A Component-based Hybrid Systems Verification and Implementation Tool in KeYmaera X (Tool Demonstration)*

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

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

Abstract. Safety-critical cyber-physical systems (CPS) should be analyzed using formal verification techniques in order to gain insight into and obtain rigorous safety guarantees about their behavior. For practical purposes, methods are needed to split modeling and verification effort into manageable pieces and link formal artifacts and techniques with implementation. In this paper we present a tool chain that supports component-based modeling and verification of CPS, generation of monitors, and systematic (but unverified) translation of models and monitors into executable code. A running example demonstrates how to model a system in a component-based fashion in differential dynamic logic (dL), how to represent and structure these models in the syntax of the hybrid systems theorem prover KeYmaera X (which implements dL), and how to prove properties in KeYmaera X. The verified components are the source for translation into executable C code, which can be run on controlled components (e. g., a robot). Additionally, we demonstrate how to generate monitors that validate the behavior of uncontrolled components (e. g., validate the assumptions made about obstacles).

1 Introduction

To ensure safe operation of cyber-physical systems (CPS), their behavior should be analyzed in safety analysis using formal verification techniques. However, monolithic models and their analysis become unnecessarily complex with increasingly large systems. Hence, techniques are needed to split both modeling and verification effort into more manageable pieces. At the same time, the correctness properties that are verified formally for a model also have to hold for

* This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) under grant number FA8750-18-C-0092 and FA8750-12-2-0291, AFOSR FA9550-18-1-0120, and by the Austrian Science Fund (FWF) P28187-N31.

R. Chamberlain et al. (Eds.): CyPhy 2018/WESE 2018, LNCS 11615, pp. 91–110, 2019.
DOI: 10.1007/978-3-030-23703-5_5

the actual implementation. When translating a model into an implementation, however, any gaps need to be overcome between modeling concepts beneficial for verification (e. g., nondeterministic control and real numbers) and those appropriate for implementation (e. g., deterministic control and machine floating points) in a way that preserves correctness (assuming correct C compilation).

Formal verification has been used successfully for hybrid system models, both using model checking [6] and theorem proving [17] techniques. For more complex applications, monolithic hybrid systems models are impractical compared to models that provide more structuring principles into smaller submodels. The real gain of component-based hybrid systems modeling techniques is realized, however, when the division into smaller components of less responsibilities is not just available when describing the models but also exploited during their formal verification by compositional proofs [14]. Finally, the full benefit of component-based hybrid system modeling needs a way of correctly implementing the components in a way that faithfully fits the intended interactions of the model.

Sound axiomatizations [15,16], verification tools [9], cross-verification in other provers [3], and provably correct compilation tools to executables [4] are known for hybrid systems themselves. While these are compositional in the programming language operators of hybrid programs [15,16,17], extensions to component-based hybrid system models remain an important challenge.

Thus, this paper takes a useful step toward these goals by developing a component-based hybrid systems modeling and verification tool built into the KeYmaera X prover for hybrid systems [9]. No soundness-critical extensions are needed for the verification in KeYmaera X, because the implementation is in tactics outside its soundness-critical kernel [8]. We take a pragmatic approach for component-based implementation of hybrid systems by generating C code that is informally inspected to be correct, but does not yet provide the degree of rigor of generating implementations of hybrid systems correctly and bridging floating point vs. real arithmetic using a chain of theorem provers [4]. Given the added value of generating code in the well-known C language, we argue that our pragmatic choice is useful in practice to enable an easy integration into an existing infrastructure of embedded and cyber-physical systems. In addition to the challenges of nondeterminism in the models, we tackle the challenges specific to component-based systems: generating code for ownsystem components with sensing/actuation interfaces to external components (e. g., obstacles). As in [4], our main ingredient to obtain correct integration with external control components is the provably correct monitor synthesis from ModelPlex [13].

2 Preliminaries

Differential Dynamic Logic (dL*).* For specifying and verifying correctness statements about hybrid systems, we use *differential dynamic logic* (dL) [15,17,16], which is a real-valued first-order dynamic logic for hybrid systems and supports *hybrid programs* as a program notation for hybrid systems.

