Runtime Verification of Correct-by-Construction Driving Maneuvers

Alexander Kittelmann[1,2(✉)], Tobias Runge[1,2], Tabea Bordis[1,2], and Ina Schaefer[1,2]

[1] TU Braunschweig, Brunswick, Germany
[2] Karlsruhe Institute of Technology, Karlsruhe, Germany
{alexander.kittelmann,tobias.runge,tabea.bordis, ina.schaefer}@kit.edu

Abstract. Cyber-physical systems play an increasingly vital role in our everyday lives by leveraging technology to mitigate human error. These systems are inherently safety-critical, which requires the highest standards in quality assurance. Therefore, designing safe behaviors for these systems in a manageable fashion and maximizing trust early on by formally verifying them against a formal specification mandates a software engineering process that prioritizes appropriate abstractions in the early design phase. However, even if models are formally verified at design time, their appropriateness in the real world stills needs to be validated at runtime, as specifications are usually incomplete. In this work, we introduce a methodology for refining verified cyber-physical systems modeled by hybrid mode automata to executable source code amenable for runtime verification. In particular, we employ ArchiCorC, which lifts the *correctness-by-construction* paradigm for programs to component-based architectures, and comes with facilities for code generation. Subsequent simulations of the executable and verified maneuvers allow to validate their initial requirements in a diverse set of scenarios.

Keywords: Correctness-by-construction · Cyber-physical systems · Runtime verification

1 Introduction

Cyber-physical systems are ubiquitous in many products that we use in our daily lives, including avionic systems [45], automobiles [15], robotics [30], and even medical equipment [25]. To reduce development complexity, their safety-critical nature mandates sophisticated elicitation of requirements and formal reasoning techniques (e.g., formal verification) during the design stage. Model-based development is such an approach in modern software engineering, which allows to express cyber-physical systems in a way that makes their analysis, visualization, and simulation tractable. The goal is to start with an abstract behavioral model of a system that is already amenable to various analyses (e.g., scalable formal

verification) to identify conceptual design flaws as early as possible. Moreover, the typical model-based design process spans multiple layers. That is, a *refining process* is applied to add more details to the current model and to perform more complex analyses. Eventually, the model is refined to executable source code.

A challenge is that the initial set of requirements used for successfully verifying an abstract model's behavior can often not guarantee that the system behaves *completely safe* after deployment. That is, elicitated requirements are typically incomplete for real-world execution of cyber-physical systems. Therefore, it is of paramount importance to *validate* the executable model under real-world conditions. To be cost-effective, the validation process aims at inspecting completeness of requirements by *simulating* the modeled behavior using one of many data-driven validation techniques (i.e., run-time verification) [33]. Model parts that violate any safety concerns can then be localized and improved, resulting in an iterative development process. Examples of such simulation and analysis tools include MATLAB/SIMULINK [3], PTOLEMY II [41], and AADLSIM [8]. Most of these simulation frameworks focus on rich specification languages, problems orthogonal to functional safety (e.g., uncertainty or performance), and are tied to specific modeling languages.

An important challenge is the gap between design model and the actual implementation that is simulated. Most of the aforementioned approaches already work with detailed enough design models amenable to simulation (e.g., MATLAB/SIMULINK [3]). However, this puts a lot of burden onto the early development of such systems due to high complexity in the modeling phase. Moreover, formal verification (e.g., establishment of correctness proofs) becomes intractable for these models. In contrast, maximizing abstraction of the initial design model is necessary to make formal verification scalable. However, this requires to *add* details to the implementation model before simulation, which increases the chances to introduce new defects and invalidates verification and simulation results.

In this work, we address this challenge by presenting an abstract formalism to specify and verify behavior of cyber-physical systems, and then derive *correct-by-construction* implementations including facilities for monitoring for virtual simulation and run-time verification. In Fig. 1, we give a high-level overview of the key ingredients of our proposed verification and validation pipeline, which depicts an iterative and incremental development process consisting of four consecutive steps.

First, we propose to model and specify the intended behavior of cyber-physical systems with so-called *hybrid mode automata*. To verify these models and to generate correctness proofs, we translate them to differential dynamic logic ($d\mathcal{L}$) [40] and employ the interactive theorem prover KEYMAERA X [34]. Second, the modeled hybrid mode automata are then translated to a component-based architecture. In particular, we exploit ARCHICORC [22], which builds on top of CORC [44] (a tool suite for correct-by-construction programs) and enables a developer to establish a component-based *correct-by-construction architecture*. Third, the architecture is automatically translated to source code in a general-purpose programming language (e.g., JAVA or C++). Although the code genera-

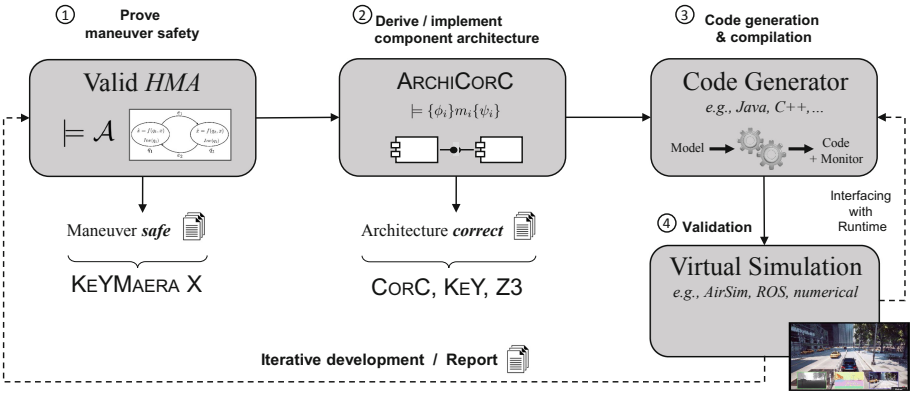


Fig. 1. Schematic overview of steps in the proposed verification and validation pipeline.

tion itself is unverified per se, each part of this process is verifiable when following best practices. That is, behaviors of components ideally follow the correctness-by-construction approach and checking validity of connections between components (horizontally and vertically) reduces to satisfiability checks based on their interface specifications. Moreover, the implementation also comes with facilities for monitoring, which are automatically generated from the formal specification.

Finally, the derived executable is verified on a set of scenarios (i.e., run-time verification by simulation). For the visual simulation environments, we integrate AirSim [47] by Microsoft, which is used for ground and air vehicles, and the robot operating system (ROS) together with Gazebo, which is a mature framework and visualization environment for robotic systems. Validation is supported by the automated monitoring of safety conditions. Monitoring functions are automatically generated from the hybrid mode automata as part of the third step. This allows to classify monitor violations as *passable* (i.e., violation does not have an impact on safety), *severe* (i.e., violation does have a high chance of impacting safety), or even *fatal* (e.g., vehicle collided with an object).

In summary, we make the following contributions.

- **Technique:** We propose a verification and validation pipeline for behaviors of cyber-physical systems that starts with an abstract model amenable to formal verification, and allows to derive a correct-by-construction and fully-functional implementation amenable to run-time verification.
- **Tool support:** We implemented our technique as part of the tool chain Skeditor by leveraging ArchiCorC, a model-based framework for developing correct-by-construction component-based architectures. As simulation environment, we integrated AirSim [47] by Microsoft.
- **Evaluation:** We performed an empirical study on five case studies in the context of automated driving, which differ in complexity and safety requirements. In our study, we verified at run-time that none of the case studies violated their safety requirements, which affirms that deriving fully-functional behaviors from highly-abstract verified models is feasible.

2 Workflow by Example

In this section, we aim at exemplifying our proposed workflow on an automatic distance control for road vehicles.

