Architectural Abstractions for Hybrid Programs

Ivan Ruchkin, Bradley Schmerl, David Garlan
Institute for Software Research
Carnegie Mellon University
Pittsburgh, PA, USA
{iruchkin, schmerl, garlan}@cs.cmu.edu

ABSTRACT

Modern cyber-physical systems interact closely with continuous physical processes like kinematic movement. Software component frameworks do not provide an explicit way to represent or reason about these processes. Meanwhile, hybrid program models have been successful in proving critical properties of discrete-continuous systems. These programs deal with diverse aspects of a cyber-physical system such as controller decisions, component communication protocols, and mechanical dynamics, requiring several programs to address the variation. However, currently these aspects are often intertwined in mostly monolithic hybrid programs, which are difficult to understand, change, and organize. These issues can be addressed by component-based engineering, making hybrid modeling more practical. This paper lays the foundation for using architectural models to provide component-based benefits to developing hybrid programs. We build formal architectural abstractions of hybrid programs and formulas, enabling analysis of hybrid programs at the component level, reusing parts of hybrid programs, and automatic transformation from views into hybrid programs and formulas. Our approach is evaluated in the context of a robotic collision avoidance case study.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architecture; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

Keywords

architectural view, hybrid program, cyber-physical system

1. INTRODUCTION

Component-based software engineering has been successful at reaping numerous benefits of designing systems with clear components and interfaces. Approaches to reuse components with contracts [20] and reason about dynamic assemblies [7] have advanced engineering to build larger, more complex systems cheaper, faster, and with fewer errors. Formal and verified composition mechanisms help design, refine, and verify large safety-critical embedded systems with reduced complexity [3, 4, 20].

For a class of systems, often called cyber-physical systems (CPS) [16] or mechatronics, the progress of compositional engineering has been somewhat slower. Software in such systems interacts very closely with complex continuous physical processes, like mechanical movement or heat exchange, which are poorly represented by typical discrete software languages and tools [16]. Lacking a way to form coherent abstractions of these physical phenomena, software engineering methods have to abstract continuous physics out, which leads to unsatisfying results. For example, software engineers have to indirectly rely on hard deadlines to not "miss" too much time instead of explicitly reasoning about processes that take place during this time [9].

The field of hybrid systems, on the other hand, has been developing ways to model CPS with direct representation of physical dynamics. In particular, models that combine discrete and continuous state changes, like hybrid automata [2] and hybrid programs (HP) [21], give an ability to verify critical system properties without abstracting away physicality. Differential dynamic logic ($\mathsf{d}\mathcal{L}$) [22] is an approach to describe and prove logical statements over HPs. $\mathsf{d}\mathcal{L}$ relies on semi-automated theorem proving to produce a proof certificate that establishes that a system satisfies a property.

Despite the successes of hybrid systems in tackling continuous phenomena, it is still a challenge to incorporate hybrid models into scalable and cost-efficient model-driven engineering (MDE) for modern CPS. Many hybrid models are monolithic blocks where diverse aspects like controller design, component communication protocol, and mechanical dynamics are intertwined and cannot be easily separated, as exemplified by many programs in [22]. At the same time, multiple models are required to represent a system adequately, and it is difficult to separate parts of each model, reuse them, and analyze new models for correct integration of parts. These issues lead to hybrid approaches like $\mathsf{d}\mathcal{L}$ being impractical for MDE.

MDE of large-scale safety-critical CPS needs to *both* formally guarantee properties of discrete-continuous interaction *and* reduce engineering costs through composition and reuse. This paper shows how architectures for hybrid programs can be used to address both of these needs. Logical verification with $\mathsf{d}\mathcal{L}$ specifies and proves properties of hybrid systems, and the architectural abstractions provide benefits

CBSE'15 May 04–08, 2015, Montreal, QC, Canada

http://dx.doi.org/10.1145/2737166.2737167

of component-based engineering: reuse of complex specifications across multiple models and high-level analysis of these models. The approach allows logical reasoning about hybrid programs at a component level. We demonstrate our approach on an existing case study [18] of collision avoidance, which is a problem commonly addressed in CPS [6, 17].

This work makes several significant contributions. First is a formal definition and semantics of HP architecture views and formulas, which enables design and fully automated generation of HPs using higher-level abstractions. Second is applications of HP architectures: type-based reuse and information flow analysis for HP architectures that advance practical MDE with hybrid models. Third is an implementation of an HP architecture plugin in AcmeStudio [24] to support manipulation and analysis of HP views, and generation of HPs from the views.

The next section introduces the robotic collision avoidance case study, which we will use throughout the paper for illustration. Sec. 3 and 4 describe, respectively, basic $\mathtt{d}\mathcal{L}$ constructs and motivating issues for componentizing with HPs. Addressing these issues, in Sec. 5 we propose architectural abstractions that represent hybrid programs and formulas. In Sec. 6 we describe how our approach supports model-based engineering with HP. Afterwards, we evaluate our approach on the robotic collision avoidance case study in Sec. 7, survey the related work and conclude the paper.

2. ROBOTIC COLLISION AVOIDANCE

Vehicle collision avoidance is a classic CPS problem, used to illustrate the needs of hybrid modeling [6, 15, 17]. However, reasoning about collision avoidance remains a challenge for design and verification: even the most sophisticated autonomous vehicle systems like those from Cornell or MIT do not achieve practical safety [10]. This is in large part due to absence of MDE methods that combine formal safety guarantees with means to cope with the system's complexity.

To help motivate the problem and illustrate our approach, consider MDE of an autonomous wheeled robot moving in a 2D space with other obstacles [18]. The robot can sense obstacles in its vicinity (e.g., with a camera, laser scanner, or a sonar) and determine its own position (GPS or 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 this tactic. The goal is creating a robot that avoids collisions and dangerous navigation. In this discussion, we will concentrate on modeling systems most relevant to collision avoidance: tactical planning, movement control, and movement itself.

Collision safety requirements may have different interpretations, from passive safety (collisions are allowed when the robot is stopped), to passive friendly safety (a moving obstacle needs an opportunity to stop) [15], to absolute safety (collisions should not occur under any circumstances) [6]. A fundamental tradeoff in modeling is that stricter notions of safety lead to unnecessarily conservative behaviors or unrealistic assumptions. For example, a robot may remain stationary forever in a crowded area if it adopts the criterion of absolute safety. It is essential for a modeler 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.

