A Component-based Approach to Hybrid Systems Safety Verification*

Andreas Müller[1], Stefan Mitsch[1], Werner Retschitzegger[1], Wieland Schwinger[1], and André Platzer[2]

[1] Department of Cooperative Information Systems
Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria
{andreas.mueller,stefan.mitsch,wieland.schwinger,werner.retschitzegger}@jku.at
[2] Computer Science Department
Carnegie Mellon University, Pittsburgh PA 15213, USA
aplatzer@cs.cmu.edu

Abstract. We study a component-based approach to simplify the challenges of verifying large-scale hybrid systems. Component-based modeling can be used to split large models into partial models to reduce modeling complexity. Yet, verification results also need to transfer from components to composites. In this paper, we propose a component-based hybrid system verification approach that combines the advantages of component-based modeling (e. g., reduced model complexity) with the advantages of formal verification (e. g., guaranteed contract compliance). Our strategy is to decompose the system into components, verify their local safety individually and compose them to form an overall system that provably satisfies a global contract, without proving the whole system. We introduce the necessary formalism to define the structure and behavior of components and a technique how to compose components such that safety properties provably emerge from component safety.

Keywords: component-based development, hybrid systems, formal verification

1 Introduction

The hybrid dynamics of computation and physics in safety-critical cyber-physical systems (CPS), such as driver assistance systems, self-driving cars, autonomous robots, and airplanes, are almost impossible to get right without proper formal analysis. To enable this analysis, CPS are modeled using so called *hybrid system models*. At larger scales of realistic hybrid system models, formal verification of monolithic models becomes quite challenging. Therefore, component-based modeling approaches split large models into partial models, i. e., co-existing or interacting components (e. g., multiple airplanes in a collision avoidance maneuver). Even though this can lead to component-based models with improved

* Work partly funded by BMVIT grant FFG BRIDGE 838526, OeAD Marietta Blau grant ICM-2014-08600, FWF P28187-N31, and ERC PIOF-GA-2012-328378.

E. Ábrahám and M. Huisman (Eds.): IFM 2016, LNCS 9681, pp. 441–456, 2016.
DOI: 10.1007/978-3-319-33693-0_28

structure and reduced modeling complexity, component verification results do not always transfer to composite systems without appropriate care.

This paper generalizes our previous work [18], which was limited to traffic flow models (i. e., port conditions limited to maximum values, contracts limited to load restrictions, components limited to interfaces and predefined behavior), to a more generic approach to *make hybrid system theorem proving modular on a component level*. The approach exploits component contracts to compose *verified components* and their *safety proofs* to a verified CPS. Differential dynamic logic $\mathsf{d}\mathcal{L}$ [21,22], the hybrid systems specification and verification logic we are working with, is already compositional for each of its operators and, thus, a helpful basis for our approach. Reasoning in $\mathsf{d}\mathcal{L}$ splits models along the $\mathsf{d}\mathcal{L}$ operators into smaller pieces. In this paper, we build compositionality for a notion of components and interfaces on top of $\mathsf{d}\mathcal{L}$. We focus on modeling a system in terms of components that each capture only a part of the system's behavior (as opposed to monolithic models) and a way to compose components by connecting their interfaces (as opposed to basic program composition with $\mathsf{d}\mathcal{L}$ operators). Component-based hybrid systems verification is challenging because both local component behavior (e. g., decisions and motion of a robot) and inherently global phenomena (e. g., time) co-occur, because components can interact virtually (e. g., robots communicate) and physically (e. g., a robot manipulates an object), and because their interaction is subject to communication delays, measurement uncertainty, and actuation disturbance. Typically, our components are open systems [11], which are described and verified in isolation from other components, separated by interfaces with assumptions about the environment that provide guarantees about the behavior of components. If needed, they can be turned into a closed system [11] by including a model of a specific environment.

This paper focuses on (i) lossless and instantaneous interaction between components (allows uncertainty and delay in dedicated "ether" components, e. g., sense the speed of a car precisely without measurement error), (ii) components without physical entanglement (allows separated continuous dynamics, e. g., robots drive on their own, but do not push each other), and (iii) components without synchronized communication (parallel composition of continuous dynamics, simplification to any sequential interleaving for discrete dynamics, e. g., robots can sense their environment, but not negotiate with each other).

With this focus in mind, we define the structure and behavior of a notion of components and a technique how to compose components such that safety properties about the whole system emerge from component safety proofs (e. g., robots will not collide when staying in disjoint spatial regions). We illustrate our approach with a vehicle cruise control case study.

2 Preliminaries: Differential Dynamic Logic

For specifying and verifying correctness statements about hybrid systems, we use *differential dynamic logic* ($\mathsf{d}\mathcal{L}$) [21,22], which supports *hybrid programs* as a program notation for hybrid systems. $\mathsf{d}\mathcal{L}$ models can be verified using KeYmaera X [8],

which is open source and has been applied for verification of several case studies.[3] The syntax of hybrid programs is generated by the following EBNF grammar:

$$\alpha ::= \alpha;\beta \mid \alpha \cup \beta \mid \alpha^* \mid x := \theta \mid x := * \mid \{x_1' = \theta_1, \ldots, x_n' = \theta_n \ \&\ H\} \mid ?\phi \ .$$

The sequential composition $\alpha;\beta$ expresses that β starts after α finishes. The non-deterministic choice $\alpha \cup \beta$ follows either α or β. The non-deterministic repetition operator α^* repeats α zero or more times. Discrete assignment $x := \theta$ instantaneously assigns the value of the term θ to the variable x, while $x := *$ assigns an arbitrary value to x. $\{x' = \theta \ \&\ H\}$ describes a continuous evolution of x (x' denotes derivation with respect to time) within the evolution domain H. The test $?\phi$ checks that a condition expressed by ϕ holds, and aborts if it does not. A typical pattern $x := *; ?a \leq x \leq b$, which involves assignment and tests, is to limit the assignment of arbitrary values to known bounds.