Table 1: Operators of differential dynamic logic (dL) formulas

dL	Operator	Meaning
$\theta_1 \sim \theta_2$	comparison	true iff $\theta_1 \sim \theta_2$ with $\sim \in \{>, \geq, =, \neq, \leq, <\}$
$\neg\phi$	not	true iff ϕ is false
$\phi \wedge \psi$	and	true iff both ϕ and ψ are true
$\phi \vee \psi$	or	true iff ϕ is true or if ψ is true
$\phi \rightarrow \psi$	implies	true iff ϕ is false or ψ is true
$\phi \leftrightarrow \psi$	equivalent	true iff ϕ and ψ are both true or both false
$\forall x\, \phi$	universal quant.	true iff ϕ is true for all values of variable x in $\mathbb{R}$
$\exists x\, \phi$	existential quant.	true iff ϕ is true for some values of variable x in $\mathbb{R}$
$[\alpha]\phi$	$[\cdot]$ modality	true iff ϕ is true after all runs of hybrid program α
$\langle\alpha\rangle\phi$	$\langle\cdot\rangle$ modality	true iff ϕ is true after at least one run of α

Table 2: Hybrid program statements (Q is a formula, α, β are hybrid programs)

Statement	Effect
$\alpha;\ \beta$	sequential composition where β starts after α finishes
$\alpha\ \cup\ \beta$	nondeterministic choice, following either alternative α or β
α^*	nondeterministic repetition, repeating α n times for any $n \in \mathbb{N}$
$x := \theta$	discrete assignment of the value of term θ to variable x (jump)
$x := *$	nondeterministic assignment of an arbitrary real number to x
$\big(x_1' = \theta_1, \ldots,$ $x_n' = \theta_n \& Q\big)$	continuous evolution of x_i along the differential eq. system $x_i' = \theta_i$ restricted to remain in evolution domain Q at all times
$?Q$	test if formula Q holds at current state, otherwise abort

Operators of dL and their informal meaning, are summarized in Table 1, and comprise the usual comparison operators, boolean operators and quantifiers. Additionally, dL supports modalities to reason about the state after at least one, respectively all runs of a hybrid program. The syntax and informal semantics of hybrid programs are summarized in Table 2. For example, a hybrid program

$$\alpha \;\equiv\; (y := *;\ ?y \leq z;\ t := 0;\ \{x' = y, t' = 1\ \&\ t \leq 10\})^* \tag{1}$$

picks any real value for y that does not exceed z, resets time t to zero, and then in the ODE continuously evolves the value of x according to the fixed slope y while simultaneously increasing the value of t with constant slope 1. The ODE stops nondeterministically at any time, but before $t \leq 10$ becomes false; then the program repeats by the * operator. A corresponding dL formula

$$x = 0 \wedge z < 0 \rightarrow [\alpha]x \leq 0 \tag{2}$$

states that starting in a state, where x is 0 and z is negative, each run of the above program α leads to a state, where x is less or equal 0.

KeYmaera X. KeYmaera X [9] is an automated and interactive theorem prover for dL and hybrid programs. KeYmaera X is implemented in Scala, expands upon functionality by introduction of tactics [8] and is based on a significantly smaller soundness-critical core than other hybrid systems verification tools, which makes it easier to ensure correct verification results. The valid example formula (2) can, for instance, be verified in KeYmaera X. We will introduce the concrete ASCII syntax of KeYmaera X later alongside our running example in Section 3.

3 Component-based Verification Tool

As the complexity of CPSs increases, monolithic models and analysis techniques become unnecessarily challenging. As already established for discrete software, decomposition into subsystems with contracts is essential in taming the complexity of larger systems. Thus, we have explored compositional modeling and verification techniques for hybrid systems [14] that conclude safety of the entire system from separate isolated safety arguments about its components and their interaction with the environment. The KeYmaera X hybrid system theorem prover allows us to bundle and analyze the ingredients of our component-based approach—component models, specifications, and lemmas of satisfied proof obligations—in a single input format.

ASCII Syntax. Models and specifications are provided to KeYmaera X in the dL ASCII syntax, which is a straightforward ASCII rendition of Tables 1 and 2, e. g., using `A->B` for $A \to B$ and using `A&B` for $A \wedge B$. The ASCII notation `alpha++beta` is used for alpha∪beta. For improved readability in longer examples, braces {...} are used for grouping differential equation systems and other program operators. Like in C programs, assignments etc. end with explicit semicolons.

The dL ASCII syntax is the basis for named entries in `.kyx` files, which consist of an optional `SharedDefinitions` block, with global definitions for the entire archive, and multiple named `ArchiveEntry` blocks, which themselves consist of optional definitions (`Definitions`), system variables (`ProgramVariables`), a (safety) specification in dL (`Problem`), and optional tactic scripts[3] (`Tactic`). Each of these blocks must be closed with an `End` statement.

```
ArchiveEntry "Example Formula (2)"
  Definitions    /* constants, functions, properties, programs */
    Real z;
    HP a ::= { {y:=*; ?y<=z; t:=0; {x'=y, t'=1 & t<=10} }* };
  End.
  ProgramVariables Real x, z; End.      /* variables */
  Problem x=0 & z<0 -> [a;]x<=0 End.    /* specification in dL */
  Tactic "Auto Proof"   /* tactic script, produces proof/lemma */
    master
  End.
End.
```

[3] The tactics language Bellerophon [8] for verification of hybrid systems provides a way to convey insights by programming hybrid systems proofs.

