IPL: An Integration Property Language for Multi-Model Cyber-Physical Systems

Ivan Ruchkin, Joshua Sunshine, Grant Iraci,
Bradley Schmerl, and David Garlan

Institute for Software Research, Carnegie Mellon University
Pittsburgh, PA, USA

Abstract. Design and verification of modern systems requires diverse models, which often come from a variety of disciplines, and it is challenging to manage their heterogeneity – especially in the case of cyber-physical systems. To check consistency between models, recent approaches map these models to flexible static abstractions, such as architectural views. This model integration approach, however, comes at a cost of reduced expressiveness because complex behaviors of the models are abstracted away. As a result, it may be impossible to automatically verify important behavioral properties across multiple models, leaving systems vulnerable to subtle bugs. This paper introduces the *Integration Property Language (IPL)* that improves integration expressiveness using modular verification of properties that depend on detailed behavioral semantics while retaining the ability for static system-wide reasoning. We prove that the verification algorithm is sound and analyze its termination conditions. Furthermore, we perform a case study on a mobile robot to demonstrate IPL is practically useful and evaluate its performance.

1 Introduction

Today, complex software systems are often built by multidisciplinary teams using diverse engineering methods [1,2]. This diversity is particularly apparent in *cyber-physical systems* (CPS) where software control interacts with the physical world. For instance, a mobile robot needs to brake in time to avoid collisions, compute an efficient long-term plan, and use a power model of its hardware to ensure it has sufficient energy to complete its missions. To satisfy each of these requirements, engineers may use heterogeneous models that vary in formalisms, concepts, and levels of abstraction. Even though these models are separate, interdependencies naturally occur because they represent the same system.

Mismatches between such implicitly dependent models may lead to faults and system failures. For example, the 2014 GM ignition switch recall was caused by an unanticipated interaction between electrical and mechanical aspects of the ignition switch [3]. This interaction led to the switch accidentally turning off mid-drive and disabling the car's software along with airbags, power steering, and power brakes. This mismatch between the electrical, mechanical, and software designs caused dozens of deaths and large financial losses.

To prevent such issues, inconsistencies or contradictions need to be detected by *integrating* the heterogeneous models. This can be done by checking properties that involve multiple models and formalisms, which we term *integration*

properties. Model integration is difficult [4] and checking integration properties is often done informally through inspection, and is limited in rigor and outcomes. One way to improve this would be to map diverse semantics and property checks into a single unifying model. Unfortunately, it is hard (and sometimes impossible) to do so, as in the case of unifying stateful and stateless models [5].

A common way to integrate heterogeneous models is to create and relate simplified abstractions. One such abstraction is *architectural views* — behaviorless component models annotated with types and properties [6,7,8]. Since views are easier to reason about than heterogeneous models, structural consistency checks can be formalized and automated [9]. However, model integration through views sacrifices behavioral expressiveness of integration properties, meaning that sophisticated interactions become uncheckable.

We perceive a *foundational gap* between the limited expressiveness of integration properties and the need to discover complex inconsistencies of several models. State-of-the-art integration approaches are limited in what is exposed from models. Exposing too little leads to insufficiently expressive analysis. Exposing too much leads to limited flexibility and extensibility of integration methods.

To help bridge this gap, this paper introduces the *Integration Property Language (IPL)* – a formal specification and verification method for integration properties based on architectural views. IPL's goal is to systematically express and automatically check properties that combine system behaviors and static abstractions, enabling end-to-end verification arguments over multiple models.

The main design principle behind IPL is to combine first-order logical reasoning across many views with "deep dives" into behavioral structures of individual models as necessary. IPL syntax interleaves first-order quantification over *rigid* constructs (defined by views) and temporal modalities that bind the behavior of *flexible* terms (changing according to models). Built upon existing satisfiability solvers and model checkers, IPL uses a sound reasoning algorithm to modularize the problem into subproblems that respective tools interpret and solve.

This paper makes three contributions: (1) a *formalized modular syntax and semantics of IPL*, instantiated for two modal logics; (2) an *algorithm to verify validity of IPL statements*, with a soundness proof and termination conditions; and (3) a *modeling case study of a mobile robot*, with several integration properties to evaluate practical applicability and performance of the IPL prototype.

The paper is organized as follows. Sec. 2 introduces an illustrating scenario of integration. Sec. 3 describes related work. Sec. 4 gives an overview of the IPL design, while Sec. 5 provides the details of the IPL syntax, semantics, and the verification algorithm. Sec. 6 provides a case study and a theoretical analysis of the algorithm. We conclude the paper with limitations and future work.

2 Motivating Integration Case

Consider an autonomous mobile robot, such as TurtleBot (`http://turtlebot.com`), that navigates to a goal location through a physical environment using its map. The environment contains charging stations for the robot to replenish its

battery. The robot has an adaptive software layer that monitors and adjusts the execution to minimize mission time and power consumption.

In the design of this system (more detail in Sec. 6.1), we have two models: a power prediction model and a planning model. The power prediction model M_{po} is a parameterized set of linear equations that estimates the energy required for motion tasks, such as driving straight or turning in place. The model is a statistical generalization of the data collected from the robot's executions. Given a description of a motion task, the model produces an estimate of required energy.

The planning model M_{pl} finds a path to a goal by representing the robot's non-deterministic movements on a map, along with their time and power effects, in a Markov Decision Process (MDP) [10]. The model's state includes the robot's location and battery charge. Whenever (re)planning is required, the PRISM probabilistic model checker [11] resolves non-determinism with optimal choices, which are fed to the robot's motion control. Although inspired by M_{po}, M_{pl} is not identical to it because of various modeling choices and compromises, for example it does not explicitly model turns.

These two models interact during execution: M_{po} acts as a safeguard against the plan of M_{pl} diverging from reality and leading to mission failure. M_{pl} only needs to be triggered when the robot is going to miss a deadline or run out of power. Otherwise, the robot avoids running the planner to conserve power[1]. If M_{pl} has overly conservative energy estimates compared to M_{po}, 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.