To specify safety properties about hybrid programs, dL provides a modal operator $[\alpha]$. When ϕ is a dL formula describing a state and α is a hybrid program, then the dL formula $[\alpha]\phi$ expresses that all states reachable by α satisfy ϕ. The set of dL formulas relevant for this paper is generated by the following EBNF grammar (where $\sim \in \{<, \leq, =, \geq, >\}$ and θ_1, θ_2 are arithmetic expressions in $+, -, \cdot, /$ over the reals):

$$\phi ::= \theta_1 \sim \theta_2 \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \rightarrow \psi \mid \phi \leftrightarrow \psi \mid \forall x \phi \mid \exists x \phi \mid [\alpha]\phi \ .$$

Notation: Variables. In dL (and thus throughout the paper) all variables are real-valued. We use V to denote a set of variables. $FV(.)$ is used as an operator on terms, formulas and hybrid programs returning only their free variables, whereas $BV(.)$ is an operator returning only their bound variables.[4] Similarly, $V(.) = FV(.) \cup BV(.)$ returns all variables (free as well as bound).

Notation: Indices. Throughout this paper, subscript indices represent enumerations (e. g., x_i). Superscript indices are used to further specify the kinds of items described by the respective variables (e. g., v^{out} represents an *output* variable). If needed, a double (super- and subscript) one-letter index is used for double numeration (e. g., x_i^j represents element j of the vector x_i).

3 Modeling and Verification Steps

In this section we present the modeling and verification steps in our component-based verification approach (cf. Fig. 1). To illustrate the steps, we will use an example of a vehicle cruise control system, which consists of an actuator component adapting the vehicle speed according to a target speed chosen by a cruise control component. The vehicle moves *continuously*, while the control behavior is described by a *discrete* control part (e. g., choose velocity and acceleration). The goal is to keep the actual velocity in some range $[0, V]$, where V denotes a maximum velocity. Note that we model components fully symbolically, which means that each component represents actually a family of concrete components.

[3] cf. `http://symbolaris.com/info/KeYmaera.html`

[4] *Bound variables* of a hybrid program are all those that may potentially be written to, while *free variables* are all those that may potentially be read [23].

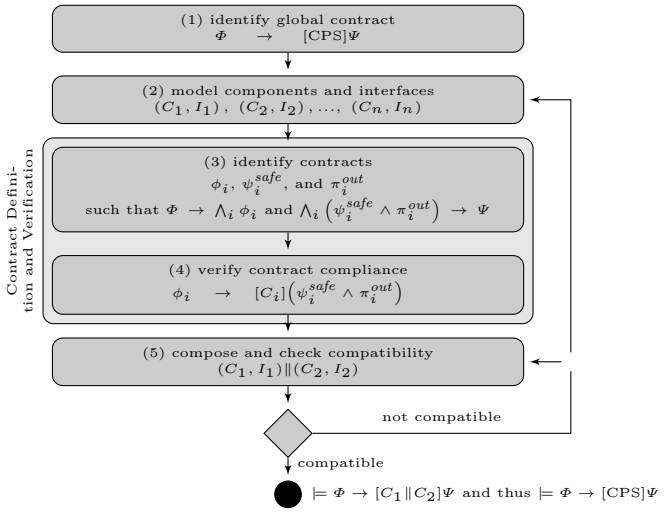


Fig. 1: Steps for component-based modeling and verification

The approach consists of the following steps:

(1) identify global contract: Before decomposing the system, it is important to learn what properties the system as a whole should fulfill (e. g., supported by domain experts). The global contract specifies the initial state of the whole system (Φ, e. g., initially the velocity is 0) as well as its overall safety property (Ψ, e. g., the velocity will stay in the desired range).

(2) model components and interfaces: Find recurring parts or natural splitting points for implementations (e. g., we split our cruise control system in a cruise controller and an actuator). The number of different components should be kept small, so that the verification effort remains low; still, there have to be sufficiently many components that can be instantiated to assemble the system. Modeling components and their interfaces is a manual effort (e. g., by modeling experts). A component has a behavior, while its interface defines public input ports and output ports, see Def. 2 and Def. 3 later.

(3) identify contracts: For each component and its interface, we identify initial states ϕ_i (e. g., initial target velocity is 0), a safety property ψ_i^{safe} (e. g., actual velocity does not exceed V), as well as an output contract π_i^{out} (e. g., target velocity is always in the desired range), see Def. 4 later. These properties have to be chosen such that the global contract follows by *refinement* or *dominance* [4]: $\Phi \rightarrow \bigwedge_i \phi_i$ and $\bigwedge_i(\psi_i^{safe} \wedge \pi_i^{out}) \rightarrow \Psi$.

(4) verify contract compliance: Verify that components satisfy their contracts formally, in our case (hybrid programs and $\mathsf{d}\mathcal{L}$), with KeYmaera X.

(5) compose and check compatibility: Construct the system by connecting component ports to compose verified components in parallel, see Def. 5 later. Any component can be instantiated multiple times in the whole system (e. g.,

instantiate maximum velocity parameters of a cruise control with actual values; connect the controller with the actuator). In order to transfer proofs about components to a global system proof, the compatibility of the components must be checked (see Theorem 1 in Section 4.2, which is proved under these compatibility assumptions). Intuitively, the *compatibility check* ensures that the values *provided* for symbolic parameters of an output port of one component instance are *compatible* with the values *required* on a connected input port of the next instance, see Def. 6 later (e. g., the controller cannot demand target speeds outside the target range).

The main result of this process is that the component safety proofs—done for compatible components in isolation—transfer to an arbitrarily large system built by instantiating these components (cf. Theorem 1).

4 Component-based Modeling

In this section we introduce essential modeling idioms and definitions for the presented steps. Section 4.1 introduces components (cf. step (2)) and their contracts (cf. step (3)). Similarly, Bauer et al. [3] show how a contract framework can be built generically. Section 4.2 introduces composition (cf. step (5)) and ensures that the local properties transfer to the overall system.

4.1 Components and Contracts

Components can observe a shared global state, and modify their internal state.

