for an analysis, in the form of assertions over multiple models in the context of the analysis. 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 8.4.

7.1 Domain Signatures and Analysis Contexts

To specify contracts for analyses, one first needs to define the formal basis that is used to capture 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 differentiate two parts of the execution context: the part that remains the same after the execution, and the part that changes due to the execution.

I address the above requirement by separating the *signatures* (symbols for specification) of models/views and the *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 the 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 5.3) and semantics (Section 5.4).

Consider a set of models ($\mathbb{M}$) and a set of views ($\mathbb{V}$) that conform to these models. Each model $\mathcal{M}$ in $\mathbb{M}$ contains a signature ($\Sigma^{\mathcal{M}}$), an interpretation ($I^{\mathcal{M}}$), and a structure (Γ), as defined in Definition 4 (Section 5.2). Each view $\mathcal{V}$ in $\mathbb{V}$ contains a signature ($\Sigma^{\mathcal{V}}$), an interpretation ($I^{\mathcal{V}}$), and a structure of architectural elements ($\mathbb{E}^{\mathcal{V}}$), as defined in Definition 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 31 (Domain Signature)**.** A *domain signature* (Σ) is a tuple of model and view signatures: $(\Sigma_1^{\mathcal{M}} \ldots \Sigma_n^{\mathcal{M}}, \Sigma_1^{\mathcal{V}} \ldots \Sigma_m^{\mathcal{V}})$.

Intuitively, a domain is a set of related concepts, definitions, and types in some application (e.g., thread scheduling) where several analyses can operate. For example, a view signature for a hardware view (e.g., one introduced in Section 6.1) would have threads ($Thrds$) and CPUs ($CPUs$) as type labels for architectural elements. Views also contain properties, which are represented as functions of architectural elements, such as thread periods ($\mathsf{Per} : Thrds \mapsto \mathbb{Z}$) and processor frequencies ($\mathsf{CPUFreq} : CPUs \mapsto \mathbb{Z}$). A model signature for a battery model can contain a symbol for its current charge (a state variable). These signatures can belong to a domain that represents the effects of scheduling on power consumption.

Characterized by the signatures of related models ($\Sigma^{\mathcal{M}}$) and views ($\Sigma^{\mathcal{V}}$), a domain signature stays fixed over the execution of multiple analyses. Model signatures provide syntactic elements for specifying appropriate contexts, in terms of important aspects of structure and behavior of the models. Thus, the context contains state variables (e.g., battery charge) and modal functions, which have interpretations change with state (and are thus determined by a modal interpretation, $I^{\mathcal{M}}$, on model structure $\Gamma^{\mathcal{M}}$). Using modal functions is a more complex way to encode the state of a model, where each state determines a concrete function mapped to a symbol. For instance, preemption between threads is a dynamic attribute of a system, encoded as a function $\mathrm{canprmt}$. 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. Both view and model signatures also contain standard sorts (such as Booleans $\mathbb{B}$ and integers $\mathbb{Z}$) in a background signature (Σ^B), since it is shared between models and views. The symbols for standard operators like addition ($+ : \mathbb{Z} \times \mathbb{Z} \mapsto \mathbb{Z}$) are part of the background signature. The model and view interpretations contain background interpretations (I^B) that interprets such symbols over background structures (Γ^B).

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: $(\mathsf{M}_{\text{sch}}, \mathsf{M}_{\text{sec}}, \mathsf{M}_{\text{cpu}}, \mathsf{M}_{\text{rek}}, \mathsf{V}_{\text{sch}}, \mathsf{V}_{\text{sec}}, \mathsf{V}_{\text{cpu}}, \mathsf{V}_{\text{rek}})$. 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 32 (Analysis Context)**.** For a given domain signature $\Sigma = (\Sigma_1^{\mathcal{M}} \dots \Sigma_n^{\mathcal{M}}, \Sigma_1^{\mathcal{V}} \dots \Sigma_m^{\mathcal{V}})$, an *analysis context* (Υ) is a tuple of the following pairs: for each view signature, Υ contains a pair of a view interpretation $I^{\mathcal{V}}$ and a view structure $\Gamma^{\mathcal{V}}$; and for each model signature, Υ contains a pair of a model interpretation $I^{\mathcal{M}}$ and a model structure $\Gamma^{\mathcal{M}}$:

$$\Upsilon = ((I_1^{\mathcal{V}}, \Gamma_1^{\mathcal{V}}) \dots (I_n^{\mathcal{V}}, \Gamma_n^{\mathcal{V}}), (I_1^{\mathcal{M}}, \Gamma_1^{\mathcal{M}}) \dots (I_m^{\mathcal{M}}, \Gamma_m^{\mathcal{M}})).$$

An analysis context represents the parts of system design, in terms of view types, that can be changed by the analysis. The context is tied to a specific domain, which is 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 $Thrds$ in $\mathcal{V}$, it can be interpreted as $I^{\mathcal{V}}(Thrds) = \{t_1, t_2, t_3\}$. The periods of these threads (in milliseconds) are determined by a function Per, which can be interpreted as follows: $I^{\mathcal{V}}(\mathsf{Per}) = \{t_1 \mapsto 40, t_2 \mapsto 50, t_3 \mapsto 60\}$.

Given a runtime state q, modal interpretation gives meaning to symbols in $\Sigma^{\mathcal{M}}$: some model function $f : A_i \times \dots \times A_j \mapsto A_k$ is defined as $q(f) : I^{\mathcal{M}}(A_i) \times \dots \times I^{\mathcal{M}}(A_j) \mapsto I^{\mathcal{M}}(A_k)$, 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 model structure contains 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). In other words, $\Gamma^{\mathcal{M}}$ contains the set of executions of the system defined by $\mathcal{M}$. The form of the structure traces varies depending on the modal formalisms (e.g., in PCTL the structure would be a set of paths and an induced probability measure; see Subsection 5.4.4 for details). For more information on model structures, see Definition 4 in Section 5.2.

To simplify further checking, the views are combined into a single view based on the mappings between them [25, 235]. The view signatures, view interpretations, and structures are combined into a single tuple $(\Sigma^{\mathcal{V}}, I^{\mathcal{V}}, \Gamma^{\mathcal{V}})$:

$$\Sigma^{\mathcal{V}} = \Sigma_1^{\mathcal{V}} \cup \dots \cup \Sigma_n^{\mathcal{V}}, \tag{7.1}$$

$$I^{\mathcal{V}} = I_1^{\mathcal{V}} \cup \dots \cup I_n^{\mathcal{V}}, \tag{7.2}$$

$$\Gamma^{\mathcal{V}} = \Gamma_1^{\mathcal{V}} \cup \dots \cup \Gamma_n^{\mathcal{V}}. \tag{7.3}$$

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 $I^{\mathcal{V}}$ and $I^{\mathcal{M}}$ as $I = I^{\mathcal{V}} \cup I^{\mathcal{M}}$.

7.2 Analysis Contracts

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

Definition 33 (Analysis)**.** Given a domain signature (Σ), an *analysis* (A) is a function that maps an input analysis context (Υ^i) of Σ symbols to an output analysis context (Υ^o) of Σ symbols: $\mathsf{A}(\Upsilon^i) = \Upsilon^o$.

As an example, consider an analysis that minimizes CPU frequencies (static CPU frequency scaling), which will be further discussed in the validation chapter (see $\mathsf{A}_{\text{FreqSc}}$ in Subsection 8.4.1). To produce an output context Υ^o, this analysis changes part of the input context Υ^i — the view interpretation $I^{\mathcal{V}}$ for a property CPUFreq. The changed interpretation maps the symbol CPUFreq to another function, which in turn maps $CPUs$ to different numbers than in Υ^i. Effectively, the analysis changes the frequencies of CPUs in a view. The rest of Υ^o (the structure and the interpretation of other symbols) is identical to Υ^i.

A contract for 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 34 (Analysis Contract)**.** An *analysis contract* ($\mathcal{C}$) for an analysis (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 $\mathbb{I} \subseteq \Sigma^{\mathcal{V}}$ from Σ that A reads.
- Outputs are a subset of view symbols $\mathbb{O} \subseteq \Sigma^{\mathcal{V}}$ from Σ that A writes.
- Assumptions $\mathsf{a}_1 \dots \mathsf{a}_n$ are IPL statements (Definition 10) over the signatures in Σ: $\mathbb{A} = \{\mathsf{a}_1 \dots \mathsf{a}_n\}$, where $\mathsf{a}_i \in \text{FORMULA}, i \in [1, n]$. For A to be executed, these statements must be satisfied by the input context $\Upsilon^i = ((I_1^{\mathcal{V}}, \Gamma_1^{\mathcal{V}}) \dots (I_n^{\mathcal{V}}, \Gamma_n^{\mathcal{V}}), (I_1^{\mathcal{M}}, \Gamma_1^{\mathcal{M}}) \dots (I_m^{\mathcal{M}}, \Gamma_m^{\mathcal{M}}))$:

$$\Gamma_1^{\mathcal{V}} \dots \Gamma_n^{\mathcal{V}}, \Gamma_1^{\mathcal{M}} \dots \Gamma_n^{\mathcal{M}} \models \mathsf{a}_i, \text{ for each } i \in [1, n].$$

- Guarantees $\mathsf{g}_1 \dots \mathsf{g}_m$ are IPL statements (Definition 10) over the signatures in Σ: $\mathbb{G} = \{\mathsf{g}_1 \dots \mathsf{g}_m\}$, where $\mathsf{g}_i \in \text{FORMULA}, i \in [1, m]$. These statements must be satisfied by the output context $\Upsilon^o = ((I_1^{\mathcal{V}}, \Gamma_1^{\mathcal{V}}) \dots (I_n^{\mathcal{V}}, \Gamma_n^{\mathcal{V}}), (I_1^{\mathcal{M}}, \Gamma_1^{\mathcal{M}}) \dots (I_m^{\mathcal{M}}, \Gamma_m^{\mathcal{M}}))$:

$$\Gamma_1^{\mathcal{V}} \dots \Gamma_n^{\mathcal{V}}, \Gamma_1^{\mathcal{M}} \dots \Gamma_n^{\mathcal{M}} \models \mathsf{g}_i, \text{ for each } i \in [1, m].$$

For example, the bin packing analysis assigns threads to CPUs. Its contract would have $Thrds, CPUs \in \mathbb{I}$ and $\mathsf{CPUBind} \in \mathbb{O}$, where CPUBind is a function mapping $Thrds$ to $CPUs$. A guarantee of this analysis could be, for instance, that the threads on each CPU meet their deadlines. The exact formalizations of contracts for case studies can be found in Section 8.4.

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: $\mathsf{A}(\Upsilon_1) \neq \mathsf{A}(\Upsilon_2)$. 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 ($\Upsilon^o = \mathsf{A}(\Upsilon^i)$ differs from Υ^i in terms of o: $I_i(\mathsf{o}) \neq I_o(\mathsf{o})$).

Definition 35 (Analysis Dependency)**.** An analysis A_1 (with contract $\mathcal{C}_1$) *is dependent* on an analysis A_2 (with contract $\mathcal{C}_2$), denoted $\mathrm{depends}(\mathsf{A}_1, \mathsf{A}_2)$, if the inputs of A_1 overlap with the outputs of A_2: $\mathcal{C}_1.\mathbb{I} \cap \mathcal{C}_2.\mathbb{O} \neq \emptyset$.

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 $\mathbb{AN}$ with contracts, an ordering $\mathcal{O} = \langle \mathsf{A}_1 \cdots \mathsf{A}_n \rangle$ of $\mathbb{AN}$ is *correct* if each analysis in the ordering is not dependent on any of the downstream analyses:

$$\forall i \in [1, n] \cdot \forall j \in [1, i) \cdot \neg\, \mathrm{depends}(\mathsf{A}_j, \mathsf{A}_i). \tag{7.4}$$

Some analyses may form a *dependency cycle* — such a sequence of analyses $\mathsf{A}_1 \ldots \mathsf{A}_n$ ($n \geq 2$) that $\mathrm{depends}((,\mathsf{A})_1, \mathsf{A}_2) \wedge \mathrm{depends}((,\mathsf{A})_2, \mathsf{A}_3) \wedge \cdots \wedge \mathrm{depends}((,\mathsf{A})_n, \mathsf{A}_1)$. Consider two analyses that optimize some parameter in a model for two competing qualities (e.g., prediction accuracy and run-time performance). These analyses form a dependency cycle: that parameter is an input and an output for both analyses, making them mutually dependent. Often, mutually dependent analyses are also mutually exclusive and not meant to be executed for the same design. Thus, dependency cycles are an important special case that needs to be handled during analysis execution, as explained in the next section.

Finally, the appropriateness of a context is defined via satisfaction of assumptions and guarantees, with checking done on the per-case basis, for given Υ^i and $\Upsilon^o = \mathsf{A}(\Upsilon^i)$. A contract $\mathcal{C}$ 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 36 (Appropriate Context)**.** A context Υ is *appropriate* for analysis A with contract $\mathcal{C} = (\mathbb{I}, \mathbb{O}, \mathbb{A}, \mathbb{G})$ if Υ satisfies the assumptions of A, and $\mathsf{A}(\Upsilon)$ satisfies the guarantees of A:

$$\forall \mathsf{a} : \mathbb{A} \cdot \Upsilon \models \mathsf{a},$$
$$\forall \mathsf{g} : \mathbb{G} \cdot \mathsf{A}(\Upsilon) \models \mathsf{g}.$$

For example, the aforementioned frequency scaling analysis is only applicable to deadline-monotonic systems, which is an assumption that constrains the pre-analysis context. The aforementioned bin packing analysis is only appropriate if it creates a schedulable system, making this absence of deadline misses a guarantee, applied to the post-analysis context.

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 *Thrds* symbol, and does not change the *Thrds* 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 $\gamma = (\mathsf{A}, \mathrm{depends}(,))$ for a given set of analyses $\mathbb{AN}$. The analyses from $\mathbb{AN}$ constitute the nodes of the graph, and the edges follow the dependency relation.

Presence of cycles in γ determines whether the analyses can be executed. If γ is a cyclic graph, there is no correct ordering of $\mathbb{AN}$. Indeed, assuming that some ordering $\mathcal{O}$ exists, consider a sub-sequence $\mathcal{O}'$ of $\mathcal{O}$, containing only the elements of $\mathcal{O}$ that correspond to the cycle in γ. The first element of $\mathcal{O}'$ is dependent on at least one of its successors (due to the nodes of $\mathcal{O}'$ forming a cycle in γ). Therefore, substituting the first element of $\mathcal{O}'$ for A_i in Equation (7.4) leads to $\mathcal{O}'$ and, hence, $\mathcal{O}$ not being correct orders. Without cycles, γ is a Directed Acyclic Graph (DAG), and any topologically-sorted ordering of its nodes is a correct ordering of $\mathbb{AN}$.

Therefore, an ordering $\mathcal{O}$ for a set of analyses $\mathbb{AN}$ is computed by: (i) constructing γ for $\mathbb{AN}$; (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 (7.4)), and executes analyses only in appropriate contexts (as defined by Definition 36). An informal summary of the algorithm is shown in Figure 7.1. The algorithm takes a domain signature Σ, an input context Υ, a set of analyses $\mathbb{AN}$ annotated with contracts , and one analysis $\mathsf{A} \in \mathbb{AN}$ as the goal analysis. The algorithm performs a correctly-ordered execution of analyses (with their output Υ_o) — or aborts if such execution is not possible.

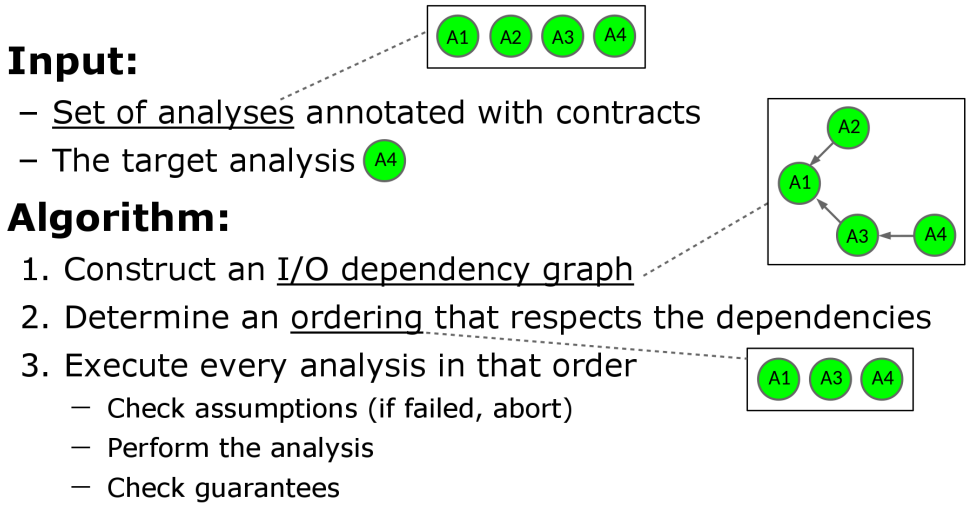


Figure 7.1: An illustration of the analysis execution algorithm. Green circles represent analyses, and arrows represent analysis dependencies.

The analysis execution algorithm follows these steps:

1. Construct a dependency graph γ.
2. Determine an ordering $\mathcal{O}$ of $\mathbb{AN}$ 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 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 A on Υ and update $\Upsilon = \mathsf{A}(\Upsilon)$.
   (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 $\mathsf{A}(\Upsilon) \models \mathbb{G}$, 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. Therefore, the conditions number 2 and 3 of successful integration (see the end of Section 2.2) are ensured by AEP.

The execution algorithm contains a non-determinstic choice of the execution order, from multiple possible orders in γ. It is possible that some of the orders lead to violations of assumptions or guarantees, whereas others would not. If during an order the execution is aborted, it can be beneficial to revert and try other orders. Various algorithms can be created and evaluated for this search, but they are outside of the scope of this thesis.

Notice that the checking of analysis contexts (steps 3a and 3c) is performed when an analysis is executed on concrete models (i.e., an Υ is available). The decision to delay the checks until right before/after the analysis execution is due to two reasons. First, concrete models enable more expressive and detailed checking, with IPL specifically. Second, checking guarantees after an analysis executes can verify the implementation of that analysis (corresponding to case 4 described in the end of Section 4.3), thus reducing the necessary trusted computing base. In theory, the checking can be performed in a model-free way, before the analyses are executed. This possibility is explored in Chapter 10.

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 be 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 AEP 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 [79]. ACTIVE has been archived [238] 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.

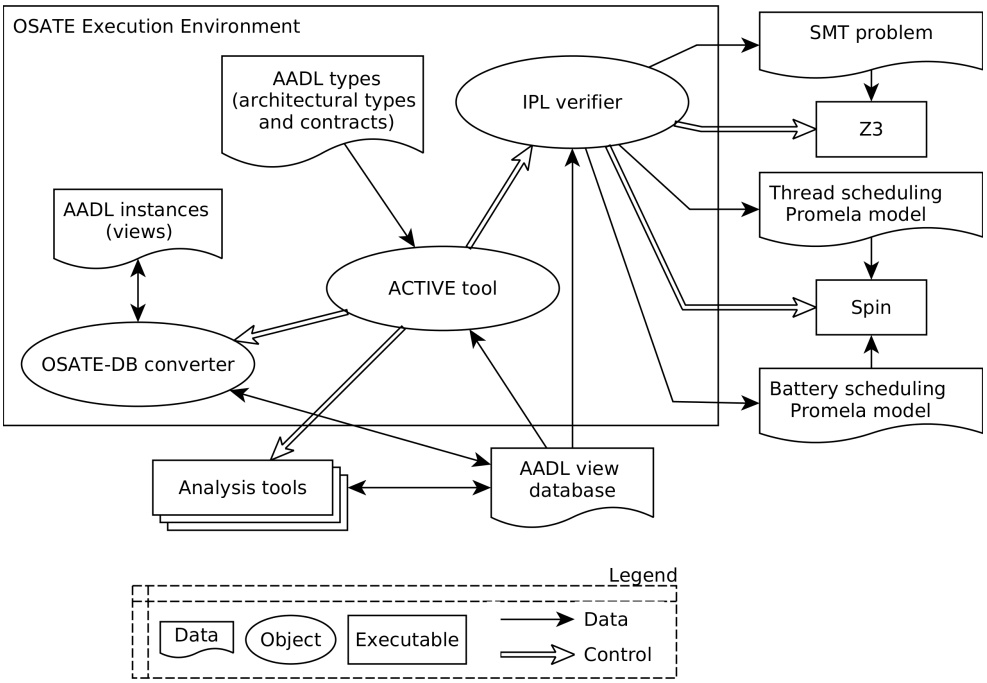


Figure 7.2: The architecture of the analysis execution platform.

Figure 7.2 depicts the architecture of AEP. Analysis contracts $\mathcal{C}$ 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 on the 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} (see Section 5.7 for details).

Chapter Summary

This chapter presented the third and final part of the integration approach — the Analysis Execution Platform. The platform relies on annotations of analyses with contracts, which capture the data dependencies of the analyses and their expectations about the execution context. This chapter also presented an algorithm that the platform uses to execute analyses with guaranteed prevention of data and context mismatches.

Chapter 8

Validation

This chapter presents the studies and evidence that support the claims formulated in Section 1.1. First, I revisit the integration argument, and formally assess its soundness in Section 8.1. The empirical 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 8.2) investigates the four claims related to IPL (expressiveness, soundness, applicability, and customizability). These claims are studied in two validation contexts: energy-aware adaptation for a mobile robot (context 1) and thread/battery scheduling for a quadrotor (context 3). A theoretical evaluation of IPL's algorithm soundness can be found earlier, in Section 5.6.
2. The validation of Part II (Section 8.3) investigates the four claims of integration abstractions (expressiveness, soundness, applicability, and customizability). These claims are investigated 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 evaluated. 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 evaluated. Theoretical evaluations of the integration abstractions can be found earlier, in Subsections 6.2.6 and 6.3.3.
3. The validation of Part III (Section 8.4) investigates the two claims related to analysis contracts (soundness and applicability). Soundness is evaluated theoretically, and both claims are evaluated practically in two validation contexts: thread/battery scheduling for a quadrotor and reliable/secure sensing for an autonomous vehicle.

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

8.1 Theoretical Evaluation of Soundness

This section revisits and details the integration argument from Section 4.4, bringing together the soundness results for IPL verification (Section 5.6), views (Subsection 6.2.6), and behavioral properties (Subsection 6.3.3).