Concern	**Variations**
Tactic	Avoiding obstacles, passing intersection, arriving at goal.
Physical space	Unconstrained, constrained box, intersection.
Desired property	Passive safety, passive friendly safety, liveness.
Robot trajectory	Grid, lines, arcs, spirals.
Obstacle behavior	Stationary, moving nondeterministically, 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 1: Concerns and variations in robotic collision avoidance modeling.

Safety definition isn't the only varying concern that affects modeling of robotic collision avoidance. Another is the set of assumptions about the robot's mechanical and embedded systems: What kind of trajectories can the robot move in? How well can acceleration be controlled? How precise and immediate is sensing of obstacles? Yet another concern is obstacles' behavior: Can obstacles move with arbitrary speed or acceleration? Can obstacles switch between stationary and moving? Are obstacles trying to avoid a collision? Variability in answering these questions affects the robot's decisions and guarantees that can be obtained from models. Therefore, an engineer has a large modeling space to explore when developing these systems. We summarize some of the high-level concerns that underly modeling of robotic collision avoidance in Tab 1.

Existing modeling approaches used in practice fall short in providing both strong formal guarantees as well as component-based structure for robotic collision avoidance. Classic embedded software modeling, like SysML, fails to take into account the physical implications of variants such as those listed in Tab. 1 and the effects they have on the system design. Simulation [23] of various sensing, movement, and actuation designs are often used to gain confidence in and compare designs. However, these simulations are not exhaustive, and the large number of potential states given the large space of variations above means that higher-confidence alternatives to simulations are required.

The hybrid CPS modeling approaches like $\mathtt{d}\mathcal{L}$ [21] allow for formal proofs of safety properties, but struggle with large model spaces: even for the relatively simple case of a robot and an obstacle, there is a number of concern combinations that a modeler has to address by creating several independent *model variants*. For example, one model variant may tackle liveness in an intersection with imprecise sensing, while another may model safely avoiding obstacles using precise sensing. Currently, modelers copy and change variants manually. But this approach becomes intractable when the number of variants becomes large: modelers have limited

or no support to create and differentiate model variants, or analyze proper fit between parts of the model.

Thus, a successful collision avoidance modeling approach requires, (i) an explicit formal representation of continuous physics; (ii) a way to compose and reuse model parts that relate to a particular concern, such as sensing with bounded error; and (iii) a higher-level analysis of model parts to ensure correct composition. We introduce an approach of hybrid program architectures that satisfies these requirements. Our approach rests upon the logical foundation of hybrid programs and $\mathsf{d}\mathcal{L}$, which we describe in the next section.

3. BACKGROUND: HP AND DDL

The robot collision avoidance system is Sec. 2 involves a blend of continuous (robot movement) and discrete (controller software) parts. Classically, software models like state machines [14] discretize time and state. Although this gives relative simplicity and power to apply automated verification techniques like model checking, discretization makes reasoning about continuous processes cumbersome [16]. Hence, there is a need for modeling formalisms that embrace continuous dynamics instead of abstracting it away.

Differential dynamic logic (DDL, or $\mathsf{d}\mathcal{L}$) [21] is a one such promising verification approach for cyber-physical systems. $\mathsf{d}\mathcal{L}$ incorporates discrete transitions and continuous evolutions into its semantic model, making it convenient to represent robotic decisions and movement [18]. $\mathsf{d}\mathcal{L}$ relies on hybrid programs (HPs)[1] to encode system state changes over variables. To cover broader classes of systems and abstract away irrelevant details (e.g., the exact behavior of robot surroundings or the exact timing moments, which may be unpredictable), HPs employ non-determinism in variable values and control transitions.

The basic syntax of HPs is shown in Tab. 2. The flow of programs is controlled by the sequential composition (;), nondeterministic choice ($\cup$), and non-deterministic repetition (*). The semantics of a HP is formally defined over the state represented by its variables. The state is changed through value assignments and continuous evolutions. Continuous evolutions advance variable values along differential equations within an evolution domain F, continuing for an arbitrary amount or stop immediately. A test operator cuts off execution branches; it is commonly used in conjunction with non-deterministic assignment: $x := *; ?x > 0$ cuts off all non-positive values of variable x.

Given a hybrid program α, one can write logical assertions with a $\mathsf{d}\mathcal{L}$ formula ϕ:

$$\phi ::== \theta_1 \sim \theta_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \forall x\phi \mid [\alpha]\phi \mid \langle\alpha\rangle\phi, \qquad (1)$$

where θ_1 and θ_2 are real arithmetic expressions and $\sim \in \{<, \leq, =, \geq, >\}$. Other operators like $\wedge$ and $\rightarrow$ are derived from the operators in (1). 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 take a form of $\varphi \rightarrow [\alpha]\phi$ or $\varphi \rightarrow \langle\alpha\rangle\phi$. Consider a robot with position x, velocity v, and acceleration a trying to reach its goal g in a 1D space. The robot can arbitrarily choose an acceleration ($a := *$) between full throttle ($a \leq A$, where A is the maximum possible acceleration) and full braking ($-b \leq a$, where b is the maximum possible braking power), but cannot drive backwards ($v \geq 0$). The robot's control alternates with physical dynamics of kinematic movement ($v' = a, x' = v$) in a non-deterministic loop. For example, the following formula expresses that if a robot hasn't yet reached its goal ($x < g$), there exists an execution (expressed with the $\langle\rangle$ modality) where the robot reaches its goal ($x \geq g$):

$$x < g \rightarrow \langle\ (a := *; ? -b \leq a \leq A; \{v' = a, x' = v, v \geq 0\})^*\ \rangle(x \geq g) \qquad (2)$$

This formula can be fed to the automated theorem prover KeYmaera [22] that builds a proof tree via proof rules. Simple models may be proved fully automatically, while more complicated ones require manual steps and specification of differential invariants. Once a proof is found, it establishes that a property holds as a logical consequence of the model. This is a stronger guarantee than can be given by simulation and numerical analysis approaches. Furthermore, such guarantees factor in continuous processes, unlike these other approaches that rely upon purely discrete models.

A $\mathsf{d}\mathcal{L}$ model variant is a specification file unrelated to other variants. Each variant contains one $\mathsf{d}\mathcal{L}$ formula that logically specifies the desired property (cf. 2). A proved variant is supplemented with a proof and contributes to the system design. If a variant is shown to not hold the property, a counterexample situation is generated to demonstrate the property violation, giving insight into future modeling. Thus, maintaining multiple model variants is an integral part of modeling with $\mathsf{d}\mathcal{L}$, further increasing the space of model variants mentioned in Sec. 2.