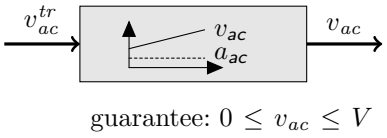
Definition 1 (Global Variables). *The global variables* V^{global} *are a set of variables shared by all components. It contains the variable* t*, which represents the system time, is initially set to* 0*, and increases linearly with rate* 1*. None of the global variables can ever be bound in any part of any component.*

In the following paragraphs, we define components, which have a behavior (e. g., how a cruise controller chooses a target velocity), and interfaces, which consist of input ports (e. g., the current velocity received by cruise control) and output ports (e. g., the new target velocity as provided by cruise control). We define the behavior of a component in the canonical order of a control part followed by a plant, which enables the definition of a structured composition operation for components and interfaces.

Definition 2 (Component). *A component* C *is defined as a tuple*

$$C = (\text{ctrl}, \text{plant}), \text{ where}$$

- ctrl *is the discrete control part of a hybrid program (HP) and does not contain continuous parts (i. e., differential equations), and*
- plant *is the continuous part of the form* $\{x_1' = \theta_1, ..., x_n' = \theta_n \& H\}$ *for* $n \in \mathbb{N}$ *i. e., an ordinary differential equation with evolution domain constraint* H*.*



(a) Actuator illustration

$$ctrl_{ac} \equiv a_{ac} := \frac{v_{ac}^{tr} - v_{ac}}{\varepsilon};\ t_{ac}^{0} := t \quad (1)$$

$$plant_{ac} \equiv \{v'_{ac} = a_{ac}\ \&\ t - t_{ac}^{0} \leq \varepsilon\} \quad (2)$$

$$\pi_{ac}^{in}(v_{ac}^{tr}) \equiv 0 \leq v_{ac}^{tr} \leq V \quad (3)$$

$$\pi_{ac}^{out}(v_{ac}) \equiv 0 \leq v_{ac} \leq V \quad (4)$$

(b) Formal component/interface

Fig. 2: Actuator component/interface example (C_{ac}, I_{ac})

The interface of a component consists of input and output ports, which can have contracts (i. e., π^{in} and π^{out}, e. g., value range for the target velocity).

Definition 3 (Interface). *An interface I is defined as a tuple*

$$I = \left(V^{\text{in}}, \pi^{\text{in}}, V^{\text{out}}, \pi^{\text{out}}\right), \text{ where}$$

- V^{in} *is a set of input variables,* V^{out} *is a set of output variables,*
- $\pi^{\text{in}} : V^{\text{in}} \to \mathcal{P}$ *specifies an input predicate ($\mathcal{P}$ represents the set of all logical formulas) representing input requirements and assumptions, exactly one per input variable (i. e., input port), accordingly for* $\pi^{\text{out}} : V^{\text{out}} \to \mathcal{P}$,
- $\forall v \in V^{\text{in}} : V(\pi^{\text{in}}(v)) \subseteq \left(\mathrm{V} \setminus V^{\text{in}}\right) \cup \{v\}$, *i. e., no input predicate can mention other input variables, which lets us reshuffle port ordering.*

An interface I is called admissible *for a component C, if* $(BV(\text{ctrl}) \cup BV(\text{plant})) \cap V^{\text{in}} = \emptyset$, *i. e., none of the input variables are bound in* ctrl *or* plant.

Consider our running example of the vehicle cruise control, where the actuator component chooses the acceleration according to a target velocity (cf. Fig. 2). As illustrated in Fig. 2a, the component has a single input port to receive a target velocity and a single output port to provide the current velocity.

Fig. 2b describes this component and interface formally: The actuator receives a target speed between 0 and V on its single input port v_{ac}^{tr}, cf. (3). It is a time-triggered controller with sampling period ε. The controller chooses the acceleration of the vehicle such that it will not exceed the target velocity until the next run and stores the current system time, cf. (1). The plant adapts the velocity accordingly and runs for at most ε time to enforce the sampling period, cf. (2). The single output port yields the resulting actual velocity, which still has to be in range between 0 and V, cf. (4).

Definition 4 (Contract). *Let C be a component, I be an admissible interface for C, and ϕ be a formula over the component's variables* V, *which determines the component's initial state. Let ψ^{safe} be a predicate over the component's variables* V, *i. e., a property describing the desirable target system state (i. e., a* safety

property*). We define* $\psi \overset{def}{\equiv} \psi^{\text{safe}} \wedge \Pi^{\text{out}}$*, where* $\Pi^{\text{out}} \equiv \bigwedge_{v \in V^{\text{out}}} \pi^{\text{out}}(v)$ *is the conjunction of all output guarantees. The* contract *of a component* C *with its interface* I *is defined as*

$$\text{Cont}(C, I) \;\equiv\; t = 0 \wedge \phi \rightarrow [(\text{in}; \text{ctrl}; \{t' = 1, \text{plant}\})^*]\psi$$

with input $\text{in} \overset{def}{\equiv} \big(v_1 := *; ?\pi^{\text{in}}(v_1)\big); ...; \big(v_r := *; ?\pi^{\text{in}}(v_r)\big)$ *for all* $v_i \in V^{\text{in}}$.

As the input predicates are not allowed to mention other inputs, the order of inputs in *in* is irrelevant. We call a component with an admissible interface that provably satisfies its contract to be *contract compliant*. This means, if started in a state satisfying ϕ, the component only reaches states that satisfy safety ψ^{safe} and all output guarantees π^{out} when all inputs satisfy π^{in}.

In our running example of Fig. 2, the actuator component has an output guarantee $\pi^{out} \equiv (0 \leq v_{ac} \leq V)$ (i. e., the speed must always be in range), and when starting from the initial conditions $\phi \equiv (v_{ac} = 0 \wedge \varepsilon > 0 \wedge V > 0)$ (i. e., vehicle initially stopped) it can provably guarantee safety[5] $\psi^{safe} \equiv 0 \leq v_{ac} \leq V$.