To formally reason about cyber-physical systems, they are often mathematically modeled by *hybrid systems* [1,7,16], which mix discrete and continuous behavior in a single formalism and abstract away from unnecessary details. Modeling approaches include *hybrid automata* [2] and *hybrid programs* [40].

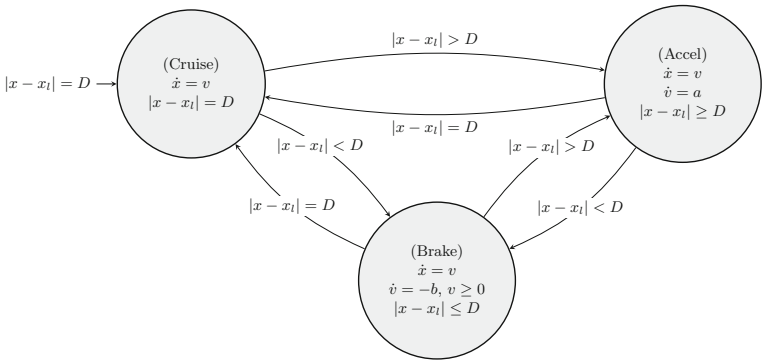


Fig. 2. Simplified hybrid system of a vehicle with automatic distance control.

In Fig. 2, we give an example of a simplified hybrid automaton representing the aforementioned automatic distance control with three possible states, namely Cruise, Accel, and Brake. Variables x and x_l define the current position on a straight line of both the host vehicle and the leading vehicle, and constant D represents the ideal distance between them. Moreover, variables v, a, and b define the current velocity, acceleration, and braking force, respectively. The goal of the distance control is to ensure that the distance between both vehicles remains approximately equal to D. The vehicle is in cruise mode when the distance to the leading vehicle is equal to distance D, which also means that acceleration a is set to zero (i.e., resulting in constant speed). In each control cycle, the current state evolves by applying the respective differential equations and is then evaluated, such that the automaton may transition into a different state. That is, if the leading vehicle increases or decreases the distance, the automaton switches to either state Accel or state Brake. The condition $v \geq 0$ in state Brake ensures that velocity v will not be allowed to become negative.

Step 1: Hybrid Mode Automaton. For modeling the behavior of cyber-physical systems, we propose *hybrid mode automata*, which is a customized combination of *hybrid automata* [16] and *mode automata* [28,29]. Hybrid mode automata consist of a number of discrete modes with guarded transitions between them. Although their expressiveness is equal to the expressiveness of hybrid automata, a key difference is that a *discrete* and simple program is projected

onto each mode, which is executed each time a mode is entered. This reduces the number of transitions by allowing more than a single computation step per mode. Furthermore, each mode is allowed to have exactly one system of ordinary differential equations that represents how variables change over time. In Listing 1, we present an excerpt of a possible variation of the distance control as a hybrid mode automaton in our own domain-specific language [20].

```
1  automaton Distance                    init Cruise
2    input variables:                    mode Accel:
3      x, x_l, D : ℝ                       controller:
4    output variables:                       if(D<100.0) {a := C} else
5      a : ℝ                                 {a := *; assume a ≤ A};
6                                          dynamics:
7    assumption:                             {x'=v, v'=a & v >= 0}
8      |x − x_l| ≥ D                       transitions:
9    guarantee:                              to Cruise when |x − x_l| = D;
10     |x − x_l| ≥ D                         to Brake when |x − x_l| < D;
11                                         mode Cruise: ...
```

Listing 1. Excerpt of a hybrid mode automaton for the distance control.

Besides transitions, each `mode` is comprised of a `controller` and `dynamics` part. The `controller` part lets a user define how variables may change (without elapsed time after entering the mode), while the dynamics part specifies the physical evolution of variables over time. Here, mode Accel assigns acceleration variable a the constant C if the distance to the leading vehicle is less than 100 m, and otherwise may assume any value for a that is less then the (constant) maximum acceleration A. Additionally, we allow to specify *assumptions* and *guarantees* of the complete automaton using first-order logical formulas (with real-valued arithmetic). Assumptions state the initial condition before executing the automaton, while guarantees state safety *invariants* that are not allowed to be violated during *any time* of the execution. Execution of such automata can be formally verified against the pair of assumptions and guarantees using a deductive calculus.

Step 2 and 3: Correct-by-Construction Architecture and Code Generation. After a maneuver represented by the hybrid mode automaton is modeled and verified, we use it to generate a component-based architecture amenable to runtime execution (e.g., for the purpose of simulation). As mentioned before, nondeterminstic assignments must be concretized manually during the implementation phase, which could potentially violate the safety guarantees. To propose a guideline for developers to implement nondeterminstic assignments in a correct-by-construction fashion, we employ the tool suite ArchiCorC [22]. At its core ArchiCorC [22] is a architecture modeling framework build on top of CorC [44]. CorC enables users to start with a specification (i.e., precondition and postcondition) and to develop correct-by-construction algorithms in an imperative language applying only sound refinement rules.

A hybrid mode automaton is translated to ArchiCorC's own domain-specific language consisting of required and provided interface definitions, as well as a textual representation for the atomic component that links provided methods to real implementations. In particular, a component resulting from a hybrid mode automaton provides an interface that includes (1) a *control method* representing a single execution of the automaton (i.e., mode switch and execution of the discrete program), (2) methods for nondeterminstic assignments that must be implemented by hand (ideally using CorC), and (3) a method for monitoring the current state. The following interfaces in ArchiCorC's interface language exemplify this translation for the automatic headway control.

```
archicorc_interface IHeadwayReq {
  // Parameters
  double D;
  // Input variables
  double x, x_l;
}
```

```
archicorc_interface IHeadwayProv {
  // Output variables
  double a;
  // State ID
  int state_id;

  //@ requires |x − x_l| ≥ D;
  //@ ensures |x − x_l| ≥ D;
  void ctrlStep(void);

  // Resolve nondet. assignment
  //@ requires |x − x_l| > D ∧ D ≥ 100;
  //@ ensures |x − x_l| > D ∧ D ≥ 100 ∧ a ≤ A;
  double a_1();

  // Monitoring
  bool monitorSatisfied(State prior);
}
```

As illustrated above, key element of the provided interface is the `ctrlStep` method that advances the state each cycle. Automated code generated for the `ctrlStep` methods resembles the structure of the corresponding hybrid mode automaton. We translate the corresponding hybrid mode automaton to a *logical formula*, which is then used as postcondition, while using the hybrid mode automaton's assumption and environmental conditions as precondition. Finally, we add a method to the component for *monitoring* whether any control actions violate the original (and verified) controller model. Input argument `State` is a simple structure used as shorthand for the collection of input and output variables. We discuss the generation of monitor code in the next section in more detail.

Step 4: Simulation and Monitoring. Finally, the generated implementation is executed in a simulation environment to validate whether the verified maneuver is appropriate in practice. To rule our any defects besides an insufficient specification, it is important to ensure that the executable implementation adheres to the same correctness guarantees as the modeled and verified controller. For this, we apply runtime monitoring that reports on violations during runtime with respect to the modeled controller. On a higher level, execution of the hybrid mode automaton is split into executing the modes and transitioning between them. In general, this results in disjunctions of modes including infor-

mation about when a mode can be executed (i.e., using the guards of incoming transitions per mode). Our automatic headway controller results in the disjunction

$$\Big((|x - xl| > D \wedge \mathsf{Accel}) \vee (|x - xl| = D \wedge \mathsf{Cruise}) \vee (|x - xl| < D \wedge \mathsf{Brake})\Big).$$

3 Modeling and Verifying Maneuvers with Hybrid Mode Automata

In this section, we formalize *hybrid mode automata*, which are used for modeling, specifying, and verifying behaviors of cyber-physical systems. In Sect. 3.1, we present syntax and semantics of hybrid mode automata. In Sect. 3.2, we present how to translate hybrid mode automata to *differential dynamic logic* for proving their correctness against a specification.