The argument aims to check a ground-truth integration property over structural models $\mathcal{M}_1^s \dots \mathcal{M}_p^s$ and behavioral models $\mathcal{M}_1^b \dots \mathcal{M}_q^b$. Previously, in Section 4.4, the integration property was simplified as a predicate $\mathrm{integprop}$ over sets of model elements and behaviors. Here, I refine the predicate's form, denoting it as a predicate ip over quantified view variables and behaviors that depend on these variables:

$$Q_1 \boldsymbol{e}_1^{\mathcal{M}} \dots Q_n \boldsymbol{e}_n^{\mathcal{M}} \cdot \mathrm{ip}(\boldsymbol{e}_1^{\mathcal{M}} \dots \boldsymbol{e}_n^{\mathcal{M}}, \Omega_1 \dots \Omega_q). \tag{8.1}$$

The goal is to show that, given the necessary assumptions, the satisfaction of Equation (8.1) can be inferred from satisfaction of a corresponding IPL formula over appropriate abstractions. Before proceeding, I make several assumptions about the above formulation of an integration property, to make it checkable by my integration approach:

- The formula should be in its prenex normal form. Quantifier alternation is allowed, and for convenience segments of the same quantifiers are represented with a single quantifier over a vector of variables ($\boldsymbol{e}_i^{\mathcal{M}}$, corresponding to the i-th segment).
- The quantified variables are over domains that are subsets of model elements from structural models $\mathcal{M}_1^s \dots \mathcal{M}_p^s$. The domains are not shown for brevity.
- The behavior sets $\Omega_1 \dots \Omega_q$ are defined by different behavioral models from $\mathcal{M}_1^b \dots \mathcal{M}_q^b$. Each behavior set is drawn from $\mathbb{\Omega}$ of its respective model using the values of the quantified variables, $\mu_1 \dots \mu_q$. Thus, $\Omega_1 \dots \Omega_q$ are implicit functions[1] of $\boldsymbol{e}_1^{\mathcal{M}} \dots \boldsymbol{e}_n^{\mathcal{M}}$.

Creating Behavioral Properties

The first step is to replace the behavior sets with behavioral properties. Due to the assumed expressiveness of behavioral property languages of the respective models (Definition 28 in Subsection 6.3.3), predicate ip can be replaced with an equivalent predicate ip' over behavioral properties ($l_1 \dots l_m$) on the respective models. Specifically, ip' uses the meaning of $[\![l]\!]_{\Omega}^{\mu}$ instead of Ω, where l is written in some behavioral language $\mathbb{L}$, over variables with values μ, for a model that contains Ω. Notice that the number of behavioral properties (m) can be different from the number of behavioral models (q), since constraints on traces may be expressed using several behavioral properties. This step is not possible if the behavioral languages are not expressive enough to represent ip equivalently.

$$Q_1 \boldsymbol{e}_1^{\mathcal{M}} \dots Q_n \boldsymbol{e}_n^{\mathcal{M}} \cdot \mathrm{ip}'(\boldsymbol{e}_1^{\mathcal{M}} \dots \boldsymbol{e}_n^{\mathcal{M}}, [\![l_1]\!]_{\Omega_1}^{\mu_1} \dots [\![l_m]\!]_{\Omega_m}^{\mu_m}), \tag{8.2}$$

where $\mathbb{L}_i$ and Ω_i ($i = 1..m$) are selected from the behavioral property languages and traces of models $\mathcal{M}_1 \dots \mathcal{M}_p$.

Creating Views

The second step is creating views for the structural models, with the goal of using matching predicates and an IPL formula (written in next step) to infer ip'. For each model $\mathcal{M}_1^s \dots \mathcal{M}_p^s$,

[1]This dependency is not shown in the notation for convenience, but it is handled in the integration argument.

a view is produced ($\mathcal{V}_1 \ldots \mathcal{V}_p$) using a viewpoint that has certain constraints on its matching predicates, as articulated below.

Suppose that $\mathcal{M}_i^s$ ($i = 1..n$) is a structural model (with a view $\mathcal{V}_i$) that contains the quantification domain of $Q_i \boldsymbol{e}_i^{\mathcal{M}}$ in Equation (8.2). Suppose also $\left|\boldsymbol{e}_i^{\mathcal{M}}\right| = l$. Then, $\boldsymbol{e}_i^{\mathcal{M}}$ should correspond to some view vector $\boldsymbol{e}_i^{\mathcal{V}}$ over some subset of $\mathcal{V}_i$, and $\left|\boldsymbol{e}_i^{\mathcal{V}}\right| = k$. The matching predicate mp_i should have dimensions (k, l).

If $Q_i \equiv \forall$, then view $\mathcal{V}_i$ is required to be complete with respect to mp_i. If $Q_i \equiv \exists$, then view $\mathcal{V}_i$ is required to be sound with respect tomp_i. These expectations satisfy part of the preconditions of the model-view reasoning theorem (Theorem 2).

Writing IPL Formula

An IPL formula is a constraint over view elements and behavioral properties. The IPL formula has the same quantifiers as Equation (8.1), but the quantified variables are over architectural elements of views, and the sizes of vector variables are set according to the respective matching predicates. That is, if a model element vector $\boldsymbol{e}_i^{\mathcal{M}}$ is of size l, then the corresponding matching predicate is mp_i with dimensions (k, l), and the corresponding view element vector $\boldsymbol{e}_i^{\mathcal{V}}$ is of size k.

$$Q_1 \boldsymbol{e}_1^{\mathcal{V}} \ldots Q_n \boldsymbol{e}_n^{\mathcal{V}} \cdot \mathrm{ipl}(\boldsymbol{e}_1^{\mathcal{V}} \ldots \boldsymbol{e}_n^{\mathcal{V}}, [\![l_1]\!]_{\Omega_1}^{\mu_1} \ldots [\![l_m]\!]_{\Omega_m}^{\mu_m}). \tag{8.3}$$

The above IPL formula is written in a way that, when conjoined with matching predicates for all model/view variables, implies the desired integration property ip' over the model variables. The possibility of finding such views and matching predicates relies on the assumption of view expressiveness (see Subsection 6.2.6). In this implication the quantification domains of model and view variables are the same as in Equations (8.1) and (8.3), respectively.

$$\begin{aligned} \forall \boldsymbol{e}_1^{\mathcal{V}} \ldots \boldsymbol{e}_n^{\mathcal{V}}, \boldsymbol{e}_1^{\mathcal{M}} \ldots \boldsymbol{e}_n^{\mathcal{M}} \cdot \mathrm{ipl}(\boldsymbol{e}_1^{\mathcal{V}} \ldots \boldsymbol{e}_n^{\mathcal{V}}, [\![l_1]\!]_{\Omega_1}^{\mu_1} \ldots [\![l_m]\!]_{\Omega_m}^{\mu_m}) \wedge \\ \bigwedge_{i=1..n} \mathrm{mp}_i(\boldsymbol{e}_i^{\mathcal{V}}, \boldsymbol{e}_i^{\mathcal{M}}) \rightarrow \mathrm{ip}'(\boldsymbol{e}_1^{\mathcal{M}} \ldots \boldsymbol{e}_n^{\mathcal{M}}, [\![l_1]\!]_{\Omega_1}^{\mu_1} \ldots [\![l_m]\!]_{\Omega_m}^{\mu_m}). \end{aligned} \tag{8.4}$$

Verifying IPL Formula

The IPL formula (Equation (8.3)) is checked using the IPL verification algorithm (Algorithm 1 in Subsection 5.5.2). The algorithm reasons over view elements and the rigid part of the IPL specification. It finds a set SV of valuations of quantified view variables, and instantiates each model $\mathcal{M}_i^b$ on each valuation μ. These instances define the sets of behaviors $\Omega_1 \ldots \Omega_m$, on which behavioral queries (Q) are run. Each behavioral query has rigid subformulas $\textsc{ratom}_1 \ldots \textsc{ratom}_k$ that are evaluated according to the IPL semantics based on the valuation μ. These queries are required to be sound, so the returned values of queries for each μ are equal to the meaning of the respective behavioral properties, and coincide with the IPL semantics:

$$\mathsf{Q}(l_i, \Omega_i, \mu_i) = [\![l_i([\![\textsc{ratom}_1]\!]_{\mu_i} \ldots [\![\textsc{ratom}_k)]\!]_{\mu_i}]\!]_{\Omega_i}^{\mu_i}, i = 1..n.$$

Due to Corollary 2 of the theorem of IPL verification soundness, the IPL formula holds on these models if and only if the algorithm indicates so. If the IPL formula is satisfied, then I apply the theorem of model-view reasoning (Theorem 2), which has its preconditions satisfied by view constraints and Equation (8.4). The theorem states that ip' (Equation (8.2)) holds. Since Equation (8.2) and Equation (8.1) are equivalent, then ip holds as well. Hence, if the IPL verification passes, the original integration property is satisfied.

If the IPL formula is not satisfied, then no implication about the original integration property can be made. In this case, an engineer can examine the models and views that prevented the satisfaction. After these artifacts are corrected, the property may pass (then the above implication applies), or return another counterexample to analyze.

The above steps prove the following theorem:

Theorem 3 (Soundness of Integration Approach)**.** If for some integration property of form shown in Equation (8.1) there exists an equivalent rewriting ip' (Equation (8.2)) using behavioral properties, and the conditions of theorems 1 and 2 hold for some collection of views and viewpoints, and Equation (8.3) is satisfied, then that integration property is satisfied.

Notice that a stronger version of this theorem would involve bi-implication: the IPL formula holds if an only if the integration property holds. For this theorem, a stronger version of the model-view reasoning theorem is needed (see the end of Subsection 6.2.6). This stronger version is sufficient because the IPL verification theorem, the rewriting of ip with ip', and replacing behavioral properties with queries are all steps that preserve equivalence.

The rest of this chapter presents the empirical studies of modeling method integration, grouped by parts of the integration approach.

8.2 Validation of Part I: Integration Property Language

IPL has been validated theoretically for soundness (see Section 5.6), and empirically in the context of Systems 1 and 3 (below).

8.2.1 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 one 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 the 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 [267] on the robotic system that was described in Section 3.1. Since that system carried out adaptation using multiple models, it was 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[2] of IPL based on Eclipse (Oxygen 1a) and OSATE2 (version 2.3.0) [78]:

- Explore the available artifacts in search for structural and behavioral models.
- Determine a relationship between the models that may lead to a design not satisfying a critical requirement.[3] Informally state the desired property.
- Create integration abstractions for the models (details in Section 8.3).
- Specify an integration property in IPL.
- Execute the verification algorithm on the integration property.
- If the verificaiton fails, trace the counter-example to the specific elements of abstractions, as well as models if needed.
- 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[4] because power appeared to be a safety-critical and cross-cutting concern. Specifically, I studied the consistency relationships between three models (see Section 3.1 for more information):

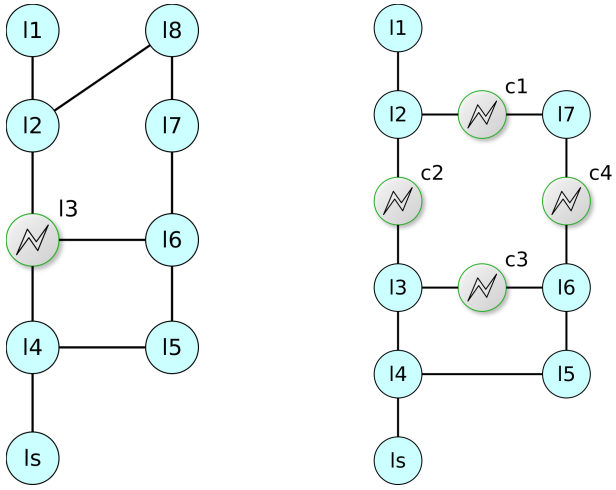
- A planning model $\mathcal{M}_{\text{plan}}$, which determines the robot's plan by solving an MDP. Each model encodes a map and energy costs in the MDP. Approximately 10 variants of this model were discovered, each with different features (explained below). Another 10 variants were created to complete the space of models.

[2]The IPL implementation has been archives [241] and is also available at `https://github.com/bisc/IPL`.

[3]This process has been performed informally, by comparing the available models and requirements.

[4]The case study models have been archived [241] and are also available online: `https://github.com/bisc/IPLProjects`.

- A power model $\mathcal{M}_{\text{power}}$, which determines the energy cost of a task based on its parameters (distance of motion, time, and the robot's configuration). While the system used one power model, feature variance led to considering up to 6 power model variants per map.
- A map model $\mathcal{M}_{\text{map}}$, which determines the locations and the possibility of moving between them. Two significantly different maps were discovered, but minor differences between map versions led 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 37 (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 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.

Atomic tasks are combined to form missions:

Definition 38 (Mission)**.** A *mission* is a finite sequence of atomic tasks with contiguous and non-self-intersecting[5] locations. A *power-successful mission* can be completed without draining

[5]No implementations of $\mathcal{M}_{\text{plan}}$ allowed for self-intersecting trajectories.

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:

- 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.

Integration Abstractions

Here I briefly summarize the integration abstractions used in this study. More details on how missions are modeled with views and behavioral properties can be found in Subsection 8.3.1, which is devoted to the evaluation of integration abstractions used for System 1.

To perform these integration checks, I created two types of integration abstractions for the above models. The first type includes structural views $\mathcal{V}_{\text{power}}$, based on $\mathcal{M}_{\text{power}}$ and $\mathcal{M}_{\text{map}}$ (see), enumerates all atomic tasks possible in $\mathcal{M}_{\text{map}}$ as components. The required energy for each task is recorded as a component property, based on the equations of $\mathcal{M}_{\text{power}}$.

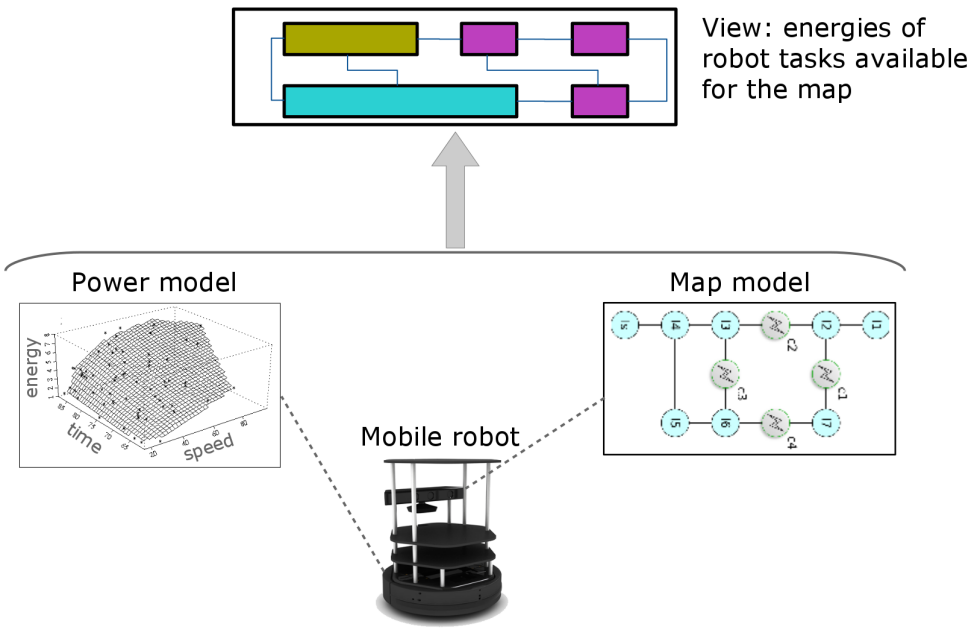


Figure 8.2: A power view $\mathcal{V}_{\text{power}}$ is created for a pair of models: $\mathcal{M}_{\text{power}}$ and $\mathcal{M}_{\text{map}}$.

A task power view ($\mathcal{V}_{\text{power}}$) is built to represent all possible atomic tasks ($\overline{Tasks}$) on a given map $\mathcal{M}_{\text{map}}$ (which defines a set of locations $\overline{Locs}$), and the energy of these tasks, according to $\mathcal{M}_{\text{power}}$. As shown in Figure 8.2, one $\mathcal{V}_{\text{power}}$ is created from a pair of models: $\mathcal{M}_{\text{map}}$ and $\mathcal{M}_{\text{power}}$. Each task ($t \in \overline{Tasks}$) is a component in $\mathcal{V}_{\text{power}}$ associated with two adjacent locations (satisfying the predicate $\mathrm{adjacent}$): its start (l_1, stored in $start$ property) and its end (l_2, stored in end property). The task is also annotated with its energy ($energy$), based on the estimate from $\mathcal{M}_{\text{power}}$. Some views also considered the robot's heading, adding the starting and ending heading properties. An excerpt of AADL code for one task component in $\mathcal{V}_{\text{power}}$ is shown in Figure 8.3, indicating how the energy, identifier, the type of the task, and starting and ending location and heading are assigned based on $\mathcal{M}_{\text{power}}$ and $\mathcal{M}_{\text{map}}$.

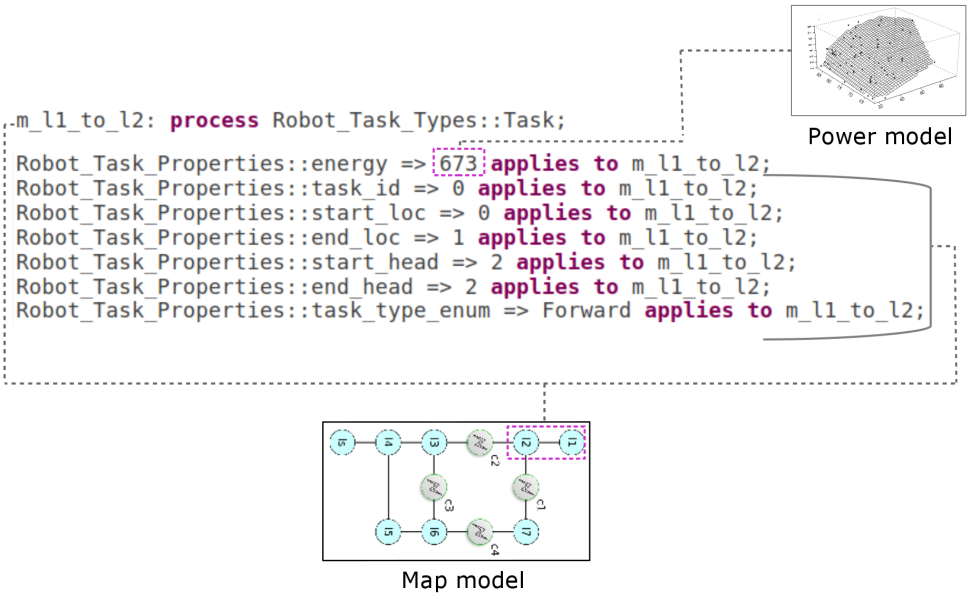


Figure 8.3: AADL code for one task in $\mathcal{V}_{\text{power}}$, moving between two locations in $\mathcal{M}_{\text{map}}$.

The desired relations between $\overline{Tasks}$ and $\overline{Locs}$ are represented by two matching predicates: mp_1 is used to define soundness (and therefore uses conjunction), and mp_2 is used to define completeness (and therefore uses implication).

$$\mathrm{mp}_1((t),(l_1,l_2)) \equiv \mathrm{adjacent}(l_1,l_2) \wedge t.start = l_1 \wedge t.end = l_2, \tag{8.5}$$

$$\mathrm{mp}_2((t),(l_1,l_2)) \equiv \mathrm{adjacent}(l_1,l_2) \rightarrow t.start = l_1 \wedge t.end = l_2. \tag{8.6}$$

The behavioral properties in this case study are expressions in a PCTL-based language, defined as a PCTL plugin in Subsection 5.3.3. The state variables of $\mathcal{M}_{\text{plan}}$ are the robot's current location ($\underline{loc}$) and current battery charge ($\underline{bat}$). The model parameters are the initial location ($\underline{initloc}$), initial battery charge ($\underline{initbat}$), and the robot's goal location ($\underline{goal}$). A model also defines a constant of the maximum battery capacity ($\overline{maxbat}$). The values for such properties included boolean and real domains. The behavioral queries were computed on an MDP from $\mathcal{M}_{\text{plan}}$, and were assumed to be sound.

These abstractions were linked in IPL specifications via quantified variables, which had the set $\overline{Tasks}$ as their quantification domain.

Integration Properties

As discussed in Section 3.1, the intent of MMI for this system is to verify consistency between $\mathcal{M}_{\text{power}}$ and $\mathcal{M}_{\text{plan}}$. Since the map model may cause inconsistencies, two goals need to be achieved:

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

Notice that the first property above is dependent on the second one: if $\mathcal{M}_{\text{power}}$ and $\mathcal{M}_{\text{plan}}$ 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 as a separate verification problem. Thus, consistency between $\mathcal{M}_{\text{plan}}$ and $\mathcal{M}_{\text{map}}$ means that $\mathcal{M}_{\text{plan}}$ has been constructed to plan in the exactly same map as $\mathcal{M}_{\text{map}}$. For $\mathcal{M}_{\text{power}}$, consistency with $\mathcal{M}_{\text{map}}$ means that its view $\mathcal{V}_{\text{power}}$ was constructed for the same map as in $\mathcal{M}_{\text{map}}$.

Thus, to perform the model integration for $\mathcal{M}_{\text{power}}$ and $\mathcal{M}_{\text{plan}}$, three types of integration properties should be satisfied:

1. If $\mathcal{M}_{\text{power}}$ considers a mission power-successful, then $\mathcal{M}_{\text{plan}}$ should do so.
2. If $\mathcal{M}_{\text{plan}}$ considers a mission power-successful, then $\mathcal{M}_{\text{power}}$ should do so.
3. $\mathcal{M}_{\text{plan}}$ and $\mathcal{M}_{\text{power}}$ 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 $\mathcal{M}_{\text{plan}}$, a behavioral property, and $\mathcal{V}_{\text{power}}$, as shown in Figure 8.4.