4.2 Composition of Components

Now that we have defined the structure and behavior of single components and their interfaces, we specify how to compose a number of those components by defining a syntactic composition operator for components. Differential dynamic logic follows the common assumption in hybrid systems that discrete actions do not consume time, i. e., multiple discrete actions of a program can happen instantaneously at the same real point in time. Time only passes during continuous evolution measured through t' in *plant*. Hence, if we disallow direct interaction between the controllers of components,[6] we can compose the discrete *ctrl* of multiple components in parallel by executing them sequentially in any order, while keeping their plants truly parallel through $\{x_1' = \theta_1, \ldots, x_n' = \theta_n \;\&\; H\}$. Interaction between components is then possible by observing plant output.

Such interaction, which exchanges information between components, will be defined by connecting ports when composing components through their interfaces. The port connections are represented by a mapping function $\mathcal{X}$, which assigns an output port to an input port for some number of input ports. In this paper, we focus on instantaneous lossless interaction, where the input variable v instantaneously takes on the value of the output port it is connected to, cf. $v := \mathcal{X}(v)$ in Def. 5. Other interaction patterns can be modeled by adapting Def. 5. For example, measurement with sensor uncertainty Δ is $v := *;\; ?\,(\mathcal{X}(v) - \Delta \leq v \leq \mathcal{X}(v) + \Delta)$, which yields a modified compatibility check.

As we do not require all ports to be connected, the mapping function is a partial function. Ports which are not connected become ports of the composite, while ports which are connected become internal variables.

[5] Note that in this case the output property and the safety property coincide. This is not necessarily always the case.

[6] Def. 5 restricts how variables between components can be shared.

Definition 5 (Parallel Composition). *Let C_i denote one of n components*

$$C_i = (\mathrm{ctrl}_i, \mathrm{plant}_i) \text{ for } i \in \{1, ..., n\}$$

with their corresponding admissible interfaces

$$I_i = \left(V_i^{\mathrm{in}}, \pi_i^{\mathrm{in}}, V_i^{\mathrm{out}}, \pi_i^{\mathrm{out}}\right) \text{ for } i \in \{1, ..., n\}$$

where $\left(V_i^{\mathrm{in}} \cup V_i^{\mathrm{out}} \cup \mathrm{V}(\mathrm{ctrl}_i) \cup \mathrm{V}(\mathrm{plant}_i)\right) \cap \left(V_j^{\mathrm{in}} \cup V_j^{\mathrm{out}} \cup \mathrm{V}(\mathrm{ctrl}_j) \cup \mathrm{V}(\mathrm{plant}_j)\right) \subseteq V^{\mathrm{global}}$ *for $i \neq j$, i. e., only variables in V^{global} are shared between components, and let*

$$\mathcal{X} : \left(\bigcup_{1 \leq j \leq n} V_j^{\mathrm{in}}\right) \rightharpoonup \left(\bigcup_{1 \leq i \leq n} V_i^{\mathrm{out}}\right)$$

be a partial (i. e., not every input must be mapped), injective (i. e., every output is only mapped to one input) function, connecting inputs to outputs. We define $\mathcal{I}^{\mathcal{X}}$ as the domain of $\mathcal{X}$ (i. e., all variables $x \in V^{\mathrm{in}}$ such that $\mathcal{X}(x)$ is defined) and $\mathcal{O}^{\mathcal{X}}$ as the the image of $\mathcal{X}$ (i. e., all variables $y \in V^{\mathrm{out}}$ such that $y = \mathcal{X}(x)$ holds for some $x \in V^{\mathrm{in}}$).

$$(C, I) \overset{def}{\equiv} ((C_1, I_1) \| ... \| (C_n, I_n))_{\mathcal{X}}$$

is defined as the composite of n components and their interfaces (with respect to $\mathcal{X}$), where

- *the sensing for non-connected inputs remains unchanged*

$$\mathrm{in} \equiv \underbrace{v_k := *; ?\pi^{\mathrm{in}}(v_k); \ldots; v_s := *; ?\pi^{\mathrm{in}}(v_s)}_{\textit{open inputs}} \text{ for } \{v_k, \ldots, v_s\} = V^{\mathrm{in}} \setminus \mathcal{I}^{\mathcal{X}}$$

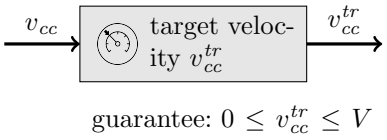
- *the order in which the control parts (and the respective port mappings) are executed is chosen non-deterministically (considering all the $n!$ possible permutations of $\{1, ..., n\}$), so that connected ports become internal behavior of the composite component*

$$\begin{aligned} \mathrm{ctrl} \equiv\ & (\mathrm{ports}_1; \mathrm{ctrl}_1; \mathrm{ports}_2; \mathrm{ctrl}_2; ...; \mathrm{ports}_n; \mathrm{ctrl}_n) \cup \\ & (\mathrm{ports}_2; \mathrm{ctrl}_2; \mathrm{ports}_1; \mathrm{ctrl}_1; ...; \mathrm{ports}_n; \mathrm{ctrl}_n) \cup \\ & ... \\ & (\mathrm{ports}_n; \mathrm{ctrl}_n; ...; \mathrm{ports}_2; \mathrm{ctrl}_2, \mathrm{ports}_1; \mathrm{ctrl}_1) \end{aligned}$$

  with $\mathrm{ports}_i \overset{def}{\equiv} \underbrace{v_j := \mathcal{X}(v_j); \ldots; v_r := \mathcal{X}(v_r)}_{\textit{connected inputs}}$ *for* $\{v_j, \ldots, v_r\} = \mathcal{I}^{\mathcal{X}} \cap V_i^{\mathrm{in}}$,

- *continuous parts are executed in parallel, staying inside* all *evolution domains*

$$\mathrm{plant} \equiv \{\underbrace{x_1^{(1)\prime} = \theta_1^{(1)}, \ldots, x_1^{(k)\prime} = \theta_1^{(k)}}_{\textit{component } C_1}, \ldots, \underbrace{x_n^{(1)\prime} = \theta_n^{(1)}, \ldots, x_n^{(m)\prime} = \theta_n^{(m)}}_{\textit{component } C_n} \ \& \ H_1 \wedge \ldots \wedge H_n\} \ ,$$