The bulk of a model variant is devoted to one or several HPs, which are wrapped into a $\mathsf{d}\mathcal{L}$ formula and formalize state changes using operations in Tab. 2. Additionally, a model variant includes a section containing declarations of constants, such as b for maximum braking, and variables, such as a, v, and x to model robot's kinematics. All constants and variables are global, so any part of the HP can read and write them. This makes it difficult to tell what variables and instructions are closely associated with each other, and which are not. Thus, each variant is a monolithic block of variables, operators, and logical statements, with many modeling concerns intertwined with each other.

Statement	Informal meaning
$\alpha;\beta$	Sequential composition; first executes α and then β
$\alpha \cup \beta$	Non-deterministic choice; executes either α or β
α^*	Non-deterministic repetition; executes α 0 or more times
$x := \theta$	Assignment of value θ to variable x
$x := *$	Assignment of an arbitrary value to variable x
$x_1' = \theta_1, \ \ldots$ $x_n' = \theta_n \ \&F$	Continuous evolution of x_i along differential equations $x_i' = \theta_i$ restricted to an evolution domain specified by formula F
$?F$	Test if formula F holds; proceed if yes, otherwise abort.

Table 2: Syntactic constructs of hybrid programs.

[1]The *hybrid* underlines a combination of discrete and continuous semantics.

In the current practice of dL modeling, model variants are derived from each other by copying and modifying the code, adopting a "clone-and-own" approach. For instance, to derive a variant with sensor uncertainty, a model with no sensor uncertainty is copied, the sensed values are augmented with bounded errors, and the controller is adjusted to handle these changes. As a result, many variants share redundant chunks of HP code that specify the same aspects.

The monolithic structure and redundant code of dL model variants make it difficult to understand, modify, and reuse them. These barriers impede the exploration of variation space, making dL costly to apply in practice despite its promising formal guarantees. If, however, we could separate model variants into components, it would provide an essential structure within each variant to start organizing the variant space and reusing model fragments. Having observed the need for component abstractions, we turn our attention now to the challenges and benefits of componentizing hybrid programs.

4. TOWARDS COMPONENTIZING HP

Component abstractions for hybrid programs could facilitate reuse of common model fragments, thus structuring the variant space and reducing its exploration effort. In addition, component abstractions would enable reasoning about composition errors that are otherwise implicit (much like components themselves). However, it is not obvious how to add component abstractions to HPs and dL: tightly coupled programs do not yield themselves to easy decomposition, and complex dL formulas mix several HPs in the same specification. The rest of this section discusses the challenges and benefits of a component-based HP in more detail.

4.1 Challenge: Tight Coupling within Model Variants

Each model variant combines a variety of intertwined concerns (e.g., Tab. 1). Modifications to such highly coupled models may penetrate many parts of the model. In particular, new variables may be necessary to represent richer state, new constants may be needed to place constraints on the state, new differential equations may be required to describe new trajectories, and new control choices may be needed to actuate the robot safely.

To exemplify this challenge, let us consider different patterns for modeling time in dL:

- Event-triggered timing (ETT). Time is not represented in a model as a variable. Instead, event conditions are part of an evolution domain constraint F. For example, the system can execute until the robot has to brake, which is when time flow is interrupted and the control is handed to the robot.
- Local continuous timing (LCT) with bounded nondeterministic 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 we chose to use, for instance, local continuous timing, then a number of changes must be made throughout the model: first of all, t needs to be reset in the loop, but the spot needs to be carefully chosen depending on whether other parts of the loop use t. Second, the differential equations and evolution domain constraints need to be updated. Furthermore, t and ε need to be added to the variable and constant declarations, respectively. Finally, control decisions are very likely to change to accommodate a possible delay of ε seconds. Thus, even a relatively simple change impacts many parts of a hybrid program, which makes it difficult to encapsulate this change in a component.

4.2 Challenge: Multi-Program Formulas

Some dL formulas go beyond the simple structure $\varphi \rightarrow [\alpha]\psi$. For example, passive friendly safety requires that for all executions of a robot and a moving obstacle [RobotObst], a robot should always stop or be far enough from the obstacle to stop (*RobotFar*). Assuming the obstacle ⟨DetailedObst⟩ is far enough from the robot (*ObstFar*), should have an opportunity to stop and avoid collision (*Safe*). The following formula from [18] captures this property:

$$Pre \rightarrow [\mathsf{RobotObst}](\mathit{RobotFar} \wedge (\mathit{ObstFar} \rightarrow \langle \mathsf{DetailedObst} \rangle \mathit{Safe})) \tag{3}$$

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

The challenge of componentizing such multi-program formulas arises from mixing the logical and imperative parts: the initial condition for a HP is mixed with another HP and is separated from it with logical predicates. In addition, part of the state space is implicitly shared. So the fundamental question is: How can multi-program formulas be decomposed into and rebuilt from reusable components?

4.3 Benefit: Reuse of Model Fragments

Reuse of model fragments is highly desirable for dL modeling to facilitate model space exploration. To illustrate this point, we extracted sets of robot's physical equations[2] from [18] in Tab. 3. In the simplest case, a robot is moving with velocity v and acceleration a along a line in a binary orientation $o \in \{1, -1\}$. A slightly more complicated case is with movement along a grid net, defined by directions $o_{fb}, o_{hv} \in \{1, -1\}$, and a line with direction defined by $dx, dy \in [0; 1]$. Modeling movement in arcs of fixed radius r requires representing rotational velocity ω and linking it to a. To enable spinning on a single spot ($r = 0$), $w' = \frac{a}{r}$ needs to be rewritten with a new helper variable s as $s' = a, s = wr$, introducing yet another physical model. Finally, the model of spiral movement does not link rotational velocity ω with a.

[2]Non-linear functions like sin are avoided in the interest of provability [22].

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

Table 3: Versions of the robot's physical dynamics.

Each row in Tab. 3 captures a complex mechanical movement. Changing between these dynamics and introducing them to new models currently relies on tedious and error-prone manual editing, which is likely to introduce errors in models, especially when several variants are modified at the same time. Instead, identifying and reusing these dynamics at a higher level of abstraction would distinguish existing model variants based on their physical model and reduce the effort of deriving new ones.

4.4 Benefit: Information Flow Analysis

Even though a $\mathsf{d}\mathcal{L}$ formula may be well-formed and provable, it may still have implicit modeling errors that lead to a system's safety violation. For instance, a robot may read an obstacle's velocity V_o right after the obstacle assigns it in every loop iteration and use this perfect information to avoid the obstacle with a narrow margin. Although this usage does not violate the model's formal safety specification, any implementation would fail to achieve this perfect prediction of the obstacle's next action, making the margin too narrow to avoid a collision.