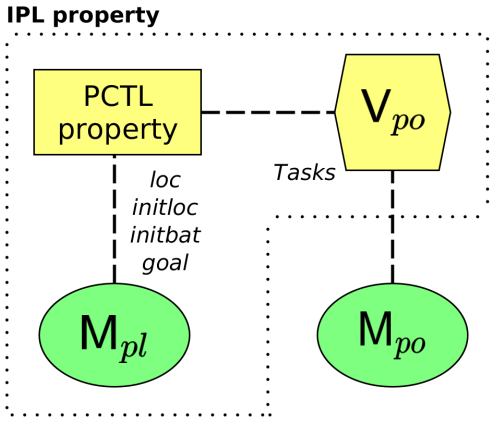


Figure 8.4: The context of integration properties for $\mathcal{M}_{\text{power}}$ and $\mathcal{M}_{\text{plan}}$. The dotted line indicates the scope of the property.

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

"For any three tasks from $\mathcal{M}_{power}$ in a sequence $\langle$go straight, rotate, go straight$\rangle$"

$$\forall t_1, t_2, t_3 : \overline{Tasks} \cdot t_1.type = t_3.type = \text{STR} \wedge t_2.type = \text{ROT} \wedge \tag{8.7}$$

"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 $\mathcal{M}_{plan}$ 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 $\overline{err}_{cons}$)."

$$\{\![\underline{initloc} = t_1.start, \underline{goal} = t_3.end, \underline{initbat} = \Sigma_{i=1}^{3} t_i.energy + \overline{err}_{\text{cons}}]\!\} = 1.$$

The above property uses quantification and constraints to construct a mission from the atomic tasks of $\mathcal{V}_{\text{power}}$ (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 $\mathcal{M}_{\text{power}}$) 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 $\mathcal{M}_{\text{plan}}$ 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 $\mathcal{M}_{\text{plan}}$ equals the energy estimate from $\mathcal{M}_{\text{power}}$ plus $\overline{err}_{\text{cons}}$, 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 $\mathcal{M}_{\text{plan}}$.

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

Property 4. If $\mathcal{M}_{\text{power}}$ considers a mission (up to four go-straight tasks long) power-successful, $\mathcal{M}_{\text{plan}}$ should consider this mission power-successful as well — allowing for $\overline{err}_{\text{cons}}$ 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"

$$t_1.end = t_2.start \wedge t_2.end = t_3.start \wedge t_3.end = t_4.start \wedge$$

"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"

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

"and that is a power-successful mission in $\mathcal{M}_{power}$"

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

"will correspond to such executions in $\mathcal{M}_{plan}$ that visit all sequence points"

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

"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."

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

In a variant of $\mathcal{V}_{\text{power}}$ 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 $\mathcal{M}_{\text{plan}}$ to $\mathcal{M}_{\text{power}}$. This time, I use an existentially quantified variable for a power budget. Of interest are situations when, given a power budget, $\mathcal{M}_{\text{plan}}$ successfully completes a mission, but this mission fails in $\mathcal{M}_{\text{power}}$ even with a large enough budget to offset the consistency error. This property assumes that the mission success is monotonic with respect to the initial energy: if a mission succeeds with a certain initial energy, it will succeed with a larger amount of energy. The converse is assumed to hold about mission failure with insufficient initial energies. While verifying this property, 100 distinct vectors (SV) of free variable values were found to satisfy the search formula for this property. Thus, 100 missions were considered to determine whether this property holds.

Property 5. If $\mathcal{M}_{\text{plan}}$ considers a mission power-successful (of four go-straight tasks long), $\mathcal{M}_{\text{power}}$ 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"

$$t_1.end = t_2.start \wedge t_2.end = t_3.start \wedge t_3.end = t_4.start \wedge$$

"without self-intersections"

$$\text{distinct}(t_1.start, t_2.start, t_3.start, t_4.start, t_4.end)\ \rightarrow$$

"there exists such an energy budget greater than the energy expected by $\mathcal{M}_{power}$"

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

"that if $\mathcal{M}_{plan}$, going through all the sequence points"

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

"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 $\mathcal{M}_{power}$ should also be power-successful that budget."

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

The third integration property — consistency of $\mathcal{M}_{\text{power}}$ and $\mathcal{M}_{\text{plan}}$ 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 $\mathcal{M}_{\text{power}}$ and $\mathcal{M}_{\text{map}}$ are consistent, and that $\mathcal{M}_{\text{map}}$ and $\mathcal{M}_{\text{plan}}$ are also consistent. It is easier to quantify specific locations by using an explicit model of a map.

Starting with $\mathcal{M}_{\text{power}}$ and $\mathcal{M}_{\text{map}}$, two views are necessary: $\mathcal{V}_{\text{power}}$ (which has already been used above) and $\mathcal{V}_{\text{map}}$ (which contains map locations as components, and their connectivity is recorded as a property of nodes, with more detail in Section 8.3). In short, $\mathcal{V}_{\text{map}}$ 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 $\mathcal{M}_{\text{power}}$ and $\mathcal{M}_{\text{map}}$ consistency is exemplified with the two properties below.

Property 6. Any location in $\mathcal{M}_{\text{map}}$ is reachable in $\mathcal{M}_{\text{power}}$.

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

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

Property 7. Every straight motion task in $\mathcal{M}_{\text{power}}$ corresponds to an edge in $\mathcal{M}_{\text{map}}$.

"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."

$$l_1.id = t.start \wedge l_2.id = t.end \wedge l_1 \in l_2.edges \wedge l_2 \in l_1.edges.$$

In a similar way, one can assure that $\mathcal{V}_{\text{map}}$ is consistent with $\mathcal{V}_{\text{power}}$. Given that the above properties are satisfied, I turn to consistency between $\mathcal{M}_{\text{map}}$ and $\mathcal{M}_{\text{plan}}$.

Property 8. Every location in $\mathcal{M}_{\text{map}}$ exists in $\mathcal{M}_{\text{plan}}$.

"Any location from $\mathcal{V}_{map}$ exists in $\mathcal{M}_{plan}$"

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

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 $\mathcal{M}_{\text{plan}}$ does not have more locations than $\mathcal{V}_{\text{map}}$ 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 $\mathcal{M}_{\text{plan}}$ with IDs smaller than the minimal ID found in $\mathcal{M}_{\text{map}}$.

"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 $\mathcal{M}_{plan}$ 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 the purposes of this integration scenario: if a location is not reachable in the MDP, it would not affect other verification. Finally, one can demonstrate that the edges are consistent between $\mathcal{M}_{\text{plan}}$ and $\mathcal{M}_{\text{map}}$ by showing two properties below.

Property 10. Any pair of locations with an edge in $\mathcal{M}_{\text{map}}$ can be traversed directly in $\mathcal{M}_{\text{plan}}$.

"For any task, the behaviors in $\mathcal{M}_{plan}$ going from its start to its end"

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

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

$$\{\![\underline{initloc} = t.start, \underline{goal} = t.end, \underline{initbat} = t.energy + 1]\!\} = 1.$$

Property 11. Any pair of locations without an edge in $\mathcal{M}_{\text{map}}$ cannot be traversed directly in $\mathcal{M}_{\text{plan}}$.

"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 $\mathcal{M}_{plan}$ aiming to go between them (without intermediate steps),"

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

"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, one can conclude that $\mathcal{M}_{\text{power}}$ and $\mathcal{M}_{\text{plan}}$ are consistent with respect to a given $\mathcal{M}_{\text{map}}$.

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 $\mathcal{M}_{\text{power}}$, $\mathcal{M}_{\text{plan}}$, and $\mathcal{M}_{\text{map}}$ 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. I discovered 17 errors listed in Table 8.1.

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

- Errors in the models (or the model generation code). These errors are integration issues that the approach intends to discover. Most of these issues were fixed as described in the last column of Table 8.1.
    - Leading to aggressive faults: the issues that would lead to possible violation of safety constraints (not running out of power in this scenario).
    - Leading to conservative faults: the issues that would lead to being overly conservative and avoiding behaviors that are otherwise preferred.

    - Not leading to faults: the issues that are inconsistencies in models, but do not affect the system's real-world behavior.
- Errors in the integration artifacts: views, behavioral properties, and IPL formulas. These errors are empirical false positives, which the approach does not aim to discover. All of these errors were corrected upon their discovery, as described in the last column of Table 8.1.
    - View errors: errors in constructing views.
    - Behavioral property errors: errors in specifications of behavioral properties.
    - IPL formula errors: errors in IPL specifications (not only behavioral parts).

Since each error was discovered by running IPL verification of integration properties, the presented 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 an error-prone process that can introduce a significant number of false positives. It is necessary to perform separate quality assurance on views to separate these issues from true positives (i.e., inconsistencies between models).
- Additional 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 optimization: transitions in $\mathcal{M}_{\text{plan}}$ 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)	Add a guard checking $\underline{bat} >=$ for every transition in $\mathcal{M}_{\text{plan}}$
2	Negative rotation energy in $\mathcal{M}_{\text{plan}}$ when facing south and turning to face southeast	Model error, aggressive fault	In the code for generation of $\mathcal{M}_{\text{plan}}$, 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	$\mathcal{M}_{\text{power}}$ used a linear regression model that maximized the fit to the experimental data, without the constraint of passing through $(0, 0)$. In contrast, $\mathcal{V}_{\text{power}}$ 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 $\mathcal{M}_{\text{power}}$ to a function that maps empty tasks to 0 required energy
4	Inconsistent required energy in $\mathcal{M}_{\text{plan}}$	Model error, conservative fault	Unknown, possibly a copying mistake of the model creator	The required energy for the $l_3 \rightarrow l_6$ edge was higher than in $\mathcal{M}_{\text{power}}$ and 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 $\mathcal{M}_{\text{power}}$ and $l_6 \rightarrow l_3$

5	Slight mismatch (0.1 second) between $\mathcal{M}_{\text{power}}$ and a variant of $\mathcal{M}_{\text{plan}}$ in the required times for forward moves	Model error, no fault	Unknown, possibly rounding of times or distances by the model creator	An integration property of precise consistency ($\overline{err}_{\text{cons}} = 0$) fails, but no perceivable impact on planning or execution	Adjust the failing integration property to have $\overline{err}_{\text{cons}} > 0$
6	Unnecessarily wide ranges for location variables in some $\mathcal{M}_{\text{plan}}$	Model error, no fault	Unknown, possibly a typo of the model creator	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	Replace the range with $[0..8]$
7	Unnecessarily wide ranges for location variables in another $\mathcal{M}_{\text{plan}}$	Model error, no fault	Unknown, possibly a typo of the model creator	A model contains 12 locations, but the variable's type has 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	Replace the range with $[0..11]$
8	Edge $l_3 \rightarrow l_6$ missing in a map view	View error	A researcher's (author's) typo during view creation	The robot would not consider paths that use $l_3 \rightarrow l_6$ (while these paths are feasible in reality), thus possibly missing optimal plans.	Add the missing edge to the view
9	Edge $l_s \rightarrow l_4$ missing in a map view	View error	A researcher's (author's) typo during view creation	The robot would not consider the paths that use $l_s \rightarrow l_4$ (while these paths are feasible in reality), thus possibly missing optimal plans	Add the missing edge to the view

10	Rotation results do not match the robot headings	IPL formula error	An error of the researcher (author): the constraints on matching headings in consecutive moves were only applied (incorrectly) in cases of empty moves	The integration property fails (false positive). No impact on planning or execution	Add constraints on directions to all consecutive moves of the robot
11	A view of $\mathcal{M}_{\text{power}}$ did not assign the values of starting and ending heading for forward motion tasks	View error	An accidental omission in view generation code	The robot would consider invalid missions with mismatched headings	Fix the code by adding the missing values
12	Overlapping IDs between charging, empty, and rotation tasks in several views of $\mathcal{M}_{\text{power}}$	View error	An error of the researcher (author) during manual view creation	A robot would consider missions with invalid combinations of tasks	Assign non-overlapping IDs to all tasks
13	Mismatch between turns in views of $\mathcal{M}_{\text{power}}$ for two different maps	View error	An accidental omission of turns 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	Correct the generation code to include all turn components
14	Empty tasks assigned an incorrect type (charging tasks) in a view of $\mathcal{M}_{\text{power}}$	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 $\mathcal{M}_{\text{plan}}$ and integration properties	IPL formula error	The specification of charging behavior in an integration property allowed any charging, whereas $\mathcal{M}_{\text{plan}}$ only allowed charging if $\underline{bat} > 1500$ mwh	The integration property considered missions that are prohibited in $\mathcal{M}_{\text{plan}}$	Add constraints in the integration properties that allow charging only when $\underline{bat} < 1500$ 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 in integration properties was too low to complete tasks in $\mathcal{M}_{\text{plan}}$	IPL formula error	To enable charging (as described in the previous inconsistency), integration properties constrain the energy upon arrival to charging stations. This change led to constraining the initial energy to be lower than that of the required energy for a sequence of tasks (in $\mathcal{M}_{\text{plan}}$) before a charging station	Considering missions that are not valid	Add a constraint to the integration properties that the energy has to be sufficient to perform tasks between charging stations

Table 8.1: Integration errors that occurred in the study.

Performance

Map	# of steps	Variable length?	Char?	Rot?	$\|SV\|$	Sat. 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. “Char” stands for “charging”, “Rot” for rotation.

I evaluated the performance of an Eclipse-based IPL implementation using variants of the $\mathcal{M}_{\text{power}}$-to-$\mathcal{M}_{\text{plan}}$ 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 solutions for the search formula (SV), total time, saturation time, interpretation time, time in SMT, and time in model checking. The study 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: IntelÂő Core i7-7600U, Ubuntu 17.04, Eclipse Oxygen 1a, OSATE 2.3.0 (debug mode) [79], Z3 solver 4.5.0 [60], PRISM model checker 4.4.beta [154] with Rabinizer 3.1 [146]. The dataset and its analysis have

been archived [241] and are also available online.[6]

The high-level findings from the performance experiments are as follows:

- Verification times vary from dozens of seconds to over 6 hours. Counts of solutions (SV) vary from dozens to over a thousand.
- Longer missions lead to increase in both saturation and interpretation times, whereas missions with more features primarily affect the saturation process.
- Model checking times grew linearly with increases in mission length across feature groups, with little response to increases in mission features.
- Saturation times grow substantially with more features, especially when considering rotations due to additional quantified variables and constraints.
- IPL's overhead (i.e., the verification time spent outside of SMT solving and model checking), averaged across all the verification runs, was small: 0.74% (stdev 0.78%) of the total verification time of each run.
- IPL's memory demands were not limiting to the verification, since at most one external tool ran at a time.

Conclusions for Evaluation on System 1

Returning to the questions posed in Subsection 8.2.1, 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 one 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.

8.2.2 Evaluation of IPL on System 3

IPL was secondarily evaluated in the context of real-time scheduling and battery design for a quadrotor. This evaluation was opportunistic: at the start of this project, I did not set a goal of discovering or checking integration properties. Instead, the study focused on integration of analyses (for more details, see Subsection 8.4.1). Nevertheless, two integration properties in analysis contracts required expressive specification of relations between structures and behaviors of models. An early prototype of IPL was used to specify and check those properties. The goal of

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

this section is to demonstrate that IPL can be applied to different domain (in this case, aerospace) and a behavioral language based on a different logic (in this case, LTL).

Models and Integration Abstractions

In IPL applications, this system was represented with three models: the CPU model (M_{cpu}), the scheduling model (M_{sch}), and the software concurrency model (M_{rek}). M_{cpu} is a structural model of the available CPUs and threads, which can reduce the frequency of CPUs to make the system more energy-efficient. M_{sch} is a behavioral model that schedules threads in a way that prevents any deadline misses.[7] M_{rek} is a software concurrency model that is based on the source code that is executed in the threads, and is treated as a structural model in this case study. For further details about these models, see Subsection 8.4.1.

For the above models, I used two types of views and one behavioral property language. The thread scheduling view (V_{sch} for M_{sch} and V_{rek} for M_{rek}) exposes Threads ($Thrds$) as components with deadlines (Dline), periods (Per), and worst-case execution times (WCET) as their properties. The CPU view (V_{cpu}) exposes CPUs ($CPUs$) as components, with the CPU frequency (CPUFreq), it maximum value ($\mathsf{CPUFreq}_{\mathsf{max}}$), and thread-to-CPU bindings (CPUBind) as properties. The matching predicates are straightforward: the views need to provide components for all threads and CPUs, via a one-to-one mapping with models M_{sch} and M_{cpu}, respectively. The properties of these components in V_{sch} and V_{cpu} need to be consistent with the data M_{sch}, M_{rek}, and M_{cpu}. The behavioral property language was based on the LTL plugin described in Subsection 5.3.2. The only modal predicate in the language was $\mathrm{canprmt}(t_1, t_2)$, which evaluates to truth only in states where thread t_1 can preempt thread t_2, according to the chosen scheduling policy. More information on the abstractions in this case study can be found in Subsection 8.3.3. The abstractions were linked with quantified variables with quantification domains of $CPUs$ and $Thrds$.

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 two models are not fully 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 equivalent to deadline-monotonic scheduling (DMS) in a particular model (e.g., a model with rate-monotonic scheduling (RMS) and the period equal to the deadline for each thread), even though it is not deadline-monotonic by design.

To keep M_{sch} and M_{cpu} non-conflicting, I specified and verified 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}.

[7] Deadline misses may result in a failure of a real-time system.

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, in each moment, a thread can preempt (i.e., take over the CPU for execution) only threads with greater deadlines. 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"

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

"should only bind pairs of threads that"

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

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

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

This property can be checked by the IPL verification algorithm. Using $\mathsf{V}_{\mathsf{sch}}$ and $\mathsf{V}_{\mathsf{cpu}}$, 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 $\mathsf{M}_{\mathsf{sch}}$. 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 before CPU frequencies are scaled down, which occurs after changing thread-to-CPU bindings. If verification succeeds, it is guaranteed that deadlines will not be missed, and the power consumption has been minimized (i.e., that $\mathsf{M}_{\mathsf{sch}}$ and $\mathsf{M}_{\mathsf{cpu}}$ are integrated correctly).

Integration Property 2: Safe Concurrency and Thread Scheduling

In the same case study, another property was evaluated the model of thread scheduling ($\mathsf{M}_{\mathsf{sch}}$) and the safe concurrency model ($\mathsf{M}_{\mathsf{rek}}$). The integration goal was to apply the safe concurrency checking from $\mathsf{M}_{\mathsf{rek}}$ to the scheduling, and this analysis is only valid under implicit deadlines and fixed-priority scheduling. Satisfaction of these conditions is the integration property to verify.

To express the above applicability conditions as an integration property, I use the set of threads from $\mathsf{V}_{\mathsf{rek}}$ (with their properties like period and deadline) and behavioral properties from $\mathsf{M}_{\mathsf{sch}}$ below in two formulas. One (fully rigid) formula constrains threads to be implicit deadlines, and the other, mixed (rigid and flexible) formula for $\mathsf{M}_{\mathsf{sch}}$ expresses the fixed-priority scheduling:

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

$$\forall t \cdot \mathsf{Per}(t) = \mathsf{Dline}(t) \wedge \tag{8.8}$$

$$\forall t_1, t_2 \cdot \mathsf{G}\ (\mathrm{canprmt}(t_1, t_2) \rightarrow (\mathsf{G} \neg \mathrm{canprmt}(t_2, t_1))). \tag{8.9}$$

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 $\mathsf{M}_{\mathsf{sch}}$. 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. It was 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 8.4), again supporting Claim 3 (applicability). Finally, Claim 2 (soundness) is in part supported by correctly identifying model integration issues using verification.

8.2.3 Summary for Evaluation of IPL

Section 8.2 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.

Below I 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.

8.3 Validation of Part II: Integration Abstractions

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

8.3.1 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 8.2.1. The main focus of the study was to discover, specify, and verify integration properties (for details see Subsection 8.2.1). Below I address the secondary focus of this study — 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 $\mathcal{V}_{\text{power}}$) can represent the tasks and missions that the robot can accomplish (particularly, in $\mathcal{M}_{\text{plan}}$) and represent them with required energies (from $\mathcal{M}_{\text{power}}$); and whether behavioral properties in PCTL can constrain the robot in $\mathcal{M}_{\text{plan}}$ to the missions described in $\mathcal{M}_{\text{power}}$.
- *Soundness:* whether the views are sound and complete (in terms of the tasks and missions of the robot) with respect to the relevant matching predicates (specifically, Equations (8.5) and (8.6) in Subsection 8.2.1); and whether the checking of behavioral properties in PCTL produces a sound result and terminates.
- *Applicability:* whether the views can satisfy various constraints on tasks and their ordering, and accommodate different features and modes of the robot (such as charging); and whether the PCTL checking produces results within the practical limits on time and memory (at design time).
- *Customizability:* whether the views and PCTL properties are extensible for new tasks and mission features that are not currently present in the study.

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

Views for System 1

As described in Subsection 8.2.1 and further detailed in Subsection 8.2.1, the role of the power view ($\mathcal{V}_{\text{power}}$) for the regression power model ($\mathcal{M}_{\text{power}}$) is to represent a set of atomic tasks (see Definition 37) that can be performed on a given map. The atomic tasks are sequentially composed into missions (see Definition 38) using quantified variables in IPL formulas. Instances of $\mathcal{V}_{\text{power}}$ are created automatically by an implementation of the power viewpoint algorithm, which creates an instance of $\mathcal{V}_{\text{power}}$ based on a given map ($\mathcal{M}_{\text{map}}$). Encoded in AADL, $\mathcal{V}_{\text{power}}$ is separated into a declaration part, where all task components are declared (see Figure 8.5), and a value-setting part, where all property values are set (see Figure 8.6). Once $\mathcal{V}_{\text{power}}$ is created, an integration property compares the missions constructed from $\mathcal{V}_{\text{power}}$ and $\mathcal{M}_{\text{plan}}$ to check that energy is consistent in $\mathcal{M}_{\text{power}}$ and $\mathcal{M}_{\text{plan}}$.