(a) Cruise controller illustration

$$ctrl_{cc} \equiv v_{cc}^{tr} := *;\ ?\big(0 \leq v_{cc}^{tr} \tag{5}$$

$$\wedge \left|v_{cc}^{tr} - v_{cc}\right| \leq \delta_V\big) \tag{6}$$

$$\pi_{cc}^{in}(v_{cc}) \equiv 0 \leq v_{cc} \leq V \tag{7}$$

$$\pi_{cc}^{out}(v_{cc}^{tr}) \equiv 0 \leq v_{cc}^{tr} \leq V \tag{8}$$

(b) Formal component/interface

Fig. 3: Cruise controller component/interface example (C_{cc}, I_{cc})

- *the respective sets of variables are merged, where* $V^{\text{in}} = \bigcup_{1 \leq i \leq n} V_i^{\text{in}} \backslash \mathcal{I}^{\mathcal{X}}$, $V^{\text{out}} = \bigcup_{1 \leq i \leq n} V_i^{\text{out}} \backslash \mathcal{O}^{\mathcal{X}}$, *i. e., ports not connected within the composite component remain input and output variables of the resulting interface,*
- *input port requirements of all interfaces are preserved, except for connected inputs, i. e.,* $\pi^{\text{in}} : V^{\text{in}} \to \mathcal{P}$ *becomes* $\pi^{\text{in}}(v)$*, accordingly for* $\pi^{\text{out}}(v)$*:*

$$\pi^{\text{in}}(v) \equiv \begin{cases} \pi_1^{\text{in}}(v) & \text{if } v \in V_1^{\text{in}} \setminus \mathcal{I}^{\mathcal{X}} \\ \dots & \\ \pi_n^{\text{in}}(v) & \text{if } v \in V_n^{\text{in}} \setminus \mathcal{I}^{\mathcal{X}} \end{cases} \quad \pi^{\text{out}}(v) \equiv \begin{cases} \pi_1^{\text{out}}(v) & \text{if } v \in V_1^{\text{out}} \setminus \mathcal{O}^{\mathcal{X}} \\ \dots & \\ \pi_n^{\text{out}}(v) & \text{if } v \in V_n^{\text{out}} \setminus \mathcal{O}^{\mathcal{X}} \end{cases}.$$

To demonstrate parallel composition in our running example, we first introduce a cruise controller component (cf. Fig. 3). The cruise control selects a target velocity from the interval, but keeps the difference between the current (received) velocity and the chosen target velocity below δ_V (cf. (5)–(6)). That way, the acceleration set by the actuator component is bounded by δ_V/ε (i. e., the vehicle does not accelerate too fiercely). We connect this cruise controller component to the actuator component (cf. Fig. 2), as illustrated in Fig. 4.

Remark 1. Note that verifying the hybrid program for a composite according to Def. 5 would require a proof of each of the $n!$ branches of *ctrl* individually, as they all differ slightly. For a large number of components, this entails a huge proof effort. Previous non-component-based case studies (e. g., [13,16,17]), therefore, chose only one specific ordering. Our component-based approach verifies all possible orderings at once, because the permutations are all proven correct as part of proving Theorem 1 below in this paper.

Remark 2. This definition of parallel composition uses a conjunction of all evolution domains, which resembles synchronization on the most restrictive component (i. e., as soon as the first and most restrictive condition is no longer fulfilled all plants have to stop and hand over to *ctrl*). A more liberal component might be forced to execute its control part because the evolution domain of a more restrictive component did no longer hold. For example a component increasing a counter on every run of its control is then forced to count although its own

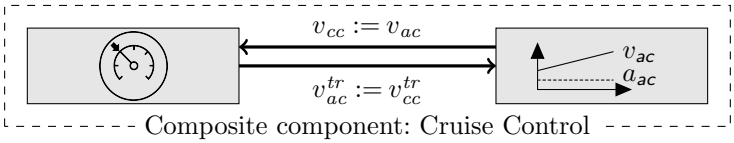


Fig. 4: Cruise control composed of a cruise controller and an actuator by Def. 5. The port connections $\mathcal{X} = \{(v_{cc}, v_{ac}),\ (v_{ac}^{tr}, v_{cc}^{tr})\}$ replace the input port $v_{ac}^{tr} := *;\ ?(0 \leq v_{ac}^{tr} \leq V)$ with an internal port assignment $v_{ac}^{tr} := v_{cc}^{tr}$, provided the compatibility check $[v_{ac}^{tr} := v_{cc}^{tr}]\left(\pi_{cc}^{out}(v_{cc}^{tr}) \rightarrow \pi_{ac}^{in}(v_{ac}^{tr})\right)$ succeeds, cf. Def. 6, and accordingly for the second port.

evolution domain might have allowed it to postpone control. If this is undesired, a component's control can be defined as $ctrl_i \cup ?\text{true}$, which would allow the component to skip when forced to run its control part.

Remark 3. We define this composition operation for any number of components, since it is *not associative*, because the composition of three components results in $3! = 6$ possible execution orders, whereas composing two components and adding a third yields only $2! + 2! = 4$ of the possible 6 execution orders.

Note that Def. 5 replaces the non-deterministic input guarded by a test from Def. 2 with a deterministic assignment that represents instantaneous and lossless interaction between components (i. e., $ports_i$), as illustrated in Fig. 4. Hence, the respective output guarantees and input assumptions must match. For instance, a cruise controller component demanding velocities $0 \leq v_{cc}^{tr} \leq 70$ is compatible with an actuator $0 \leq v_{ac}^{tr} \leq 100$, but not the other way around.