More abstractly, this problem is rooted in *incorrect information flow*: hybrid programs do not restrict exchange of data between entities, thus allowing information exchange that would not be possible in reality. A solution to this problem would distinguish acting entities, impose information flow constraints, and analyze HPs to find violations.

Let us consider a more detailed example of information exchange. A modeler sketches a variant for the intersection tactic to model the safety of robot's decision to cross an intersection. A robot advances along the x axis, and a moving (unfriendly) obstacle is approaching the intersection (x_o, y_r) along the y axis. The model sketch takes the following form:

$$
\begin{aligned}
\mathit{ISect} \equiv\ & \mathit{Pre} \rightarrow [&& (4)\\
& (\ (a_o := *; ? -b \leq a_o \leq A; && (5)\\
& \quad (?x_r > x_o \vee \mathit{Safe}; a_r := *; ? -b \leq a_r \leq A) && (6)\\
& \qquad \cup\ (?\mathit{CanGoBefore} \vee \mathit{CanGoAfter}; && (7)\\
& \qquad\quad a_r := *; ?0 \leq a_r \leq A) && (8)\\
& \qquad \cup\ (?v_r = 0; a_r := 0) \cup\ a_r := -b; && (9)\\
& \quad t := 0; && (10)\\
& \quad \{x'_r = v_r, v'_r = a_r, && (11)\\
& \qquad y'_o = v_o, v'_o = a_o, && (12)\\
& \qquad t' = 1, t \leq \varepsilon, v_r \geq 0, v_o \geq 0\}\ )^* && (13)\\
&](\|(x_r, y_r) - (x_o, y_o)\| > 0) && (14)
\end{aligned}
$$

Given certain preconditions *Pre* (4), formula *ISect* states that a collision does not occur (14) in any execution of the hybrid program (5)–(13). The program gives the obstacle a non-deterministic choice of acceleration (5). The robot has several decision branches: it may choose any acceleration if it has already crossed the intersection or is safely far away from it (6), accelerate fully to pass before or after the obstacle (8), remain stopped, or brake (9). The timing pattern is LCT, and the robot and obstacle cannot drive backwards (13). The conditions *Safe*, *CanGoBefore*, and *CanGoAfter* are at the heart of the robot's decision-making and need to be determined by a modeler.

Now imagine that the sketch *ISect* is handed over to another modeler to complete it with the robot's branch decisions. These decisions can be made differently depending on the information available to the robot. There are two options: (i) sense y_o and assume the obstacle's velocity bounds are V_{max} and V_{min}; and (ii) sense y_o and V_o and assume the obstacle's acceleration bounds are A_{max} and $A_{min} \geq 0$. In case of (i) it is possible to estimate the obstacle's slowest time to the intersection as $t_{int} \equiv (y_r - y_o)/V_{min}$ and define $\mathit{CanGoAfter} \equiv x_r + V_r t_{int} > x_o$. For (ii) the modeler can instead approximate t_{int} as a root of $y_r - y_o + V_o t_{int} + A_{min} t_{int}^2/2 = 0$.

Options (i) and (ii) critically rely on different assumptions about the system design: whether the obstacle's velocity is available to the robot and what variables are bounded. If the first modeler intended (i) based on knowledge of the robot's sensors and obstacle parameters, but the second modeler instead chooses (ii), it is possible for the condition *CanGoAfter* to lead to a collision because the model would not fit the reality. Since the original *ISect* sketch does not provide any guidance or constraint on what information about the obstacle is available to the robot, analogous interaction errors may also happen in the specification of *CanGoBefore* and *Safe* across many model variants. Detecting these errors with the original $\mathsf{d}\mathcal{L}$ semantics would be nearly impossible due to the monolithic structure of a HP that does not accociate between variables and statements with any part of model.

To summarize this section, componentizing HPs is difficult due to tight coupling of model fragments and complex logical combinations of hybrid programs. However, if introduced, component abstractions would facilitate reuse and enable high-level analysis of information flow. In the following section we overcome the componentization challenges with an architectural approach that gives a component-based representation to HPs.

5. ARCHITECTURES FOR HP

This section develops architectural abstractions for hybrid programs. Our goal is to represent hybrid programs and multi-program $\mathsf{d}\mathcal{L}$ formulas at a component level, generating plain HPs from reused model fragments. To achieve the goal, we define the following: (i) a component as a primary entity for information hiding and reuse, (ii) a connector for information exchange and HP code transformation, (iii) an architectural view – a collection of components and connectors – to represent a single HP, and (iv) a logical formula over views to represent a model variant.

We base our approach on the Acme architecture description language that describes hierarchical assemblies (views) of components and connectors with properties [13]. Com-

ponents connect through ports, to which connectors attach their roles. Acme facilitates component-based reuse through the notion of a *style* – a collection of component, connector, port, and role types that can be reused in many views. Acme is not limited to software structures: it has also been extended to capture physical entities (cf., Bhave's CPS styles [5]). Constructs introduced in this section give meaning to Acme's elements in terms of HPs and $\mathsf{d}\mathcal{L}$. This way, Acme provides high-level manipulation, reuse, and analysis, HPs provide imperative specification, and $\mathsf{d}\mathcal{L}$ provides logical specification and verification.

5.1 Components

First, we need to distinguish the acting entities of hybrid programs – *hybrid program actors*. Actors encapsulate a state, expose part of it through ports, and combine discrete control and continuous physics:

DEFINITION 1. Hybrid program actor *HPA is a component instance that is characterized by a tuple:*

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

The state of an actor is a set of variables and constraints: $State \equiv (Vars, Constr)$, where $Vars \equiv \{v_i\}$ is set of a typed variables[3], and $Constr \equiv \{\varphi_i\}$ is a set of state constraint formulas, defined by Eq. 1, over variables in *Vars*. For example, for a robot with 1D Line dynamics from Tab. 3, $State \equiv (\{x, v, a, o\}, \{o \in \{-1, 1\}\})$.

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

An actor's control is a hybrid program: $Ctrl \equiv \alpha$, defined as a sequence of operators from Tab. 2 over variables in *Vars* and *Ports*. *Ctrl* describes computations executed by the actor. For example, statements 6–9 are a prototype of robot's *Ctrl*, pending replacement of x_o with an appropriate port.