3.1 Formalization of Hybrid Mode Automata

As presented in the previous section, we propose to model the behavior of cyber-physical systems by means of *hybrid mode automata*. We first give a definition of the syntax of hybrid mode automata and explain specific parts of it afterwards.

Definition 1 (Syntax of Hybrid Mode Automata ($\mathcal{HMA}$)). *A* hybrid mode automaton $\mathcal{A}$ *is a tuple* $\langle Q, q^0, V^{\mathsf{in}}, V^{\mathsf{out}}, \mathsf{Trans}, \mathsf{Dyn}, \mathsf{Ctrl}\rangle$ *where:*

- Q *is a finite set of modes,*
- $q^0 \in Q$ *is the initial mode,*
- V^{in} *and* V^{out} *are sets of input and output variables in* $\mathbb{R}$*, respectively. We require that* $V^{\mathsf{in}} \cap V^{\mathsf{out}} = \emptyset$*,*
- $\mathsf{Trans} \subseteq Q \times G(V) \times Q$ *is the set of transitions, which are labeled by a first-order logical formula* G *over the input and output variables. We use the notation* $q \xrightarrow{g} q'$ *for* $(q, g, q') \in \mathsf{Trans}$*,*
- $\mathsf{Dyn} : Q \to \mathsf{ODE}$ *maps a mode to a (possibly empty) system of ordinary differential equations,*
- $\mathsf{Ctrl} : Q \to \mathsf{HMAL}_{\mathsf{disc}}$ *maps a mode to the discrete control part.*

The two elements of hybrid mode automata that need further explanation are the maps Dyn and Ctrl. As described in Sect. 2, each mode is associated with a set of ordinary differential equations and a sequence of *discrete computation steps* (e.g., assigning a value to a specific variable). Semantically, the discrete computations are executed sequentially and instantaneously (i.e., without elapsed time) after a mode is entered. Afterwards, the differential system runs for indefinite amount of time and lets specific variables evolve accordingly. In particular, to model discrete computations and ODEs, we adopt a simplified version of the syntax and semantics of *hybrid programs* provided by Platzer [40].[1]

[1] Further information on the syntax and semantics of hybrid programs beyond our application in this work can be found in A. Platzer's textbook *Logical Foundations of Cyber-Physical Systems* [40].

We denote by ODE the set of differential systems, where an element $ode \in$ ODE has the following syntax:

$$ode \equiv x_1' = f_1(x_1), \ldots, x_n' = f_n(x_n) \,\&\, H. \tag{1}$$

Here, $x_1, \ldots, x_n$ are variables that change over time by equations $f_1, \ldots, f_n$, and H is an *evolution constraint* in first-order logic that restricts the maximum evolution of these variables. For example, the ODE $\left[v' = -2 \& v \geq 0\right]$ decrements variable v by 2 over an arbitrary amount of time but only as long as v remains non-negative.

The language we use for implementing discrete computations in modes will be denoted HMAL (for *Hybrid Mode Automata Language*) and is a subset of hybrid programs (excluding nondeterministic repetitions and nondeterministic choice). We provide the following syntax for HMAL.

Definition 2 (Syntax of HMAL). *The* syntax *of language* HMAL *is defined by the following grammar, where S_1 and S_2 are programs of* HMAL*, $x, x_1, \ldots, x_n$ are real-valued variables, θ is a term, P, Q, and H are first-order logical formulas in real arithmetic, and $x' = f(x)$ is a system of ODEs:*

$$\begin{aligned} S_1, S_2 ::= {} & S_1; S_2 \mid x := \theta \mid \textbf{havoc}\ x_1, \ldots, x_n \mid \textbf{assume}\ H \mid \textbf{skip} \\ & \mid \textbf{if}(H)\{S_1\}\textbf{else}\{S_2\} \mid \textbf{assert}\ H \end{aligned}$$

Language HMAL consists of seven constructs. *Sequential composition* first runs the program defined by S_1 and then the program defined by S_2. *Discrete assignment* assigns a value of term Θ to variable x. *Nondeterministic assignment* represented by the keyword **havoc** $(\cdot)$ assigns arbitrary real numbers to variables $x_1, \ldots, x_n$. These assignments must be concretized when deriving an executable implementation. **assume** H is used to check that a particular formula holds at the respective position. **skip** is shorthand for **assume** true. The selection statement **if**(H) $\{S_1\}$ **else** $\{S_2\}$ runs program S_1 if condition H holds, and runs program S_2 otherwise. Finally, an *assertion* is similar to an assumption, where the checkable condition H valuates to either true or false depending on the current state. A violation of H will not just abort the current run, but the program will transition to a designated *error state*, which itself does not have outgoing transitions. This way, we mark violations of H explicitly as erroneous behavior.

After presenting the syntax of both HMAL and ODEs, we present their semantics next. Execution semantics of HMAL and ODEs are each based on the transitioning between *states*. In particular, Let $\mathbb{R}$ represent the set of real numbers and let V denote the set of real-valued variables. A state is a function σ from V to $\mathbb{R}$, i.e., $\sigma : V \to \mathbb{R}$.

Definition 3 (Semantics of HMAL and ODEs). *Let V be a finite set of real-valued variables and let Σ be the set of all possible states. The* semantics *of a program $S \in$ HMAL leads to the following denotational definition of the transition relation $[\![S]\!]_{\mathsf{HMAL}}, [\![S]\!]_{\mathsf{ODE}} \subseteq \Sigma \times \Sigma$, where $\sigma, \sigma' \in \Sigma$ represent the initial and final state, respectively, and σ^{error} is the error state:*

- $[\![S\,;\,S']\!]_{\mathsf{HMAL}} = \{(\sigma,\sigma') \mid (\sigma,\sigma_{im}) \in [\![S]\!]_{\mathsf{HMAL}}, (\sigma_{im},\sigma') \in [\![S']\!]_{\mathsf{HMAL}}\}$ *with intermediate state* σ_{im},
- $(\sigma,\sigma') \in [\![x := \Theta]\!]_{\mathsf{HMAL}}$ *iff* $\sigma'(x) = \mathtt{eval}(\Theta,\sigma)$ *and* $\forall y \in V$ *with* $x \neq y$ *it follows that* $\sigma(x) = \sigma'(y)$,
- $(\sigma,\sigma') \in [\![\textbf{havoc}\ x_1,\ldots,x_n]\!]_{\mathsf{HMAL}}$ *iff* $\forall x \in \{x_1,\ldots,x_n\}$, $\forall y \in V$ *with* $x \neq y$ *it follows that* $\sigma(y) = \sigma'(y)$,
- $(\sigma,\sigma') \in [\![\textbf{assume}\ H]\!]_{\mathsf{HMAL}}$ *iff* $\sigma = \sigma'$ *and the assignment of variables in state* σ *satisfies formula* H *(i.e.,* $\sigma \models H$*)*,
- $(\sigma,\sigma') \in [\![\textbf{if}(H)\{S_1\}\textbf{else}\{S_2\}]\!]_{\mathsf{HMAL}}$ *iff* $(\sigma,\sigma') \in [\![\textbf{assume}\ H; S_1]\!]_{\mathsf{HMAL}} \cup [\![\textbf{assume}\ \neg H; S_2]\!]_{\mathsf{HMAL}}$,
- $\Big[(\sigma,\sigma^{\mathsf{error}}) \in [\![\textbf{assert}\ H]\!]_{\mathsf{HMAL}}$ *iff* $(\sigma,\sigma) \in [\![\textbf{assume}\ \neg H]\!]_{\mathsf{HMAL}}\Big]$ *and* $\Big[(\sigma,\sigma') \in [\![\textbf{assert}\ H]\!]_{\mathsf{HMAL}}$ *iff* $(\sigma,\sigma') \in [\![\textbf{assume}\ H]\!]_{\mathsf{HMAL}}\Big]$,
- *For ODEs:* $(\sigma,\sigma') \in [\![x' = f(x)\,\&\,H]\!]_{\mathsf{ODE}}$ *iff* $\gamma : [0,r] \to \Sigma$ *is a solution of the ODE* $x' = f(x)$ *with* $\gamma(0) = \sigma$, $\gamma(r) = \sigma'$, *and each state in between* $\gamma(0)$ *and* $\gamma(r)$ *satisfies formula* H *with respect to differential equation* $x' = f(x)$ *(i.e.,* $\gamma \models x' = f(x)\,\&\,H$*).*