Integrating these two models means ensuring that their estimates of required energy do not diverge. One threat to integration is the difference in modeling of turns: M_{pl} models turns implicitly, combining them with forward motions into single actions to reduce the state space and planning time. In M_{po} however, turns are explicit tasks, separate from forward motion. This potential inconsistency can be checked with the following integration property: "*the difference in energy estimates between the two models should not be greater than a predefined constant* $\overline{err_cons}$". The purpose of this property is to enable end-to-end safety arguments (e.g., not running out of power or arriving before a deadline). Instead of (inaccurately) assuming equivalence of M_{po} and M_{pl}, this property would provide a rigorous estimate of $\overline{err_cons}$,[2] which can be used to assert that the battery cannot run out because its charge is always greater than $\overline{err_cons}$.

It is far from straightforward to verify this property. First, the abstractions are different: M_{pl} describes states and transitions (with turns embedded in them), whereas M_{po} describes a stateless relation. Second, there is no single means to express such integration properties formally: PCTL (Probabilistic Computation Tree Logic [11]) is a property language for M_{pl}, but M_{po} does not come with a reasoning engine. Finally, even if these obstacles are overcome, the models are often developed by different teams, so they need to stay separate and co-evolve.

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

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

The integration property can be checked in several ways. A direct approach is to develop a "supermodel" containing M_{pl} and M_{po} as sub-models. A supermodel would query M_{po} from each state of M_{pl}. Although accurately detecting violations, this method is not tractable for realistic models of hundreds of thousands of states. Furthermore, the property would be hardcoded in the supermodel implementation, which would need to be developed anew for other properties.

Another approach relies on abstraction of models through *architectural views*. The views are hierarchical arrangements of discrete static instances (architectural elements) with assigned types and properties (defined in Sec. 5.1). Typically, when views are used to integrate multiple models [12], the verification is confined to the views to take advantage of their relatively simple semantics (without temporal behaviors or dynamic computation). One could encode all possible M_{pl} behaviors (i.e., trajectories of locations, turns, and energies) in views, also encoding them as atomic motion tasks of M_{po}. This approach, again, leads to either intractability or approximation (e.g., only recording the number of turns in each path), which in turn would not have soundness guarantees.

In this paper we pursue the integration approach that combines specifications over behaviors and views as necessary. For now, we provide an informal version of the integration property, which will be formalized in the end of Sec. 5.2.

Property 1 (Consistency of M_{po} and M_{pl})**.** *For any three sequential M_{po} tasks ⟨go straight, rotate, go straight⟩ that do not self-intersect and have sufficient energy, any execution in M_{pl} that goes through that sequence in the same order, if initialized appropriately, is a power-successful mission (modulo $\overline{err_cons}$).*

It is challenging to systematically express and verify such properties while holding the models modular and tractable. Notice how missions in M_{po} need to correspond to missions in M_{pl}; e.g., the initial charge of M_{pl} needs to be within $\overline{err_cons}$ of the expected mission energy in M_{po}. Specifications like Prop. 1 are enabled by our solution design and the language syntax (Secs. 4 and 5).

3 Related Work

Model Integration Model-based engineering relies on a variety of formalisms, including synchronous, timed, and hybrid models [5].When models are similar, it is easier to find unifying abstractions, like in the case of consistency checking for software models [13,14,6] or model refinement [15,16,17]. We, however, target a broader scope of cyber-physical models that were not intended for integration, leading to more challenging problems [18,4].

Integration approaches for CPS models can be seen along a spectrum from structural (operating on model syntax) to semantic (operating on behavior) ones [19]. One structural approach is to use *architectural views* — abstract component models [20,7]. Views have been extended with physical descriptions for consistency checking via graph mappings [12] and arithmetic constraints [21]. Other recent structural approaches include model transformations [22], ontologies [23], and metamodels [24]. Model transformations are typically forced to either map models to the same semantics or abandon one or more in favor of

new meanings. This paper extends the view-based structural approach to write formalized statements that affect many semantic universes.

On the semantic end, one approach is to relate model behaviors directly [25]. Although theoretically elegant, this approach suffers from limited automation and creating inter-model dependencies. Other semantic approaches relate model behaviors through proxy structures. Well-known examples include the Ptolemy II environment [26] and the GEMOC studio [27]. In contrast to these works on heterogeneous simulation, we focus on logical verification of multiple models. Another example is the OpenMETA toolchain for domain-specific language integration [28]. The toolchain contains automated support for verifying individual CPS models (e.g., bond graphs) based on their logically-defined interfaces. OpenMETA's integration language (CyPhyML), however, commits to continuous-trajectory semantics [29], whereas IPL allows arbitrary plug-in behaviors. Our work builds on a prototype of a FOL/LTL contract formalism [30], which we extend by providing a full-fledged language (as opposed to a stitching of two statements) with a sound verification algorithm and a plugin system.

Logics, Satisfiability, Model Checking This paper is related to quantified Computation Tree Logic (QCTL) [31] and well-researched combinations of first-order logic (FOL) [32] and linear temporal logic (LTL) [33], going back to the seminal work of Manna and Pnueli [34] on first-order LTL, which has been instantiated in many contexts [35,36]. Typically, such work focuses classical properties of logics and algorithms, such as decidability and complexity.We, instead, focus on expressiveness and modularity — practical concerns for CPS. For example, IPL differs from the trace language for object models [36] in that we do not create a full quantification structure in each temporal state. In contrast, IPL is modular with existing models and delegates behavioral reasoning to them.

An ambitious approach is to directly combine arbitrary logics, at the cost of high complexity and limited automation (as in fibred semantics [37]). Even when modular [38], combining logics merges their model structures, which may lead to tractability challenges in practice. We opt to keep models completely separate, thus reducing complexity and overhead.