Each of the desired integration qualities are evaluated on $\mathcal{V}_{\text{power}}$ at two levels: for *individual tasks* and their *types* (which are directly represented in the view), and for *missions*, which are 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 8.2.1).

```
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 8.5: Task declarations in an instance of $\mathcal{V}_{\text{power}}$ in the AADL syntax.

The *expressiveness* of $\mathcal{V}_{\text{power}}$ 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 $\mathcal{V}_{\text{power}}$. In cases when a task does not require values for certain properties (e.g., an empty task does not have a specific orientation), it can be left unspecified. Omitting these values leads to underspecified SMT constraints on the uninterpreted functions that encode these properties of the view elements. The expressiveness of tasks is limited by the locations of the map: the tasks that start or end not in one of the map's locations cannot be specified. From the perspective of missions, $\mathcal{V}_{\text{power}}$ allows to specify only missions up to a given finite length. This constraint is not limiting for this case study: each map has an upper bound on mission length because missions are not allowed to self-intersect (in accordance with the dynamics of $\mathcal{M}_{\text{plan}}$). The maximum mission length is 7 tasks for the map variant 1 (Figure 8.1a) and 10 tasks for the map variant 2 (Figure 8.1b).

The *soundness* of $\mathcal{V}_{\text{power}}$ in terms of tasks (i.e., containing only tasks of valid types between existing map locations) relies on the correctness of the power viewpoint algorithm, which takes a map and a set of required task types as inputs, 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 8.2.1), to the point where the algorithm produces sound views in practice. Their soundness has been confirmed by manual inspection and testing that involved checking "trivial" properties, such as "every forward motion task has an inverse in the view". In terms of missions, the exact encoding of the definition in IPL ensures soundness. Any sequence

```
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 8.6: Task properties and their values from an instance of $\mathcal{V}_{\text{power}}$ in the AADL syntax.

of tasks that is constrained to be contiguous and non-self-intersecting is considered a mission, in accordance with Definition 38. 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 guaranteeing that any sequence of tasks from $\mathcal{V}_{\text{power}}$ that fits these constraints a valid mission.

The *completeness* of $\mathcal{V}_{\text{power}}$ in terms of tasks (i.e., containing all tasks pertinent to a given map) is based on the expressiveness of $\mathcal{V}_{\text{power}}$. As discussed above, $\mathcal{V}_{\text{power}}$ can encode any task that is relevant for the robot on a given map, thus leading to a complete view in terms of tasks. Their completeness was verified by manual inspection and testing. In terms of missions, completeness relied on exact encoding of missions in IPL: given a sufficient number of quantified variables, any finite mission in a given map can be encoded using first-order logic constraints. In this case study, for any combination of mission features, the intended set of missions was encodable 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 practical applicability, leading to large specifications.

The *applicability* of power views faced two challenges in this system. The first challenge for $\mathcal{V}_{\text{power}}$ is that certain tasks and missions have tacit constraints that are automatically satisfied in $\mathcal{M}_{\text{plan}}$. For instance, the robot does not always start facing in the direction of its first move, and an extra rotation task is needed to encode missions. Another example is that in some versions of $\mathcal{M}_{\text{plan}}$ only allow charging once the battery charge is below a certain level, and the aforementioned constraints are needed. These caveats led to $\mathcal{V}_{\text{power}}$ generating some missions that cannot be executed in $\mathcal{M}_{\text{plan}}$, and hence unsatisfied integration properties. However, as mentioned before, IPL allows arbitrary logical specifications over views, these constraints are satisfied by using additional variables and constraints in 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 fidelity of visual sensing) of the robot may be required for integration with the same $\mathcal{M}_{\text{plan}}$. This variability is handled using multiple instances of viewpoints: each mode and each combination of features is encoded as a separate viewpoint, creating multiple views for a pair of $\mathcal{M}_{\text{power}}$ and $\mathcal{M}_{\text{map}}$. As a result, each view of $\mathcal{M}_{\text{power}}$ has to be compared with an appropriate version of $\mathcal{M}_{\text{plan}}$ that sets the same mode and uses the same mission features. Thus, this applicability challenge is addressed using the flexibility of views and viewpoints, but it requires extra computational resources for multiple executions of the verification algorithm.

Finally, $\mathcal{V}_{\text{power}}$ 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 $\mathcal{V}_{\text{power}}$ 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

Model integration for System 1 used behavioral properties specified in PCTL for $\mathcal{M}_{\text{plan}}$ to constrain the robot to a particular mission. When constrained to a mission, the robot would have to use the energy that $\mathcal{V}_{\text{power}}$ estimates for that mission, equal to the sum of all tasks' energies. A mission is power-successful according to $\mathcal{M}_{\text{plan}}$ if and only if the robot reaches the goal location in $\mathcal{M}_{\text{plan}}$.

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 $t_1 ... t_4$, starting in the starting location of the first task ($t_1.start$) with a sufficiently charged battery, equal to the sum of energies of $t_1 \dots t_4$:

$$\begin{aligned} P_{max=?}[&(\mathsf{F}\ \underline{loc} = t_2.start) \wedge (\mathsf{F}\ \underline{loc} = t_3.start) \wedge (\mathsf{F}\ \underline{loc} = t_4.start) \wedge \\ &((\underline{loc} = t_1.start)\ \mathsf{U}\ (\underline{loc} = t_2.start\ \mathsf{U}\ (\underline{loc} = t_3.start\ \mathsf{U}\ \underline{loc} = t_4.end)))] \\ &\{\!|\underline{initloc} = t_1.start, \underline{goal} = t_4.end, \underline{initbat} = \Sigma_{i=1}^{4} t_i.energy + \overline{err}_{\text{cons}}|\!\}. \end{aligned} \tag{8.10}$$

The expectation of *expressiveness* for behavioral properties is as follows: the behavioral language should provide sufficient means to express the integration constraints and queries so that the model does not need to be manually changed outside of the behavioral language. In System 1, the expressiveness challenge for behavioral properties of $\mathcal{M}_{\text{plan}}$ is two-fold: (i) constraining the robot's actions to the individual tasks represented by quantified variables, and (ii) enforcing the ordering of these tasks. 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: $\mathcal{M}_{\text{plan}}$ to constrains the robot to make necessary moves in order to visit the locations and make the nested until expression hold. However, the nested until operators do not fully address (ii) because the semantics the until operator in PCTL (see Subsection 5.4.4) allow the robot to skip intermediate locations of the mission. For instance, the temporal constraint $\underline{loc} = l_1\ \mathsf{U}\ (\underline{loc} = l_2\ \mathsf{U}\ (\underline{loc} = l_3))$ is satisfied both by the intended mission that goes through all three locations ($l_1 \rightarrow l_2 \rightarrow l_3$), and by an unintended mission that skips l_2 ($l_1 \rightarrow l_3$) — assuming

that the model has a direct transition from l_1 to l_3. Therefore, to address (ii), additional constraints using the future modality (F) were added to the first line of Equation (8.10).

Charging tasks are expressed implicitly in PCTL properties: if a robot visits a charging station, it gets a choice of charging in $\mathcal{M}_{\text{plan}}$; 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 formed by tasks from $\mathcal{V}_{\text{power}}$.

The *soundness* of behavioral queries is based to the tools that implement the behavioral queries. For $\mathcal{M}_{\text{plan}}$, queries are checked by the probabilistic model checker PRISM. Depending on the configuration of PRISM, some queries may not terminate (by 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, but does not guarantee 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 $\mathcal{M}_{\text{plan}}$. 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* [145], showed impractically long times (several minutes) and memory consumption (over one gigabyte) for one conversion of a property. By using a different conversion tool, Rabinizer 3 [146], it was possible to speed up behavioral checks, making them applicable to all the formulas in this study. The times for behavioral checking were comparable to those for SMT checking (see Table 8.2 in Subsection 8.2.1). Thus, the lack of termination guarantees resulted in limited applicability of PCTL properties.

Customizability of behavioral properties was not required or evaluated in this case study. This customizability is determined by the behavioral model and the logic that describes its properties. In the System 1 study, the logical operators (determined by PCTL and the PRISM input language) sufficiently constrained $\mathcal{M}_{\text{plan}}$ for the purposes of this integration scenario. The initial and state variables of $\mathcal{M}_{\text{plan}}$ have also been sufficient to express the constraints on tasks and missions, making customization of PCTL syntax or semantics unnecessary. However, if needed, additional state variables could be added to the behavioral language. Moreover, the ability to plug in various modal logics contributes to the customizability of IPL, as discussed in Subsection 8.2.1.

8.3.2 Evaluation of Integration Abstractions on System 2

The investigation of integration abstractions for the collision avoidance system was part of an independent study, which had the goal finding appropriate integration abstractions for hybrid programs and $\mathsf{d}\mathcal{L}$ properties. In particular, these abstractions were required to relate hybrid programs to component-based models and integration approaches. This relation is a challenge: hybrid programs do not inherently have an explicit component structure (although it has been recently added by other authors as well [198]), and therefore are difficult to abstract via views (which are component-based).

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 enable reasoning about hybrid programs using the view abstractions. Thus, integration is achieved using the same representations that are used for theorem-proving for $\mathsf{d}\mathcal{L}$ specifications with KeYmaera [93].

The integration abstractions for hybrid programs were defined earlier in this thesis: a description of HP views can be found in Subsection 6.2.4, 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, using the following interpretations of these qualities in this context:

- *Expressiveness:* whether HP views can sufficiently represent component structures of 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 can be a sound and complete representation of hybrid programs with respect to the HP syntax; and whether the checking of $\mathsf{d}\mathcal{L}$ view formulas is sound and guaranteed to terminate.
- *Applicability:* whether HP views contain the information necessary to generate HPs in practice and can support other practical concerns (such as different timing models).
- *Customizability:* whether views can be customized to represent various aspects of HPs; 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) [194, 195]. To support this evaluation, a prototype tool for creation and analysis of these abstractions was implemented in AcmeStudio [243]. The tool and models are available in an archive [239].

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 HP views, see Definition 27 in Subsection 6.3.2). The views should reflect the common component-like and varying parts of hybrid programs: actors (e.g., robots and obstacles), sensing, control, physical dynamics, and so on. In other words, the viewpoint for constructing views of HPs ($\mathcal{VP}^{\text{hp}}$) needs to be sensitive to such variance in the HPs. The important dimensions of this variance for robotic collision avoidance are summarized in Table 3.1 in Section 3.2. If these aspects were not represented in abstractions (i.e., the abstractions of different models were the same), then integration would lack the expressiveness to identify inconsistencies in these aspects. On the other hand, if the abstractions were completely different for similar models, they would not adequately represent the common patterns in HPs that affect integration.

As an example of variance between HPs, consider the robot's possible physical dynamics provided 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

Name	Equations
1D Line	$x' = ov, v' = a, v \geq 0$
Grid	$x' = \frac{(1+o_{hv})o_{fb}}{2}v, y' = \frac{(1-o_{hv})o_{fb}}{2}v, v' = a, v \geq 0$
2D 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.

yet another physical model. Finally, the model of spiral movement is similar to arcs, but 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 views. In this case, an *actor type* is a partially specified actor, with some properties ($State$, $Prts$, $phys$) without fully assigned values (a fully specified actor is defined in (Definition 19). Thus, a fully specific actor can be composed from an arbitrary number of types. If actor a satisfies types $\mathcal{A}$ and $\mathcal{B}$, then the following holds:[8]:

$$\mathcal{A}.State \cup \mathcal{B}.State \subseteq a.State,$$
$$\mathcal{A}.Prts \cup \mathcal{B}.Prts \subseteq a.Prts,$$
$$\mathcal{A}.phys, \mathcal{B}.phys \subseteq a.phys.$$

Extending a type with a sub-type is equivalent to having both types: $(a \in (\mathcal{A} \sqsubseteq \mathcal{B})) \Leftrightarrow (a \in \mathcal{A}) \wedge (a \in \mathcal{B})$. This approach enables representation and reuse of varying elements of HPs. For example, notice that spiral dynamics (Table 8.3) is a more general case of arcs without spinning, so I extend the former with the latter: ARCNOSPINDYNT $\equiv$ SPIRALDYNT $\cup (\emptyset, \emptyset, \emptyset, \{w' = \frac{a}{r}\})$. Then SPIRALDYNT is reused every time an actor is declared with it or ARCNOSPINDYNT.

Variance in actor interactions can be represented with an HP connector, which determines the Trf function (see Subsection 6.2.4 for its definition). This connector encapsulates modeler's expertise about common transformations of the actors' programs, such as IPSC (defined in Subsection 6.2.4) or the *Immediate Bounded Error Sensing (IBES)* connector. Instead of manually introducing new variables, constraining them, and weaving into the code, a modeler achieves the same effect automatically with an HP connector.

The evidence of reusing the view types in HP views supports the claim of expressiveness. Many primitive fragments of hybrid programs, such as $SeqC$, contributed in all model variants. An HP view as a whole was reused three times in a model variant where the robot reaching the goal was modeled at different time moments. This demonstrates utility of architectural $\mathsf{d}\mathcal{L}$ formulas.

[8]The control property $ctrl$, however, is not composed from multiple types: it is required that there is a single source of controller, be it an actor instance or one of actor types.

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. It was surprising that HP connectors were used 28 times (IPS being most common) – almost twice in each model – demonstrating a large amount of component interaction that the views made explicit. Overall, this reuse shows that HP views are sufficiently expressive to compare and differentiate the relevant dimensions of variance in hybrid programs.

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

$$\phi \rightarrow [\mathcal{V}^{\text{hp}}]\psi$$
$$\phi \rightarrow \langle\mathcal{V}^{\text{hp}}\rangle\psi,$$

where ϕ is the HP's precondition, ψ is its postcondition, and $\mathcal{V}^{\text{hp}}$ is its view.

The remaining two properties referenced more than one HP in a property specification. Consider a property below that describes passive friendly safety (this concept is defined in Section 3.2). This property requires that for all executions of a robot and a moving obstacle ($[\text{ROBOTOBST}]$), a robot should always stop or be far enough from the obstacle to stop ($RobotFar$). Assuming the obstacle $\langle\text{DETAILEDOBST}\rangle$ 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 [193]) captures this property:

$$\begin{aligned} Pre \rightarrow [\text{ROBOTOBST}](&RobotFar \wedge \\ &(ObstFar \rightarrow \langle\text{DETAILEDOBST}\rangle Safe)) \end{aligned} \tag{8.11}$$

This $\mathsf{d}\mathcal{L}$ formula includes two hybrid programs that execute independently from each other: once ROBOTOBST, which contains a robot and a non-deterministic obstacle, stops at some point, program DETAILEDOBST starts executing. DETAILEDOBST does not have the robot's code in it explicitly (assuming the robot 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 (8.11) 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 the views have disjoint state spaces, extra statements were added 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

The *soundness* and *completeness* of *HP views* depend on the implementation of $\mathcal{VP}^{\text{hp}}$. This viewpoint is implemented according to Option (ii) described in Subsection 6.2.5: a model can be generated from a given view automatically, but views have to be manually created for existing models. The soundness and completeness are evaluated in relation to the matching predicates described in Subsection 6.2.4, applied to actors, connectors, and the composer of the view. In short, the actors should account for all of the HP code, when transformed by connectors and composed by the composer.

Before assessing soundness and completeness, the first step is to guarantee conformance. Given a view $\mathcal{V}^{\text{hp}}$ and a hybrid program $\mathcal{M}$, I use $\mathcal{V}^{\text{hp}}$ to generate a conforming model $\mathcal{M}'$ according to Definition 23, and check whether $\mathcal{M}$ is equivalent to $\mathcal{M}'$. Upon manual inspection, all 15 views were found conforming to the original models.

Soundness of $\mathcal{VP}^{\text{hp}}$ means that, informally, every element of an HP view maps to some element in an HP according to the matching predicates. 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 part of the view. Upon manual inspection, all 15 views were found sound with respect to $\mathcal{VP}^{\text{hp}}$.

When a view is complete, informally, every model element is guaranteed to be represented in the view. Since the views have been established by splitting the original HPs into actors and connectors, it is sufficient to check that every variable and operator is present in some view. All 15 views were found complete with respect to $\mathcal{VP}^{\text{hp}}$.

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 [93]. This assurance technique does not guarantee termination for arbitrary programs and formulas: in the general case, human assistance may be needed to complete some proof branches, although in practice most proofs are automatically computable. 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 HP views makes it impossible to guarantee their soundness a priori. However, this study has shown that careful creation and checking 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 (i.e., 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 propagating variables, consider different patterns for modeling time in HPs:

- Event-triggered timing (ETT). The time is not represented in a model as an explicit variable. Instead, event conditions are part of an evolution domain constraint F. For example, the system can execute until a certain distance from an obstacle, which is when the time flow is interrupted and the control is handed to the robot, giving it an opportunity to brake.
- 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 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 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 $GCT \equiv ((\{T\}, \emptyset), \emptyset, \emptyset, \{T' = 1\})$, 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 of state models that encapsulate control algorithms, as it is done in Sphinx [194] and component-based contracts for hybrid systems [198].

Application of $\mathsf{d}\mathcal{L}$ *view formulas* was more straightforward than that of HP views: once the views were created, the $\mathsf{d}\mathcal{L}$ 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, which is possible using the provided definitions. It was also possible to encapsulate time in a global actor, although large controller code remained part of some actors without further modularization or reuse. Encoded as $\mathsf{d}\mathcal{L}$ view formulas, behavioral properties did not encounter substantial applicability challenges in this study.

Customizability

The main focus of this study was checking whether views can be *customized* to represent hybrid programs. Therefore, in this study customizability was linked to expressiveness. As discussed earlier, this study demonstrated the creation of a customized viewpoint $\mathcal{VP}^{\text{hp}}$, 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. $\mathcal{VP}^{\text{hp}}$ 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 and views can be used together for the same model, and are not necessarily mutually exclusive abstractions. Specifically, the original $\mathsf{d}\mathcal{L}$ formulas were customized to specify properties for views. To check $\mathsf{d}\mathcal{L}$ view formulas, HPs are generated from the views. Therefore, a model can have both a view abstraction and a behavioral property abstraction, which are related to each other. The practical implication is that views can replace models as engineer-facing artifacts, eliminating the effort required for co-evolution (described in Subsection 6.2.5).

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.

8.3.3 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 8.4.1.

The models for the quadrotor were inspired by the literature and constructed manually (see the details in Subsection 8.4.1), as opposed to being taken from an existing engineering project or dataset. This circumstance makes views *sound* and *complete* a priori: the views were constructed by the researches as proxies of possible models. Hence, only behavioral 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 views and LTL properties can maintain 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 Thread Schedulability and Battery Design

Views were applied as a uniform representation of multiple models related to real-time schedulability and battery design. I created several views in this study, all specified in AADL:

- The scheduling view (V_{sch}) for M_{sch} contains $Thrds$ as components with Dline, their periods, and worst-case execution times as their properties.
- Data security view (V_{sec}) for M_{sec} contains $Thrds$ as components with their security levels.
- The CPU view (V_{cpu}) for M_{cpu} contains $CPUs$ as components, with CPUFreq, $\mathsf{CPUFreq}_{\mathsf{max}}$, and CPUBind as component properties.
- The Rek view (V_{rek}) for M_{rek} has the same viewpoint as V_{sch}: it represents $Thrds$ and their schedulability properties.
- The thermal runaway view (V_{tr}) for M_{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_{bsch}) for M_{bsch} 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 merge implements a basic 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, making them potentially insufficient for checking of analysis contracts. This challenge was addressed by separating the behavioral models from views and expressing behavioral constraints in LTL (which are 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 8.4.1. The behavioral dynamics were modeled separately with Spin models for thread scheduling and battery scheduling, discussed in Subsection 8.4.1. 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 (which are presented in Subsection 8.2.2), SMT performance was near-instantaneous: an SMT query for a merged view took less than 2 seconds.

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 and *completeness* of views have not been evaluated in this case study: the views primarily served as an interchange medium for information, rather than an abstraction for particular existing models. Thus, by construction the views represented the ground truth of the system's design, and hence were 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} using LTL-based expression, was presented in Subsection 8.2.2. That property provides a way to reason about preemption patterns by using the model's reasoning engine (Spin), 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 Spin engine on the Promela model of the scheduler.

Another LTL property was used for querying M_{bsch}. To detect whether a thermal runaway reaction is likely, the following IPL specification iterates through batteries to ensure that thermal neighborhoods (explained in Section 3.3) enable sufficient heat transfer (the terms TN and K are defined in Subsection 8.4.1):

Property 14. In all batteries the thermal neighborhoods 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, the *expressiveness* of behavioral properties for this system is sufficient for querying the temporal dynamics of thread preemption in a scheduler (i.e., which threads are allowed to preempt each other in given circumstances) and thermal neighborhoods in a battery (i.e., how the thermal connectivity of cells is allowed to change over time). One limitation of standard LTL is that it does not allow to compare 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 that they are an appropriate proxy of a thermal runaway, rather than a more complex function of the state, which would be harder to express and check in LTL. Thus, with a more expressive (and still checkable) behavioral property language, it would be possible to specify complex expressions over patterns of thermal connections, potentially leading to a more precise expression of conditions of thermal runaway.

The *soundness* of checking 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.

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 (as discussed 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

Threads	DMS/RMS time (s)	EDF Time (s)
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 behavioral checking for M_{sch} with Spin. MEMLIM indicates that the verification exceeded the memory limit of 30 Gb.

Cells	FGURR Time (s)	FGWRR Time (s)	GPWRR Time (s)
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 behavioral checking for M_{bsch} with Spin. MEMLIM indicates that the verification exceeded the memory limit of 30 Gb.

model can be checked individually, as opposed to their parallel composition. In this study, if M_{sch} was combined with M_{bsch}, the verification times would have been intractable: various possible interleavings of transitions would lead to state-space explosion.

I evaluated the verification of LTL properties on the two aforementioned Promela models using a general-purpose Amazon EC2 virtual machine (`aws.amazon.com/ec2`) 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, LTL properties for M_{sch} are checkable for CPUs that run up to 10 threads. LTL properties for M_{bsch} are checkable for batteries with up to 25 cells. These numbers match the scale of a realistic moderately-sized CPS. Moreover, the memory issues can be mitigated by increasing the size of random-access memory, so this limitation is not fundamental to the approach.

A secondary applicability challenge was to ensure a non-contradictory shared interpretation of thread IDs between views and LTL properties, in the context of M_{sch}. Constructing a consistent background interpretation (I^B) is a responsibility of integration abstractions described in Subsection 6.3.4, in order to enable domain transfer in IPL (Subsection 5.4.1). In this study,

for instance in Property 12 described in Subsection 8.2.2, SMT found pairs of threads that need their deadline-monotonicity checked in M_{sch}. The thread 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.

Customizability of behavioral properties for this system required finding state variables that represent the dynamics relevant to the integration properties. The state variables were not scalars (like the battery charge in System 1), but vectors instead (e.g., the number of thermal neighbors of each kind). To represent them, I used a function TN that takes an index and returns the value at that index in the vector. Furthermore, the function is interpreted differently in every state, allowing the values to change. Similarly, in the property of behavioral deadline monotonicity (Property 12 in Subsection 8.2.2), such a modal function had to take references to architecture components, which added the domain transfer challenge: converting component references to integers. This challenge was addressed at the interface between views and behavioral properties. 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 8.4.1.

