Runtime Verification of Correct-by-Construction Driving Maneuvers

Alexander Kittelmann\textsuperscript{1,2(✉)}, Tobias Runge\textsuperscript{1,2}, Tabea Bordis\textsuperscript{1,2}, and Ina Schaefer\textsuperscript{1,2}

\textsuperscript{1} TU Braunschweig, Brunswick, Germany
\textsuperscript{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, elicited 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 (dL) [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 ARCHI-CoR [22], which builds on top of CoR [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.

The text provides a detailed description of the verification and validation pipeline for behaviors of cyber-physical systems. It 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. The pipeline includes the following steps:

1. **Prove maneuver safety** using KeyMAERA X to prove maneuver safety.
2. **Derive / implement component architecture** using ARCHICORC to derive the component architecture.
3. **Code generation & compilation** using Code Generator to generate the code.
4. **Virtual Simulation** using tools like AirSim or ROS for simulation.
5. **Validation** and **Report** for final validation.

The text explains that each part of this process is verifiable when following best practices. 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. 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). The visual simulation environments integrate tools like AirSim [47] by Microsoft and the robot operating system (ROS) together with Gazebo. 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, the text makes 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 a 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 \[40\].

\[
\begin{align*}
\dot{x} &= v \\
| x - x_l | &= D \\
| x - x_l | &> D \\
\end{align*}
\]

\[
\begin{align*}
\dot{x} &= v \\
v &= a \\
| x - x_l | &= D \\
| x - x_l | &< D \\
| x - x_l | &> D \\
\end{align*}
\]

\[
\begin{align*}
\dot{x} &= v \\
v &= -b, v \geq 0 \\
| x - x_l | &\leq D \\
| x - x_l | &= D \\
| x - x_l | &< D \\
\end{align*}
\]

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
2 input variables: x, x_l, D : \mathbb{R}
3 output variables: a : \mathbb{R}
4 assumption: |x - x_l| \geq D
5 guarantee: |x - x_l| \geq D

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 than 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, nondeterministic assignments must be concretized manually during the implementation phase, which could potentially violate the safety guarantees. To propose a guideline for developers to implement nondeterministic 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 nondeterministic 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);
    //@ 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 out 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

\[
\left( (|x - x_l| > D \land \text{Accel}) \lor (|x - x_l| = D \land \text{Cruise}) \lor (|x - x_l| < D \land \text{Brake}) \right).
\]

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 (HMA)).** A hybrid mode automaton \( A \) is a tuple \((Q, q^0, V^{in}, V^{out}, \text{Trans}, \text{Dyn}, \text{Ctrl})\) 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 \( V^{in} \cap V^{out} = \emptyset \),
- \( \text{Trans} \subseteq Q \times G(V) \times Q \) 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 \((q,g,q') \in \text{Trans}, \)
- \( \text{Dyn} : Q \rightarrow \text{ODE} \) maps a mode to a (possibly empty) system of ordinary differential equations,
- \( \text{Ctrl} : Q \rightarrow \text{HMAL}_{\text{disc}} \) maps a mode to the discrete control part.

The two elements of hybrid mode automata that need further explanation are the maps \( \text{Dyn} \) and \( \text{Ctrl} \). 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 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:

\[
ode \equiv x'_1 = f_1(x_1), \ldots, x'_n = f_n(x_n) & H.
\]  

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 \( \left[ v' = -2v & v \geq 0 \right] \) 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, \( \theta \) is a term, \( P, Q \), and \( H \) are first-order logical formulas in real arithmetic, and \( x' = f(x) \) is a system of ODEs:

\[
S_1, S_2 ::= S_1; S_2 | x := \theta | \text{havoc } x_1, \ldots, x_n | \text{assume } H | \text{skip} | \text{if}(H)\{S_1\}\text{else}\{S_2\} | \text{assert } H
\]

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 \( \theta \) 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 \( (H) \{S_1\} \text{else} \{S_2\} \) 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 \( H \) evaluates 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 \( \sigma \) from \( V \) to \( \mathbb{R} \), i.e., \( \sigma : V \rightarrow \mathbb{R} \).