Both constructs, language HMAL and the definition of ODEs, are inspired by the syntax and semantics of hybrid programs [37–40]. The language of hybrid programs itself is only a simple nondeterministic programming language with support for differential equations. The reason to closely follow hybrid programs is twofold. First, hybrid programs provide a very simple programming model, which is expressively sufficient for us to describe the intended controller logic at this design stage. However, hybrid programs additionally provide means for nondeterministic repetition and nondeterministic choice, which we eliminated. The reason for the former is that a mode only performs one execution per time step before it transitions, which makes repetition unnecessary. The reason for the latter is that nondeterministic choice needs to be resolved when deriving a concrete implementation. We restrict language HMAL to nondeterministic assignment only, which is simpler to resolve. The second reason is that we aim at leveraging $d\mathcal{L}$ [37–40] for verifying hybrid mode automata. As $d\mathcal{L}$ is defined over hybrid programs, it is natural to only make some necessary modifications. Finally, we can give a definition of the execution semantics of hybrid mode automata.

Definition 4 (Execution Semantics of Hybrid Mode Automata). *Let* σ_i^{in} *and* σ_i^{out} *denote valuations of variables in* V*. A* valid run *of a hybrid mode automaton* $\mathcal{A} = \langle Q, q^0, V^{\mathsf{in}}, V^{\mathsf{out}}, \mathsf{Trans}, \mathsf{Dyn}, \mathsf{Ctrl}\rangle$ *is a sequence of mode switches* $l_0,\ldots,l_k \in Q$ *of the form* $\mathsf{run}_{\mathcal{A}} = \langle\sigma_0^{\mathsf{in}}, l_0, \sigma_0^{\mathsf{out}}\rangle,\ldots,\langle\sigma_n^{\mathsf{in}}, l_n, \sigma_n^{\mathsf{out}}\rangle$ *such that*

- σ_i^{in} *and* σ_i^{out} *are input and output valuations of variables in* V *in mode* l_i,
- *for all* $k = 0,\ldots n$,

$$(\sigma_k^{\mathsf{in}},\sigma_k^{\mathsf{out}}) \in \{(\sigma^{\mathsf{in}},\sigma^{\mathsf{out}}) \mid (\sigma^{\mathsf{in}},\sigma^{\mathsf{im}}) \in [\![\mathsf{Ctrl}(l_k)]\!]_{\mathsf{HMAL}}, (\sigma^{\mathsf{im}},\sigma^{\mathsf{out}}) \in [\![\mathsf{Dyn}(l_k)]\!]_{\mathsf{ODE}}\},$$

– *the initial execution is performed after taking the first transition from initial mode* q^0*, such that* $\langle q^0, G, l_0\rangle \in \mathsf{Trans}$ *and* $\sigma_0^{\mathsf{in}} \models G$*,*
– *for each* $i = 0, \ldots, n-1$*,* $\langle \sigma_i^{\mathsf{in}}, l_i, \sigma_i^{\mathsf{out}}\rangle$ *is followed by* $\langle \sigma_{i+1}^{\mathsf{in}}, l_{i+1}, \sigma_{i+1}^{\mathsf{out}}\rangle$ *if and only if there exists a transition* $(l_i, G, l_{i+1}) \in \mathsf{Trans}$ *and* $\sigma_{i+1}^{\mathsf{in}} \models G$*.*

Execution of a hybrid mode automaton begins with taking a transition from the initial mode, and afterwards sequentially running the currently active mode's discrete program and ODEs. We assume that guards of outgoing transitions of a mode do not overlap. If no applicable outgoing transition exists, we assume a self-transition. Assuming a logical clock, this ensures that each cycle the discrete program of the currently active mode is executed, and that the dynamical systems evolves over time. That is, the execution semantics of a hybrid mode automaton resembles a trace semantics.

3.2 Verification Based on Differential Dynamic Logic

Differential dynamic logic ($d\mathcal{L}$) [37–40] is a *first-order modal logic* for specifying and proving safety properties of hybrid programs. That is, formulas in $d\mathcal{L}$ that do not contain modalities are classical first-order logical formulas with real arithmetic. Additionally, the modal operators $[\alpha]$ and $\langle\alpha\rangle$ for a hybrid program α express special reachability properties. Essentially, $d\mathcal{L}$ formula $[\alpha]\phi$ is *true* iff ϕ is true for all reachable states of α (i.e., upon complete execution of α), and $d\mathcal{L}$ formula $\langle\alpha\rangle\phi$ is *true* iff ϕ is true in some reachable state of α.

As language HMAL is defined with the semantics of hybrid programs in mind, we prove the correctness of hybrid mode automata for a given specification by translating them to $d\mathcal{L}$. For this, we define the translation function $\mathtt{trans}_{\mathcal{HMA}}$ that relates hybrid mode automata and hybrid programs. As depicted in Listing 1, a user defines valid assumptions Φ (i.e., precondition) and guarantees Ψ (i.e., postcondition) of the respective behavior in first-order logic with real-valued arithmetic. Assumptions must hold prior to executing the hybrid mode automaton, whereas guarantees are invariants that must hold at every real point in time during the continuous dynamics. The safety requirement in $d\mathcal{L}$ is then expressed as

$$\Phi \rightarrow [\mathtt{trans}_{\mathcal{HMA}}]\,\Psi \tag{2}$$

For example, the automatic distance control assumes initially that the difference in positions of host and leading vehicle is greater than the minimal allowed distance. Only then can it guarantee that the difference in positions will remain greater throughout the execution.

To prove the validity of $d\mathcal{L}$ formulas deductively, a sound set of axioms and proof rules (i.e., a deductive calculus) is required. The KeYmaera X theorem prover [13] implements such a calculus for $d\mathcal{L}$ and is built on top of a *small trusted kernel* written in Scala to also increase trust in the tool support itself. In Fig. 3, we give a schematic overview of the modeling and verification procedure. For brevity, we omit the definition of $\mathtt{trans}_{\mathcal{HMA}}$ that translates hybrid mode

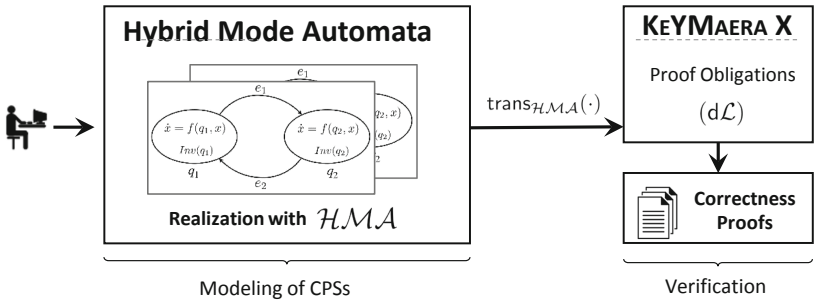


Fig. 3. Schematic overview of the verification process.

automata to $d\mathcal{L}$, as it is only of technical nature and can be found in previous work [20]. After we proved validity of a hybrid mode automaton with respect to the pair of assumptions and guarantees, the next goal is to derive a correct-by-construction implementation amenable to runtime verification. This way, we may rule out the introduction of new implementation defects and can focus on validating completeness of the specified safety guarantees.