An actor's physics is a set of differential equations with an evolution domain constraint: $Phys \equiv \{x_i' = \theta_i \& F\}$ over variables in *Vars* and *Ports*. Each line in Tab. 3 is an instance of robot's *Phys*. The goal of separating physics from control is that the former is often shared among many model variants, while the latter may be more specific the variant's combination of concerns.

5.2 Component Interaction

We have introduced the acting entities of hybrid programs. Now we represent channels, through which interaction between hybrid actors happens, and the effects of these channels, like adding a bounded error to sensing, with *hybrid program connectors*:

DEFINITION 2. *Given* $\{HPA_i\}$, HP connector *HPC is a connector instance that is characterized by a tuple:*

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

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

Roles $Roles \equiv \{r\}$ distinguish different responsibilities of ports attached to a HP connector, such as a sender or a receiver. A mapping between roles and ports *RTP* associates each role with a port on an actor: $RTP \equiv Roles \to \bigcup HPA_i.Ports$. 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 *Roles* and *RTP* that define *what a connector is*, Trf defines *how it works*. Formally, Trf maps a set of actors and their attachments to a set of transformed actors:

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

Consider a simple immediate precise sensing (IPS) connector that senses the precise value of a variable and returns the result immediately. It has two roles: $Roles = \{Sense, Sensed\}$. Let actor a_1 use its port p_1 to sense the value of p_2 from actor a_2 through an IPS. The IPS transformation derives $\dot{a_1}$ from a_1 and $\dot{a_2}$ from a_2[4]:

$$IPS.\mathsf{Trf}(\{a_1, a_2\}, Roles, RTP) \equiv \{\dot{a_1}, \dot{a_2}\} \text{ s.t.} \quad (15)$$

$$\begin{aligned} \dot{a}_1.State = a_1.State, \quad & \dot{a}_2.State = a_2.State, \\ \dot{a}_1.Ports = a_1.Ports \setminus \{p_1\}, & \dot{a}_2.Ports = a_2.Ports \setminus \{p_2\}, \\ \dot{a}_1.Ctrl = a_1.Ctrl\ [p_1/p_2], \quad & \dot{a}_2.Ctrl = a_2.Ctrl, \\ \dot{a}_1.Phys = a_1.Phys, \quad & \dot{a}_2.Phys = a_2.Phys. \end{aligned}$$

IPS replaces the readings of variable p_1 in a_1 with readings of p_2 in $a_1.Ctrl$. A more complicated connector is an immediate bounded error sensing (IBES) connector that delivers a sensing result immediately with a bounded error $\delta \geq 0$. It has three roles: $\{Sense, Sensed, Bounds\}$. If we add another port p_3 to a_1 above and connect role *Bounds* to it ($RTP(Bounds) = p_3 \in a_1.Ports$), the IBES transformation[5] replaces p_1 in $a_1.Ctrl$ with a new variable λ that stores the sensing result with error δ:

$$IBES.\mathsf{Trf}(\{a_1, a_2\}, Roles, RTP) \equiv \{\dot{a_1}, \dot{a_2}\} \text{ s.t.} \quad (16)$$

$$\begin{aligned} \dot{a_1}.State.Vars &= a_1.State.Vars \cup \{\lambda, \delta\} \\ \dot{a_1}.State.Constr &= a_1.State.Constr \cup \{\delta \geq 0\}, \\ \dot{a_1}.Ports &= a_1.Ports \setminus \{p_1, p_3\}, \\ \dot{a_1}.Ctrl &= \lambda := *; ?p_2 - \delta \leq \lambda \leq p_2 + \delta; \\ &\qquad (a_1.Ctrl\ [p_1/\lambda, p_3/\delta]), \\ \dot{a_1}.Phys &= a_1.Phys. \end{aligned}$$

Analogously, we can define other connectors: an uncertain actuation connector for adding an uncertainty factor after changes in a control variable, a delayed precise sensing connector that stores a delayed value for one loop iteration, and a delayed bounded error connector that adds both a delay and a bounded error. Connectors can be applied to a single component by connecting all roles to the component's ports: for example, to model robot's imprecise information about its own location, IBES's *Sense* and *Sensed* have to be associated with the same port.

By invoking Trf of each connector, we can reduce a set of actors and connectors to a set of actors without connectors. We will call this the Transform Connectors (TC)

[4]We use $\alpha[a/b]$ to mean substitution of a for b in HP α.
[5]IBES handles $\dot{a_2}$ the same way as IPS in (15).

procedure:$\{HPA\} \times \{HPC\} \to \{HPA\}$. TC transforms and removes all connectors in a non-deterministic order. In theory, a different order of transformation may result in semantically different programs. Nevertheless, in practice we did not find cases where the meaning would depend on the transformation order. We proceed under the assumption that there is a single set of components that TC returns.

So, on the one hand, HP connectors represent information exchange between actors, such as sensing. On the other hand, HP connectors encapsulate common HP transformations, which can be used to obtain new model variants in a disciplined manner, e.g., by replacing IPS with IBES. HP connectors do not handle composition: we separate composition from interaction between actors. In the next subsection we introduce the means to create composite actors.

5.3 Component Composition

To enable automated generation of HPs, we need to bridge the gap between HP actors and hybrid programs. To this end, we will compose the actors until there is a single mega-actor, which then generates a HP. There are, however, several ways to compose actors. Therefore, we encapsulate a mechanism of composition in a *composer* (similar to component glue [3], director [23], and coordinator [7] in related work):

DEFINITION 3. *A* composer *CPR is a pair* (Compose, ToHP)*, where* Compose *is a function that maps several actors into one actor:* $\{HPA\} \to HPA$*, and* ToHP *is a function that maps HPA to a hybrid program.*

A composer implements a method of creating aggregate actors with Compose until the system is represented by a single actor, which is then converted into a hybrid program using ToHP. In the general case Compose can be arbitrarily complicated and is out of this paper's scope. Instead, we focus on the *sequential composer SeqC* that is implicitly used throughout all collision avoidance models in [18]. This composer orders actors' executions in a given sequence:

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

$$\begin{aligned} \dot{a}.State &= a_1.State \cup \cdots \cup a_n.State, \\ \dot{a}.Ports &= a_1.Ports \cup \cdots \cup a_n.Ports, \\ \dot{a}.Ctrl &= a_1.Ctrl; \ldots; a_n.Ctrl, \\ \dot{a}.Phys &= \{a_1.Phys, \ldots, a_n.Phys\}. \end{aligned} \tag{17}$$

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

$$SeqC.\mathsf{ToHP}(a) \equiv (a.Ctrl; a.Phys)^* \tag{18}$$

We envision other kinds of composers as well; for example, a non-deterministic composer with $\cup$, an alternating composer with a flag, or a truly parallel composer[6]. Nevertheless, *SeqC* is sufficient for the purposes of this work.