8.3.4 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 8.4.2), while views provided a convenient representation of models for these contracts. This section focuses only on the 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 engineering project or existing set of data. Therefore, the views were a priori a ground-truth representation, yielding no insight into their *soundness* or *completeness* (in relation to existing concrete 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 for this system and domain.

Views for Reliability, Trustworthiness, and Control

In this study, the modeling goal for views is to represent the structural elements that are relevant to inter-domain vulnerabilities. The interactions of analyses in this study are determined by the static information from $\mathcal{M}_{\text{fmea}}$, $\mathcal{M}_{\text{trust}}$, and $\mathcal{M}_{\text{ctrl}}$, which I encode the respective AADL views: $\mathcal{V}_{\text{fmea}}$, $\mathcal{V}_{\text{trust}}$, and $\mathcal{V}_{\text{ctrl}}$ (see Subsection 8.4.2 for their definitions). The views are built based on the basic types and structures of an existing AADL model for an autonomous vehicle, created by McGee et al. [181]. 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). I 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 that operates in that mode: sensors ($m.Sns$), controllers ($m.Ctrls$), and actuators.[10] Usage of modes differentiates these views from those for Systems 1–3, since the dynamics in those systems were complex enough to warrant accessing a model through behavioral properties. In contrast, the dynamics of changing attackers is relatively simple in System 4 and can be encoded as modes in views.

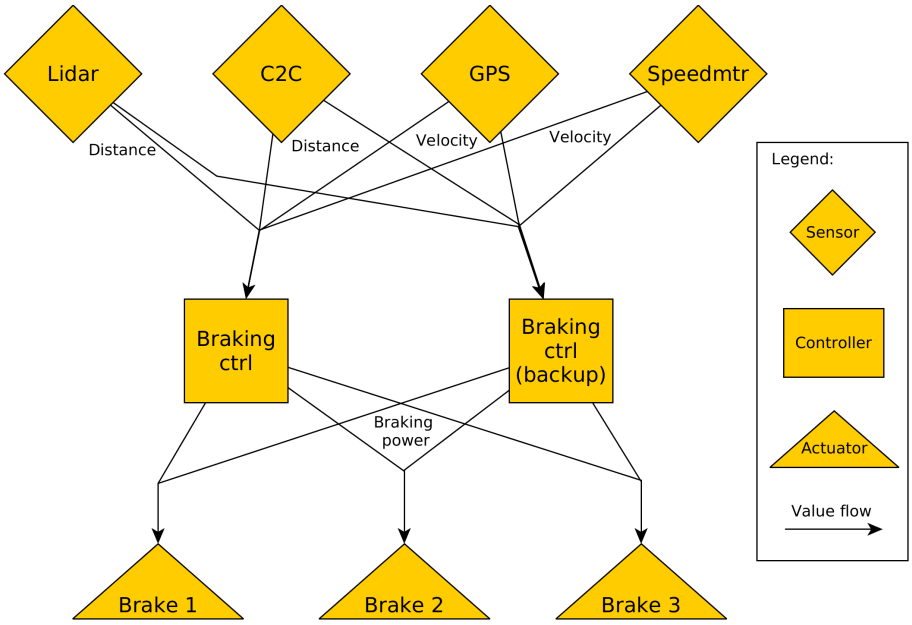


Figure 8.7: An architectural view of the braking subsystem in a self-driving car.

[9]The AADL model with analysis contracts has been archived [238] and is also available online (`github.com/bisc/collision_detection_aadl`).

[10]Actuators are critical components of the system, but they are not modeled explicitly because the focus is on interaction between sensors and controllers.

To simplify the checking of contracts, all views are merged into a single view (see Figure 8.7), by merging identical elements from various views. This operation has been performed manually. The following elements comprise the resulting architectural model:

- Sensors (Sns) have the following properties:
  - Sensed variables $VarsS \subseteq Vars$: 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 ($\mathbb{B}$): whether the sensor is providing data. This property does not presuppose that the data is trustworthy or compromised.
  - Trustworthiness Trust ($\mathbb{B}$): whether the sensor can be compromised by the attacker and is sending untrustworthy data. I use this boolean abstraction of trust for demonstrating how a vulnerability is introduced. For sensors with $\mathsf{Trust} = \bot$ I assume that an attacker can compromise them in any quantity and at any point of time. Even with this relatively simple abstraction, one can exploit a vulnerability, as shown in Table 3.5. More sophisticated models may consider numeric or multidimensional trustworthiness [209] for more precise estimation of confidence in sensor data.
  - Probability of mechanical failure P_{fail} ($\%$): the probability of a sensor mechanically malfunctioning and remaining broken ($\mathrm{avail} = \bot$) 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 ($Ctrls$) have the following properties:[11]
  - Required variables $VarsR \subseteq Vars$: 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 ($\mathbb{B}$): analogous to sensors, whether the controller is turned on by the user or engineer.
  - Availability avail ($\mathbb{B}$): 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 $\mathrm{ctrlsafe}$ ($\mathbb{B}$): whether the controller meets the control performance, safety, and stability requirements.
- System modes Mds (i.e., different configurations) have the following properties:

[11] Although controllers are physical elements and can be attacked, in this system I focus on sensor attacks and assume that direct controller attacks do not occur. 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.

- Required fault-tolerance α_{fail} (%): the maximum acceptable probability of the system's random failure. The final design is expected to malfunction not more likely than α_{fail}.
- Attacker model $atkm$ (internal or external): the type of the attacker considered in the system design. For simplicity, I consider only one dimension — whether the attacker is internal or external. If required, one could model other dimensions such as local or remote attacker. Each attacker model defines a sensor vulnerability evaluation function $\text{isvuln} : Sns \to \mathbb{B}$ that determines whether a particular sensor can be attacked by this attacker. This function abstracts out the 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, as indicated in Table 3.4, is always satisfied, $\text{isvuln} \equiv \top$.

Qualities of Integration

The *expressiveness* provided by the 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 the (limited) dynamism in the system.

This study presents no evidence to support *soundness* or *completeness* of views: the views are created as a single ground-truth medium, and are sound and complete by construction.

The views did not face any outstanding *applicability* challenges in this study: the views successfully captured the information and were analyzable in practice with an SMT solver. In part, this circumstance is due to reusing an existing AADL model from related work with data schemas that were a priori appropriate for this system.

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 the 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 8.4.2.

8.3.5 Summary for Evaluation of Integration Abstractions

Section 8.3 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 in specialized ways (e.g., modes in views or modal functions), thus taking advantage of the views' customizability. In some cases, expressiveness was limited in order to guarantee soundness and applicability (in particular, when checking behavioral properties).

- *Soundness:* soundness and completeness of views are critical to integration arguments in the case study systems, and have been achieved by a combination of automated generation, testing, manual inspection, and iterative debugging. Soundness of behavioral queries is also critical, 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 it to expressiveness. Regardless of their nature, most applicability concerns were satisfactorily addressed in these case studies, supporting Claim 3.
- *Customizability:* in all four case studies, integration abstractions were successfully tailored to the respective 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 for the domain's behavioral models. The customizability of the abstractions often enabled representations that are sufficiently expressive and checkable at the same time, linking customizability to soundness and expressiveness. These findings support Claim 4.

8.4 Validation of Part III: Analysis Execution Platform

The analysis execution platform (AEP) was studied in the context of Systems 3 and 4, since only these systems feature multiple dependent analyses with sophisticated execution contexts. This section describes the case studies of analysis integration in those two systems.

8.4.1 Evaluation of AEP on System 3

The investigation of analysis execution for the quadrotor (System 3, Section 3.3) was an independent case study, with the aim to discover and prevent conflicts between CPS analyses. The discovery process was performed as a literature review: I looked for widespread analyses of embedded systems for correctness, schedulability, and security that are 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 8.2.2.

The 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 (see Subsection 8.3.3 for details). Then, a contract was written for each analysis in terms of the view elements, implementing cases 1 and 2 from Section 4.5). Finally, I performed experiments with analysis execution.

This evaluation focused on the *applicability* and *soundness* of AEP. Soundness means executing the analyses with a guarantee of preventing inconsistencies or design errors (due to missed dependencies and context mismatch). Applicability in this case refers to the ability of the platform to execute the analyses while resolving the dependencies without cycles and avoiding scalability issues. 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 8.3.3.

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 the 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 to specify 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 specifying analysis contracts are represented with a signature Σ_{Sched} (Definition 31). The central sets in Σ_{Sched} are threads ($Thrds$) and CPUs ($CPUs$), which are modelled as component types in AADL. Thus, the meaning of these symbols is established with $I^{\mathcal{V}}_{Sched}$ (or just $I^{\mathcal{V}}$ in this context) 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 32). Σ_{Sched} also contains symbols for 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 ($SecCls$) and thread scheduling policies ($SchedPols$):

$$I^{\mathcal{V}}_{Sched}(SecCls) = \{\mathbf{normal}, \mathbf{secret}, \mathbf{topsecret}\}, \qquad (8.12)$$
$$I^{\mathcal{V}}_{Sched}(SchedPols) = \{\mathbf{rms}, \mathbf{dms}, \mathbf{edf}\}.$$

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 q the function *itself* might be different): $q(\mathrm{canprmt}) : T \times T \mapsto \mathbb{B}$, such that $q(\mathrm{canprmt}(t_1, t_2)) = \top$ iff t_1 can preempt t_2 in state q. To instantiate M_{sch}, one needs a set of threads (sharing the same CPU) and a scheduling policy as initialization parameters.

In addition to the above view and model symbols, the domain signature contains background sorts — Booleans $\mathbb{B}$, integers $\mathbb{Z}$, and reals $\mathbb{R}$ — that are shared between views and behavioral models and can be used in contract specifications.

Now I describe how the above symbols are interpreted. The view symbols (components and their properties) are interpreted on concrete instances of AADL models, which play the role

[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	$Thrds \mapsto \mathbb{Z}$	Thread's period.
Dline	$Thrds \mapsto \mathbb{Z}$	Thread's deadline.
WCET	$Thrds \mapsto \mathbb{Z}$	Thread's worst-case execution time.
ThSecCl	$Thrds \mapsto SecCls$	Thread's security class.
CPUSchedPol	$CPUs \mapsto SchedPols$	CPU's scheduling policy.
CPUFreq	$CPUs \mapsto \mathbb{R}$	CPU's normalized frequency[12].
notcoloc	$Thrds \mapsto 2^{Thrds}$	A thread t is mapped to a set of threads that should not share the same CPU as t.
CPUBind	$Thrds \mapsto CPUs$	Thread-to-CPU binding.
ThSafe	$C \mapsto \mathbb{B}$	Flag for whether a CPU's threads are thread-safe.
Voltage	$() \mapsto \mathbb{R}$	Required system voltage[13].

Table 8.6: Properties of threads and CPUs in Σ_{Sched}.

of views and constitute a context Υ. $SchedPols$ and $SecCls$ are interpreted as enumerations 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 [120]) 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 $\left|I^{\mathcal{V}}(C)\right|$ processors, and each running task is allocated to a processor dynamically. For each periodic real-time task (t), a state q interprets the following propositions:

- $\mathsf{Prior}(t) : \mathbb{Z}$ – the priority of t.
- $\mathsf{Run}(t) : \mathbb{B}$ – whether a job of t is dispatched on a processor.
- $\mathsf{InQ}(t) : \mathbb{B}$ – 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 M_{sch}, and threads t_1, t_2, $q(\text{canprmt})(t_1, t_2)$ is $\top$ if and only if the following holds in state 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 $q(\text{canprmt})(t_1, t_2)$ 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 needs to represent variance in execution time, without exceeding each thread's maximum execution time. The program does so without using explicit time counters (which would substantially increase the number of states) in the following way. 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 $\mathrm{canprmt}$ 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 of batteries, which change 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 ($Batts$), which are rectangular arrays of cells (with $batrows$ rows and $batcols$ columns). Three mutually exclusive policies for scheduling cell connections ($ConnSchedPols$) 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 ($\mathbf{GPwRR}$) [142, 143]. One goal of these policies is to select the cells to discharge to maintain a constant output voltage (Voltage), which is a property that intersects with the scheduling domain. Note how specifying domain signatures makes it possible to naturally capture domain overlaps through views. A given voltage is maintained by arranging a number of cells in series (SerialReq) and in parallel (ParalReq). Another goal of connection scheduling is to ensure a battery lifetime that matches the product specifications (which is represented as a flag HasReqdLifetime). The background sorts ($\mathbb{B}, \mathbb{Z}, \mathbb{R}$) 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 $\mathsf{TN} : Batts \times \mathbb{Z} \mapsto \mathbb{Z}$. 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 [141]: 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 triggers a thermal runaway in battery b if it violates the condition:

$$\sum_i \mathsf{K}(b, i) \times q(\mathsf{TN}(b, i)) \geq 0 \tag{8.13}$$

[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
Voltage	$() \mapsto \mathbb{R}$	Required system voltage.
$batrows$	$Batts \mapsto \mathbb{Z}$	Battery's cell rows.
$batcols$	$Batts \mapsto \mathbb{Z}$	Battery's cell columns.
$BatConnSchedPols$	$Batts \mapsto ConnSchedPols$	Battery's cell scheduling policy.
SerialReq	$Batts \mapsto \mathbb{Z}$	Number of cells required to connect in series.[16]
ParalReq	$Batts \mapsto \mathbb{Z}$	Number of cells required to connect in parallel.[17]
K	$Batts \times \mathbb{Z} \mapsto \mathbb{Z}$	Weight of cells with i thermal neighbors.
HasReqdLifetime	$Batts \mapsto \mathbb{B}$	Flag whether a battery has the required lifetime.

Table 8.7: Properties of batteries in Σ_{Batt}.

The exact values of K are not known a priori, and are determined experimentally for each type of batteries. 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 ($batrows$, $batcols$) and requirements (SerialReq, 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 $x \in [0, batrows - 1]$ and $y \in [0, batcols - 1]$:

- $\mathsf{CellCharge}(c)$ is the charge of c. To simplify model checking I 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 $\mathbf{idle}$.
- $\mathsf{Gr}(c)$ 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 $\mathsf{IsTNbr}(c_1, c_2)$,

[16] SerialReq is a battery-specific form of the voltage output requirement.
[17] ParalReq is a battery-specific form of the electrical current output requirement.

if: (i) $c_1 \neq c_2$; (ii) $\mathsf{Gr}(c_1) = \mathsf{Gr}(c_2)$; (iii) $|c_1.x - c_2.x| + |c_1.y - c_2.y| \leq TNDIST$ [18]; (iv) $\mathsf{CellCharge}(c_1) = \mathsf{CellCharge}(c_2) = \top$; and (v) $\mathsf{CellSt}(c_1) = \mathsf{CellSt}(c_2) = \textbf{discharging}$. The number of thermal neighbors of cell c is $\mathsf{TNbr}(c) = |\{c' \in \chi \cdot \mathsf{IsTNbrs}(c, c')\}|$. Finally, $\mathsf{TN}(b, i) = |\{c' \in \chi \cdot \mathsf{TNbrs}(c) = i\}|$.

The above Kripke structure is implemented as a single-process Promela program. The program maintains the state variables CellCharge, CellSt, Gr, and TN 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., 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 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 ParalReq groups. **GPWRR** assembles groups by packing as many charged cells into each group as possible. Then it selects the top ParalReq most charged groups to discharge. Within each group, all schedulers select 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 subsumes any high-fidelity model of charge. Thus, the M_{bsch} represents the logic of the cell schedulers and abstracts away the exact charge of the cells. These modeling choices contribute to the soundness of the approach: the model is an overapproximation that does not miss 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 ($\mathsf{A}_{\text{SecAlloc}}$) has contract $\mathcal{C}_{\text{SecAlloc}} : \mathbb{I} = \{Thrds, \mathsf{ThSecCl}\}$, $\mathbb{O} = \{\text{notcoloc}\}$, $\mathbb{A} = \emptyset$, and $\mathbb{G} = \{\mathsf{g}\}$ where g is:

$$\forall t_1, t_2 : Thrds \cdot \mathsf{ThSecCl}(t_1) \neq \mathsf{ThSecCl}(t_2) \rightarrow t_1 \in \text{notcoloc}(t_2).$$

Thus, $\mathsf{A}_{\text{SecAlloc}}$ makes no assumptions, but guarantees that threads with different security classes are never co-located.

The bin packing analysis ($\mathsf{A}_{\text{BinPack}}$) has contract $\mathcal{C}_{\text{BinPack}}$: $\mathbb{I} = \{Thrds, CPUs, \text{notcoloc}, \mathsf{Per}, \mathsf{WCET}, \mathsf{Dline}\}$, $\mathbb{O} = \{\mathsf{CPUBind}\}$, $\mathbb{A} = \emptyset$, and $\mathbb{G} = \{\mathsf{g}\}$ where g is:

$$\forall t_1, t_2 : Thrds \cdot t_1 \in \text{notcoloc}(t_2) \rightarrow \mathsf{CPUBind}(t_1) \neq \mathsf{CPUBind}(t_2).$$

[18] For the calculations I use $TNDIST = 2$.

Thus, $\mathsf{A}_{\text{BinPack}}$ makes no assumptions but guarantees that threads that should not be co-located are never scheduled on the same CPU.

Frequency scaling ($\mathsf{A}_{\text{FreqSc}}$) has contract $\mathcal{C}_{\text{FreqSc}}$: $\mathbb{I} = \{\mathit{Thrds}, \mathit{CPUs}, \mathsf{CPUBind}, \mathsf{Dline}\}$, $\mathbb{O} = \{\mathsf{CPUFreq}\}$, $\mathbb{A} = \{\mathsf{a}\}$, and $\mathbb{G} = \emptyset$, where a is:

$$\forall t_1, t_2 : \mathit{Thrds} \cdot t_1 \neq t_2 \wedge \mathsf{CPUBind}(t_1) = \mathsf{CPUBind}(t_2) \rightarrow \\ \mathsf{G}\,(\mathrm{canprmt}(t_1, t_2) \rightarrow \mathsf{Dline}(t_1) < \mathsf{Dline}(t_2)).$$

Thus, $\mathsf{A}_{\text{FreqSc}}$ 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 equivalent to DMS in some model, e.g., rate-monotonic scheduling in a model with threads equal to deadlines, even though it is not deadline-monotonic by design. This property has been discussed in more detail for validation of IPL in Subsection 8.2.2, and mentioned for validation of LTL properties as integration abstractions in Subsection 8.3.3.

Model checking with REK (A_{Rek}) has contract $\mathcal{C}_{\text{Rek}}$: $\mathbb{I} = \{\mathit{Thrds}, \mathit{CPUs}, \mathsf{Per}, \mathsf{Dline}, \mathsf{WCET}, \mathsf{CPUBind}\}$, $\mathbb{O} = \{\mathsf{ThSafe}\}$, $\mathbb{G} = \emptyset$, and $\mathbb{A} = \{\mathsf{a}_1, \mathsf{a}_2\}$ where:

$$\mathsf{a}_1 \triangleq \forall t : \mathit{Thrds} \cdot \mathsf{Per}(t) = \mathsf{Dline}(t),$$
$$\mathsf{a}_2 \triangleq \forall t_1, t_2 : \mathit{Thrds} \cdot \mathsf{G}\,(\mathrm{canprmt}(t_1, t_2) \rightarrow \mathsf{G}\,\neg\,\mathrm{canprmt}(t_2, t_1)).$$

The Rek model checker [39] 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. 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 8.2.2 and mentioned for validation of LTL properties as integration abstractions in Subsection 8.3.3.

Thermal runaway ($\mathsf{A}_{\text{ThermRun}}$) has contract $\mathcal{C}_{\text{ThermRun}}$: $\mathbb{I} = \{\mathit{Batts}, \mathit{batrows}, \mathit{batcols}, \mathsf{Voltage}\}$, $\mathbb{O} = \{\mathsf{K}\}$, $\mathbb{A} = \emptyset$, 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 $i : \mathbb{Z} \in [0, 3]$. $\mathsf{A}_{\text{ThermRun}}$ determines K through experimentation, adjusting K so that acceptable heat propagation patterns satisfy (8.13), and the unacceptable ones violate it. Note that $\mathsf{A}_{\text{ThermRun}}$ 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 ($\mathsf{A}_{\text{BatSched}}$) has contract $\mathcal{C}_{\text{BatSched}}$: $\mathbb{I} = \{\mathit{Batts}, \mathit{batrows}, \mathit{batcols}\}$, $\mathbb{O} = \{\mathit{BatConnSchedPols}, \mathsf{HasReqdLifetime}, \mathsf{SerialReq}, \mathsf{ParalReq}\}$, $\mathbb{A} = \emptyset$, and $\mathbb{G} = \{\mathsf{g}\}$ where g is:

$$\forall b : \mathit{Batts} \cdot \mathsf{G}\,(\mathsf{K}(b,0) \times \mathsf{TN}(b,0) + \mathsf{K}(b,1) \times \mathsf{TN}(b,1) + \\ \mathsf{K}(b,2) \times \mathsf{TN}(b,2) + \mathsf{K}(b,3) \times \mathsf{TN}(b,3) \geq 0).$$

$\mathsf{A}_{\text{BatSched}}$ computes a battery cell connectivity scheduler that maximizes the battery lifetime given the battery characteristics and output requirements. It sets the flag $\mathsf{HasReqdLifetime}$ indicating whether the battery, given its selected scheduler, meets the lifetime requirement. Since

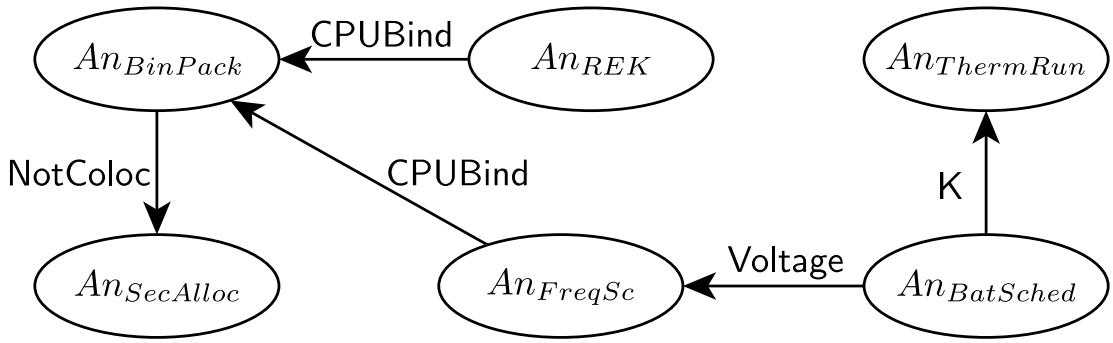


Figure 8.8: Analysis dependency graph for System 3.

the scheduling is not aware of the thermal runaway, the determined scheduler needs to be verified against the thermal runaway pattern, hence the guarantee. $\mathsf{A}_{\text{BatSched}}$ 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 different designs of the quadrotor, to examine the application of AEP 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 8.8. 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 an assumption or guarantee fails in the middle of an 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 $I^{\mathcal{V}}\mathsf{Per} = \{t_1 \mapsto 100, t_2 \mapsto 150, t_3 \mapsto 200\}$, $I^{\mathcal{V}}(\mathsf{Dline}) = \{t_1 \mapsto 100, t_2 \mapsto 90, t_3 \mapsto 200\}$, $I^{\mathcal{V}}(\mathsf{WCET}) = \{t_1 \mapsto 10, t_2 \mapsto 15, t_3 \mapsto 20\}$ are allocated to a single CPU. Before analysis $\mathsf{A}_{\text{FreqSc}}$ is applied to determine CPU frequencies, its assumption $\mathcal{C}_{\text{FreqSc}}.\mathsf{a}$ is verified. Recall that $\mathcal{C}_{\text{FreqSc}}.\mathsf{a}$ 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 $\mathcal{C}_{\text{FreqSc}}.\mathsf{a}$ 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 $\mathcal{C}_{\text{FreqSc}}.\mathsf{a}$ (assuming the tasks release their first jobs at the same time). Thus, AEP not only prevents incorrect

usage of $\mathsf{A}_{\text{FreqSc}}$, but also extends the application of this analysis to EDF, beyond its original scope (i.e., DMS designs).