Definition 6 (Compatible Composite). *The composite of n components with interfaces $((C_1, I_1)\|...\|(C_n, I_n))_{\mathcal{X}}$ is a* compatible composite *iff*

$$\text{CPO}(I_i) \equiv [v := \mathcal{X}(v)]\left(\pi_j^{\text{out}}(\mathcal{X}(v)) \rightarrow \pi_i^{\text{in}}(v)\right)$$

is valid for all input ports $v \in \mathcal{I}^{\mathcal{X}} \cap V_i^{\text{in}}$, for all interfaces I_i and where I_j is the interface containing the port that is connected to the input port v of I_i. We call $\text{CPO}(C_i)$ *the* compatibility proof obligation *for the interfaces I_i and say the interfaces I_i are* compatible *(with respect to $\mathcal{X}$) if* $\text{CPO}(I_i)$ *holds.*

In other words, $((C_1, I_1)\|...\|(C_n, I_n))_{\mathcal{X}}$ is a *compatible* composite if all internal port connections are appropriate, i. e., if the guarantee of the output port implies the requirements of the respective input port to which it is connected.

Composite Contracts. Now that we have defined components and interfaces, their contracts, and how to compose them to form larger composites, we prove that the contracts of single components transfer to composites if compatible.

Theorem 1 (Composition Retains Contracts). *Let C_1 and C_2 be components with admissible interfaces I_1 and I_2 that are* contract compliant *(i. e., their contracts are valid)*

$$\models t = 0 \land \phi_1 \rightarrow [(\mathrm{in}_1; \mathrm{ctrl}_1; \{t' = 1, \mathrm{plant}_1\})^*] (\psi_1) \text{ and} \tag{9}$$

$$\models t = 0 \land \phi_2 \rightarrow [(\mathrm{in}_2; \mathrm{ctrl}_2; \{t' = 1, \mathrm{plant}_2\})^*] (\psi_2) \tag{10}$$

and compatible *with respect to $\mathcal{X}$ (i. e., compatibility proof obligations are valid)*

$$\models [v := \mathcal{X}(v)] \left(\pi_1^{\mathrm{out}}(\mathcal{X}(v)) \rightarrow \pi_2^{\mathrm{in}}(v)\right) \text{ and} \tag{11}$$

$$\models [v := \mathcal{X}(v)] \left(\pi_2^{\mathrm{out}}(\mathcal{X}(v)) \rightarrow \pi_1^{\mathrm{in}}(v)\right) \tag{12}$$

$$\text{for all input ports } v \in \mathcal{I}^{\mathcal{X}} \cap V_{1,2}^{\mathrm{in}} \ .$$

Then, the parallel composition $C_3, I_3 = ((C_1, I_1) \| (C_2, I_2))_{\mathcal{X}}$ satisfies the contract

$$\models t = 0 \land (\phi_1 \land \phi_2) \rightarrow [(\mathrm{in}_3; \mathrm{ctrl}_3; \{t' = 1, \mathrm{plant}_3\})^*](\psi_1 \land \psi_2) \tag{13}$$

with in_3, ctrl_3, *and* plant_3 *according to Def. 5.*

The proof for Theorem 1 can be found in [19], along with a generalization to n components. This central theorem allows us to infer how properties from single components transfer to their composition. As such, it suffices to prove the properties for the components and conclude that a similar property holds for the composite, without explicitly having to verify it. The composite contract states that, considering both pre-conditions hold (i. e., $\phi_1 \land \phi_2$), all states reached by the parallel execution of the components, both post-conditions hold (i. e., $\psi_1 \land \psi_2$).

5 Case Study: Vehicle Cruise Control

To illustrate our approach, we used a running example of a simple *vehicle cruise control system*. The overall system requirement was to keep the velocity v_{ac} in a desired range $[0, V]$ at all times, i. e., $0 \leq v_{ac} \leq V \rightarrow [CruiseControl] 0 \leq v_{ac} \leq V$. We split the system into two components, cf. Fig. 4: an actuator component adapts velocity according to a target v_{ac}^{tr} provided by a cruise control component as v_{cc}^{tr}. If the cruise control component (Fig. 3) provides a valid target velocity to the actuator (i. e., $0 \leq v_{ac}^{tr} \leq V$), the actuator component (Fig. 2) ensures to keep the actual velocity in the desired range (i. e., $0 \leq v_{ac} \leq V$), thus ensuring the overall system property. Additionally, the actuator provides the current velocity on an output port that is read by the controller, acting as a feedback loop.

Following Def. 4, we derive contracts for each component, which consists of initial conditions ϕ (cf. (14)–(15)), safety conditions ψ^{safe} (cf. (16)) and the port conditions (cf. (4) and (8)). Maximum speed $V > 0$ and cycle time $\varepsilon > 0$ must be known. Additionally, the controller initializes $v_{cc}^{tr} = 0$ and $\delta_V > 0$. The actuator restricts the initial velocity to $0 \leq v_{ac} \leq V$.

$$\phi_{cc} \equiv v_{cc}^{tr} = 0 \land \varepsilon > 0 \land V > 0 \land \delta_V > 0 \tag{14}$$

$$\phi_{ac} \equiv 0 \leq v_{ac} \leq V \land \varepsilon > 0 \land V > 0 \tag{15}$$

$$\psi_{ac}^{safe} \equiv 0 \leq v_{ac} \leq V \tag{16}$$

The set of global variables follows accordingly (cf. Def. 1): $\mathrm{V}^{global} = \{\varepsilon, V, t\}$.

After verifying[7] both contracts *Cont*(C_{cc}, I_{cc}) and *Cont*(C_{ac}, I_{ac}), we want to compose the components to get the overall system, using the mapping function $\mathcal{X} = \{(v_{cc}, v_{ac}),\ (v_{ac}^{tr}, v_{cc}^{tr})\}$. Therefore, we have to check the compatibility proof obligations for both connected ports (cf. Fig. 4). Then the overall system property directly follows from the contract of the actuator component.

Splitting a system into components reduces the model complexity considerably, since a component needs to know neither about the differential equation systems of other components, nor about their control choices. In combined models, we have to analyze all the possible permutations of control choices, while in the component-based approach, by Theorem 1 we can guarantee correctness for all possible sequential orderings, without the proof effort entailed by listing them explicitly.