Now that we defined actors, their interaction, and their composition, we are ready to introduce *HP architectural views* that represent hybrid programs:

DEFINITION 4. *A* HP architectural view *HPV is a tuple* $(\{HPA\}, \{HPC\}, CPR)$.

[6]A parallel composition operator in dL is work in progress.

Thus, a HP architectural view contains actors, which interact through connectors, and a composer. To obtain a hybrid program α from HPV, the first step is to transforms components using connectors with the TC procedure. Next, *HPC* compose the actors to form a single *HPA* and generate a HP. In other words:

DEFINITION 5. *A HP architectural view HPV* represents *hybrid program* α*, denoted* $\alpha = \mathcal{R}^V_{HP}(HPV)$*, if*

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

This definition establishes an algorithm to obtain a HP from its architectural view. Now a HP can be edited at a higher level by manipulating actors and connectors. Notice that *State* does not yet contribute to the hybrid program since it is a logical, not operational, property like *Ctrl* or *Phys*. The remaining step is enabling logical specification over HP views.

5.4 Architectural DDL Formulas

As discussed in Sec. 4.2, dL formulas may incorporate several hybrid programs. Hence just embedding a logical specification in a view would limit us to simple logical formulas. Our goal is to build a logical specification layer on top of HP views, making it possible to reuse a HP view in several variants that target different properties. To represent a broad variety of formulas, including (3), we define an *architectural dL formula* the following way:

DEFINITION 6. *An* architectural dL formula *over HP views* $HPV_1, \ldots, HPV_n$ *is a dL formula over variables from these views* $Vars_1 \cup \cdots \cup Vars_n$*, parametric terms* $Constr_1, \ldots, Constr_n$*, and hybrid programs* $HPV_1, \ldots, HPV_n$.

Given several views, one can express a system property in a formula that combines the views. To obtain a plain dL formula from the architectural dL formula, one needs to replace $Constr_i$ with $\mathsf{Compose}(\mathsf{TC}(HPV_i)).State.Constr$ and replace HP view references HPV_i with their generated code $\mathcal{R}^V_{HP}(HPV_i)$.

With the introduced formulas, Eq. 3 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*. Other conditions like *RobotFar*, *ObstFar*, and *Safe* will need to be specified without referencing the views since these conditions are specific to the property. Since views have disjoint state spaces, extra statements are necessary to relate the state of *RobotObst* after finishing and the state of *DetailedObst* before starting.

To conclude this section, architectural abstractions (actors, connectors, composers, and formulas) represent hybrid programs and dL formulas. We described an algorithm to transform an architectural dL formula to a normal dL formula. The next section illustrates how these abstractions benefit CPS modeling.

6. SUPPORTING HP MODELING

This section demonstrates how architectural models support analysis and reuse of hybrid programs.

6.1 Information Flow Analysis

We assume now that every model variant is encoded as an architectural dL formula with at least one HPV. We

stated in Sec. 4.4 that the goal is to analyze information flow at the component level. An input to the analysis is *HPV* before TC, Compose, or ToHP procedures are carried out. The output is whether the desired constraints on information flow were satisfied.

The first step is ensuring *information hiding among actors*: each actor's control and physics do not use other actors' state. This separation is embedded in Def. 1: an actor's control and physics programs should be formulas over *only State* and *Ports* of this component. This helps eliminate many model variants that would not be compatible, for instance, a controller that worked with straight line physics (Tab. 3) does not work with arcs because the set of control variables is different.

Separated actors can bridge their variable sets by adding connectors and exchanging data. For the *ISect* model in Sec. 4.4, we create actors R, O, and Int and adding connectors for robot to sense x_{int}, y_{int}, y_o, and V_{min}. Thus we indicate that y_o and V_{min} are the only variables that the robot senses from the obstacle, and that it cannot operate any others like a_o. This critical separation between *what is an obstacle is* and *what the robot knows about the obstacle* limits the implementation of *Ctrl* in the model to the realistically available variables.

Our architectural abstractions allow modelers to customize information flow constraints in Acme. For example, consider the issue of "cheating" by reading control variables right after they were set (Sec. 4.4). This can happen if a variant combines immediate sensing with *SeqC*. To detect this problem an extra constraint for view $\mathsf{TC}(HPV)$ can be defined:

$$\forall a_1 : HPA \mid \forall v \in a_1.State.Vars \mid \nexists a_2 : HPA \mid \\ v := \_ \in Sub(a_1.Ctrl) \wedge v \in Sub(a_2.Ctrl) \wedge \\ HPV.CPR = SeqC(\dots, a_1, \dots, a_2, \dots),$$

where $Sub(\varphi)$ is a set of all subformulas of φ and $\_$ is a wildcard for any formula. Satisfaction of this constraint, checked with a solver in Acme, guarantees that control variables are not read outside the actor in the same loop iteration.

6.2 Reuse with Architectural Types

To address the need for model fragment reuse (Sec. 4.3), we add types to HP actors and connectors. For simplicity, an *actor type* is a partially specified actor as in Def. 1. An actor can have an arbitrary number of types. Thus actor a satisfies types $a : \mathcal{A}$ and $a : \mathcal{B}$ if[7]:

$$\mathcal{A}.State \cup \mathcal{B}.State \subseteq a.State,$$
$$\mathcal{A}.Ports \cup \mathcal{B}.Ports \subseteq a.Ports,$$
$$\mathcal{A}.Phys, \mathcal{B}.Phys \subseteq a.Phys.$$

Type extension is equivalent to having both types: $a \in (\mathcal{A} \sqsubseteq \mathcal{B}) \equiv a \in \mathcal{A} \wedge a \in \mathcal{B}$. This simple approach enables powerful reuse. For example, notice that spiral dynamics (Tab. 3) is a more general case of arcs w/o spinning, so we extend the former with the latter: $\mathsf{ArcNoSpinDynT} \equiv \mathsf{SpiralDynT} \cup (\emptyset, \emptyset, \emptyset, \{w' = \frac{a}{r}\})$. Then SpiralDynT is reused every time an actor is declared with it or ArcNoSpinDynT.