Next, suppose the quadrotor has a single battery with $batrows = batcols = 4$, and a voltage requirement $\mathsf{ParalReq} = \mathsf{SerialReq} = 3$. It has been observed in related work [141] 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 (8.13) simulates this intuition.

After executing $\mathsf{A}_{\text{BatSched}}$ on the above design, which picks a battery scheduler, the IPL algorithm verifies its guarantee $\mathcal{C}_{\text{BatSched}}.\mathsf{g}$. Since $\mathsf{A}_{\text{BatSched}}$ 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 (8.13) 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 $\mathsf{A}_{\text{BatSched}}$ does not.

The above experiments provide additional confidence in *soundness* of the approach to analysis integration. Specifically, it shows that in practice, AEP 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 8.3.3) and soundness of behavioral queries to correctly determine whether a context is appropriate (discussed for System 3 in Subsection 8.3.3).

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 8.3.3 (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 8.3.3).

Another applicability concern is the 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 can be used beyond their original intended context: if the contract specification holds, then the context is appropriate — even if the original creator of the analysis did not envision that. This benefit has been demonstrated for A_{Rek} and $\mathsf{A}_{\text{FreqSc}}$.

To summarize, in this case study, six analyses for a quadrotor design were integrated by the means of contracts. The study has found that AEP 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 8.3.4.

8.4.2 Evaluation of AEP on System 4 

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 a Jeep through its sensors [106], published at the time of this case study.

To investigate this application 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. The description of the views for the autonomous car and their evaluation can be found in Subsection 8.3.4. After the views were constructed, the analysis contracts were written and evaluated in terms of dependencies and contexts of the analyses. This section focuses only on applicability and soundness of AEP:[19]

- *Soundness:* the ability of AEP to execute the analyses according to their dependencies and only in appropriate contexts.
- *Applicability:* the ability of AEP to avoid dependency loops and reflect the analysis creator's intent in the contracts.

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 8.3.4. To remind the reader, the architectural types are sensors (Sns), controllers ($Ctrls$), and modes (Mds). 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.

The FMEA analysis (A_{fmea}) searches for a component redundancy structure [20] that is capable of withstanding the expected random failures of individual components and create a system with a probability of failure no larger than α_{fail}. Hence, one output of FMEA is an architecture of sensors and controllers.

[19]The expressiveness and customizability were not evaluated directly because these qualities were determined by the views that were used by the platform.

[20]This analysis is constrained by the costs of components (in terms of the available funds, physical space, and other resources): the trivial solution of replicating each sensor a large number of times would typically not be acceptable.

Another output of FMEA is a set of likely failure modes. [21] The output contains the failure modes (i.e., system configurations with some unavailable sensors, for which $\mathrm{avail} = \bot$) that need to be checked for the system to be safe.

A typical FMEA assumption is that the random mechanical/hardware failures are independent across 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 A_{fmea} is as follows:

- Inputs: P_{fail}, α_{fail}.
- Outputs: Sns, $Ctrls$, Mds.
- Assumption. *Component failure independence:* if one component fails, another component is not more likely to fail:
$$\forall c_1, c_2 \in Sns \cup Ctrls \ \cdot P(\neg c_1.\,\mathrm{avail} \mid \neg c_2.\,\mathrm{avail}) \leq P(\neg c_1.\,\mathrm{avail}) + \epsilon_{fail}.$$
- Guarantee. *Functioning controllers:* each controller variable is provided by some sensor:
$$\forall m \in Mds \cdot \forall c \in m.Ctrls \cdot \forall v \in c.VarsR \cdot \exists s \in m.Sns \cdot v \in s.VarsS.$$

The sensor trustworthiness analysis $\mathsf{A}_{\text{trust}}$ determines the possibility of each sensor being compromised (represented with a flag Trust) given their placement, power status, availability, and the selected attacker model ($atkm$). To avoid ambiguity, I assume that unpowered and unavailable sensors cannot be compromised. Therefore, $\mathsf{A}_{\text{trust}}$ marks a sensor as untrustworthy if and only if the sensor is powered, available, and vulnerable for the given attacker model:

$$\forall s \in Sns : \neg \mathsf{A}_{\text{trust}}(s) \iff s.\mathsf{Pow} \wedge s.\,\mathrm{avail} \wedge atkm.\,\mathrm{isvuln}(s).$$

$\mathsf{A}_{\text{trust}}$ treats failures differently compared to 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 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 $\mathsf{A}_{\text{trust}}$ is specified as follows:

- Inputs: Sns, Place, Pow, avail, $atkm$.

[21] The definition of probability for failure modes may differ depending on the system requirements. For example, one may consider failure modes with probabilities $\geq 0.1\alpha_{\text{fail}}$.

- Output: Trust.
- Assumption. *Component failure dependence:* some components are likely to fail together:
$$\exists c_1, c_2 \in Sns \cup Ctrls : P(\neg c_1.\,\text{avail} \mid \neg c_2.\,\text{avail}) \geq P(\neg c_1.\,\text{avail}) - \epsilon_{trust}$$
- Guarantee. *Correct trustworthiness assignment:* a sensor is not trustworthy if and only if it is vulnerable for the considered attacker model:
$$\forall m \in Mds, s \in m.Sns \cdot s.\mathsf{Trust} = \bot \iff m.atkm.\,\text{isvuln}(s).$$

The control safety analysis (A_{ctrl}) determines whether the control has the required performance standards: it 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 to refine the contracts further.

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 related work [75], 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 that at least half of sensors are trustworthy by establishing a mapping function f (for each variable) between trustworthy and untrustworthy sensors. Existence and surjectivity[22] 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 A_{ctrl}:

- Inputs: Sns, $VarsS$, $Ctrls$, $VarsR$.
- Output: ctrlsafe.
- Assumption. *Minimal sensor trust* — for each untrusted sensor there is at least one different trusted sensor [23]:
$$\begin{aligned}
&\forall m \in Mds\ \forall c \in m.Ctrls, v \in c.VarsR \cdot \\
&\quad \exists f :\ Sns \to Sns\ \cdot \forall s_u \in m.Sns \cdot \\
&\qquad v \in s_u.VarsS \wedge s_u.\mathsf{Trust} = \bot \to \\
&\qquad \exists s_t \in m.Sns \cdot v \in s_t.VarsS \wedge s_t.\mathsf{Trust} = \top \wedge f(s_t) = s_u.
\end{aligned}$$
- Guarantees: not specified.

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 ($s.\mathsf{Trust} = \bot \iff atkm.\,\text{isvuln}(s)$), the system's control is safe ($\text{ctrlsafe} = \top$), 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.

[22] A *surjective* function maps some argument to every value in its range.

[23] This assumption can be written in a simpler form, "at least half of the sensors are trustworthy": $\forall m \in Mds \cdot |m.S_{trustworthy}|/|m.Sns| \geq 0.5$. Unfortunately, such statements cannot be verified using state-of-the-art SMT, and theories with set cardinalities have not been implemented for SMT yet.

Analysis Execution

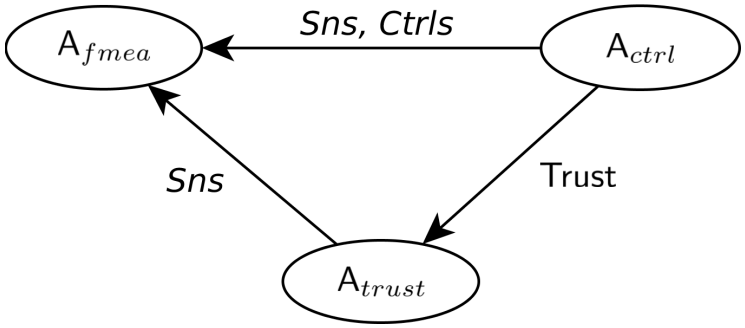


Figure 8.9: Dependencies of analyses for System 4.

Before the analyses are executed, their dependencies need to be resolved. Dependency calculation for A_{fmea}, $\mathsf{A}_{\text{trust}}$, and A_{ctrl} yielded the following dependency graph (shown in Figure 8.9):

- A_{fmea} does not depend on any analyses considered in this case study.
- $\mathsf{A}_{\text{trust}}$ depends on A_{fmea} that outputs Sns — an input for $\mathsf{A}_{\text{trust}}$.
- A_{ctrl} depends on A_{fmea} that outputs Sns and $Ctrls$ — inputs for A_{ctrl}.
- A_{ctrl} depends on $\mathsf{A}_{\text{trust}}$ that outputs Trust — part of an assumption for A_{ctrl}.

I executed the analyses according to these dependencies in the ACTIVE tool (see Section 7.4), in the order of A_{fmea}, $\mathsf{A}_{\text{trust}}$, and A_{ctrl}. For example, if the user changes $atkm$ and tries to execute A_{ctrl}, $\mathsf{A}_{\text{trust}}$ is executed first so that the assumption of A_{ctrl} is verified on values of Trust that are consistent with $atkm$. Moreover, before $\mathsf{A}_{\text{trust}}$ is executed, A_{fmea} is executed since $\mathsf{A}_{\text{trust}}$ (and 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 8.4.1). The assumptions and guarantees for analysis contexts used three types of specifications: first-order deterministic statements, second-order deterministic statements, and first-order probabilistic statements. All contracts other than the second-order deterministic ones are expressible and checkable in IPL. The deterministic logical contracts 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 [112]). 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 [154] or MRMC (Markov Reward Model Checker) [138].

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 (a mix of automated verification

and manual inspection) found no violations, providing secondary evidence of soundness.

In this study, the *applicability* of AEP 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 loop by letting A_{fmea} work on the set of sensors, and have A_{trust} perform trustworthiness calculations on the set of sensors. Thus, the applicability of AEP was demonstrated.

To summarize, the case study of System 4 has shown applicability and soundness of AEP by integrating analyses from several domains (other than real-time schedulability and battery design, as in System 3). It was shown that the inputs, outputs, assumptions, and guarantees for the domain of reliability/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 8.3.4.

8.4.3 Summary for Evaluation of AEP

Section 8.4 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, provided there are no circular dependencies. The soundness of context checking follows from implementing the checks of assumptions and guarantees using the IPL verification. These results provide evidence for Claim 2. The soundness of properties to which IPL is not currently applicable has not been studied.
- *Applicability:* AEP 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. The evidence supporting Claims 1 and 4 can be found in Section 8.3.

Chapter Summary

This chapter presented multiple validation studies from two perspectives. First, a theoretical analysis of soundness showed that when views that are appropriately sound and complete, and behavioral queries are sound, the model integration using the presented approach is sound (i.e., the integration properties can only be satisfied by models that are inconsistency-free). Second, four empirical validation studies assessed customizability, applicability, and expressiveness of the approach in diverse real-world settings, leading to a conclusion that the approach satisfies the claims made in the thesis statement (see Section 1.1).

Chapter 9

Related Work

This chapter discusses the research related to this thesis, split into 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 for CPS

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 formal languages of this category include Alloy [129], TLA+ [156], Z , Object-Z [247], VDM-SL [160], and B [157]. The models in these languages 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 [118] and FSP [174], transition systems [264], Harel statecharts [113], UML statecharts [74] and Promela/Spin [120], PlusCal [156], Petri nets [69], dynamic logic (accompanied with JML specifications) [114, 131], 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 formal notations as Event-B [1], UML [74], SysML [63], and CML [265]. These methods can be considered modeling frameworks using several related models. Relating these models at the language level makes it easy to check some consistency properties (e.g., referential integrity), but does not fully solve the problem of model integration. Combinations of models allow for multiple flavors 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 [163]: the necessary granularity of discretization is usually unknown, and high enough granularity is often impractical due to limited scalability. However, many of the these models have been extended to support continuity (e.g., Hybrid Event-B [15]), as discussed in the next section.

9.1.2 Continuous and Hybrid Modeling Methods

On the other side of the spectrum are classic modeling methods, which rely on differential and difference equations [262]. 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., 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 (explicitly) 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 [91]), described by ordinary differential equations. These models enable simulation, theoretical analysis (stability, safety, robustness) and, in some cases, admit closed-form solutions to simplify prediction and other analyses.

Continuous models are often used in control engineering. In practice, causal signal-flow notations (Simulink[58] and SCADE [73]) are used for designing control and plant (environment) models, enabling simulation analysis of empirical properties like rise time, overshoot, and setting time [47]. Signal-flow models were extended with discrete state descriptions (see StateFlow [111])

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 [136] and by statistical model checking [216]. Controller design can be bounded by more abstract, provably safe verification models, such as those designed in KeYmaera X [93].

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 [23, 214]. Typical analyses of hybrid systems are based on forward/backward reachability: given a set of states, determine the reachable or past states by following the system dynamics. To compute flowpipes, automated tools (e.g., SpaceEx [89] and Flow* [42]) use various geometric approximations of state sets, such as rectangular hulls, polyhedra, and ellipsoids. Another way to analyze hybrid systems is to specifying their invariants and proving them, as done for the differential dynamic logic and hybrid programs [215]. 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 8.3.2.

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 Kronos [268] and UPPAAL [159]. Timed automata were used as a semantic basis for component-based models such as BIP [20] and EAST-ADL [178], 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. Recently, there have been advances in finding algebraic invariants [217] and connecting hybrid systems to CPS implementations (e.g., VeriPhy [29]). This thesis also alleviates some of these difficulties from the modeling standpoint, by designing component-based integration abstractions (see Section 6.2) [236].

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 Integration 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.

9.2.1 Software and Systems Architecture

My approach builds on a special type of discrete models that originate in the field of software architecture [244]. Architectural models are hierarchical collections of components and connectors. The two architecture description languages used in this thesis are AADL [80] and Acme [98]. 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) [254], software architecture relies on a concept of a viewpoint — a perspective from which an architectural model (or, a view) is created [117, 128]. This concept allows me to use views to represent the parts of models relevant to a particular integration scenario, as described in Subsection 6.2.2.

Multiple prior works have investigated the problem of consistency between software views [45, 67, 199, 223]. 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 [210], which is used in views that were extended to represent physical elements of CPS and perform consistency via graph mappings [26] and arithmetic constraints [227]. 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 [104].

A relatively recent modification of architectural models — Distributed Emergent Ensembles of Components (DEECo) [35, 36] — 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 6.2.4), 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.

9.2.2 Logic and Verification

The other integration abstraction (behavioral properties) draws on the field of logic, in particular on modal logics and model checking [50, 112, 176]. 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 [13, 154]. 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 plugins.

The design of IPL is inspired by combinations of the first-order logic [30] with various modal logics. The LTL plug-in draws on the seminal work of Manna and Pnueli [176] on first-order LTL, which has been instantiated in many contexts [46, 99]. Another example is the Quantified Computation Tree Logic (QCTL) [57]. 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 [46] in that I do not create a full quantification structure in each temporal state. Instead, the IPL design follows earlier proposals of combining open reasoning systems [102], 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 Satisfiability Modulo Theories (SMT) solving [204]. 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 [203]. The use 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 [60]) can use heuristics to solve problem instances in undecidable theory combinations. IPL interfaces with SMT solvers using the SMT-LIB textual format [19], which enables seamlessly switching between SMT solvers as needed.

The concept of an analysis in this thesis has been borrowed from earlier research on open analytic models [201]. This concept generalizes a broad set of operations on models, such as model transformations, which have been studied extensively for software models [100, 109, 148, 158, 229, 259]. 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.

9.2.3 Compositionality, Contracts, and Dependencies

The analysis contracts are inspired by assume-guarantee reasoning that have been used extensively for compositional verification, dating back to at least applications of contracts to programming languages [185]. 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 [201] 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 [129]). 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 [220, 223, 269]. Often, a dependency is a relation between two artifacts where one artifact reads or uses the other (e.g., for compilation or execution). A dependency between decisions or operations is a relation where the subsequent decision/operation requires the preceding one to be completed. In CPS, dependencies can be represented in the Dependency Modeling Language (DML) [220], 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 [64]). 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 Existing 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 [67, 71,

207]. 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, 118, 247]. 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 [104, 137, 250].

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 decompositions e.g. into components) to semantic ones (reconciling models/analyses at the level of their meaning. e.g. the behaviors they enable) [234]. In the end of this section I review the approaches that combine the two perspectives. From the standpoint of classifications of integration problems [10, 11, 263], I focus on approaches to data, platform, and process integration — the categories of integration problems addressed in this thesis.

9.3.1 Structural Approaches

Several structural approaches extend the software consistency methods for CPS. One example is *model transformations* [166, 178]. 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 [205] 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 [132], which assumes a shared semantic basis of models in order to make models and their logical properties equivalently translatable 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 [68]. 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* [258] and *metamodels* [255]. In integration, metamodels are used to represent relations between models by connecting the types of their elements [184]. 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 [186, 187], 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 [212, 255]. 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 [66]. 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) [255]. 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 [24, 53, 206, 242]. The platform-based design approach, supported by the metroII environment [59], 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 [26] 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 [194] 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 6.2.4 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).

9.3.2 Semantic Approaches

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 [224, 225], as well as the prior work on refinement [169], simulation, and bi-simulation relations [88, 101, 103]. This approach bypasses the issue of syntactic differences between formal notations, 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 [126], and discrete transition systems were related to hybrid automata [14] using contract automata (which conceptually represent a behavior relation).

Another semantic approach is implemented in the OpenMETA toolchain, developed at Vanderbilt University, for integration of domain-specific languages [251, 252]. This toolchain is organized in accordance with platform-based design [140] and tackles model integration on three levels: models, tools, and execution. At the model level, this approach uses CyPhyML – a component-based integration language [246] for describing semantic mappings (written as FORMULA [130] 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 [219]. 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 [165] 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 [55] and combinations of VDM [28] with bond graphs [261] and 20-sim [85]. 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 [94]). One example is hybridization that develops one logic's features on top of another logic, leading to potential reuse of modalities and operators [16].

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 [149], combinations of logics merge their model structures, which may lead to tractability challenges in practice. My approach keeps models separate, allowing heterogeneous behavioral semantics to be changed independently from each other.

9.3.3 Mixed Approaches

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 [37]. 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 [127] 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 [21] (translated to timed automata) and BIP [20] (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) [265], which combines several models, including SysMl, VDM, and CSP. This approach relies on multiple viewpoints and combines architectural abstractions with multiple behavioral notations. 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 successful a priori integration of models based on view abstractions. My approach, in contrast, enables a posteriori integration when the notations and models are not necessarily picked from a known set.

Finally, some works focus directly on interactions between CPS analyses. For example, JANI [34] 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 [119]. 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.

Chapter Summary

To summarize, several key differences distinguish this thesis from the existing work. 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 simpler analyses). Finally, my approach does not require a shared underlying semantics or meta-model of 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. Second, I summarize the limitations of the approach, including the concerns about 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 of Applicability

I describe the scope of applicability of the integration approach in four dimensions:

1. The models that the approach can integrate.
2. The analyses that the approach can integrate.
3. The integration properties that the approach can check for the above models and analyses.
4. The domains and systems to which the approach can be applied.

I start by considering the scope in terms of the *models* that can be integrated. The structural abstractions (views) apply to a wide range of structures in models, representing static hierarchical key-value information in these structures. This representation is independent of the particular form of dynamics and behaviors. However, views make three assumptions about structural models:

- There is a finite number of relevant model elements, and they can be represented by a finite number of view elements. Without this assumption, the saturation step of the IPL verification algorithm (Section 5.5) is not guaranteed to terminate.
- Structural models follow the closed-world assumption: the elements that are not specified in the model are assumed to not be part of it. Thus, views cannot handle incomplete models or models with uncertain membership of elements. Every model is assumed to be fully described. This assumption is required because views and viewpoints do not have a mechanism to handle model elements that may be available, but are not part of the model. Note that a model can be incomplete with respect to one viewpoint, but complete (and therefore within the scope) with respect to another viewpoint. I return to the discussion of model completeness later in this chapter.

- Structural models do not contain uncertainties that would lead to unknown or undefined values of properties for view elements. For instance, if a power model has a range of potential energy values for some process, this range cannot be accurately represented with a real-valued scalar property of a view element. This assumption can be satisfied by making the view more complex (e.g., including multiple properties such as mean and variance) or relaxing the matching predicate to allow for inaccuracies (which may negatively affect the checking of the desired integration property).