The symbols defined in the `Definitions` and the variables defined in the `ProgramVariables` can be used in the `Problem` block or in other definitions. The named `Tactic` blocks, if provided, each list a Bellerophon [8] tactic to verify the current `Problem`.

3.1 Running Example

To illustrate the concepts of our component-based verification approach and to demonstrate the capabilities of our verification tool, we use a running example of a *robot* that has to avoid collision with an *obstacle*, see Fig. 1.

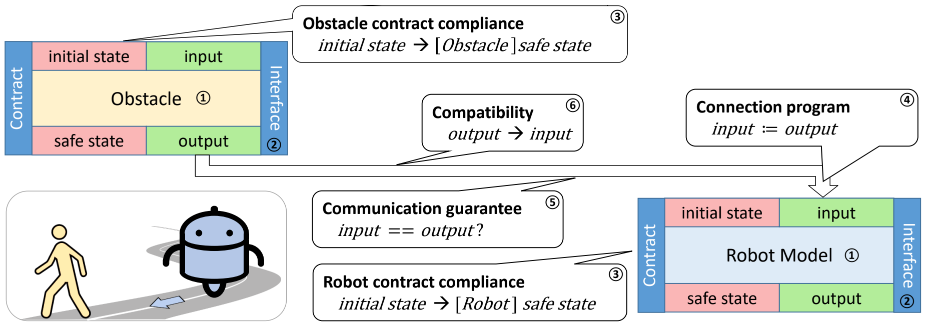

Fig. 1: Running Example: Robot collision avoidance

The speed of the obstacle is limited to at most S. The robot regularly receives the obstacle's position to ensure that it stays at a safe distance from the obstacle. Additionally, the robot can receive desired speed suggestions (e. g., from a remote control, or by a user, which will not be modeled as part of the example), which guide the robot's speed. In our example, the difference between updates of the speed suggestion is limited to avoid sudden speed changes. The overall *safety property* of the system is that robot and obstacle should never crash. If they meet, the robot must be stopped.

Shared definitions. Shared definitions in the beginning of a `.kyx` file are used to define global constants, facts, and programs, which can be accessed in all archive entries. The components in the example share facts about constants, such as the maximum difference `D()` between speed suggestions, the speed limit `S()` of obstacles, each ≥ 0, and the maximum control cycle time `ep()>0` that limits the plant runtime (i. e., the time until the next controller run and sensor update). They also define program `skip` as the trivial test that always passes.

```
SharedDefinitions
  Real D();           /* maximum change in desired speed request */
  Real S();                                      /* speed limit */
  Real ep();                           /* maximum control cycle time */
  Bool globalFacts <-> ( D()>=0 & S()>=0 & ep()>0 );     /* facts */
  HP skip ::= { ?true; };                  /* skip as trivial test */
End.
```

3.2 Component-based Deductive Verification by Contracts

Our component-based deductive verification approach [14] bases on individual *components* with *interfaces* and local *contracts*, which are *composed* in a *compatible* fashion to create a safe model of the overall system and *make hybrid system theorem proving modular on a component level*. Under certain precisely formalized compatibility conditions on how components are connected, the components and their contracts ensure that their compositions directly inherit safety from the safety of the components. Users of our approach [14] provide the following specifications and lemmas (circled numbering as in Fig. 1):

1. Component models ① [14, Def. 1], design parameters [14, Def. 3], interfaces ② [14, Def. 4] and component contract compliance lemmas ③ [14, Def. 5] define what components guarantee about their behavior in isolation.
2. Connections and connection programs ④ [14, Remark 1] define how the components interact in the composed system.
3. Communication guarantee lemmas ⑤ [14, Def. 7] and compatibility lemmas ⑥ [14, Def. 8] ensure that the interaction in the system happens between compatible components.

The component composition tactic in KeYmaera X then combines these individual lemmas into a safety proof of the overall composed system. In the following, we briefly recap the necessary definitions of previous work [14] and apply our component-based verification approach to the robot example.

Components, Interface and Contract Compliance

Components. As usual, each component combines a model of the dynamic behavior with an interface defining how the component receives inputs and provides outputs through ports. The component behavior model consists of a control *ctrl* (comprising exclusively discrete computations), a physical plant *plant* and internal communication *cp* between the sub-components of non-atomic components (`skip` for atomic components). Components do not share any variables, except for global constants accessed throughout the system.