The benefit of component-based verification becomes even larger when replacing components in a system. For example, we can easily replace the cruise control from Fig. 3 with a more sophisticated controller that takes the target velocity as user input from an additional input port. After verifying the user guided cruise control component, we only have to re-check the compatibility proof obligations. In a monolithic model, in contrast, the whole system including the actuator component must be re-verified.

6 Related Work

CPS Verification. Hybrid automata [2] are popular for modeling CPS, and mainly verified using reachability analysis. Unlike hybrid programs, hybrid automata are not compositional, i. e., for a hybrid automaton it is not sufficient to establish a property about its parts in order to establish a property about the automaton. Techniques such as assume-guarantee reasoning or hybrid I/O automata [14], which are an extension of hybrid automata with input- and output-ports, address this issue. Our approach here shares some of the goals with hybrid I/O automata and also uses I/O ports. But we target compositional reasoning for *hybrid programs*, where the execution order of statements is relevant, so that our approach defines how parallel composition results in interleaving of hybrid programs. Furthermore, we define compositional modeling for hybrid programs such that *theorem proving* of the entire system is reduced to proving properties about the components and *simple composition checks*. Hybrid process algebras (e. g., Hybrid χ [24], HyPA [20]) are specifically developed as compositional modeling formalisms to describe behavior and interaction of processes using algebraic equations. For verification purposes by simulation or reachability analysis, translations from Hybrid χ into hybrid automata and timed automata exist, so even though modeling is compositional, verification still falls back to monolithic analysis. We, in contrast, focus on exploiting compositionality in the proof.

Component-based CPS Modeling. Damm et al. [5] present a component-based design framework for controllers of hybrid systems with a focus on reac-

[7] All proofs were done in KeYmaera X [8].

tion times. The framework checks connections when interconnecting components: alarms propagated by an out-port must be handled by the connected in-ports. We, too, check component compatibility, but for contracts, and we focus on transferring proofs from components to the system level.

Focusing on architectural properties, Ruchkin et al. [26] propose a component-based modeling approach for hybrid-systems. Although they do not transfer verification results from components to composites, their definitions have been an inspiration for our notion of components. Ringert et al. [25] model CPS as Component and Connector (C&C) architectures using automata to describe solely the discrete behavior. They verify the translated models of single components, but do not provide guarantees about verified compositions.

Interface algebras (cf. [1,9]) are formalisms that separate component-based models into interface models and component models. Similar to our approach, the component model describes what a component does, while the interface model defines how the component can be used. It is often distinguished between interfaces with and without state, where stateful interfaces are usually viewed as concurrent games. Our approach is similar to a stateless interface algebra [1]. Similarly, Bauer et al. [3] show how a contract framework can be built generically. While useful for inspiration, these approaches focus on modeling aspects and do not consider verification.

Verification. Madl et al. [15] model real-time event-driven systems. Their models can be transformed to UPPAAL (cf. [12]) timed automata, restricting the continuous part of their models to time instead of arbitrary physical behavior (e. g., movement). Moreover, their analysis targets the *entire composition* of timed automata, thus defeating the advantages of components for verification.

A field closely related to component-based verification is assume-guarantee reasoning (AGR, e. g., [7,10]), which was originally developed as a device to counteract the state explosion problem in model checking by decomposing a verification task into subtasks. In AGR, individual components are analyzed together with *assumptions* about their context and *guarantees* about their behavior (i. e., a component's contract). AGR rules need to exercise care for circularity in the sense that the approaches verify one component in the context of the other and vice-versa, like Frehse et al. [7] (using Hybrid Labeled Transition Systems as abstraction for Hybrid I/O-Automata) and Henzinger et al. [10] (using hierarchical hybrid systems based on hybrid automata). However, existing approaches are often limited to linear dynamics, cannot handle continuity or use corresponding reachability analysis or model checking techniques. In $\mathsf{d}\mathcal{L}$, in contrast, we can handle non-linear dynamics and focus on theorem proving.

In summary, only few component-based approaches handle generic CPS with both discrete and continuous aspects (e. g., [5,15,26]), but those do not yet focus on the impact on formal verification. Related techniques for CPS and hybrid systems verification focus mainly on timed automata, hybrid process algebras, and hybrid automata with linear dynamics or end up in monolithical verification.

7 Conclusion and Future Work

We presented an approach for component-based modeling and verification of CPS that (i) splits a CPS into components, (ii) verifies a contract for each of these components and (iii) composes component instances in a way that transfers the component contracts to a composite contract. Our approach makes hybrid system verification more modular at the scale of components, and combines the advantages of component-based modeling approaches (e. g., well structured models, reduced model complexity, simplified model evolution) with the advantages of formal verification (e. g., guaranteed contract compliance).

Currently, our approach is limited to global properties that are stated relative to the initial system state. Port conditions are only allowed to mention global variables and the port variable itself, which prevents conditions on the change of a port since the last measurement (e. g., how far has a vehicle moved since the beginning vs. how far has it moved since the last measurement). This restriction can be removed with ports that remember their previous value and relate measurements over time. Additionally, we plan to (i) introduce further composition operations (e. g., sensing with measurement errors), (ii) provide further component extensions (e. g., multi-cast ports), and (iii) provide tool support to instantiate and compose components, and to generate their hybrid programs.