4 Runtime Verification of $\mathcal{HMA}$ Models

In this section, how monitor conditions of the controller model are generated to validate whether the implementation behaves as intended.

4.1 Generating Monitor Conditions for Runtime Verification

In the previous sections, we described how to derive an implementation from an $\mathcal{HMA}$ model. Although models and abstraction are necessary to reduce complexity (e.g., for asserting correctness), any model has the tendency to deviate from the real world. It is therefore important to monitor at runtime that the properties proven with respect to the model are also ensured during execution. For instance, if at some point compliance of model and implementation cannot be guaranteed anymore, it is important to initiate fallback options that may still hinder catastrophic behavior (e.g., emergency braking).

To be able to assess whether our controller behaves as intended, we additionally generate executable monitor code for the controller in a dedicated method with signature **bool** `monitorSatisfied(State prior)`. Eventually, the monitor code can be used throughout runtime to report condition violations of the controller (i.e., deviations of controller model and implementation) that would otherwise be unnoticed. In particular, the generated code compares the current state of the controller with the previous state, which we explicitly provide as argument (the current state is obtained from the atomic component itself). If

$$
\begin{aligned}
\mathsf{trans}_{\mathsf{mon}}(\mathsf{Ctrl}(q)) &\doteq \mathsf{trans}_{\mathsf{mon}}(S) \wedge V = V^{\mathsf{post}} \qquad \textit{with}\, \mathsf{Ctrl}(q) \equiv S \\
\mathsf{trans}_{\mathsf{mon}}(S_1; S_1) &\doteq \mathsf{trans}_{\mathsf{mon}}(S_1) \wedge \mathsf{trans}_{\mathsf{mon}}(S_1) \\
\mathsf{trans}_{\mathsf{mon}}(x := \theta) &\doteq x^{\mathsf{post}} = \theta \\
\mathsf{trans}_{\mathsf{mon}}(\textbf{havoc}\ x_1, \ldots, x_n) &\doteq \mathsf{true} \\
\mathsf{trans}_{\mathsf{mon}}(\textbf{assume}\ H) &\doteq H[x \mapsto x^{\mathsf{post}}] \\
\mathsf{trans}_{\mathsf{mon}}(\textbf{assert}\ H) &\doteq H[x \mapsto x^{\mathsf{post}}] \\
\mathsf{trans}_{\mathsf{mon}}(\textbf{skip} H) &\doteq \mathsf{true} \\
\mathsf{trans}_{\mathsf{mon}}(\textbf{if}(H)\{S_1\}\textbf{else}\{S_2\}) &\doteq \Big(H[x \mapsto x^{\mathsf{post}}] \wedge \mathsf{trans}_{\mathsf{mon}}(S_1)\Big) \\
&\quad \vee \Big(\neg H[x \mapsto x^{\mathsf{post}}] \wedge \mathsf{trans}_{\mathsf{mon}}(S_2)\Big)
\end{aligned}
$$

Fig. 4. Translation of a mode to a logical formula for the monitoring condition.

the current state is not a valid poststate of the given prestate according to our modeled controller, the monitor is not satisfied. This is guaranteed by using ModelPlex [32] in our process that, given a $d\mathcal{L}$ model, generates monitor code for the controller model automatically.

In particular, the monitor conditions we generate are formulas that relate the current and next state with respect to the $\mathcal{HMA}$ model. That is, a monitor condition $\mathsf{mon}(v^{\mathsf{curr}}, v^{\mathsf{post}})$ compares the previous state comprised of variables v^{curr} with the next state comprised of variables v^{post}, and valuates to false whenever the program statements of the $\mathcal{HMA}$ model lead to a different model state than observed in the current state at runtime. In Fig. 4, we depict the rules for translating the program statements of a mode of a hybrid mode automaton to the logical counterpart for the monitoring condition. That is, if a particular mode is executed on the implementation level, the monitor condition for the respective controller program of the $\mathcal{HMA}$ model is based on this translation scheme. Importantly, variables that do not change by the modeled controller must evaluate to the same value in the current and next state. We represent this by the set of unmodified variables V in the beginning of the translation, which we explicitly add to the monitor condition.

4.2 Simulation

Simulation is an integral part of our verification and validation pipeline. There exist numerous simulation environments to choose from, which all come with advantages and drawbacks. Most popular in the research community, Gazebo [23] is a simulation platform that offers a modular design that allows developers to integrate different physics engines and to create complex robotic systems with arbitrary sensor models and simple 3D worlds. Furthermore, Gazebo maintains a close relationship with the robot operating system

(ROS) [24,42], one of the most prominent open-source frameworks for personal and industrial robotic systems. Therefore, Gazebo is typically used for simulating systems based on ROS modules. Although Gazebo comes with numerous features to increase realism in simulations, its rendering engine cannot compete with engines such as the Unreal engine or Unity, which makes it difficult to create visually-rich environments close to real-world scenarios.

To focus on visually-rich environments, AirSim [47] is a recent platform based on the Unreal engine that focuses primarily on automotive vehicles and flying drones. Most appealing, AirSim comes with pre-existing physical models of automotive vehicles and numerous 3D worlds (e.g., urban neighborhood or city), which saves development time and reduces the risk of introducing insufficient physical behavior. Especially, using independently-developed models and simulation environments is necessary in our evaluation to not invalidate empirical results.

In our validation and verification pipeline, we currently integrate both simulation platforms mentioned before. We primarily use Gazebo/ROS for robotic systems, and AirSim for automotive vehicles, such as applied in our evaluation for various driving maneuvers. Moreover, the modularity of our tool suite allows to extend the current set of simulation environments and to add new ones in a plug-and-play fashion.

5 Evaluation

The above sections raised two important research questions that we try to investigate by an empirical experiment and a qualitative assessment:

RQ-1: *To what degree do safety guarantees that are verified at design time hold at execution level?*

RQ-2: *What are lessons learned drawn from our experiences following the proposed verification and validation pipeline?*

With **RQ-1**, we investigate the *feasibility* of our verification and validation pipeline. In particular, we evaluate whether safety guarantees can be transferred from the verification model to the execution model following the guidelines of ArchiCorC and inspecting the generated monitor objects. Finally, with **RQ-2**, we discuss our experiences with the proposed verification and validation process.

We characterize our non-trivial case study of a road vehicle and the evaluated subject maneuvers in Sect. 5.1. In Sect. 5.2, we present results and discuss the research questions.

5.1 Case Studies and Setup

We aim at evaluating our pipeline in the context of automated driving maneuvers. In particular, we employ AirSim [47] as simulation environment to validate

to what extent correctness guarantees of verified $\mathcal{HMA}$ models can be transferred to implementation level. For automatic driving, the final goal is indeed to develop such $\mathcal{HMA}$ models for each maneuver from a catalog of basic driving maneuvers that can be safely applied in road traffic. The purpose of this evaluation, however, is a *proof of concept* of our verification and validation pipeline.

In total, we created five maneuvers including validation scenarios in AirSim to demonstrate the applicability of our verification and validation pipeline. All studied maneuvers are modeled as hybrid mode automata, verified, and eventually implemented in C++ supported by ArchiCorC. As ArchiCorC is limited with respect to floating-point reasoning, parts of the implementation remain unverified. However, such parts are already in a form that allows to verify them when reasoning with floating-point arithmetic becomes available. A short description of the maneuvers and their safety requirements is given in the following.