The requirements for behavioral models in my approach are more restrictive than for the structural ones. First, a behavioral model has to provide interpretations to sentences of a behavioral property language, with a sound algorithm to query the values of these sentences. The algorithm should terminate, at least in practical conditions if not theoretically. This requirement is satisfied mainly by formal models. Second, similar to the case of views, the behavioral model has to be unambiguous and complete with respect to the statements in the property language. That means that the model should contain sufficient information to answer each query, and not provide ambiguous answers. As a result, models with stochastic outcomes (e.g., some simulation models) cannot be used as behavioral models in my approach: behavioral queries for such models cannot be represented as functions because the returned values may differ for the same inputs. Stronger assumptions on behavioral models lead to higher expressiveness of integration properties and reuse of analysis tools, due to using model-specific property languages. The above assumptions lead to higher expressive power and reuse of reasoning engines.

The scope is broad in terms of individual *analyses*: any analyses that can have their inputs and outputs described in terms of view elements can be integrated using my approach. This restriction requires analyses to have unambiguous underlying models, with well-defined boundaries of what parts of models are read and changed. The contexts of individual analyses can be checked if these contexts can be specified using IPL formulas, the applicability of which is described in the next paragraph. For sets of analyses, the only limitation is that their dependencies should not form dependency cycles. This restriction only excludes analyses from being automatically executed in the same dependency graph, but not from being part of AEP and executed in separate graphs. I discuss potential approaches to the issue of analysis dependency cycles below in Section 10.4.

My approach focuses on *integration properties* that rely on structure *and* behavior of models. In particular, these properties bind structural (rigid) elements and multiple isolated behavioral expressions. In a trivial case, the approach can check properties that are purely structural (i.e., statements over views) or behavioral (i.e., sentences in behavioral property languages). Although this scope of integration properties is broader than most state-of-the-art approaches (see Chapter 9), some properties cannot be directly specified or checked by my approach. In particular, IPL cannot interleave terms from different behavioral languages or directly connect traces from heterogeneous dynamics: such capabilities would likely require additional assumptions on the models, limiting other dimensions of applicability.

My approach does not specifically restrict the *application domains* and *systems*, as long as they belong the field of CPS. Given the evidence in Chapter 8, the approach can be applied to a variety of domains, provided that the assumptions on models, analyses, and integration properties hold. The benefits of the approach are likely to be experienced for systems with multiple dependent models or analyses, and cross-cutting systemic properties like timing, energy, and security.

10.2 Limitations

The limitations describe the potential shortcomings of my approach. Some limitations lead to the aforementioned scope restrictions, while others apply to the models, analyses, or 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[1] of my approach is the *presence* of models, in a syntactically complete and interpretable form. These models need to be provided by engineers as an input to the approach. This limitation exists because the approach does not create new models of the system — only abstractions that serve integration purposes. Therefore, the approach relies on other sources of models, such as manual creation by engineers or automatic generation from other models. Another reason for that limitation is that model-free reasoning about integration properties is not currently supported. For instance, an integration property cannot be currently inferred from some axioms, and requires views and behavioral models to be checked on. As a result of that limitation, the approach does not provide up-front guarantees of analysis applicability to any model. Nevertheless, my approach can be complemented with model-free methods that would reason abstractly about integration properties. For instance, if it can be derived a priori that, given certain assumptions, any model of some system would satisfy an integration property, then only the assumptions of this derivation would need to be checked, instead of the integration property. The advantage of relying on concrete models is that the specific details of a model can be considered, enabling more precise analysis than in model-free approaches.

The approach also assumes that queries of behavioral models can be *trusted*. This assumption is formalized as soundness of queries, defined in Subsection 6.3.3. Soundness of queries requires models to be complete (from the syntactic standpoint) and have sufficient information to process the query, similar to completeness of structural models described above in Section 10.1). Trustworthiness is necessary so that engineers can write integration properties that rely on the semantics of plugged-in behavioral languages. Generally, most implementations of popular behavioral languages and models are well-tested and satisfy this assumption. Note that trustworthiness is only required from behavioral checking, but not necessarily from analyses that are being integration: correctness of their implementations an analysis can be checked with a contract (see the fourth case in Section 4.3).

Views used in the approach are expected to *conform* to their respective models, and be *sound and complete* with respect to the model-view matching predicate. 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. 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). Nevertheless, it is possible to verify the view assumptions using their definitions (see Subsection 6.2.3).

[1]The only tool in this thesis that can be used without instances of models or views is the dependency analysis of AEP. View signatures are sufficient to determine analysis dependencies, see Section 7.2.

Analysis execution is limited to sets of analyses without input/output *dependency cycles*. Although relatively infrequent, these cycles cannot be resolved by the current execution platform, leaving the execution of these analyses to engineers. Dependency cycles may arise from several causes. Some cycles may be caused by the coarse granularity of view types, in terms of which the dependencies are specified. Thus, some cycles may be resolved by modeling analysis inputs and outputs more precisely in views. For instance, assigning hardware elements different types (e.g., sensors and actuators) would remove a dependency between the analyses that change sensors and the analyses that change actuators. Also, future work may develop techniques to resolve these loops, as I discuss below in Subsection 10.4.2. Moreover, some circular analyses may represent competing or incompatible approaches to interpreting models or designing systems, and therefore should not be executed within the same dependency graph.

The validation case studies have shown that the *performance* (in terms of time and memory required for verification) is adequate for 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 are caused by the existing tools of formal verification, which account for almost all verification time, while IPL itself has a remarkably low overhead, as shown in Subsection 8.2.1. The performance of those tools depends on carefully designed abstractions and constraints. In my approach, integration abstractions and properties can be finely tuned to optimize performance. 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.

Expressiveness

My approach has been shown sufficiently *expressive* for the integration scenarios available in the four case studies. However, expressiveness is limited across all three parts of the approach. Low expressiveness threatens construct validity — whether IPL specifications can represent meaningful integration properties. I review the expressiveness concerns in views, behavioral properties, and the IPL syntax.

In some cases, fixed-value element properties in 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 using a name label (e.g., "rate-monotonic scheduling"). Another potential consequence of limited view expressiveness is dependency cycles of analyses (described above). These cycles may appear due to the views not distinguishing the element types; for instance, if both CPUs and memory chips 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 mitigation is to use behavioral properties instead of views to integrate models with complex dynamics.

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, in classic LTL it is not possible to express the following statement: in the next state, a state variable (e.g., battery charge) increases by a constant 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 [17]. 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 limits expressiveness. As noted above, IPL cannot describe properties by mixing terms of behavioral languages. In such cases, the property cannot be split into subformulas that can be evaluated on separate 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 A_{ctrl} (see Subsection 8.4.2) is not expressible in SMT due to the lack of operators for set or array cardinality. However, as new reasoning theories become available (e.g., for lists, arrays, and strings), they can be incorporated into IPL to augment its native syntax, independently of the behavioral models.

Practical Concerns

The *complexity* of the proposed specifications (IPL formulas and analysis contracts) threatens their application in real-world 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 so complex that using them leads to more errors than they discover?

Essential complexity [33] 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 providing reusable solutions and templates for 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 in 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 or an existing domain is changed (i.e., a new domain signature is introduced, see Definition 31 in Section 7.1). Given a domain, a contract depends on the characteristics of 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 precisely consistent to interact correctly, whereas others produce correct interactions in the presence of bounded inconsistencies. Similarly, one engineering project may impose narrower error bounds than others. In my approach, the definition of consistency is determined by engineers depending on their needs and circumstances, as opposed to imposing a fixed a priori notion of consistency (e.g., a graph homomorphism). This flexibility enables necessary tolerance of inconsistency and uncertainty [44]. 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 a comprehensive architectural model that includes every element of the system (also known as the "base architecture" [25]). Prior work use 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, thus allowing consistency to be checked without a "global" perspective that has to be synchronized between all models. As a result, models can be changed independently, without coordinating changes to a complete architecture of the system. Nevertheless, my approach is compatible with prior work on consistency through architectural views, and if a base architecture is provided, structural consistency can be checked using the views created for IPL.

My approach preserves *independence of models* that have heterogeneous semantics. In particular, I do not attempt to unify the semantics of models directly via classic composition methods (i.e., those that create a combined model from several constituents). This decision enables integration of models that do not allow straightforward composition. 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 simultaneously used for integration and refined by their teams.

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 sub-formulas (plugins) of 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 5.3. As a result, the verification algorithm performs part of reasoning at a "shallow" level of views, "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 find unified semantics for them or compose their dynamics, which is outside of the scope of this thesis.
- Models represent a specific part or aspect of a system in detail. 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, which was articulated above. Unlike models, views are designed to represent a homogeneous, simplified, and fixed perspective on the system. Therefore, views are easier to relate and compose.
- 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 represent partially-known behavioral information. 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 *reuse of existing analyses*. Views link CPS models and architectural representations, thus enabling application of architectural analyses in schedulability, synchronization, and design synthesis [81, 116, 230, 266]. 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 designed to add new modeling methods without up-front planning, in contrast with existing top-down approaches that prescribe specific formal notations and analyses.

This work has treated viewpoints (i.e., algorithms to produce views for models, see Definition 15 in Subsection 6.2.2) as *independent entities*. Their independent treatment allows additions or changes to a viewpoint without propagation to other viewpoints. Nonetheless, dependencies

between viewpoints are compatible with the approach, which uses constraints on views for verification, without assuming anything about the process of view creation or updating unconstrained. An example of a viewpoint dependency is when a view is created using the information in another view (along with a model). Combination of viewpoints can enable reuse and reduction of manual effort in creating and updating views.

Multiple possibilities were considered for *specifying input/output dependencies* of analyses. I decided that they could be specified *only* in terms of view types — without references to model elements or behaviors. While in practice analyses change models, tracking changes in terms of model elements would require overcoming the heterogeneity of models directly. Instead, it is convenient to use views, which are already used for integration purposes. The other abstraction (behavioral properties) does not have a suitable granularity for dependency specifications: the interpretation of a behavioral property may change due to almost any model change. Therefore, representing dependencies with behavioral properties would lead to frequent dependency cycles. Specifying dependencies in terms of more granular behavioral elements (e.g., specific states or segments of traces) is feasible, but would require a unified semantics for behavioral models, which, as discussed earlier, is outside of the scope of this thesis.

One might argue that embedding one model (or its parts) into another is sufficient for integration. For example, the MontiCore framework [151] embeds embedding domain-specific languages into each other. I consider embedding to be a syntactic 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 contradictions between the embedded information 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 $\mathcal{M}_{\text{power}}$ are embedded in $\mathcal{M}_{\text{plan}}$ 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 its power model is inaccurate, 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 a specific engineering process, focusing instead on providing *techniques and 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.

Finding Integration Scenarios

In my experience from the case studies, one of the most challenging tasks of model integration is framing an initial integration scenario. Initially, a scenario is described without a detailed

understanding of how the models are related and what errors could occur. Once a scenario is understood better, it clearly delineates a set of models/analyses and an integration property that could be violated due to an interaction between these models/analyses. Below I offer my reflections on finding initial integration scenarios.

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 reduce the manual effort required to use the approach. Some of these improvements focus on *view creation*. Composition and integration of viewpoints [183], as mentioned above, can automate and reduce the effort for view creation, similarly to the previous work in non-CPS settings [67, 231]. For instance, a manually created view listing the actors in a model may be sufficient to automatically construct a timing and an energy view for these actors. If multiple views are created automatically, their creation and update can be seen as a set of meta-analyses and encoded in the execution platform for further automation. 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, one can compose views and behavioral properties, like it was done for $\mathsf{d}\mathcal{L}$ view formulas written over HP views (Section 6.3). Another example is automatically generating behavioral sub-languages (with operators and their semantics) given a view and a viewpoint, similar to language workbenches [72]. Such compositions would allow engineers to interact primarily with integration abstractions, to 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 macro is a syntactic shortcut to a 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 the *performance and scalability* of checking integration properties. The saturation process with SMT done 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 is currently done. Views can also be translated into more efficient SMT specifications. In particular, view elements encoded with uninterpreted constants instead of integer identifiers, which are used in the current implementation of IPL. Furthermore, SMT specifications can be generated selectively for the elements and types used in an integration property, leading to smaller SMT specifications that are checked faster. Querying behavioral properties can be sped up by parallelizing the queries (which is possible because these queries are independent from each other) or using parametric model checking.

A different approach to improving verification performance is to develop *deductive symbolic reasoning* for IPL specifications. A proof calculus for IPL would allow proving formulas based on a set of axioms, without using concrete models. As a result, some integration properties can be proved up-front for a class of models, and others may require checking only some assumptions on specific models. Developing deductive model-free reasoning may require behavioral plugins to include proof rules, to enable equivalent transformations of flexible subformulas. For analysis contracts, deductive reasoning would be able to discharge some of the contracts' assumptions, reducing the verification burden when an analysis is executed. I discuss a more comprehensive proposal for reasoning using analyses in the next section.

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 [197] violate privacy restrictions? Do privacy-enhancing mechanisms increase sensor noise beyond the robustness of attack detection [211]? Another interesting application domain is medical CPS: what are the design-time conditions for device models to be consistent with patient models [41]? Finally, the approach can be extended to behavioral models beyond model checking, in particular simulation, optimization, and game-theoretic models. Simulation could be used to falsify properties in signal temporal logic [175]. Game-theoretic models can be used to check equilibria and constraints on the synthesized solutions. Optimization can be incorporated with subformulas that describe convex problems and delegating them to a separate solver [245].

To reduce chances of unsatisfied assumptions and guarantees during analysis execution, it is possible to consider *all possible orders of the analyses* — instead of just one order, as in the current implementation of AEP. Multiple orders arise from multiple paths in the dependency

graph. The paths can be compared in terms of the assumptions that are likely to be satisfied, based on the guarantees of prior analyses in the path. Path with more assumptions satisfied a priori are more likely to result in successful executions of all the analyses in the path. Also, when one path fails, other paths can be executed, increasing the chance that a successful path is eventually found.

10.4.2 Long-term Research Directions

This thesis proposed a new specification approach, but the problem of *specifying complex multi-model properties* remains open [233]. Fundamentally, expressiveness of the rigid part of IPL can be extended by supporting higher-order logics. These logics would enable comprehensive properties that might currently need to be manually split. The main obstacle to using these logics directly is the absence of decidable procedures to check their satisfiability. This obstacle could be overcome by using model-free symbolic reasoning (discussed above), which could reduce a higher-order formula to multiple first-order formulas. Alternatively, in the case of second-order logics, quantified functions with bounded domains/ranges could be decidable by existing SMT solvers. Such functions can model, for instance, an uncertain allocation from threads to CPUs. Integrating second-order verification into IPL would require substantial changes to the IPL's syntax, semantics, and implementation.

One can extend multi-model integration to check properties over *incomplete models*, enabling integration at an early stage in modeling. One could replace values in views and models with (potentially high-order) constraints, which would be combined with IPL verification. This way, design space exploration would be performed together with verified integration, resulting in correct-by-construction designs. Another approach is interactive integration checking, which 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. Finally, developing the theory behind views and models may enable structural abstractions for open-world models, with multiple views of varying fidelity conforming to the same model. When provided with an integration property, the view mechanism would automatically generate an appropriate view and check it for the characteristics (e.g., soundness) necessary to support the verification of the integration property.

A different approach to address model incompleteness is to perform (part of) *IPL verification at run time*. For instance, instead of assuming that a power model returns accurate energy values for a robot's tasks, the robot could observe and approximate the energy values, continually checking integration properties online. This monitoring would be supported by a middleware that aggregates sensing and perception data from multiple models. While transferring verification to run time may provide weaker up-front guarantees than design-time verification, it may allow checking properties with higher precision and expressiveness, avoiding combinatorial explosion. Furthermore, run-time verification is promising for systems with limited models, such as rare events and systems controlled by humans.

Increasingly, models are produced by *learning from data*, as opposed to using expert insight or deriving them from first principles. For offline learning, a major challenge is finding integration abstractions of such models, since a priori knowledge about them is limited, and may be insufficient to define views or behavioral properties based on patterns in the models. Automatic mining of these patterns could result in specifications for integration of such models. Online learning

presents a threat to the verification guarantees established at design time. To alleviate this problem, one may demonstrate that some changes to a set of models preserve their integration guarantees, as was done in the single-model case [92]. Another technique for online learning is using hierarchical control and safety envelopes, with simple verifiable controllers enforcing safety boundaries on complex learning controllers [84].

A significant extension of the analysis execution platform is an automatic method of *resolving circular dependencies*, which means finding a way to execute circularly dependent analyses without sacrificing the objectives of either analysis. One approach would be to find instances of system models that are invariant to application of all the analyses that form the cycle. With respect to a particular dependency loop, such models are called *fixpoints* of the loop. A preliminary exploration has shown that at least techniques for finding fixpoints are feasible [237]. First, the analyses that are in a loop can be iteratively applied to the same model in hope of converging on a fixpoint. This process might be guided by their assumptions and guarantees. Second, constraint solving in the space of models based on the assumptions and guarantees can be used to produce models that satisfy most or all of the guarantees. Such models are more likely to be fixpoints or converge to fixpoints faster. Third, genetic search over the space of models may produce a fixpoint, if selection is based on assumptions and guarantees of the analyses in the dependency loop. Regardless of the technique, this extension would make circularly dependent analyses converge on an acceptable system design out-of-the-box. The advantage of this convergence is that the analyses would work cooperatively without modifications to their source code.

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. This extension develops a *broader notion of dependency* between analyses than the one considered in this thesis: changes to some models may change the facts known about other models, requiring re-verification. This proposed knowledge management approach would be sensitive to such changes, and may prevent multi-model errors that would otherwise be missed.

An ambitious research direction is *automatic repair* of 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: 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 might 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. The former includes the factors that

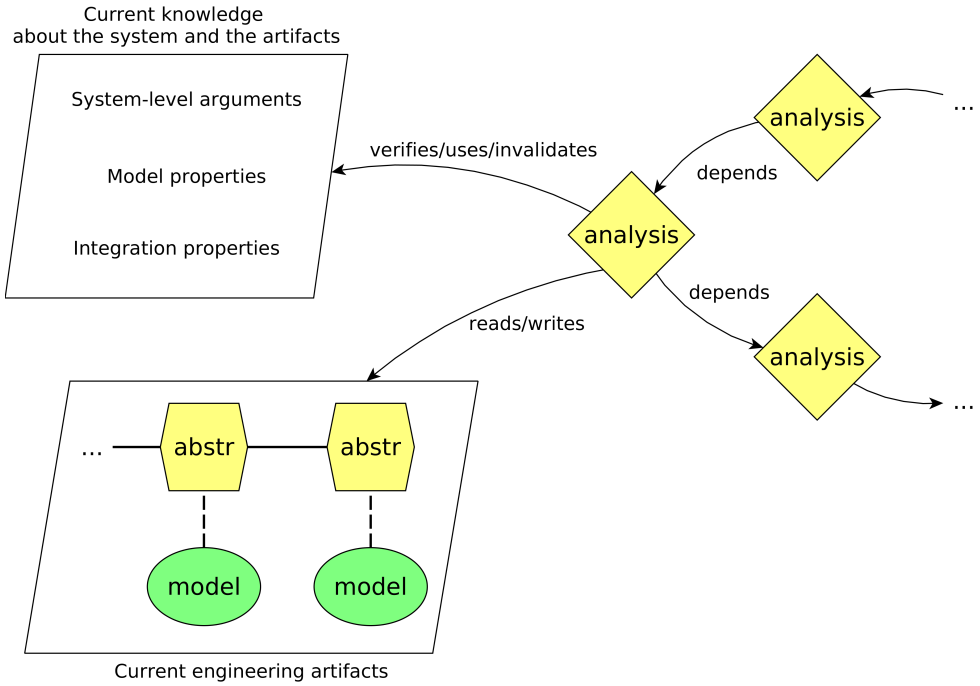


Figure 10.1: A schema of knowledge management for analyses. Arrows indicate operations.

might lead to integration issues: backgrounds of engineers, team communication, dependencies between models, and patterns in models and their languages. Investigating this perspective may clarify the context and non-functional properties that are relevant to model integration [10], as well as team and organization structures that simplify integration. The latter includes user interfaces to support creation of integration abstractions and properties, as well as displaying integration results to engineers with limited proficiency in some of the models. Developing more usable integration tools is likely to lead to improved adoption and outcomes of modeling method integration.

Chapter Summary

Assurance of complex systems remains an open and important topic, especially when multiple perspectives on the system are involved. Specialists with deep technical expertise make precise and well-informed judgments within their domain, but are not necessarily the best decision-makers when evaluating systemic qualities [90]. This thesis provided a modeling toolbox to connect heterogeneous domains of technical expertise. It is important to continue developing techniques for relating and balancing multiple sources of expertise at several levels of engineering: teams, system properties, subsystems, models, and analyses.

Chapter 11

Conclusion

This final chapter summarizes the thesis and its contributions. In this dissertation, I addressed the problem of modeling method integration (MMI), 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 an 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 (query-able 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 5.3), whereas the semantics evaluates each sentence in a way that (Section 5.4).
3. A verification algorithm for IPL statements. The algorithm checks the validity of IPL statements on a set of models and views (Section 5.5).
4. A proof of the soundness of the verification algorithm and its termination conditions (Section 5.6).
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 these contributions to the *practice of engineering cyber-physical systems:*

1. An implementation of the IPL editor and verifier in the Eclipse/OSATE environment [241].
2. A generator of SMT specifications from AADL views (part of ACTIVE) [238].
3. An implementation of the analysis execution platform based on the Eclipse/OSATE environment (part of ACTIVE) [238].
4. A generator of hybrid programs from HP views based on the AcmeStudio environment [239].
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 6.2.5).