References

1. de Alfaro, L., Henzinger, T.A.: Interface theories for component-based design. In: Henzinger, T.A., Kirsch, C.M. (eds.) Embedded Software, First Int. Workshop, EMSOFT 2001, Oct., 8-10, 2001, Proc. LNCS, vol. 2211, pp. 148–165. Springer (2001)
2. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.: Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In: Grossman, R.L., Nerode, A., Ravn, A.P., Rischel, H. (eds.) Hybrid Systems. LNCS, vol. 736, pp. 209–229. Springer (1992)
3. Bauer, S.S., David, A., Hennicker, R., Larsen, K.G., Legay, A., Nyman, U., Wasowski, A.: Moving from specifications to contracts in component-based design. In: de Lara, J., Zisman, A. (eds.) Fundamental Approaches to Software Engineering (FASE), Mar. 24 - Apr. 1, 2012. Proc. LNCS, vol. 7212, pp. 43–58. Springer (2012)
4. Benvenuti, L., Bresolin, D., Collins, P., Ferrari, A., Geretti, L., Villa, T.: Assume-guarantee verification of nonlinear hybrid systems with Ariadne. Int. Journal of Robust and Nonlinear Control 24(4), 699–724 (2014)
5. Damm, W., Dierks, H., Oehlerking, J., Pnueli, A.: Towards component based design of hybrid systems: Safety and stability. In: Manna, Z., Peled, D.A. (eds.) Time for Verification. LNCS, vol. 6200, pp. 96–143. Springer (2010)
6. Felty, A.P., Middeldorp, A. (eds.): Automated Deduction - CADE-25 - 25th Int. Conf. on Autom. Deduction, Aug. 1-7, 2015, Proc., LNCS, vol. 9195. Springer (2015)
7. Frehse, G., Han, Z., Krogh, B.: Assume-guarantee reasoning for hybrid i/o-automata by over-approximation of continuous interaction. In: Decision and Control, 2004. CDC. 43rd IEEE Conf. on. vol. 1, pp. 479–484 (Dec 2004)
8. Fulton, N., Mitsch, S., Quesel, J., Völp, M., Platzer, A.: KeYmaera X: an axiomatic tactical theorem prover for hybrid systems. In: Felty and Middeldorp [6], pp. 527–538

9. Graf, S., Passerone, R., Quinton, S.: Contract-based reasoning for component systems with rich interactions. In: Sangiovanni-Vincentelli, A., Zeng, H., Di Natale, M., Marwedel, P. (eds.) Embedded Sys. Dev., vol. 20, pp. 139–154. Springer (2014)
10. Henzinger, T.A., Minea, M., Prabhu, V.S.: Assume-guarantee reasoning for hierarchical hybrid systems. In: Benedetto, M.D.D., Sangiovanni-Vincentelli, A.L. (eds.) Hybrid Systems: Computation and Control, 4th Int. Workshop, HSCC 2001, Mar. 28-30, 2001, Proc. LNCS, vol. 2034, pp. 275–290. Springer (2001)
11. Kurki-Suonio, R.: Component and interface refinement in closed-system specifications. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM'99 - Formal Methods, Sept. 20-24, 1999, Proc. LNCS, vol. 1708, pp. 134–154. Springer (1999)
12. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. STTT 1(1-2), 134–152 (1997)
13. Loos, S.M., Platzer, A., Nistor, L.: Adaptive cruise control: Hybrid, distributed, and now formally verified. In: Butler, M., Schulte, W. (eds.) FM'11 - Formal Methods. LNCS, vol. 6664, pp. 42–56. Springer (2011)
14. Lynch, N.A., Segala, R., Vaandrager, F.W.: Hybrid I/O automata. Inf. Comput. 185(1), 105–157 (2003)
15. Madl, G., Abdelwahed, S., Karsai, G.: Automatic verification of component-based real-time CORBA applications. In: Proc. of the 25th IEEE Real-Time Systems Symp. (RTSS), 5-8 Dec. 2004. pp. 231–240. IEEE Computer Society (2004)
16. Mitsch, S., Ghorbal, K., Platzer, A.: On provably safe obstacle avoidance for autonomous robotic ground vehicles. In: Newman, P., Fox, D., Hsu, D. (eds.) Robotics: Science and Systems IX, Technische Universität Berlin, June 24-28, 2013 (2013)
17. Mitsch, S., Loos, S.M., Platzer, A.: Towards formal verification of freeway traffic control. In: ICCPS. pp. 171–180. IEEE/ACM (2012)
18. Müller, A., Mitsch, S., Platzer, A.: Verified traffic networks: Component-based verification of cyber-physical flow systems. In: 18th IEEE Intelligent Transportation Systems Conf. (ITSC). pp. 757–764. IEEE (2015)
19. Müller, A., Mitsch, S., Retschitzegger, W., Schwinger, W., Platzer, A.: A component-based approach to hybrid systems safety verification. Tech. Rep. CMU-CS-16-100, Carnegie Mellon (2016)
20. Pieter J. L. Cuijpers, Reniers, M.A.: Hybrid process algebra. J. Log. Algebr. Program. 62(2), 191–245 (2005)
21. Platzer, A.: Differential-algebraic dynamic logic for differential-algebraic programs. J. Log. Comput. 20(1), 309–352 (2010)
22. Platzer, A.: The complete proof theory of hybrid systems. In: Proc. of the 27th Annual IEEE Symp. on Logic in Computer Science, LICS 2012, June 25-28, 2012. pp. 541–550. IEEE Computer Society (2012)
23. Platzer, A.: A uniform substitution calculus for differential dynamic logic. In: Felty and Middeldorp [6], pp. 467–481
24. Ramon R. H. Schiffelers, D. A. van Beek, Man, K.L., Reniers, M.A., Rooda, J.E.: Formal Semantics of Hybrid Chi. In: Larsen, K.G., Niebert, P. (eds.) Formal Modeling and Analysis of Timed Systems. LNCS, vol. 2791, pp. 151–165. Springer (2003)
25. Ringert, J.O., Rumpe, B., Wortmann, A.: From software architecture structure and behavior modeling to implementations of cyber-physical systems. In: Wagner, S., Lichter, H. (eds.) Software Engineering 2013 - Workshopband, 26. Feb. - 1. Mar. 2013. LNI, vol. 215, pp. 155–170. GI (2013)
26. Ruchkin, I., Schmerl, B.R., Garlan, D.: Architectural abstractions for hybrid programs. In: Kruchten, P., Becker, S., Schneider, J. (eds.) Proc. of the 18th Int. ACM SIGSOFT Symp. on Component-Based Software Engineering, CBSE 2015, May 4-8, 2015. pp. 65–74. ACM (2015)