Explore World (Vehicle version). The goal of this maneuver is to randomly explore an area without colliding with any object. Due to the vehicle's kinematics, the turning circle must be considered explicitly. Safety goals are (1) respecting a maximum velocity ($v \leq v_{max}$), and (2) avoiding collision.

Safe Halt. The goal of this maneuver is to drive forward in a straight line and come to a halt if needed (i.e., either in front of an obstacle or a particular point) without collision or overstepping. Safety goals are again (1) respecting a maximum velocity ($v \leq v_{max}$), and (2) avoid collision.

Lane Keeping Assistance. This maneuver captures the lateral aspects of following a lane without deviating too far from the lane's center point. For this, the vehicle must drive on a lane with perceivable solid lane markings. Safety goal is to respect a maximum lane deviation ($y \leq y_{max}$).

Adaptive Cruise Control (Unoptimized and Optimized). This maneuver captures the longitudinal aspects of following a car while keeping a safe distance. We further split the *adaptive cruise control* maneuver into two versions, an optimized version and an unoptimized version. The optimized version resembles a more sophisticated controller on the modeled level without nondeterministic assignments, whereas the unoptimized version is simpler and delegates concretization of the acceleration part to the implementation level. Safety goal is to avoid collision with the leading vehicle.

As our evaluation aims at investigating whether our concept is feasible, we only built simple scenarios to test each driving maneuver in isolation. For the *Explore World* maneuver, we created a closed world with numerous static obstacles. For the *Safe Halt* maneuver, we used the same world, but placed the vehicle in front of an obstacle in a straight line. For the lane keeping assistance, we modeled a street and encoded the ground truth of the lane markings to eliminate sensor uncertainty. For the adaptive cruise control, we added a second leading vehicle in front of our vehicle that drives in a straight line.

Table 1. Simulation results.

Maneuver	Safety goal	Fail statistics				
		Non.-Det.	Passable	Severe	Failure rate (∅)	Sim. time
Explore world	$v \leq v_{max}$; no collision	Yes	100%	0%	1.07%	5 m
Safe halt	$v \leq v_{max}$; no collision	no	100%	0%	3.03%	ca. 6–12 s.
LKA	Lane deviation ($y \leq y_{max}$)	Yes	100%	0%	1.63%	30 s
ACC (Unoptimized)	No collision	Yes	100%	0%	2.72%	30 s
ACC (Optimized)	No collision	No	100%	0%	2.36%	30 s

5.2 Results and Insights

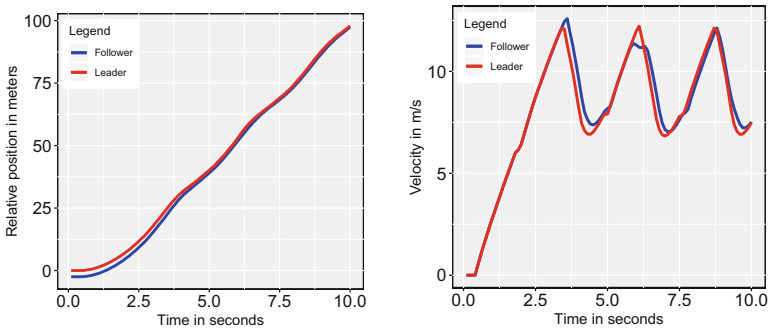
In the following, we share results obtained with our case studies. In particular, we are interested in usefulness and feasibility of our proposed approach. First, we investigate whether safety guarantees from verified $\mathcal{HMA}$ models can be transferred to the implementation level. Second, we discuss our experiences. The concrete models are contained as examples in the tool suite Skeditor [21], which integrates this work as part of its maneuver-centric development approach for cyber-physical systems.[2]

RQ-1: Validation and Safety Violations

In the following, we investigate the feasibility of our approach by simulating simulating each case study in AirSim and validating their correctness by employing the generated monitors. As mentioned before, each maneuver (i.e., $\mathcal{HMA}$ model) was successfully verified, which leads to the question whether correctness guarantees transfer to the implementation level. In Table 1, we summarize results from the performed experiments. In particular, for the fail statistics, we report three values. First, column *passable* reports on the percentage of monitor violations that we consider as acceptable. These stem from violations of nondeterministic assignments or other issues in the initial state, but do not violate the safety invariants. Second, column *severe* reports on the percentage of violations of the safety invariants. Finally, *failure rate* is the time spent in monitor-violating states with respect to the simulation time.

We observe that none of the five maneuvers violated any of the safety invariants during their simulation, which is why all reported violations are considered as *passable*. Furthermore, the failure rates are all low. However, we identified that oftentimes the initial state (i.e., a halted state with zero velocity for all five case studies) violates the monitor condition for the control part, while during movement the failure rate converges towards zero. Other times, the failure rate increases for a short amount of time. We assume that the conversion of arithmetic reals to floating-point precision results in sporadic problems.

[2] https://github.com/AlexanderKnueppel/Skeditor.



(a) Relative Positions (b) Velocities

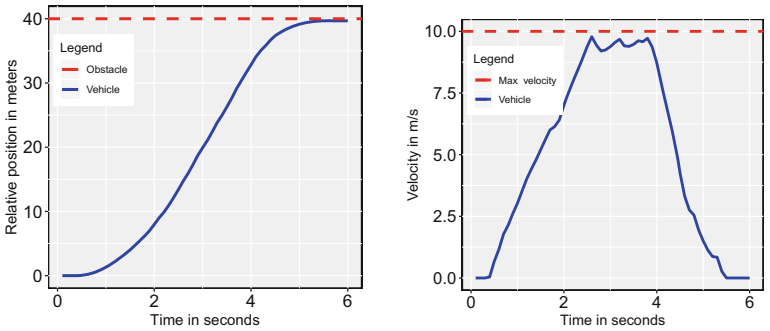
(c) Accelerations (d) Failure Rate

Fig. 5. Measurements for the *Adaptive Cruise Control* case study with the leading vehicle keeping a velocity between 8 and 12 m/s.

RQ-2: Experiences

In the following, we discuss experiences following the proposed development and simulation process using the *Optimized Adaptive Cruise Control* and *Safe Halt* case studies as two representative examples. In Fig. 5, we depict the position, velocity, acceleration, and failure rate over time of the *Optimized Adaptive Cruise Control* case study. The leading vehicle alternates between accelerating and braking as shown by the acceleration (lower left), and aims at keeping the velocity between 8 m/s and 12 m/s (upper right). The position (upper left) of the ego car shows that it tightly follows the leading car, but always keeps a minimal distance that is considered safe. The failure rate (lower right) shows the aforementioned issue, where the initial state violates the monitor condition due to imprecision, but afterwards the monitor condition is not violated anymore.

In contrast to the optimized version, the unoptimized version is simpler and provides nondeterminstic assignments for braking and acceleration. Moreover, the implementation switches accelerations only between the maximum braking force B and maximum acceleration force A. Although safety invariants still held at execution, we considered this behavior to be less convenient for human drivers.



(a) Relative Position (b) Velocity

(c) Acceleration (d) Failure Rate

Fig. 6. Measurements for the *Safe Halt* case study with a maximum velocity of 10 m/s.

We realized that optimizing this behavior is best addressed in the modeling phase, as we decided to add additional modes for more fine-grained control.