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

A HP connector type determines the Trf function, which encapsulates modeler's expertise about common transformations, such as IPS or IBES. Instead of manually introducing new variables, constraining them, and weaving into the code, a modeler achieves this with a HP connector automatically. For example, in the *ISect* program, we can derive a variant with uncertain awareness of the obstacle's position by replacing IPS with IBES, doing it more reliably and easily than manually in hybrid program code.

Type-based reuse is sometimes at odds with information hiding: some actor fragments need to be reused while being accessible to all actors, such as timing patterns in Sec. 4.1. To strike a balance between reuse and hiding, we introduce a global actor GlobalHPA, represented by a system in Acme, into every model variant. This actor is a convenient exception to information hiding discussed in Sec. 6.1: its variables are visible to all other actors. To reuse timing patterns with types $LCT \equiv (\{t, \varepsilon\}, \{\varepsilon \geq 0\}), \emptyset, t := 0, \{t' = 1, t \leq \epsilon\})$ and $GCT \equiv (\{T\}, \emptyset), \emptyset, \emptyset, \{T' = 1\})$, we let $\mathsf{GlobalHPA} : LCT$ or $\mathsf{GlobalHPA} : GCT$ to ensure consistent timing without the need to create a connector to read t, T, or ε.

In summary, architectural types allow a modeler to specify and reuse common model fragments like physics and time, thereby organizing the variant space around these fragments.

7. EVALUATION

We evaluated the proposed architectural abstractions on an extended set of model variants from a significant robotic collision case study [19, 25]. Since the models were created *prior to and without consideration* of their componentization, these models are a reasonable evaluation target. We implemented a prototype tool as a plugin to AcmeStudio [24][8] and used it to create architectures, reuse, and analyze architectural models for 15 hybrid programs and 12 $\mathsf{d}\mathcal{L}$ formulas over these programs.

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

The information flow analysis found several violations of information hiding *between HP actors*. In several models a robot directly changed obstacle's position $x_o := *; y_o := *$. Although this intended to mean selection of an arbitrary obstacle, this specification is fragile and confusing: it does not work as intended when copied to another context, such as diamond modality $\langle\rangle$, since the robot can moving the obstacle away from itself to avoid a collision. Another violation occurred in an intersection scenario similar to *ISect*: the robot's control conflated sensing of an obstacle's position

[8]The tool and models can be downloaded at `www.cs.cmu.edu/~iruchkin/dist/hparch-cbse15.zip`.

and intersection's position, which shared one of coordinates. Hence, it was impossible to tell whether the robot uses obstacle sensor data or intersection awareness. This issue may affect implementations since the measurement of obstacle is approximate and delayed, while the intersection's position is often known and fixed.

Creating architectural models exposed information hiding violations *between hybrid programs* as well. In one multi-program $\mathsf{d}\mathcal{L}$ formula, two programs shared variables *odx* and *ody*, but gave them different meaning: the first HP used them as the obstacle's speeds, and the second one used them as the obstacle's unit direction vectors. This semantic mismatch would lead to implementation errors where the same variables could be interpreted differently. We also discovered that in (3) a robot in RobotObst factors in an assumption of the worst-case braking power about an obstacle in a DetailedObst, which does not actually hold: the obstacle can decelerate arbitrarily fast, making safety easier to attain in the model than in reality. In summary, although some of the discovered violations are not bugs per se, they may become those in the process of refinement and implementation, so their detection is beneficial to MDE.

We observed a limitation of our approach in the case study: large portions of similar hybrid code were "trapped" in the robot controllers, but we were unable to reuse them because 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 statecharts that encapsulate control algorithms, as is done in Sphinx [25].

8. RELATED WORK

The logical foundations of our work lie in the modeling and verification of hybrid systems. Modeling with hybrid automata [2] and hybrid programs [22] is sufficient to mathematically express a broad class of systems. Hybrid verification has two distinct trends: deductive verification (proofs) and reachability analysis (model checking). We directly support deductive verification with $\mathsf{d}\mathcal{L}$, which has been used in domains of aircraft, collision avoidance, and traffic control. Reachability verification algorithms, most prominently ones in the SpaceEx platform [12], use approximations to estimate the automata reachability set. In this paper, we do not contribute any additional verification techniques, but we overcome the engineering limitations that these modeling formalisms have when it comes to creating complex systems.

Sphinx [25] and HP refactorings [19] are approaches to abstracting and manipulating hybrid programs. Sphinx provides UML class diagrams and statecharts to describe hybrid program behavior. By fully conforming to the $\mathsf{d}\mathcal{L}$ metamodel, it is limited to mimic programs very closely without a possibility to separate fragments of a HP or to carry out information flow analysis, lacking connectors for sensing and actuation. HP refactorings describe HP transformations that are a priori correct or have known proof obligations. Unfortunately, so far such refactorings exist between only very few model variants. Our work instead handles reuse and analysis in many more cases even when the proof obligations are unknown.

The majority of component-based frameworks do not address the continuous dynamics and its correctness directly, implicitly delegating it to other models. Mechatronic UML [4] and BIP [3] provide a basis for verification of coordination and timing properties, but neither represent or reason about the differential evolution of a system directly. In particular, the details of continuous collision control in the BeBot case study [4] are abstracted out and assumed to be correct, instead of specifying and verifying it as in our approach. μ-Kevoree [11] focuses on dynamism to resolve runtime challenges, but not design-time variation. The simulation platform Ptolemy II [23] implements rich simulations by introducing a director and the computational model for each actor, but falls short in non-determinism and exhaustiveness of conclusions.

Recent work in Distributed Emergent Ensembles of Components (DEECo) [7] replaces typical system configurations with dynamic component assemblies defined by predicate-based membership. Communication of components in assemblies is indirectly decided by mapping between component states through coordinating state automata. DEECo forms a convenient foundation for implementation and deployment of dynamic components with time guarantees. Unfortunately, runtime dynamicity [8] does not alleviate the burdens of hybrid modeling and verification tackled in this paper for two reasons. First, assemblies bind and reuse holistic components, but not fragments of components or common component transformations. Second, the physical continuities are abstracted away like in the aforementioned frameworks, making it impossible to verify a system's physical behavior and limiting it to simulation. We introduced components abstractions that open the door to component-based reasoning in the face of continuous behaviors, making DEECo applications possible as well.