The robot example consists of two components: Both measure time t in their plant. The obstacle C_o (3) nondeterministically chooses speed s_o in its controller and moves its position p_o as defined in the plant. If safe according to formula (5) (i. e., distance to measured obstacle position $\hat{p}_o$ large enough), the robot C_r (4) sets its speed s_r to the received speed suggestion $\hat{d}$ and moves its position p_r accordingly in the plant (i. e., position changes according to speed and time evolves linearly, for at most maximum control cycle time ε); otherwise it stops. These components are atomic, so have trivial internal communication $cp \equiv$ `skip`.

$$\mathrm{C}_o = (\overbrace{s_o := *; ?\big(0 \le s_o \le S\big)}^{ctrl_o},\ \overbrace{t' = 1, p_o' = s_o}^{plant_o}) \tag{3}$$

$$\mathrm{C}_r = (\underbrace{\text{if}\,(\text{Drive})\, s_r := \hat{d} \text{ else } s_r := 0}_{ctrl_r}, \underbrace{t' = 1, p_r' = s_r \ \&\ t - t^- \le \varepsilon}_{plant_r}) \tag{4}$$

$$\text{Drive} \equiv \hat{p}_o - p_r > (\hat{d} + S) \cdot \varepsilon \tag{5}$$

Components in ASCII syntax. When specifying components in KeYmaera X, we declare all component variables and define the hybrid programs for the component controller, plant, and internal connections. For example, the obstacle C_o (3) has real-valued variables for its current position p_o (`po`), its previous position p_o^- (`po0`), and its speed s_o (`so`), and it will keep track of time t (`t`) and plant start time t^- (`t0`) in the contracts that we develop subsequently.

```
Real po;    /* current obstacle position */
Real po0;   /* previous obstacle position */
Real so;    /* speed of obstacle */
Real t;     /* time */
Real t0;    /* time before plant's present control cycle */
```

In ASCII syntax, the component programs are the controller $ctrl_o$ written as `ctrlobs`, the plant $plant_o$ as `plantobs` with ASCII evolution domain constraint `true` (which can also be omitted in dL), and glue code cp_o as `cpobs`.

```
HP ctrlobs  ::= { so:=*; ?(0<=so&so<=S()); }; /* obstacle control */
HP plantobs ::= { {t'=1, po'=so & true} };      /* obstacle plant */
HP cpobs    ::= { skip; };                    /* obstacle glue code */
```

Interfaces. Possible interaction points of a component are described in its interface, which defines input ports V^{in} and output ports V^{out}, together with input assumptions π^{in} and output guarantees π^{out} that stipulate the expected respectively guaranteed range of values on each port. We use $\mapsto$ to associate assumptions and guarantees with ports.

In the robot example, the obstacle outputs its position p_o, whose change is bounded by the maximum speed S and the time $(t - t^-)$ that has passed since the last position information transmission (6). The robot comes with two input ports (7) that take a speed suggestion $\hat{d}$, which must not deviate from its previous value $\hat{d}^-$ by more than D (to avoid too sudden speed changes), and an obstacle position $\hat{p}_o$, with an input assumption that, for simplicity, exactly matches the output guarantee of the obstacle's single output port.

$$\mathrm{I}_o = \big(\overbrace{\{p_o\}}^{V_o^{out}}, \overbrace{p_o \mapsto \left|p_o - p_o^-\right| \leq S \cdot \left(t - t^-\right)}^{\pi_o^{out}}\big) \tag{6}$$

$$\mathrm{I}_r = \big(\underbrace{\{\hat{p}_o, \hat{d}\}}_{V_r^{in}}, \underbrace{\left(\hat{p}_o \mapsto \left|\hat{p}_o - \hat{p}_o^-\right| \leq S \cdot \left(t - t^-\right), \hat{d} \mapsto \left|\hat{d} - \hat{d}^-\right| \leq D\right)}_{\pi_r^{in}}\big) \tag{7}$$

Interfaces in ASCII syntax. The variable and property declarations of interfaces translate into hybrid programs and formulas [14, Def. 6]. The obstacle (6) has a single output port for its position p_o, so we define a port memory program `deltaobs` that stores the position in `po0`. The obstacle does not have input ports, so `inobs` is `skip`. The obstacle output guarantee π_o^{out}, which specifies that the position change $|p_o - p_o^-|$ over duration $(t - t^-)$ cannot exceed maximum speed S (referred to with nullary constant function symbol `S()`), is defined in `safeobs`. The boolean predicate `safeobs` takes real-valued position and time arguments. Additionally, in `initobs` we define under which initial conditions obstacles will meet their output guarantees, which will become important for verifying contract compliance.

```
HP deltaobs ::= { po0:=po; };                          /∗ port memory ∗/
HP inobs    ::= { skip; };                         /∗ read input ports ∗/
Bool safeobs(Real po, Real po0, Real t, Real t0) <–>
  ( abs(po–po0) <= S()∗(t–t0) );                   /∗ safety property ∗/
Bool initobs(Real po, Real po0, Real so) <–>     /∗ initial state ∗/
  ( po=po0 & so=0 );
```