**Definition 3 (Semantics of HMAL and ODEs).** Let \( V \) be a finite set of real-valued variables and let \( \Sigma \) be the set of all possible states. The semantics of a program \( S \in \text{HMAL} \) leads to the following denotational definition of the transition relation \( \llbracket S \rrbracket_{\text{HMAL}}, \llbracket S \rrbracket_{\text{ODE}} \subseteq \Sigma \times \Sigma \), where \( \sigma, \sigma' \in \Sigma \) represent the initial and final state, respectively, and \( \sigma^{\text{error}} \) is the error state:
Let $\sigma_{im}$ denote valuations of variables in $V$. A valid run of a hybrid mode automaton $A = \langle Q, q^0, V_{in}, V_{out}, Trans, Dyn, Ctrl \rangle$ is a sequence of mode switches $l_0, \ldots, l_k \in Q$ of the form $\text{run}_A = \langle \sigma^i_0, l_0, \sigma^o_0; \ldots; \sigma^i_n, l_n, \sigma^o_n \rangle$ such that

- $\sigma^i_0$ and $\sigma^o_0$ are input and output valuations of variables in $V$ in mode $l_0$,
- for all $k = 0, \ldots, n$,

$$((\sigma^i_k, \sigma^o_k) | (\sigma^i_k, \sigma^o_k)) \in [S; S']_{\text{HMAL}}$$
the initial execution is performed after taking the first transition from initial mode \(q^0\), such that \(\langle q^0, G, l_0 \rangle \in \text{Trans} \) and \(\sigma^\text{in}_0 \models G\).

- for each \(i = 0, \ldots, n-1\), \(\langle \sigma^\text{in}_i, l_i, \sigma^\text{out}_i \rangle\) is followed by \(\langle \sigma^\text{in}_{i+1}, l_{i+1}, \sigma^\text{out}_{i+1} \rangle\) if and only if there exists a transition \((l_i, G, l_{i+1}) \in \text{Trans} \) and \(\sigma^\text{in}_{i+1} \models G\).

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 \(\alpha\) express special reachability properties. Essentially, \(d\mathcal{L}\) formula \([\alpha]\phi\) is true iff \(\phi\) is true for all reachable states of \(\alpha\) (i.e., upon complete execution of \(\alpha\)), and \(d\mathcal{L}\) formula \(\langle \alpha \rangle\phi\) is true iff \(\phi\) is true in some reachable state of \(\alpha\).

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 \(\text{trans}_{\text{HMA}}\) that relates hybrid mode automata and hybrid programs. As depicted in Listing 1, a user defines valid assumptions \(\Phi\) (i.e., precondition) and guarantees \(\Psi\) (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

\[
\Phi \rightarrow [\text{trans}_{\text{HMA}}]\Psi \tag{2}
\]

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 \textit{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 \(\text{trans}_{\text{HMA}}\) that translates hybrid mode
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

![Fig. 3. Schematic overview of the verification process.](image-url)
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 dLC 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 HMA model. That is, a monitor condition \(\text{mon}(v_{\text{curr}}, v_{\text{post}})\) compares the previous state comprised of variables \(v_{\text{curr}}\) with the next state comprised of variables \(v_{\text{post}}\), and valuates to false whenever the program statements of the 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 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 ($v \leq v_{\text{max}}$), 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 ($v \leq v_{\text{max}}$), 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 ($y \leq y_{\text{max}}$).

**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.

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

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 \(HMA\) 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., \(HMA\) 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 \(\text{passable}\) 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 \(\text{severe}\) reports on the percentage of violations of the safety invariants. Finally, \(\text{failure rate}\) 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 \(\text{passable}\). 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.
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 nondeterministic 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.
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 VARCHOC, 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 non-deterministic 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 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