Our algorithm relies on Satisfiability Modulo Theories (SMT) [39] and model checking [40,11]. To guarantee termination, we limit ourselves to decidable combinations of background theories (like uninterpreted functions and linear real arithmetic) that admit the Nelson-Oppen combination procedure [41]. In practice, modern SMT solvers (e.g., z3 [42]) heuristically solve instances of undecidable theories. In model checking we use the usual conversion of a modal property to an automaton (Buchi, Rabin, ...) and its composition with models [43,11].

4 Integration Property Language: Design

The Integration Property Language (IPL) is intended for model integration, which informally means that models do not contradict each other. We envision the following workflow. An engineer creates or obtains system models for integration. Some of these models will be interfaced through a behavioral property

language. The other models will be accessed through static abstractions (views), created by the engineer. Then the engineer writes and checks an integration property over views and behavioral properties using IPL. If the verification fails, the engineer inspects and corrects the models and/or the property. Whenever the models change, their respective views are updated, and properties are reverified.

A primary goal of the IPL design is *applicability* to real-world model integrations. Therefore, our design focuses on these 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 support diverse CPS models, IPL should neither be tied to a particular property language or form of model behavior (discrete, continuous, or probabilistic), require the reengineering of constituent models. Thus, IPL should enable straightforward incorporation of new models and property languages.

3. Tractability. To enable automation in practice, verification of IPL specifications must be sound and implementable with practical scalability.

To support these principles, we make the following four design decisions.

A. 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. In this paper, we target two modal logics common in model-based engineering: LTL and PCTL.

B. Separation of structure and behavior. IPL explicitly treats the static (rigid) and dynamic (flexible) elements of models separately. We accomplish this using *views* (defined in Sec. 5.1) that serve as projections of static aspects of behavioral models. This separation enables tractability because static aspects can be reasoned about without the temporal/modal dimension. We support expressiveness by allowing combinations of rigid and flexible elements to appear in the syntax.

C. Multi-step verification procedure. We combine reasoning over static aspects in first-order logic with "deep dives" into behavioral models to retrieve only the necessary values. We preserve tractability by using tools only within individual well-defined semantics, without direct dependencies between models.

D. Plugin architecture for behavioral models. To create a general framework for integration, we specify several *plugin points* — APIs that each behavioral model has to satisfy. While the model itself can stay unchanged, IPL requires a plugin to use their formalism for verification. This way, IPL does not make extra assumptions on models beyond the plugin points, hence enhancing modularity.

To support expression and verification of Prop. 1, we use PCTL with M_{pl} to reason about behaviors and a view V_{po} for reasoning about the static/stateless elements of M_{po}. V_{po} serves as a *task library*, containing all atomic tasks (going straight and rotating in the motivating example) in each location/direction in the given map. Each task is annotated with its properties, such as start/end locations, distance, required time, and required energy. Each task in V_{po} is encoded as a component and contains several properties. Thus, this view allows natural composition of missions as constrained sequences of components.

5 Integration Property Language: Details

This section describes IPL by defining its syntax and formalizing its semantics. After, we provide an algorithm to check whether an IPL formula is valid.

5.1 Concepts and Preliminaries

The concept of an architectural view originates in the field of software architecture [44]. Recently, views have been adapted to represent non-software elements such as sensors and transducers in CPS [9]. We use views to extract information for IPL to analyze without needing to process all the details of models.

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

IPL uses views for modeling static, behavior-free projections of models. For example, M_{po} uses a map of locations for its tasks, and it can be exposed in a view (V_{map}) as a set of interconnected components ($\overline{Locs}$). Each component is a location, and connectors indicate direct reachability between them. We use views as an abstraction because of their composability, typing, and extensible hierarchical structure. No dynamic information (e.g., the current battery charge) is put in views so that behavioral semantics are confined to models.

Definition 2 (Formal View)**.** *A* (formal) view V *is a pair of a view signature (Σ_{V}) and its semantic interpretation (I^{V}). The signature contains a set of architectural elements ($\mathbb{E}$), their types, properties, sorts/constants, and functions/predicates. The semantic interpretation gives static meaning to the elements in the view signature, independent of state or time.*

We use formal views to define the syntax and semantics of IPL. We establish an isomorphic relationship between the two definitions by converting architectural models to SMT programs, as in prior work [30]. Both definitions of views are used throughout the paper: Def. 1 — for applied modeling (e.g., representing views in the case study), and Def. 2 — for theory behind IPL and verification.

Definition 3 (Model)**.** *A* (behavioral) model M *is a triple of an (interface) signature, an interpretation (I_q^{M}), and a structure on which it is interpreted. The signature defines symbols of state variables, modal functions/predicates, and a list of name-type pairs for initialization parameters. The parametric structure determines the model's set of behavior traces ($\mathsf{M}.trcs$) [43,45].*

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

For the rest of this section we consider a fixed set of behavioral models $\mathbb{M}$, with some of them abstracted by a fixed set of views ($\mathbb{V}$). Each view can be seen as some (implicit) function of a model. We consider two specific model property languages: LTL and PCTL, although in principle we are not limited to them.

Shared by models and views, background interpretation I^B evaluates common sorts, constants (e.g., boolean $\top$ and $\bot$), functions (e.g., addition), and predicates (e.g., equality) from background theories (e.g., the theory of equality or linear real arithmetic). Formally, we only allow theories that are decidable [46] and form decidable combinations [41], but in practice it is acceptable to use undecidable combinations for which available heuristics resolve relevant statements.

Formulas will be described over a context of views and models. Syntactically, IPL formulas are written over a *signature* (Σ) that contains symbols from $\mathbb{V}$ and $\mathbb{M}$. Semantically, a formula's context is determined by a *structure* (Γ) that contains interpretations I_q^{M}, I^{V}, and I^B along with their domains.