Contract compliance. *Contracts* are dL formulas that tie together a component's behavior with its interface. The contract compliance proof obligations [14, Def. 5] that users have to show follow the structure in the obstacle contract compliance example below:

$$\big(\overbrace{S \geq 0 \wedge D \geq 0}^{\Omega} \wedge \overbrace{s_o = 0 \wedge p_o = p_o^-}^{\phi_o}\big) \rightarrow \big[\big(\overbrace{p_o^- := p_o}^{\Delta_o}; \overbrace{s_o := *; ?\left(0 \leq s_o \leq S\right)}^{ctrl_o};$$
$$t^- := t; \underbrace{\{t' = 1, p_o' = s_o\}}_{plant_o}; \underbrace{\mathtt{skip}}_{in_o}; \underbrace{\mathtt{skip}}_{cp_o}\big)^*\big] \underbrace{\left|p_o - p_o^-\right| \leq S \cdot \left(t - t^-\right)}_{\Pi_o^{out}} \tag{8}$$

From global facts Ω about system parameters as well as initial conditions ϕ (here: initially the obstacle is stopped $s_o = 0$ and the port memory bootstrapped $p_o = p_o^-$), users must prove that all runs of a component ensure the interface output guarantees Π^{out}. The component behavior is stitched together as a hybrid program from the component specification according to [14, Def. 5] as follows:

– Δ updates the port memory (here: remember the position of the obstacle)

- *ctrl* runs the component controller (here: choose a new obstacle speed s_o, but at most maximum speed S
- $t^- := t; \{t' = 1, plant\}$ measures time and describes the effect of the control decision (here: the obstacle moves according to the chosen speed)
- *in* reads values from the component input ports (here: `skip` since obstacle has no inputs)
- *cp* transfer values between the subcomponents of the current component (here: `skip` since the obstacle is not built from smaller components)

The obstacle contract compliance proof obligation (8) follows in a straightforward way from the ASCII definitions.

```
Problem  /* obstacle contract */
  t=t0 & globalFacts() & initobs(po,po0,so) -> [{
    deltaobs; ctrlobs; t0:=t; plantobs; inobs; cpobs;
   }*]safeobs(po,po0,t,t0)
End.
```

Robot contract compliance (9) assumes that the robot is stopped initially. It stores both initial port values (Δ_r) and nondeterministically chooses values for its input ports (in_r). The robot has no output ports, but guarantees that its local safety property ψ_r^{safe} holds, which ensures that the robot will never actively crash into the obstacle.

$$(\Omega \wedge \overbrace{\hat{p}_o = \hat{p}_o^- \wedge \hat{d} = \hat{d}^- \wedge s_r = 0 \wedge \varepsilon > 0)}^{\phi_r} \rightarrow [(\overbrace{\hat{p}_o^- := \hat{p}_o; \hat{d}^- := \hat{d}}^{\Delta_r}; ctrl_r; t^- := t; \{t' = 1, plant_r\}; \underbrace{\hat{p}_o := *; ?\pi_r^{in}(\hat{p}_o); \hat{d} := *; ?\pi_r^{in}(\hat{d})}_{in_r}; cp_r)^*](\underbrace{s_r > 0 \rightarrow \hat{p}_o \neq p_r}_{\psi_r^{safe}}) \tag{9}$$

Verifying contract compliance in KeYmaera X. In order to ensure contract compliance, both contracts must be formally verified. The proof automation of KeYmaera X can complete the component proofs of the running example fully automatically. But to illustrate our component-based approach we provide proof scripts to store the lemmas that are required for deriving system safety upon composition. Such proof scripts can be included as tactics in the archive entry, using the Bellerophon tactics language [8]. The KeYmaera X web user interface [12] supports point-and-click creation of proof scripts (see Section 3.3). The tactics language provides a number of predefined named tactics (e. g., `andL` – simplify a conjunction on the left-hand side of the sequent). Most of these tactics must be applied at certain positions of the sequent (e. g., `andR(1)` – apply the tactic to the first formula on the right-hand side of the sequent) and some require additional parameters (e. g., `loop(invariant,1)` – prove a non-deterministic repetition at position 1 using loop induction with the provided `invariant`). A *semicolon* concatenates tactics, i. e., `t1;t2` executes tactic `t2` after `t1`. If a proof requires the verification of multiple proof goals (e. g., loop induction requires the verification of three branches, i. e., `invariant` holds initially, `invariant` implies target property, `invariant` is inductive) a *less-than sign* indicates that the following *comma-separated* tactics are applied to the respective branches (e. g., `t1 <(t2,t3)` – after `t1`, tactic `t2` is applied to the first branch and `t3` is applied to the second branch).