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`. 9.2.1

[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. 9.3.3

[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 36th IEEE Conference on Decision and Control (CDC)*, 1997. 2.1, 9.1.2

[8] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. S-TaLiRo: A Tool for Temporal Logic Falsification for Hybrid Systems. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 254–257. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-19835-9. 2.1

[9] 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

[10] 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, 10.4.2

[11] Fredrik Asplund, Matthias Biehl, Jad El-Khoury, and Martin Torngren. 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

[12] 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

[13] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008. ISBN 978-0-262-02649-9. 4, 9.2.2

[14] 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, 9.3.2

[15] Richard Banach, Michael Butler, Shengchao Qin, Nitika Verma, and Huibiao Zhu. Core Hybrid Event-B I: Single Hybrid Event-B machines. *Science of Computer Programming*, 105: 92–123, July 2015. ISSN 0167-6423. doi: 10.1016/j.scico.2015.02.003. URL `http://www.sciencedirect.com/science/article/pii/S0167642315000283`. 1, 9.1.1

[16] 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. 9.3.2

[17] Jeffrey Barnes. *Software Architecture Evolution*. PhD thesis, Carnegie Mellon University, 2013. 10.2

[18] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, 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

[19] 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, 9.2.2

[20] 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. 6.2.4, 9.1.2, 9.3.3

[21] Steffen Becker, Stefan Dziwok, Christopher Gerking, Christian Heinzemann, Wilhelm Schafer, 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. 9.3.3

[22] 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

[23] A. Benveniste, T. Bourke, B. Caillaud, J. Colaco, 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

[24] 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. 9.3.1

[25] 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, 6.2.3, 6.2.5, 1, 7.1, 10.3

[26] 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, 9.2.1, 9.3.1

[27] 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

[28] Dines Bjorner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*. Springer-Verlag, Berlin, Heidelberg, 1978. ISBN 978-3-540-08766-3. 9.3.2

[29] Brandon Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and Andre Platzer. VeriPhy: Verified Controller Executables from Verified Cyber-physical System Models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design*

and Implementation, PLDI 2018, pages 617–630, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5698-5. doi: 10.1145/3192366.3192406. URL http://doi.acm.org/10.1145/3192366.3192406. 1, 9.1.2

[30] Egon Borger, Erich Gradel, and Yuri Gurevich. *The Classical Decision Problem*. Springer, October 2001. ISBN 978-3-540-42324-9. 9.2.2

[31] 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

[32] 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, 6.2.2

[33] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. Addison-Wesley Professional, anniversary edition, August 1995. ISBN 0-201-83595-9. 10.2

[34] 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. 9.3.3

[35] 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. 6.2.4, 9.2.1

[36] 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. 9.2.1

[37] 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. 9.3.3

[38] 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

[39] 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, 8.4.1

[40] 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

[41] 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

[42] Xin Chen, Erika Abraham, 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. 2.1, 9.1.2

[43] 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

[44] 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

[45] 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. 9.2.1

[46] 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. 9.2.2

[47] 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

[48] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, Lecture Notes in Computer Science, pages 52–71. Springer Berlin Heidelberg, 1982. ISBN 978-3-540-39047-3. 1.1

[49] Edmund M. Clarke, William Klieber, Milos Novacek, 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

[50] 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. 5.4.3, 9.2.2

[51] 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, 6.2.2

[52] 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

[53] 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. 9.3.1

[54] 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

[55] 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. 9.3.2

[56] 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

[57] Arnaud Da Costa, Francois 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. 9.2.2

[58] 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

[59] 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. 9.3.1

[60] Leonardo De Moura and Nikolaj Bjorner. 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, 8.2.1, 9.2.2

[61] 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

[62] 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

[63] 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

[64] 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. 9.2.3

[65] 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

[66] 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, 9.3.1

[67] Remco Matthijs Dijkman. *Consistency in multi-viewpoint architectural design*. PhD thesis, Telematica Instituut, Enschede, The Netherlands, 2006. 9.2.1, 9.3, 10.4.1

[68] Vladimir Dimitrieski. Model-Driven Technical Space Integration Based on a Mapping Approach. *University of Novi Sad*, March 2018. URL http://nardus.mpn.gov.rs/handle/123456789/9309. 2.1, 9.3.1

[69] 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

[70] Alexandre Donze. Breach, a Toolbox for Verification and Parameter Synthesis of Hybrid Systems. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 167–170, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 978-3-642-14294-9. doi: 10.1007/978-3-642-14295-6_17. URL http://dx.doi.org/10.1007/978-3-642-14295-6_17. event-place: Edinburgh, UK. 2.1

[71] Alexander Franz Egyed. *Heterogeneous View Integration and its Automation*. PhD thesis, University of Southern California, 2000. 9.3

[72] Sebastian Erdweg, Tijs van der Storm, Markus Volter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriel D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. The State of the Art in Language Workbenches. In *Software Language Engineering*, pages 197–217. Springer, Cham, October 2013. doi: 10.1007/978-3-319-02654-1_11. URL http://link.springer.com/chapter/10.1007/978-3-319-02654-1_11. 10.4.1

[73] Esterel Technologies. SCADE Suite, 2015. URL http://www.esterel-technologies.com/products/scade-suite/. 2.1, 9.1.2

[74] 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. 9.1.1

[75] 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, 8.4.2

[76] Ansgar Fehnker and Franjo Ivancic. Benchmarks for Hybrid Systems Verification. In Rajeev Alur and George J. Pappas, editors, *Hybrid Systems: Computation and Control*, Lecture Notes in Computer Science, pages 326–341. Springer Berlin Heidelberg, January 2004. ISBN 978-3-540-21259-1 978-3-540-24743-2. URL http://link.springer.com/chapter/10.1007/978-3-540-24743-2_22. 1

[77] Peter Feiler and David Gluch. Modeling and Analysis with the AADL: The Basics. November 2012. URL http://www.informit.com/articles/article.aspx?p=1959953. 6.1

[78] Peter Feiler and Aaron Greenhouse. OSATE Plugin Guide. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2006. 5.7, 8.2.1

[79] Peter Feiler, Lutz Wrage, Julien Delange, and Joe Siebel. OSATE2, 2015. URL https://github.com/osate. github.com/osate. 1, 7.4, 8.2.1

[80] 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. 1, 2.1, 4.1, 5.7, 9.2.1

[81] 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

[82] 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

[83] 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. 6.2.2

[84] Jaime F. Fisac, Anayo K. Akametalu, Melanie N. Zeilinger, Shahab Kaynama, Jeremy Gillula, and Claire J. Tomlin. A General Safety Framework for Learning-Based Control in Uncertain Robotic Systems. *arXiv:1705.01292 [cs]*, May 2017. URL `http://arxiv.org/abs/1705.01292`. arXiv: 1705.01292. 10.4.2

[85] 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. 9.3.2

[86] 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

[87] 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

[88] G. Frehse, Zhi Han, and B. Krogh. Assume-guarantee reasoning for hybrid I/O-automata by over-approximation of continuous interaction. In *Proc. of the 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. 9.3.2

[89] Goran Frehse, Colas Le Guernic, Alexandre Donze, 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, 2.1, 9.1.2

[90] S. Frey, A. Rashid, P. Anthonysamy, M. Pinto-Albuquerque, and S. A. Naqvi. The Good, the Bad and the Ugly: A Study of Security Decisions in a Cyber-Physical Systems Game. *IEEE Transactions on Software Engineering*, pages 1–1, 2018. ISSN 0098-5589. doi: 10.1109/TSE.2017.2782813. 10.4.2

[91] 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

[92] Nathan Fulton and Andre Platzer. Safe Reinforcement Learning via Formal Methods: Toward Safe Control Through Proof and Learning. In Sheila McIlraith and Kilian Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, February 2-7, 2018, New Orleans, Louisiana, USA.*, pages 6485–6492. AAAI Press, 2018. 10.4.2

[93] 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. 8.3.2, 8.3.2, 9.1.2

[94] 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. 9.3.2

[95] Paul Gao, Russel Hensley, and Andreas Zielke. A road map to the future for the auto industry. *McKinsey Quarterly*, October 2014. 1

[96] 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

[97] 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

[98] 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. 1, 4.1, 9.2.1

[99] 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. 9.2.2

[100] 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. 9.2.2

[101] Antoine Girard and George J. Pappas. Approximate Bisimulation: A Bridge Between

Computer Science and Control Theory. *European Journal of Control*, 17(5âĂŞ6): 568–578, 2011. ISSN 0947-3580. doi: 10.3166/ejc.17.568-578. URL http://www.sciencedirect.com/science/article/pii/S0947358011709773. 9.3.2

[102] 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. 9.2.2

[103] Claudio 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. 9.3.2

[104] Susanne Graf, Sophie Quinton, Alain Girault, and Gregor Gossler. 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. 9.2.1, 9.3

[105] 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

[106] Andy Greenberg. Hackers Remotely Kill a Jeep on the HighwayâĂŤWith Me in It. *WIRED*, July 2015. URL http://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/. 8.4.2

[107] 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

[108] Object Management Group. Object Constraint Language, Version 2.4. Technical Report formal/2014-02-03, 2014. URL http://www.omg.org/spec/OCL/2.4. 6.3.2

[109] Esther Guerra, Juan de Lara, Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schonbock, 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. 9.2.2

[110] 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. 4

[111] Gregoire 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

[112] 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, 8.4.2, 9.2.2

[113] David Harel. Statecharts: A Visual Formalism For Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987. 2.1, 9.1.1

[114] 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

[115] T.A. Henzinger. The theory of hybrid automata. In *Proc. of Eleventh Annual IEEE Symposium on Logic in Computer Science, 1996 (LICS '96)*, pages 278–292, July 1996. doi: 10.1109/LICS.1996.561342. 2.1

[116] 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

[117] 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. 6.2.2, 9.2.1

[118] 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

[119] Sonke 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. 9.3.3

[120] 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, 8.4.1, 9.1.1

[121] Ronald A. Howard. *Dynamic Programming and Markov Processes*. Technology Press of the Massachusetts Institute of Technology, 1960. 3.1

[122] Fei Hu, Yu Lu, Athanasios V. Vasilakos, Qi Hao, Rui Ma, Yogendra Patil, Ting Zhang, Jiang Lu, Xin Li, and Neal N. Xiong. Robust Cyber-Physical 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

[123] Pengcheng Huang, P. Kumar, G. Giannopoulou, and L. Thiele. Energy efficient DVFS scheduling for mixed-criticality systems. In *2014 International Conference on Embedded Software (EMSOFT)*, pages 1–10, October 2014. doi: 10.1145/2656045.2656057. 3.3

[124] 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

[125] 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

[126] 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. 9.3.2

[127] 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`. 9.3.3

[128] 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`. 6.2.2, 9.2.1

[129] 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, 9.2.3

[130] Ethan K. Jackson, Tihamer 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`. 9.3.2

[131] 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

[132] 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. 9.3.1

[133] Jean-Baptiste Jeannin, Khalil Ghorbal, Yanni Kouskoulas, Ryan Gardner, Aurora Schmidt, Erik Zawadzki, and Andre Platzer. A Formally Verified Hybrid System for the Next-Generation Airborne Collision Avoidance System. In Christel Baier and Cesare Tinelli,

editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 21–36. Springer Berlin Heidelberg, 2015. ISBN 978-3-662-46681-0. 2.1

[134] 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

[135] 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

[136] 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

[137] 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

[138] 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. 8.4.2

[139] Oliver Kautz, Alexander Roth, and Bernhard Rumpe. Achievements, Failures, and the Future of Model-Based Software Engineering. In Volker Gruhn and Rudiger 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

[140] 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. 9.3.2

[141] 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, 8.4.1, 8.4.1

[142] 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, 8.4.1

[143] 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, 8.4.1

[144] 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

[145] Joachim Klein and Christel Baier. On-the-Fly Stuttering in the Construction of Deterministic w-Automata. In Jan Holub and Jan Zdarek, editors, *Implementation and Application of Automata*, Lecture Notes in Computer Science, pages 51–61. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-76336-9. 8.3.1

[146] 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`. 8.2.1, 8.3.1

[147] Soonho Kong, Sicun Gao, Wei Chen, and Edmund Clarke. dReach: d-Reachability Analysis for Hybrid Systems. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 200–205. Springer Berlin Heidelberg, 2015. ISBN 978-3-662-46681-0. 2.1

[148] Alexander Konigs. *Model Integration and Transformation - A Triple Graph Grammar-based QVT Implementation*. PhD thesis, TU Darmstadt, November 2008. URL `http://tuprints.ulb.tu-darmstadt.de/1194/`. 9.2.2

[149] 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`. 9.3.2

[150] 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

[151] Holger Krahn, Bernhard Rumpe, and Steven Volkel. MontiCore: a framework for compositional development of domain specific languages. *International Journal on Software Tools for Technology Transfer*, 12(5):353–372, March 2010. ISSN 1433-2779, 1433-2787. doi: 10.1007/s10009-010-0142-1. URL `http://link.springer.com/article/10.1007/s10009-010-0142-1`. 10.3

[152] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View*. Springer, 2008. 5.2.2

[153] 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

[154] 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, 5.2.2, 5.3.1, 5.3.3, 5.4.4, 6.3, 8.2.1, 8.4.2, 9.2.2

[155] Kai Lampka, Simon Perathoner, and Lothar Thiele. Component-based system design: 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

[156] 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

[157] 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

[158] Juan de Lara and Hans Vangheluwe. AToM3: A Tool for Multi-formalism and Metamodelling. 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. 9.2.2

[159] 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

[160] Peter Gorm Larsen, Kenneth Lausdahl, Nick Battle, John Fitzgerald, Sune Wolff, Shin Sahara, Marcel Verhoef, Peter W. V. Tran-Jorgensen, and Tomohiro Oda. VDM-10 Language Manual. Technical Report TR-001, The Overture Initiative, www.overturetool.org, April 2013. 9.1.1

[161] 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

[162] 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

[163] 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, 1, 9.1.1

[164] 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

[165] 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, 6.2.4, 9.3.2

[166] 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. 9.3.1

[167] 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

[168] 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

[169] Sarah M. Loos and Andre Platzer. Differential Refinement Logic. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 505–514, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4391-6. doi: 10.1145/2933575.2934555. URL `http://doi.acm.org/10.1145/2933575.2934555`. 1, 9.3.2

[170] 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

[171] 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

[172] Sergey E. Lyshevski. *Engineering and Scientific Computations Using MATLAB*. Wiley-Interscience, Hoboken, 1st edition, June 2003. ISBN 978-0-471-46200-2. 2.1

[173] 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`. 3.2

[174] Jeff Magee and Jeff Kramer. *Concurrency: State Models & Java Programs*. Wiley, April 1999. ISBN 0-471-98710-7. 2.1, 9.1.1

[175] 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

[176] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992. 9.2.2

[177] 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

[178] Raluca Marinescu. *Model-driven Analysis and Verification of Automotive Embedded Systems*. PhD thesis, Maladaren University, 2016. B., 9.1.2, 9.3.1

[179] 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

[180] 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

[181] Ethan McGee, Mike Kabbani, and Nicholas Guzzardo. Collision Detection AADL, 2013. URL https://github.com/mikekab/collision_detection_aadl. 8.3.4

[182] 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. 6.2.6

[183] 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

[184] 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. 9.3.1

[185] 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. 9.2.3

[186] 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. 9.3.1

[187] 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. 9.3.1

[188] 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

[189] 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

[190] 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

[191] 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

[192] Stefan Mitsch and Andre Platzer. ModelPlex: Verified Runtime Validation of Verified Cyber-Physical System Models. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, Lecture Notes in Computer Science, pages 199–214. Springer International Publishing, September 2014. ISBN 978-3-319-11163-6 978-3-319-11164-3. doi: 10.1007/978-3-319-11164-3_17. URL http://link.springer.com/chapter/10.1007/978-3-319-11164-3_17. 1

[193] Stefan Mitsch, Khalil Ghorbal, and Andre Platzer. On Provably Safe Obstacle Avoidance for Autonomous Robotic Ground Vehicles. In *Proc. of Robotics: Science and Systems*, 2013. 3.2, 6.2.4, 8.3.2

[194] 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, 8.3.2, 8.3.2, 9.3.1

[195] 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,

8.3.2

[196] Stefan Mitsch, Khalil Ghorbal, David Vogelbacher, and Andre Platzer. Formal verification of obstacle avoidance and navigation of ground robots. *The International Journal of Robotics Research*, 36(12):1312–1340, October 2017. ISSN 0278-3649. doi: 10.1177/0278364917733549. URL https://doi.org/10.1177/0278364917733549. 2.1, 3.2

[197] 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

[198] Andreas Muller, Stefan Mitsch, Werner Retschitzegger, Wieland Schwinger, and Andre Platzer. Tactical contract composition for hybrid system component verification. *International Journal on Software Tools for Technology Transfer*, 20(6):615–643, November 2018. ISSN 1433-2787. doi: 10.1007/s10009-018-0502-9. URL https://doi.org/10.1007/s10009-018-0502-9. 1, 8.3.2, 8.3.2

[199] 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. 9.2.1

[200] 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

[201] 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, 9.2.2, 9.2.3

[202] 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

[203] 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. 5.2.2, 9.2.2

[204] 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. 4.2, 9.2.2

[205] 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. 9.3.1

[206] 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. 9.3.1

[207] 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

[208] Pierluigi Nuzzo, Richard Murray, and Alberto Sangiovanni-Vincentelli. Methodology and Tools for Next Generation Cyber-Physical Systems: The iCyPhy Approach. *INCOSE International Symposium*, 25(1):235–249, 2015. ISSN 2334-5837. doi: 10.1002/j.2334-5837.2015.00060.x. 1

[209] 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. 8.3.4

[210] Richard F. Paige, Phillip J. Brooke, and Jonathan S. Ostroff. Metamodel-based model conformance and multi-view consistency checking. *ACM TOSEM*, 16, 2007. 9.2.1

[211] 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

[212] 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. 9.3.1

[213] A. Platzer. The Complete Proof Theory of Hybrid Systems. In *2012 27th Annual IEEE Symposium on Logic in Computer Science*, pages 541–550, June 2012. doi: 10.1109/LICS.2012.64. 2.1, 6.3.3

[214] 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

[215] Andre Platzer. *Logical foundations of cyber-physical systems*. Springer Berlin Heidelberg, New York, NY, 2018. ISBN 978-3-319-63587-3. 2.1, 9.1.2

[216] Andre Platzer and Edmund M. Clarke. Computing Differential Invariants of Hybrid Systems as Fixedpoints. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 176–189. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-70545-1. 2.1, 9.1.2

[217] Andre Platzer and Yong Kiam Tan. Differential Equation Axiomatization: The Impressive Power of Differential Ghosts. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 819–828, New York, NY, USA, 2018.

ACM. ISBN 978-1-4503-5583-4. doi: 10.1145/3209108.3209147. URL http://doi.acm.org/10.1145/3209108.3209147. 9.1.2

[218] 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, 5.3.1, 5.3.2

[219] Joseph Porter, Gabor Karsai, Peter Volgyesi, Harmon Nine, Peter Humke, Graham Hemingway, Ryan Thibodeaux, and Janos 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. 9.3.2

[220] Ahsan Qamar. *Model and Dependency Management in Mechatronic Design*. PhD thesis, KTH Sweden, Stockholm, Sweden, 2013. 9.2.3

[221] 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

[222] Radhakisan Baheti and Helen Gill. Cyber-Physical Systems. Technical report, National Science Foundation, 2011. 2.1

[223] 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. 9.2.1, 9.2.3

[224] Akshay Rajhans. *Multi-Model Heterogeneous Verification of Cyber-Physical Systems*. PhD thesis, Carnegie Mellon University, 2013. 9.3.2

[225] 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., 9.3.2

[226] 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

[227] 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, 9.2.1

[228] Raj Rajkumar, Insup Lee, Lui Sha, and John 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

[229] Andreas Rentschler. *Model Transformation Languages with Modular Information Hiding*. PhD thesis, Karlsruhe Institute of Technology, Karlsruhe, Germany, 2015. 9.2.2

[230] 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

[231] 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

[232] 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

[233] 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

[234] 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

[235] 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. 5.2.2, 7.1

[236] 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

[237] 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. 10.4.2

[238] 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

[239] Ivan Ruchkin, Bradley Schmerl, and David Garlan. Supplementary Materials for a Case Study of Architectural Abstractions for Hybrid Programs, 2018. 8.3.2, 4

[240] Ivan Ruchkin, Joshua Sunshine, Grant Iraci, Bradley Schmerl, and David Garlan. IPL: An Integration Property Language for Multi-model Cyber-physical Systems. In Klaus Havelund, Jan Peleska, Bill Roscoe, and Erik de Vink, editors, *Formal Methods*, Lecture Notes in Computer Science, pages 165–184. Springer International Publishing, 2018. ISBN 978-3-319-95582-7. 5.2.2

[241] 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, 2, 4, 8.2.1, 1

[242] 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. 9.3.1

[243] Bradley Schmerl and David Garlan. AcmeStudio: Supporting Style-Centered Architecture Development. In *Proc. of ICSE*, 2004. ISBN 0-7695-2163-0. 1, 8.3.2

[244] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, April 1996. ISBN 0-13-182957-2. 6.2.1, 9.2.1

[245] 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

[246] Gabor Simko, David Lindecker, Tihamer Levendovszky, Sandeep Neema, and Janos Sztipanovits. Specification of Cyber-Physical Components with Formal Semantics âĂŞ 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., 9.3.2

[247] 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

[248] SPARC. Robotics 2020 Multi-Annual Roadmap For Robotics in Europe, 2015. 1

[249] 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

[250] 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

[251] 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, 9.3.2

[252] J. Sztipanovits, T. Bapty, X. Koutsoukos, Z. Lattmann, S. Neema, and E. Jackson. Model and Tool Integration Platforms for CyberâĂŞPhysical System Design. *Proceedings of the IEEE*, 106(9):1501–1526, September 2018. ISSN 0018-9219. doi: 10.1109/JPROC.2018.2838530. 2.1, 9.3.2

[253] Janos Sztipanovits. Cyber Physical Systems âĂŤ 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

[254] 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. 9.2.1

[255] 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. 9.3.1

[256] 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

[257] Anton Valukas. Report to Board of Directors of General Motors Company Regarding Ignition Switch Recalls. Technical report, Jenner & Block, 2014. 1, 2.2

[258] 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. 9.3.1

[259] Daniel Varro, Gabor Bergmann, Abel Hegedus, Akos Horvath, Istvan Rath, and Zoltan 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. 9.2.2

[260] 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

[261] 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. 9.3.2

[262] Gerhard Wanner, Ernst Hairer, and Syvert Norsett. *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

[263] 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

[264] 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

[265] 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, 9.3.3

[266] 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

[267] Robert K. Yin. *Case Study Research: Design and Methods*. Sage Publications, Inc, 4th edition, October 2008. ISBN 1-4129-6099-1. 8.2.1

[268] Sergio Yovine. Kronos: A Verification Tool for Real-Time Systems. (Kronos User's Manual Release 2.2). *International Journal on Software Tools for Technology Transfer*, 1, January 2001. doi: 10.1007/s100090050009. 9.1.2

[269] 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. 9.2.3