Finally, we make additional assumptions: (i) views are pre-computed and stay up-to-date with models; (ii) views can be translated into finite SMT programs; (iii) once initialized, any model can check/query any statement in its property language; and (iv) models and views share the background interpretation.

5.2 Syntax

To support modularity, we keep track of syntactic terms that cannot be interpreted by either views or models. So we introduce the rigid/flexible separation: *flexible* terms (denoted with underlines, like $\underline{loc}$) are interpreted by I_q^{M}, and *rigid* terms (denoted with overlines, like $\overline{Tasks}$) are interpreted by I^{V}. Terms of I^B are used by both models and views (no special notation; e.g., $<$).

To embed model property languages into IPL, the syntax allows model-specific formulas to be defined as "plugins" in the grammar. That is, various property languages are usable inside IPL formulas. Thus, the syntax is split into the *native* (related to views) and *plugin* (related to property languages) parts.

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. An IPL interpreter should evaluate the native parts when it prepares a model property to verify. Consider Prop. 1 in Sec. 2 where a model evaluating $P_{max=?}$ requires interpreting native IPL term $t_2.start$.

We organize the native/plugin syntax as presented in Fig. 1. We define each syntax element (box) on top of symbols in Σ and quantified variables ($\mathbb{V}$). We build two types of subformulas: rigid atomic formulas (RATOM) from rigid terms (RTERM), and flexible atomic formulas (MATOM). Our strategy is to keep flexible and rigid syntax separate until they merge in FORMULA. In this way, we preserve modularity: compound formulas can be deconstructed into simpler ones that are evaluated by either models or views.

A *rigid term* RTERM is either a variable VAR, a constant CONST, an architectural element type ELEM, a property of a rigid term RTERM.PROP[3], a background function BFUNC, or a view function VFUNC. A *rigid atom* RATOM is a logical expression over rigid terms. See the full syntax rules in the online appendix [47].

[3] Properties are only applicable to architectural elements, references to which can be accessed in a variable or a function. We assume all expressions are well-typed.

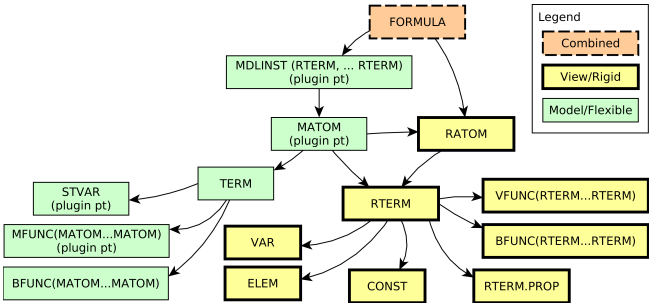

Fig. 1. IPL abstract syntax. Boxes are syntax elements, arrows — syntactic expansions.

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

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

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

Behavioral models often have parameters such as initial conditions. To provide parameter values, we introduce the fourth and outermost plugin point:

Definition 5 (Model Instantiation Clause)**.** Model instantiation clause *binds rigid terms to model parameters, wrapping* MATOM*:*

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

The values of RTERM_i are passed as parameters to the behavioral model. Finally, on top of the flexible syntax above, we can define quantification:

Definition 6 (IPL Formula)**.** IPL formulas *are logical formulas with first-order quantification over an instantiated model formula or a rigid atom.*

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

Illustrating modularity of the syntax, we give two extensions of the grammar: first with Linear Temporal Logic (LTL) [33], and second with Probabilistic Computational Tree Logic (PCTL) [11]. Here we highlight the expansion of MATOM in both plugins, while their full description is in the online appendix [47].

LTL Plugin Syntax Linear Temporal Logic (LTL) is a logic to express temporal constraints on traces [33]. We embed the usual modalities: until and next.

$$\text{TATOM}_u ::= \text{TATOM}\ \mathsf{U}\ \text{TATOM}, \text{TATOM}_x ::= \mathsf{X}\ \text{TATOM},$$
$$\text{TATOM}_a := \text{TATOM} \wedge \text{TATOM}, \text{TATOM}_n := \neg\text{TATOM},$$
$$\text{MATOM} ::= \text{TATOM} ::= \text{RATOM} \mid \text{TERM} \mid \text{TATOM}_u \mid \text{TATOM}_x \mid \text{TATOM}_a \mid \text{TATOM}_n.$$

PCTL Plugin Syntax We use extended PCTL (its variant used in PRISM) expresses probabilistic constraints over a computation tree, and its models are MDPs and discrete-time Markov chains (DTMCs) [11]. Flexible terms are as in the LTL plugin, but MATOM expands into several layered behavioral atoms.

$$\begin{aligned}
\text{PATHPROP} ::= {} & \text{RATOM} \mid \text{TERM} \mid \text{PATHPROP} \wedge \text{PATHPROP} \mid \neg\text{PATHPROP} \mid \\
& \text{PATHPROP}\ \mathsf{U}^{\leq \mathsf{k}}\ \text{PATHPROP} \mid \mathsf{X}\ \text{PATHPROP}, \\
\text{PPROP} ::= {} & P_{o\sim p}[\text{PATHPROP}],\ \text{PQUERY} ::= P_{o=?}[\text{PATHPROP}], \\
\text{MATOM} ::= {} & \text{PPROP} \mid \text{PQUERY} \mid \text{RWDPROP} \mid \text{RWDQUERY},
\end{aligned}$$

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

With the syntax defined, we encode the motivating property (Prop. 1) in IPL below. We use quantification to bind constraints on task sequences in V_{po} (with task attributes *start*, *end*, and expected *energy*) and a PCTL query for M_{pl}.

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

To summarize, IPL formulas express quantified modal constraints over symbols in Σ. We use quantification outside of flexible atoms to preserve modularity. Further, we extended the flexible part of IPL with two model property languages.

5.3 Semantics

Here we give the meaning to the IPL syntax in terms of structure Γ by reducing a formula to either Γ's model part (I_q^{M}) or Γ's the view part (I^{V}), but not both.