The proof script to verify the obstacle contract is straightforward: First, the implication is resolved (`implyR(1)`). Then, loop induction uses the safety property as invariant, and the resulting branches (base case, use case and induction step) are proved by general proof automation (`master`). The statement `done({‘message‘},{‘id‘})` creates lemmas to verify each branch (can be referred to with the provided ID). We will need these lemmas later, when we apply our composition theorem.

```
Tactic "Proof Obstacle Contract Compliance (Create Lemmas)"
  implyR(1);
  loop({‘safeobs(po,po0,t,t0)‘},1); <(
    master; done({‘Base case done‘}, {‘Obstacle Base Case Lemma‘}),
    master; done({‘Use case done‘}, {‘Obstacle Use Case Lemma‘}),
    master; done({‘Step done‘}, {‘Obstacle Step Lemma‘}) )
End.
```

Composition To safely compose individual components, [14, Def. 6] introduces a quasi-parallel, associative and commutative *composition operation* (true parallel *plant* composition and coarse-grained *ctrl* composition without interleaving) to create systems from components. The composition operation is configurable with a *connection program* to account for different options how to transfer values between ports (e. g., lossless connection vs. estimation from sensors with some uncertainty). A notion of *compatibility* ensures that connections are made only between ports whose assumptions and guarantees fit.

Connection program. A user-defined connection program determines how values are passed between ports. For example, lossless, instantaneous connection[4] directly copies the value of an output port v to an input port $\hat{v}$: $con_{ll}(v) \equiv \hat{v} := v$. The corresponding HP `con` in ASCII syntax is listed below.

```
  HP con ::= { por:=po; };    /* connection program */
```

These user-defined connection programs must provably provide certain communication guarantees [14, Def. 7]: a connection program *con* must be executable ($\langle con \rangle true$) and its effect must be expressible in a first-order logic formula ζ by relating the connected ports without side effects ($[con]\zeta$). The direct assignment in a lossless, instantaneous connection is obviously executable and its effect is trivial equality between the port values, i. e., $\zeta_{ll}(\hat{v}, v) \equiv \hat{v} = v$. To ensure that the communication guarantees hold, both properties are formally verified from the ASCII syntax specifications below.

[4] See [14] for further examples of connection programs.

```
Problem
  <con;>true  /* connection program is executable */
End.
Tactic "Proof Connection Program Executable"
  master; done({'Executable'},{'Connection Executable Lemma'})
End.
```

The effect of a lossless, instantaneous connection is summarized with the formula `Bool zeta(Real por, Real po) <-> ( po=por );` and used in the communication guarantee effect proof below.

```
Problem
  [con;]zeta(por,po)      /* communication effect */
End.
Tactic "Proof Communication Effect"
  master; done({'Effect'}, {'Communication Effect Lemma'})
End.
```

Compatibility. When connecting ports, users must prove that the connections between the components are compatible, so that an output port exclusively supplies values that are accepted by the connected input port, so formally: $\zeta(v^-, \hat{v}^-) \land \Omega \to [con(v)]\big(\pi_j^{out}(\hat{v}) \to \pi_i^{in}(v)\big)$. In our robot example, the sole connection transfers the obstacle position to the robot (the desired speed suggestion port remains unconnected), so we get one compatibility proof obligation:

$$\Big(\big(\overbrace{p_o^- = \hat{p}_o^-}^{\zeta_{ll}(p_o^-,\hat{p}_o^-)}\big) \land \big(\overbrace{S \geq 0 \land D \geq 0}^{\Omega}\big)\Big) \to \underbrace{[\hat{p}_o := p_o]}_{con_{ll}(p_o)}\Big(\underbrace{|p_o - p_o^-| \leq S \cdot \big(t - t^-\big)}_{\pi_o^{out}(p_o),\ \text{see (6)}} \to \underbrace{|\hat{p}_o - \hat{p}_o^-| \leq S \cdot \big(t - t^-\big)}_{\pi_r^{in}(\hat{p}_o),\ \text{see (7)}}\Big)\ . \tag{10}$$