In this specific case, we considered that resolving nondeterministic choice at implementation level to realize the same optimization would be significantly more difficult for two reasons. First, modes and their implementation in HMAL provide a *local* view on the parameters involved (e.g., velocity), whereas the implementation is much more detailed and scattered. The unoptimized version sets the acceleration to a fixed constant. The optimized version, however, views the acceleration as a function depending on variables and parameters, such as velocity, distance to leading vehicle, and worst-case execution time. Considering optimality of such functions increases effort during the implementation phase. Second, verifying correctness of such optimization is also more promising in the modeling phase, as numerical optimization is best addressed with differential dynamic logic.

The second example illustrating the *safe halt* case study is shown in Fig. 6. The vehicle must keep a maximum velocity of 10 m/s and starts 40 m in front of an obstacle. Similar to the unoptimized adaptive cruise control case study, the vehicle only switches accelerations between the maximum braking force B and

maximum acceleration force A (upper left). The failure rate (lower right) shows an increase in monitor violations after four seconds. Although we did not exactly locate the cause for this issue, as both safety guarantees were not violated, we assume that sensor uncertainties may play a role.

While failure rates should not be dismissed, the above experiments increase our confidence in the proposed link from verified $\mathcal{HMA}$ models to actual implementations. All evaluated maneuvers transferred their safety guarantees to the simulation. Although the first two case studies required more implementation effort (e.g., due to interfacing with AirSim and processing sensor data), the automatic code generation and reuse of existing implementations allowed us to implement the final three case studies considerably faster. Moreover, three of the five maneuvers used non-deterministic assignments in their hybrid mode automata, which had to be resolved manually by us following the correctness-by-construction approach. This ensured that no new defects were introduced and that violations of monitor conditions can, in principle, always be traced back to an incomplete specification. We therefore believe that our verification and validation pipeline is particularly valuable for virtual prototyping of maneuvers and experimenting with their set of requirements.

6 Related Work

To enable the correctness-by-construction approach in our tool chain, we integrated CorC [44] into ArchiCorC. The reason for this decision are manifold. First, the feature set and future plans of CorC are sufficient for the purpose of ArchiCorC. In particular, both tools target object-oriented languages and the small kernel of CorC's theoretical foundation increases trust in its correctness. Second, CorC and ArchiCorC are based on the same technology stack, namely the *Eclipse Modeling Framework*. This leads to easier maintenance of the bridge between both tool suits and provides better user experience, as ArchiCorC artifacts and CorC programs can all be part of the same module. Third, CorC is well-maintained and actively developed, whereas most other frameworks in the field of stepwise program construction are not maintained anymore and also never reached a level of maturity, which we would consider sufficient enough for proper integration into ArchiCorC.

In spite of its young age, CorC was already extended in several directions. First, Runge et al. [43] extended CorC with a notion of information flow control-by-construction. Instead of checking confidentiality of data post-hoc by static information-flow analyses, information flow control-by-construction defines refinement rules for constructing secure programs. Second, Bordis et al. [6] introduced VarCorC, which is an offspring of CorC that focuses on correctness-by-construction for software product lines, instead of only considering monolithic programs. Finally, ArchiCorC, as presented in this chapter, lifts CorC to an architectural level by bundling correct-by-construction implementations in software components and providing means for code generation.

In the literature, many languages, techniques, frameworks, and tools exist to formally and semi-formally address the diverse set of challenges of industrial

cyber-physical systems development. These challenges include large system sizes, heterogeneity of connected modules, stakeholders from a multitude of disciplines, requirements elicitation, and also software evolution and maintenance themselves. Popular modeling languages include AADL [10,26,31,48,49], Modelica [9,14], Alloy [17,19], UML [4,11,18,27], and its variants SysML [12,35,36] and MARTE [5,46]. While all these languages greatly contributed to the research of system's design and analysis, their purpose is (1) to provide rich modeling facilities for almost all parts of a cyber-physical system, and (2) to eventually use these models as basis for real production code. Both goals make these languages inherently complex. For instance, AADL is used to model both hardware and software architectures of real-time embedded systems in great detail. In contrast, we aim to thrive for simplicity and focus on the functional modeling of maneuvers only to scale and leverage formal verification.

7 Conclusion

We presented a verification and validation pipeline for cyber-physical systems, where we explained how verified abstract maneuvers represented by hybrid mode automata can be refined in a correct-by-construction fashion to a component-based architecture amenable to simulation and runtime verification. In particular, we employed ArchiCorC, which (1) allows to manually resolve nondeterministic assignments relying on the correctness-by-construction approach for programs and (2) automatically generates code in a general-purpose programming language that can be considered correct-by-construction as well. To validate the executable maneuver at run-time, we added functionality for automatically generating monitor conditions based on the corresponding hybrid mode automaton. As the verified model is highly abstract, monitoring ensures that the safety obligations can be checked during runtime in case of hardware issues or environmental uncertainties.

We simulated the derived controller implementations in AirSim to inspect the appropriateness of the abstract model of a maneuver. The pursued and accomplished goal is that the link from a formal $\mathcal{HMA}$ model to execution is achievable in practice for non-trivial maneuvers. We have evaluated that all five case studies indeed transferred their correctness guarantees to the execution stage.

References

1. Alur, R.: Formal verification of hybrid systems. In: Proceedings of the International Conference on Embedded Software and Systems, pp. 273–278 (2011)
2. Alur, R., et al.: The algorithmic analysis of hybrid systems. Theoret. Comput. Sci. **138**(1), 3–34 (1995)
3. Angermann, A., Beuschel, M., Rau, M., Wohlfarth, U.: Matlab-simulink-stateflow. De Gruyter Oldenbourg (2020)

4. Bernardi, S., Gentile, U., Marrone, S., Merseguer, J., Nardone, R.: Security modelling and formal verification of survivability properties: application to cyber-physical systems. J. Syst. Softw. **171**, 110746 (2021)
5. Bernardi, S., Merseguer, J.: A UML profile for dependability analysis of real-time embedded systems. In: Proceedings of the International Workshop on Software and Performance (WOSP), pp. 115–124 (2007)
6. Bordis, T., Runge, T., Knüppel, A., Thüm, T., Schaefer, I.: Variational correctness-by-construction. In: Cordy, M., Acher, M., Beuche, D., Saake, G. (eds.) Proceedings of the International Working Conference on Variability Modelling of Software-Intensive Systems (VAMOS), pp. 7:1–7:9. ACM (2020). https://doi.org/10.1145/3377024.3377038
7. Branicky, M.S.: Introduction to hybrid systems. In: Hristu-Varsakelis, D., Levine, W.S. (eds.) Handbook of Networked and Embedded Control Systems. Control Engineering, pp. 91–116. Birkhäuser, Boston (2005). https://doi.org/10.1007/0-8176-4404-0_5
8. Buzdalov, D., Khoroshilov, A.: A discrete-event simulator for early validation of avionics systems. In: Proceedings of the Workshop on Architecture Centric Virtual Integration (ACVIP), p. 28 (2014)
9. Elmqvist, H., Mattsson, S.E., Otter, M.: Object-oriented and hybrid modeling in modelica. J. Eur. des systèmes automatisés **35**(4), 395–404 (2001)
10. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley, Boston (2012)
11. France, R., Evans, A., Lano, K., Rumpe, B.: The UML as a formal modeling notation. Comput. Stand. Interfaces **19**(7), 325–334 (1998)
12. Friedenthal, S., Moore, A., Steiner, R.: A Practical Guide to SysML: the Systems Modeling Language. Morgan Kaufmann, San Francisco (2014)
13. Fulton, N., Mitsch, S., Quesel, J.-D., Völp, M., Platzer, A.: KeYmaera X: an axiomatic tactical theorem prover for hybrid systems. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 527–538. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_36
14. Gómez, F.J., Aguilera, M.A., Olsen, S.H., Vanfretti, L.: Software requirements for interoperable and standard-based power system modeling tools. Simul. Model. Pract. Theory **103**, 102095 (2020)
15. Goswami, D., et al.: Challenges in automotive cyber-physical systems design, pp. 346–354 (2012). https://doi.org/10.1109/SAMOS.2012.6404199
16. Henzinger, T.A.: The theory of hybrid automata. In: Inan, M.K., Kurshan, R.P. (eds) Verification of Digital and Hybrid Systems. NATO ASI Series, vol. 170, pp. 265–292. Springer, Berlin, Heidelberg (2000). https://doi.org/10.1007/978-3-642-59615-5_13
17. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. (TOSEM) **11**(2), 256–290 (2002)
18. Jue, W., Song, Y., Wu, X., Dai, W.: A semi-formal requirement modeling pattern for designing industrial cyber-physical systems. In: Proceedings of the Annual Conference of the IEEE Industrial Electronics Society (IES), vol. 1, pp. 2883–2888. IEEE (2019)
19. Kang, E., Adepu, S., Jackson, D., Mathur, A.P.: Model-based security analysis of a water treatment system. In: Proceedings of the International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS), pp. 22–28. IEEE (2016)