Domain Transfer Interpretation is based on *semantic domains* – collections of formal objects (e.g., numbers) in terms of which syntax elements can be fully interpreted. For IPL we define two domains: the *model domain* (D_{M}) and the *view domain* (D_{V}). D_{M} is associated with I_q^{M}, and D_{V}— with I^{V}.

Definition 7 (Belonging to semantic domain)**.** *Syntactic element s belongs to a semantic domain D if there exists an interpretation I such that $I(s) \in \mathsf{D}$.*

D_{M} and D_{V} are defined in Tab. 1: 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 Tab. 1 indicates if a syntax element, once interpreted, can be *transferred* to the other domain, i.e., if a bijection between its evaluations

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

Table 1. Semantic domains and transfer in IPL.

and some set in the other domain exists. "By value" is mapping to a constant in the other domain. "By reference" is mapping to an integer ID (e.g., for ELEM, unique integer IDs are generated for referencing in the model). Notice that view domain elements are mostly transferable to the model domain (except quantification). To support modularity, models can only transfer values of MDLINST.

Native semantics We interpret IPL formulas in the following context: Γ ($\mathbb{V}$, M_i with I_q^{M}, I^{V}, and I^B), states q, potentially infinite sequences of states $\omega \equiv \langle q_1, q_2, \ldots \rangle$, and mapping μ of variables to values. Starting from the bottom of Fig. 1 with rigid terms (RTERM), we gradually simplify the semantic context (denoted as the subscript of $[\![]\!]$ and on the left of $\models$). The meaning of standard logical operations from FORMULA and RATOM is found in the online appendix [47].

$$[\![\text{CONST}]\!]_\Gamma = I^B(\text{CONST}), [\![\text{VAR}]\!]_\mu = \mu(\text{VAR}), [\![\text{STVAR}]\!]_{\Gamma,q} = I_q^{\mathsf{M}}(\text{STVAR}),$$
$$[\![\text{VFUNC}(r_1, \ldots r_n)]\!]_{\Gamma,\mu} = I^{\mathsf{V}}(\text{VFUNC})([\![r_1]\!]_{\mathbb{V},\mu} \ldots [\![r_n]\!]_{\mathbb{V},\mu}) \quad \textit{if} \;\; r_1 \ldots r_n \in \text{RTERM},$$
$$[\![\text{ELEM}]\!]_{\Gamma,q,\mu} = I^{\mathsf{V}}(\text{ELEM}) = \{e\} \subseteq \mathbb{E}, [\![\text{RTERM.PROP}]\!]_{\Gamma,q,\mu} = I^{\mathsf{V}}(\text{PROP})([\![\text{RTERM}]\!]_{\mathbb{V},\mu}),$$
$$\Gamma, \omega, \mu \models \forall x : r \cdot f \quad \textit{iff} \;\; \Gamma, \omega, \mu' \models f, \text{ where } r \in \text{RTERM},$$
$$f \text{ is either FORMULA or RATOM}, \mu' = \mu \cup \{x \mapsto v\} \text{ for all } v \text{ in } [\![r]\!]_{\Gamma,\mu}.$$
$$\Gamma, \mu \models (a)[|p_1 \ldots p_n|] \quad \textit{iff} \;\; \mathbb{V}, \mathsf{M}([\![p_1]\!]_{\mathbb{V},\mu} \ldots [\![p_n]\!]_{\mathbb{V},\mu}), \mu \models a, \text{ where } a \in \text{MATOM}.$$

We provide a only brief summary of the plugin semantics for LTL and PCTL due to space limitations; for the full semantics see the online appendix [47].

LTL plugin semantics For LTL the model is a canonical transition system M_{ts} [40]. We evaluate TATOM and FORMULA on a sequence of states (ω). Logical operations and quantifiers are evaluated the same as natively.

$$\Gamma, \omega, \mu \models f \quad \textit{iff} \;\; \Gamma, q, \mu \models f, \text{ where } q \in \omega^{1,1}, f \in \text{TERM}.$$
$$\Gamma \models \text{FORMULA} \quad \textit{iff} \;\; \forall \omega : \mathsf{M}_{ts}.trcs \cdot \Gamma, \omega, \emptyset \models \text{FORMULA}.$$

PCTL plugin semantics PCTL formulas are evaluated on MDPs (M_{mdp}), or a DTMC M_{dtmc} if we collapse non-determinism [11]. Temporal operators mean the same as in LTL except the bounded until.

For $f \in$ PPROP and RWDPROP, $Prob^{\pi}(q, f)$ is a probability of f holding after q for policy π from Π:

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

where $f \in$ PATHPROP; $\sim \in \{<, \leq, >, \geq\}$; $\mathrm{opt}_{\pi\in\Pi}$ is $\sup_{\pi\in\Pi}$ if $o \equiv \max$, $\inf_{\pi\in\Pi}$ if $o \equiv \min$, no-op if $o \equiv \emptyset$; X_f is a random reward variable, Exp^{π} is its expectation.

Now the semantics of IPL has been fully defined, in a way that formalized Eq. 1 expresses the intent of informal Prop. 1. Formulas are evaluated modularly, by their reduction to subformulas, each of which is interpreted by I^{V}, I_q^{M}, or I^B.

5.4 Verification Algorithm

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

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

Below we step through Alg. 1 that solves Prob. 1. The algorithm uses several transformations, all of which are formally defined in the online appendix [47]. The first step is equivalently transforming f to its prenex normal form (PNF, i.e., all quantifiers occurring at the beginning of the formula), denoted $ToPNF(f)$.