The compatibility proof obligation (10) includes the input assumption (7) and output guarantee (6) for the connected port, and the communication guarantee ζ about the connection program *con*. The compatibility proof obligation is verified automatically in KeYmaera X and the resulting lemma is again stored.

```
Definitions
  Bool zeta(Real por, Real po) <->     /* communication guarantee */
       ( po=por );
  Bool ObsPosOut(Real po, Real po0, Real t, Real t0)  <->
       ( abs(po-po0) <= S()*(t-t0) );         /* output guarantee */
  Bool ObsPosIn(Real por, Real por0, Real t, Real t0) <->
       ( abs(por-por0) <= S()*(t-t0) );      /* input assumption */
End.
Problem                            /* compatibility proof obligation */
```

```
  zeta(por0,po0) & globalFacts() ->
    [por:=po;](ObsPosOut(po,po0,t,t0) -> ObsPosIn(por,por0,t,t0))
End.
```

After composing the components from the running example using lossless, instantaneous communication, we get a composite system component and interface. The system component (11) sequentially composes individual controllers and executes plants in parallel. Internally the obstacle position is transmitted using the connection program. The system component's interface (12) contains the remaining input port of the robot and indicates that previous values must be stored for connected and unconnected ports alike.

$$\mathrm{C}_{sys} = \Big(\overbrace{(ctrl_{rc}; ctrl_r; ctrl_o)}^{ctrl_{sys}}, \overbrace{(plant_r, plant_o)}^{plant_{sys}}, \overbrace{(con_{ll}(\hat{p}_o))}^{cp_{sys}}\Big) \tag{11}$$

$$\mathrm{I}_{sys} = \Big(\underbrace{\{\hat{d}\}}_{V^{in}}, \underbrace{(\hat{d} \mapsto \left|\hat{d} - \hat{d}^-\right| \leq D)}_{\pi^{in}}, \underbrace{\{\}}_{V^{out}}, \underbrace{()}_{\pi^{out}}, \underbrace{\{p_o^-, d^-, \hat{p}_o^-, \hat{d}^-\}}_{V^-}\Big) \tag{12}$$

Composition retains safety After verifying local contracts (8) and (9), compatibility among connected ports (10) and the communication guarantee for the applied connection program, the remaining question is whether the safety property holds for the composed system. [14, Thm. 1] ensures that, starting from an initial state where both initial conditions hold, all runs of the composed system satisfy the safety properties, here:

$$\vDash (t = t^- \wedge \Omega \wedge \phi_1 \wedge \phi_2 \wedge \zeta) \rightarrow$$
$$[(\Delta; ctrl; t^- := t; \{t' = 1, plant\}; in; cp)^*] \left(\psi_1^{safe} \wedge \Pi_1^{out} \wedge \psi_2^{safe} \wedge \Pi_2^{out}\right) \tag{13}$$

The proof of this theorem is implemented constructively in KeYmaera X to *automatically* derive a proof tactic that will verify that the system contract (13) holds from component proofs. This proof tactic takes the lemmas of component contract compliance, communication guarantees, and compatibility as input.

Summary The user specifies and verifies contract compliance (based on component behavior and interface) for each component according to [14, Def. 5], defines a connection program with verified communication guarantees according to [14, Def. 7], and discharges the compatibility proof obligation for connected ports according to [14, Def. 8]. The results of verified contract compliance, communication guarantees, and compatibility are stored as named lemmas and fed to our tool to retrieve a tactic for proving safety of the composed system.

3.3 Web User Interface

KeYmaera X[5] comes with a web-based user interface (UI) [12] that supports the verification of dL formulas. If a .kyx file is loaded, the UI creates a proof

[5] http://www.keymaeraX.org

attempt for each contained archive entry. If an archive entry includes a tactic, the UI calls the underlying KeYmaera X proof engine and attempts to prove the problem formula. Otherwise, the user can start a manual proof attempt and choose which sequence of tactics to apply. KeYmaera X automatically records the selected proof steps and exports the resulting proof script. Additionally, the UI allows textual input of proof steps. The proof scripts presented in this paper can be constructed with the UI, exported, and then fed into the component-based verification tool.

4 Code Generation

For model debugging and testing purposes, KeYmaera X provides (*unverified*) *code generation* from the dL data structures. The code generation tools translate hybrid programs, which are often nondeterministic (nondeterministic assignments, choices, and repetitions), into deterministic C code. This translation requires to mimic nondeterminism with the deterministic language features of C. We aim for a translation that preserves safety, which means that we want the behavior of the resulting C program be *one of* the behaviors of the hybrid program, but need not necessarily preserve all possible behaviors of the hybrid program (such refinements are verifiable for hybrid programs with differential refinement logic [11], but here we bridge different languages). The translation is systematic but not verified; especially the translation of real arithmetic into floating point arithmetic does not preserve the semantics: verified compilation to machine code and interval arithmetic computations is supported through the VeriPhy pipeline [4].