20. Kittelmann, A.: Maneuver-centric formal engineering approach for cyber-physical systemsA. Ph.D. thesis, Braunschweig, Technische Universität Carolo-Wilhelmina zu Braunschweig (2022). https://doi.org/10.24355/dbbs.084-202204121019-0
21. Knüppel, A., Jatzkowski, I., Nolte, M., Thüm, T., Runge, T., Schaefer, I.: Skill-based verification of cyber-physical systems. In: FASE 2020. LNCS, vol. 12076, pp. 203–223. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45234-6_10
22. Knüppel, A., Runge, T., Schaefer, I.: Scaling correctness-by-construction. In: Margaria, T., Steffen, B. (eds.) ISoLA 2020. LNCS, vol. 12476, pp. 187–207. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61362-4_10
23. Koenig, N., Howard, A.: Design and use paradigms for gazebo, an open-source multi-robot simulator. In: Proceedings of the International Conference on Intelligent Robots and Systems (IROS), vol. 3, pp. 2149–2154. IEEE (2004)
24. Koubaa, A.: Robot Operating System (ROS). The Complete Reference (Volume 1) SCI, vol. 625. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-26054-9
25. Lee, I., et al.: Challenges and research directions in medical cyber-physical systems. Proc. IEEE **100**(1), 75–90 (2012). https://doi.org/10.1109/JPROC.2011.2165270
26. Lin, Q., Adepu, S., Verwer, S., Mathur, A.: Tabor: a graphical model-based approach for anomaly detection in industrial control systems. In: Proceedings of the Asia Conference on Computer and Communications Security (ASIACCS), pp. 525–536 (2018)
27. Mancini, T., et al.: Parallel statistical model checking for safety verification in smart grids. In: Proceedings of the International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm), pp. 1–6. IEEE (2018)
28. Maraninchi, F., Rémond, Y.: Mode-automata: about modes and states for reactive systems. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 185–199. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0053571
29. Maraninchi, F., Rémond, Y.: Mode-automata: a new domain-specific construct for the development of safe critical systems. Sci. Comput. Program. **46**(3), 219–254 (2003)
30. Michniewicz, J., Reinhart, G.: Cyber-physical robotics - automated analysis, programming and configuration of robot cells based on cyber-physical-systems. Proc. Technol. **15**, 566–575 (2014). https://doi.org/10.1016/j.protcy.2014.09.017
31. Misson, H.A., Gonçalves, F.S., Becker, L.B.: Applying integrated formal methods on cps design. In: Proceedings of the Brazilian Symposium on Computing Systems Engineering (SBESC), pp. 1–8. IEEE (2019)
32. Mitsch, S., Platzer, A.: Modelplex: verified runtime validation of verified cyber-physical system models. Formal Methods Syst. Des. **49**(1), 33–74 (2016)
33. Müller, A., Mitsch, S., Retschitzegger, W., Schwinger, W.: Towards cps verification engineering. In: Proceedings of the International Conference on Information Integration and Web-Based Applications & Services, pp. 367–371. iiWAS 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3428757.3429146
34. Müller, A., Mitsch, S., Schwinger, W., Platzer, A.: A component-based hybrid systems verification and implementation tool in Keymaera x (tool demonstration). In: Chamberlain, R., Taha, W., Törngren, M. (eds.) CyPhy/WESE -2018. LNCS, vol. 11615, pp. 91–110. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-23703-5_5
35. Neghina, M., Zamfirescu, C.-B., Pierce, K.: Early-stage analysis of cyber-physical production systems through collaborative modelling. Softw. Syst. Model. **19**(3), 581–600 (2019). https://doi.org/10.1007/s10270-019-00753-w

36. Pagliari, L., Mirandola, R., Trubiani, C.: Engineering cyber-physical systems through performance-based modelling and analysis: a case study experience report. J. Softw. Evol. Process **32**(1), e2179 (2020)
37. Platzer, A.: Differential dynamic logic for hybrid systems. J. Autom. Reason. **41**(2), 143–189 (2008). https://doi.org/10.1007/s10817-008-9103-8
38. Platzer, A.: Logical Analysis of Hybrid Systems - Proving Theorems for Complex Dynamics. Springer, Cham (2010). https://doi.org/10.1007/978-3-642-14509-4
39. Platzer, A.: Logics of dynamical systems. In: Proceedings of the International Symposium on Logic in Computer Science (LICS), pp. 13–24. IEEE Computer Society (2012). https://doi.org/10.1109/LICS.2012.13
40. Platzer, A.: Logical Foundations of Cyber-physical Systems. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-63588-0
41. Ptolemaeus, C.: System Design, Modeling, and Simulation: Using Ptolemy II, vol. 1. Ptolemy.org Berkeley (2014)
42. Quigley, M., et al.: Ros: an open-source robot operating system. In: Procedings of the Workshop on Open Source Software, vol. 3, p. 5. Kobe, Japan (2009)
43. Runge, T., Knüppel, A., Thüm, T., Schaefer, I.: Lattice-based information flow control-by-construction for security-by-design, pp. 44–54. ACM (2020). https://doi.org/10.1145/3372020.3391565
44. Runge, T., Schaefer, I., Cleophas, L., Thüm, T., Kourie, D., Watson, B.W.: Tool support for correctness-by-construction. In: Hähnle, R., van der Aalst, W. (eds.) FASE 2019. LNCS, vol. 11424, pp. 25–42. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-16722-6_2
45. Sampigethaya, K., Poovendran, R.: Aviation cyber-physical systems: Foundations for future aircraft and air transport. Proc. IEEE **101**(8), 1834–1855 (2013). https://doi.org/10.1109/JPROC.2012.2235131
46. Seceleanu, C., et al.: Analyzing a wind turbine system: From simulation to formal verification. Sci. Comput. Program. **133**, 216–242 (2017)
47. Shah, S., Dey, D., Lovett, C., Kapoor, A.: AirSim: high-fidelity visual and physical simulation for autonomous vehicles. In: Hutter, M., Siegwart, R. (eds.) Field and Service Robotics. SPAR, vol. 5, pp. 621–635. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-67361-5_40
48. Zhang, L.: Specifying and modeling automotive cyber physical systems. In: Proceedings of the International Conference on Computational Science and Engineering (CSE), pp. 603–610. IEEE (2013)
49. Zhang, L.: Modeling large scale complex cyber physical control systems based on system of systems engineering approach. In: Proceedings of the International Conference on Automation and Computing (ICAC), pp. 55–60. IEEE (2014)