Algorithm 1 IPL verification algorithm

procedure VERIFY(f, M)
  $f \leftarrow ToPNF(f)$ ▷ Put the formula into the prenex normal form
  $f^{FA} \leftarrow FuncAbst(\hat{f})$ ▷ Replace model instances with functional abstractions
  $f^{CA} \leftarrow ConstAbst(\hat{f})$ ▷ Replace model instances with constant abstractions
  $\hat{f}^{FA} \leftarrow RemQuant(f^{FA})$ ▷ Remove FA quantifiers
  $\hat{f}^{CA} \leftarrow RemQuant(f^{CA})$ ▷ Remove CA quantifiers
  $sv \leftarrow$ all μ s.t. $\exists I \cdot I, \mu \models \hat{f}^{FA} \not\Leftrightarrow \hat{f}^{CA}$ ▷ *Saturation:* find all variable values that satisfy non-matching abstractions
  $I^F_{sv}(F_i(\mu)) \leftarrow [\![\text{MDLINST}_i]\!]_{\mathsf{M},\mu}$ for each $\mu \in sv$ ▷ *Model checking:* run model instances to interpret functional abstractions on the above values
  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

The next step is to replace occurrences of instance terms MDLINST_i (interpretation of which is yet unknown to views/SMT) with two kinds of abstractions:
1. Functional abstraction (FA). FA replaces MDLINST_i with uninterpreted functions F_i. The arguments of these functions are the free variables that are present in the syntactic subtree of MDLINST_i. (Below, $\boldsymbol{x} \equiv x_1 \dots x_n$.)

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

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

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

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

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

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

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

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

Once variable assignments sv are determined, we can construct I^F_{sv} (a subset of I^F) by directly executing behavioral checking of MDLINST_i on concrete values:

$$I^F_{sv}(F_i)(\mu) = [\![\text{MDLINST}_i]\!]_{\mathsf{M},\mu} \text{ for all } \mu \in sv \text{ and all } i \in [1, m]. \tag{2}$$

Finally, the algorithm performs a validity check by checking satisfiability of the negation of f^{FA}. f is valid iff the check fails to find an interpretation that agrees with I^F_{sv} and satisfies $\neg f^{FA}$. We implemented this algorithm in an IPL IDE based on Eclipse (`https://www.eclipse.org`, with its source code online (`https://github.com/bisc/IPL`). More information about the IDE and an illustration of Alg. 1 on the running example is in the online appendix [47].

6 Evaluation

Here we evaluate IPL from a theoretical (soundness and termination of the algorithm) and practical (checking integration for a mobile robot) standpoint.

To avoid false positives/negatives, IPL verification should produce sound results. We prove that any answer returned by Alg. 1 is correct with respect to the semantics (independently of the plugins). To be valuable, the algorithm should terminate on practical problems. We hence provide the termination conditions.

We show that interpretations of MDLINST over sv determine the formula's validity. Correctness and termination follow directly from this result in Cor. 2.

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

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

Proof Sketch. Soundness follows from straightforward instantiation. For completeness, we assume for contraction that $I^F \models f^{FA}$ and instantiate a μ that both satisfies f^{FA} and does not, depending on the interpretation. We show that $\mu \in sv$ to derive a contradiction. Full proof is in the online appendix [47]. □

Thm. 1 leads to two corollaries (see their proofs in the online appendix [47]).

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

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

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

6.1 Case Study: Adaptive Mobile Robot

To assess the practical applicability of IPL, we guided our case study with three questions: 1. What is the role of integration properties in real systems? 2. Can we specify them with IPL? 3. Is IPL verification tractable in practice?

To address these questions, we applied IPL to a system in a case study [48]. The system was chosen to meet the following criteria: it must be a running system to ensure realism, it must be from the CPS domain to ensure fit, it must include multiple models using different formalisms to evaluate IPL's expressiveness, and we had to have access to domain experts to answer questions and assess usefulness. A TurtleBot 2 robot (described in Sec. 2) implemented using the Robot Operating System (ROS) [49] for sensing, localization, and navigation, and a model-based adaptive system for planning the robot's mission-related actions meets all of these criteria. We conducted a historical review with the project's artifacts to discover relevant models and integration properties. The case study models are available online (`https://github.com/bisc/IPLProjects`).

Our case study focused on a planning model M_{pl} and a power model M_{po} because power is a prominent concern in the system and these two models have a complex dependency. Both models co-evolved throughout the project, and we collected over 10 variants of these models with of varying sophistication.

Integration Properties. An example integration property between M_{pl} and M_{po} is that they must agree on energy spent in various missions; otherwise the robot may run out of power. (A mission is made up of different energy-spending motion tasks such as forward movement, rotation, and charging. A *power-successful mission* can be done with a given initial power budget.)

View Modeling and Verification. To formalize the integration properties, we chose to create a view (V_{po}) for M_{po} and combine it with a behavioral interface

to M_{pl}. There are many ways to construct an appropriate view, and we took the route of creating a *task library* — enumerating all relevant atomic tasks.

V_{po} has to agree with M_{pl} on the task primitives, otherwise the integration check will always fail. Each motion command is an architectural element with its own *id*, *start*, *end*, and *energy* (which is computed by M_{po} given a distance and a speed, hence making V_{po} a correct view for M_{po}). The only requirement for the view is that it contains *all* the objects of interest (here, atomic tasks).

Another view is a map view (V_{map}), containing locations (as components) and their connections. We discovered 5 maps, organized in two categories. The first category contains 9 locations (including 1 charging station) and 9-10 edges. The second category contains 12 locations (including 4) and 13 edges.

Both V_{po} and V_{map} have been created by automated transformations that require the same map artifact. V_{po} requires equations from M_{po} and outputs a library of tasks encoded in the Architecture Analysis and Design Language (AADL) [50]. V_{map} outputs a list of locations in AADL. In total, we generated over 30 variants of views to represent relevant combinations of task primitives.

Using the above view abstractions, we specified dozens of integration property variants (similar to Eq. 1) for various mission features and lengths. In map-related properties, quantified variables iterate over locations. In power-related properties, quantified variable iterate over atomic tasks. Examples of these properties are highlighted in the online appendix [47].