Work that is closest to ours extends software architecture elements with hybrid system elements. AcmeViews [5] augment Acme architecture description for CPS with explicit physical elements, like efforts and flows, in order to verify consistency among multiple views. AcmeViews however do not have semantics given in terms of a hybrid program and therefore lack automated generation, reuse, and continuous dynamics. Hybrid Annex for AADL [1] extends AADL with hybrid annotations that capture variables, invariants, and differential evolution and discrete jump behaviors. Hybrid Annex demonstrates the power of architectural modeling, although only partially supporting reuse of the variation space with component types; it does not support rich connectors, and so reasoning about transformation and information flow is hardly possible. Our work advances the described state of the art of hybrid modeling through high-level design, reuse, analysis, and generation of hybrid programs.

9. CONCLUSION

This paper presented architectural abstractions to capture hybrid programs and $\mathsf{d}\mathcal{L}$ formulas: architectural formulas express critical properties over several views, each consisting of HP actors, connectors, and a composer. Our approach provides formal guarantees for hybrid programs and supports engineering of model variants through reuse of model fragments and information flow analysis. The evaluation on a case study of robotic collision avoidance demonstrated the viability of architectural modeling for hybrid programs.

The implications of this work go beyond reuse and analayis of hybrid programs for theorem proving. This paper's link between architecture and hybrid models promises seamless model-driven engineering for cyber-physical systems. On the

one hand, compatibility with classic architectural concepts lets us take advantage of architectural results in synchronization, schedulability, and dynamic adaptation. On the other hand, we believe that the principles of this work are not specific to hybrid programs, and will allow us to bring the power of hybrid verification to the system level without isolating software reasoning from continuous physics.

Architectural abstractions for hybrid programs open several directions for future research. First, knowledge of architectural structure and properties can facilitate hybrid proving. Second, controller specification can be reused and analyzed further, potentially by relating hybrid programs to other control models like Simulink. Finally, another interesting direction is providing architectural analyses for hybrid models, e.g., taint analysis for security and error propagation analysis.

10. ACKNOWLEDGMENTS

The authors thank Stefan Mitsch, Nathan Fulton, and André Platzer for contributing their ideas, providing robot models, and giving feedback on this work.

This work was supported in part by the National Science Foundation under Grant CNS-0834701, the National Security Agency, and the U.S. Department of Defense 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. The Software Engineering Institute (SEI) is a Federally Funded Research and Development Center managed by Carnegie Mellon University. Any views, opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, the National Security Agency, United States Department of Defense, ASD(R&E), or the SEI.

11. REFERENCES

[1] E. Ahmad, B. R. Larson, S. C. Barrett, N. Zhan, and Y. Dong. Hybrid annex: An AADL extension for continuous behavior and cyber-physical interaction modeling. In *Proc. of HILT 2014*, pages 29–38, 2014.

[2] R. Alur, T. A. Henzinger, and H. Wong-toi. Symbolic analysis of hybrid systems. In *Proc. of CDC'97*, 1997.

[3] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *Proc. of IEEE SEFM*, pages 3–12. IEEE, 2006.

[4] S. Becker and C. Brenner. The MechatronicUML design method – process, syntax, and semantics. Technical Report TR-RI-14-337, 2014.

[5] A. Bhave, B. Krogh, D. Garlan, and B. Schmerl. View consistency in architectures for cyber-physical systems. In *Proc. of IEEE/ACM ICCPS'11*, pages 151 –160, Apr. 2011.

[6] S. Bouraine, T. Fraichard, and H. Salhi. Provably safe navigation for mobile robots with limited field-of-views in dynamic environments. *Autonomous Robots*, 32(3):267–283, Apr. 2012.

[7] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil. DEECO: An ensemble-based component system. In *Proc. of CBSE'13*, pages 81–90, New York, NY, USA, 2013.

[8] T. Bures, P. Hnetynka, and F. Plasil. Strengthening architectures of smart CPS by modeling them as runtime product-lines. In *Proc. of CBSE'14*, pages 91–96, New York, NY, USA, 2014. ACM.

[9] P. Derler, E. A. Lee, S. Tripakis, and M. Torngren. Cyber-physical system design contracts. In *Proc. of ACM/IEEE ICCPS'13*, pages 109–118. ACM, 2013.

[10] L. Fletcher et al. The MIT-cornell collision and why it happened. In *The DARPA Urban Challenge*, number 56 in STAR, pages 509–548. Springer, 2009.

[11] F. Fouquet, B. Morin, F. Fleurey, O. Barais, N. Plouzeau, and J.-M. Jezequel. A dynamic component model for cyber physical systems. In *Proc. of CBSE'12*, pages 135–144. ACM, 2012.

[12] G. Frehse et al. SpaceEx: Scalable verification of hybrid systems. In *Computer Aided Verification'11*, pages 379–395. Springer, 2011.

[13] D. Garlan, R. Monroe, and D. Wile. Acme: Architectural description of component-based systems. *Foundations of component-based systems*, pages 47–67, Jan. 2000.

[14] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 1987.

[15] Kristijan Macek, Dizan Alejandro Vasquez Govea, Thierry Fraichard, and Roland Siegwart. Towards safe vehicle navigation in dynamic urban scenarios. *Automatika*, 2009.

[16] E. A. Lee. CPS foundations. In *In Proc. of DAC'10*, pages 737–742, New York, NY, USA, 2010. ACM.

[17] J. A. Misener. Cooperative intersection collision avoidance system (CICAS): Signalized left turn assist and traffic signal adaptation. *PATH Research Report*, Mar. 2010.

[18] S. Mitsch, K. Ghorbal, and A. Platzer. On provably safe obstacle avoidance for autonomous robotic ground vehicles. In *Proc. of Robotics: Science and Systems*, 2013.

[19] S. Mitsch, J.-D. Quesel, and A. Platzer. Refactoring, refinement, and reasoning. In *Proc. of Formal Methods'14*, pages 481–496. Springer, 2014.

[20] M. Ozkaya and C. Kloukinas. Design-by-contract for reusable components and realizable architectures. In *Proc. of CBSE'14*, pages 129–138. ACM, 2014.

[21] A. Platzer. Differential dynamic logic for hybrid systems. *Journal of Automated Reasoning*, 41(2):143–189, 2008.

[22] A. Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, 2010.

[23] C. Ptolemaeus. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, Sept. 2013.

[24] B. Schmerl and D. Garlan. AcmeStudio: Supporting style-centered architecture development. In *Proc. of ICSE'04*, pages 704–705. IEEE, 2004.

[25] Stefan Mitsch, Grant Olney Passmore, and Andre Platzer. Collaborative verification-driven engineering of hybrid systems. *Mathematics in Computer Sc.*, 8(1):71–97, 2014.