Outcomes. To answer our first question (see top of Sec. 6.1), we discovered that complex integration properties appear when several models contain interrelated data (in our case, locations, connections between them, and energy expenditures for tasks). These properties serve as steps in safety reasoning that would otherwise use oversimplified and unsupported assumptions (e.g., models agree on energy). If these assumptions are not satisfied, the system falls short of its goals. Thus, IPL fills in an important niche of reasoning for multi-model systems.

To answer the second question, we focused on multiple variants of power-related integration properties for M_{po} and M_{pl}.We were able to represent all relevant point-to-point missions up to a bounded number of recharging actions. The end-to-end power-safety argument for the robot relies on these integration properties: if M_{po} has worst-case error $\overline{err_pow}$, M_{pl} has worst-case error $\overline{err_mdp}$, the worst-case consistency error is $\overline{err_cons}$, then to not run out of power, the battery has to have at least $g(\overline{err_pow}, \overline{err_mdp}, \overline{err_cons})$ charge during any execution, where g is some function (addition in simple cases). Thus, we observed that integration properties verify bounds of consistency errors, which are inputs to end-to-end safety arguments.

We discovered several critical inconsistencies in the models we observed: (1) the MDP does not check whether the battery was enough for the last step (thus, in some missions the robot would run out); (2) turn energy was inconsistent making one turn action add energy to the battery (caused by a bug in the model generation code); (3) M_{po} and M_{pl} disagreed significantly in their energy predictions for tasks with near-zero times because of the non-zero y-intersect in M_{po} (recall that it was constructed using regression). We therefore conclude that IPL is capable of finding model inconsistencies in real-world projects.

Performance. We evaluated the performance of the Eclipse-based IPL implementation using the power-related property variants. Specifically, we executed 24 verification runs by varying the number of tasks and the map and toggling the mission features—variable length missions, charging, and rotations. IPL's performance is reasonable for practical purposes, with a remarkably low overhead. Although larger missions with more features led to substantially longer times, IPL finished within several hours. The details are in the online appendix [47].

7 Discussion

This paper makes a significant step towards bridging the semantic gap between heterogeneous CPS models. The Integration Property Language enables systems engineers to specify expressive properties over behavioral and static semantics of multiple models in a way that is both modular and extensible. IPL specifications are soundly checkable with a combination of SMT solving and model checking. The case study showed that IPL can encode relevant real-world integration properties and verify them in reasonable times.

IPL relies on existing views, models, and analysis tools for reasoning. It also shares their limitations on automation and performance. In practice, extra automation or manual effort is required for views to remain up-to-date with models. IPL performance is limited by satisfiability solving for many constraints and quantified variables. Improvements in the state-of-the-art satisfiability and model checking should lead to comparable improvements in the IPL performance.

IPL allows behavioral checking to be carried out independently of where its inputs come from, thereby supporting custom workflows in diverse engineering disciplines. This freedom, however, comes at a cost of expressiveness: we could not allow complete transfer of view functions to $\mathsf{D_M}$ (Tab. 1 allows it only for transferable arguments), which would need callbacks from model checking to views to evaluate a view function. This feedback loop would create a dependency from models to views and negatively impact modularity and extensibility of IPL.

Future work will focus on three areas: (1) incorporating other property languages into IPL and conducting more case studies, (2) handling models (such as Simulink) that are widely used in CPS but do not have a rigorous property language, and (3) an analysis of scalability and effectiveness with respect to other integration methods (e.g., the "supermodel" approach).

Acknowledgments

This material is based on research sponsored by AFRL and DARPA under agreement number FA8750-16-2-0042. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the AFRL, DARPA or the U.S. Government.

References

1. Mosterman, P.J., Zander, J.: Cyber-physical systems challenges: a needs analysis for collaborating embedded software systems. Software & Systems Modeling **15**(1) (February 2016) 5–16
2. Fitzgerald, J., Larsen, P., Pierce, K., Verhoef, M., Wolff, S.: Collaborative Modelling and Co-simulation in the Development of Dependable Embedded Systems. In Mry, D., Merz, S., eds.: Integrated Formal Methods. Volume 6396 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2010) 12–26
3. Valukas, A.: Report to Board of Directors of General Motors Company Regarding Ignition Switch Recalls. Technical report, Jenner & Block (2014)
4. Sztipanovits, J., Koutsoukos, X., Karsai, G., Kottenstette, N., Antsaklis, P., Gupta, V., Goodwine, B., Baras, J., Wang, S.: Toward a Science of Cyber-Physical System Integration. Proc. of the IEEE (2011)
5. Alur, R.: Principles of Cyber-Physical Systems. The MIT Press, Cambridge, Massachusetts (April 2015)
6. Dijkman, R.M.: Consistency in multi-viewpoint architectural design. PhD thesis, Telematica Instituut, Enschede, The Netherlands (2006)
7. Maoz, S., Ringert, J.O., Rumpe, B.: 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, New York, NY, USA, ACM (2013) 444–454
8. Reineke, J., Tripakis, S.: Basic Problems in Multi-View Modeling. In: Proc. of the 19th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Springer Berlin Heidelberg (January 2014)
9. Bhave, A.: Multi-View Consistency in Architectures for Cyber-Physical Systems. PhD thesis, Carnegie Mellon University (December 2011)
10. Howard, R.A.: Dynamic Programming and Markov Processes. Technology Press of the Massachusetts Institute of Technology (1960)
11. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic Model Checking. In Bernardo, M., Hillston, J., eds.: Formal Methods for Performance Evaluation. Number 4486 in Lecture Notes in Computer Science. Springer Berlin Heidelberg (2007) 220–270
12. Bhave, A., Krogh, B., Garlan, D., Schmerl, B.: View Consistency in Architectures for Cyber-Physical Systems. In: IEEE/ACM International Conference on Cyber-Physical Systems (ICCPS). (2011)
13. Nuseibeh, B., Kramer, J., Finkelstein, A.: A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification. IEEE Trans. Softw. Eng. **20**(10) (October 1994) 760–773
14. Egyed, A.F.: Heterogeneous View Integration and its Automation. PhD thesis, University of Southern California (2000)
15. Hoare, C.A.R.: Communicating Sequential Processes. Commun. ACM **21**(8) (August 1978) 666–677
16. Smith, G.: The Object-Z Specification Language. Springer (2000)
17. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. 1st edn. Cambridge University Press, New York, NY, USA (2010)
18. Karsai, G., Sztipanovits, J.: Model-Integrated Development of Cyber-Physical Systems. In Brinkschulte, U., Givargis, T., Russo, S., eds.: Software Technologies for Embedded and Ubiquitous Systems. Volume 5287 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2008) 46–54

19. Ruchkin, I.: Integration Beyond Components and Models: Research Challenges and Directions. In: Proc. of the Third Workshop on Architecture Centric Virtual Integration (ACVI), Venice, Italy (2016) 8–11
20. Kruchten, P.: The 4+1 View Model of Architecture. IEEE Software **12** (1995) 42–50
21. Rajhans, A., Bhave, A., Loos, S., Krogh, B., Platzer, A., Garlan, D.: 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)
22. Marinescu, R.: Model-driven Analysis and Verification of Automotive Embedded Systems. PhD thesis, Maladaren University (2016)
23. Vanherpen, K., Denil, J., David, I., De Meulenaere, P., Mosterman, P.J., Torngren, M., Qamar, A., Vangheluwe, H.: Ontological reasoning for consistency in the design of cyber-physical systems, IEEE (April 2016) 1–8
24. Torngren, M., Qamar, A., Biehl, M., Loiret, F., El-khoury, J.: Integrating viewpoints in the development of mechatronic products. Mechatronics (2013)
25. Rajhans, A., Krogh, B.H.: Heterogeneous verification of cyber-physical systems using behavior relations. In: Proc. of the 15th ACM conference on Hybrid Systems:Computation and Control (HSCC), New York, NY, USA, ACM (2012) 35–44
26. Lee, E.A., Neuendorffer, S., Zhou, G.: System Design, Modeling, and Simulation using Ptolemy II. Ptolemy.org (2014)
27. Combemale, B., Deantoni, J., Baudry, B., France, R., Jezequel, J.M., Gray, J.: Globalizing Modeling Languages. Computer **47**(6) (June 2014) 68–71
28. Sztipanovits, J., Bapty, T., Neema, S., Howard, L., Jackson, E.: 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)
29. Simko, G., Lindecker, D., Levendovszky, T., Neema, S., Sztipanovits, J.: Specification of Cyber-Physical Components with Formal Semantics Integration and Composition. In: Model-Driven Engineering Languages and Systems. Springer Berlin Heidelberg (January 2013) 471–487
30. Ruchkin, I., de Niz, D., Chaki, S., Garlan, D.: Contract-based Integration of Cyber-physical Analyses. In: Proc. of the Intl. Conf. on Embedded Software (EMSOFT), New York, NY, USA, ACM (2014) 23:1–23:10
31. Costa, A.D., Laroussinie, F., Markey, N.: Quantified CTL: Expressiveness and Model Checking. In: CONCUR 2012 Concurrency Theory. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (September 2012) 177–192
32. Borger, E., Gradel, E., Gurevich, Y.: The Classical Decision Problem. Springer (October 2001)
33. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, 1977. (October 1977) 46–57
34. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag (1992)
35. Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Combination Methods for Satisfiability and Model-Checking of Infinite-State Systems, Springer Berlin Heidelberg (July 2007) 362–378
36. Cimatti, A., Roveri, M., Susi, A., Tonetta, S.: Formalizing requirements with object models and temporal constraints. Software & Systems Modeling **10**(2) (August 2009) 147–160
37. Gabbay, D.M.: Fibred Semantics and the Weaving of Logics Part 1: Modal and Intuitionistic Logics. The Journal of Symbolic Logic **61**(4) (1996) 1057–1120

38. Konur, S., Fisher, M., Schewe, S.: Combined model checking for temporal, probabilistic, and real-time logics. Theoretical Computer Science **503** (September 2013) 61–88
39. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). J. ACM **53**(6) (2006) 937–977
40. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. **8**(2) (April 1986) 244–263
41. Nelson, G., Oppen, D.C.: Simplification by Cooperating Decision Procedures. ACM Trans. Program. Lang. Syst. **1**(2) (October 1979) 245–257
42. De Moura, L., Bjrner, N.: 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, Berlin, Heidelberg, Springer-Verlag (2008) 337–340
43. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)
44. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., Stafford, J.: Documenting Software Architectures: Views and Beyond. 2 edn. Addison-Wesley Professional (October 2010)
45. Gurfinkel, A., Shoham, S., Meshman, Y.: SMT-based Verification of Parameterized Systems. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE 2016, New York, NY, USA, ACM (2016) 338–348
46. Kroening, D., Strichman, O.: Decision Procedures - An Algorithmic Point of View. Springer (2008)
47. Ruchkin, I., Sunshine, J., Iraci, G., Schmerl, B., Garlan, D.: Appendix for IPL: An Integration Property Language for Multi-Model Cyber-Physical Systems. (2018) http://acme.able.cs.cmu.edu/pubs/uploads/pdf/fm2018-appendix.pdf.
48. Yin, R.K.: Case Study Research: Design and Methods. 4th edn. Sage Publications, Inc (October 2008)
49. Quigley, M., Gerkey, B., Smart, W.D.: Programming Robots with ROS: A Practical Introduction to the Robot Operating System. 1 edn. O'Reilly Media (November 2015)
50. Feiler, P.H., Gluch, D.P., Hudak, J.J.: The Architecture Analysis & Design Language (AADL): An Introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University (2006)