Component-based hybrid systems typically model the interaction of controllers with their environment, and therefore combine ownsystem components (e. g., the robot) with environment components (e. g., obstacles). These components are fundamentally different in nature, which is reflected in their implementation: We generate *control code* from the controller models of the ownsystem components, and *monitoring code* to monitor whether or not the actual physical environment behaves according to the assumptions made in the model. Monitoring code can also be generated from the controller models of the ownsystem components, which is useful to sandbox untrusted control code (e. g., highly optimized controllers, or controllers that use learning).

4.1 Control Code

Static semantics. In order to declare C data structures to represent constant system parameters and state variables, we analyze the static semantics of hybrid programs via their free and bound variables [16]. Uninterpreted function symbols and variables that are free but not bound in the program (and thus only read) become declared as system parameters. Bound variables that are chosen nondeterministically are interpreted as system inputs and must be provided, e. g., by the user, sensors, or through optimization procedures. All other bound variables are state variables and computed by the generated control code. Interpreted function symbols min, max, and abs are translated to library function calls.

```
typedef struct {     /* constant parameters that never change */
  long double ep;    /* reaction time bound */
  long double S;     /* obstacle maximum speed */
  long double D;     /* remote control maximum speed */
} parameters;
```

```
typedef struct {    /* component input variables */
  long double po;   /* obstacle position */
  long double dr;   /* desired speed (advisory) */
} inputs;
```

```
typedef struct {    /* state variables of controller and plant */
  long double pr;   /* robot position */
  long double sr;   /* robot speed */
  long double t;    /* time */
  long double por0; /* previous obstacle position (port memory) */
  long double dr0;  /* previous desired speed (port memory) */
  long double t0;   /* plant start time */
} state;
```

The sensors and actuators are accessed from the generated control code through callback functions. Sensors provide the latest sensor values for each of the input variables, whereas actuators take as input the current and unmodifiable state including the control decisions of the controller.

```
typedef inputs (*Sensors)(void);
typedef void (*Actuators)(state const* const);
```

These data structures are used as arguments in the generated control code with signature

```
state ctrl(state const* const current,
           parameters const* const params,
           Sensors sense,
           Actuators actuate )
```

The implementation of the control code is derived using the translation rules discussed next.

Overview. In order to resolve nondeterminism in the execution paths of hybrid programs, the generated code uses backtracking and therefore operates on a copy of the state and tracks the success of statements in the following data structure:

```
struct { state state; bool success; } result = {
  .state = current, .success = false }
```

The effects (collected in `result.state`) of unsuccessful statements are reverted before attempting alternative executions with backtracking (e. g., the second branch of a nondeterministic choice). We use the notation `C(·)` to denote compilation of a `dL` formula, term, or hybrid program into C.

Terms and formulas. The translation of terms and formulas from `dL` to C rests on an appropriate representation of real arithmetic in machine-executable floating point arithmetic. Even though unsound, here we opt for code readability and translate variables, number literals, and terms into double-precision floating point representations. Provable safety needs a sound translation of real arithmetic to floating/fixed point interval arithmetic as in VeriPhy [4].

Real-valued number literals n are compiled to double-precision floating point literals C(n) $\rightsquigarrow$ `nL`, read-only constant variables c are compiled to system parameters C(c) $\rightsquigarrow$ `params->c`, nondeterministically chosen variables x are compiled to the corresponding field in the inputs data structure C(x) $\rightsquigarrow$ `sense()->x`, and deterministically computed state variables x are compiled to the corresponding field in the result state data structure C(x) $\rightsquigarrow$ `result.state.x`. Terms are straightforward translations of the basic arithmetic operators, with exponentials and interpreted functions `abs`, `min`, and `max` translated to C math library calls. Formulas are straightforward translations of the boolean operators.

Deterministic statements. Assignments $x := e$ are translated directly to C: `result.state.x = C(e); result.success = true;` where the success variable indicates that the assignment succeeded. Tests $?F$ are straightforward assignments to the program success flag: `result.success = C(F);` leaves the state unchanged. Conditionals if (F) a else b, which are syntactic sugar for $(?F; a) \cup (?\neg F; b)$, are straightforward C conditionals: `if (C(F)) C(a) else C(b)`. Sequential composition $a; b$ executes hybrid program b after successful execution of HP a, which is translated to `C(a); if (result.success) { C(b); }`.

Nondeterministic statements. Nondeterministic assignments in hybrid programs are often used to model inputs to the controller, such as sensor values or optimization procedures when the controller has a variety of different options that are considered similar in terms of safety (e. g., controller is free to choose any acceleration if speeding up is safe). We therefore interpret nondeterministic assignments $x := *$ as control inputs to the generated controller by reading from the sensor input callback function: `result.state.x = sense()->x;`.

Nondeterministic choice. Nondeterministic choices are resolved eagerly by executing the *first successful branch*, regardless of whether later branches would also be executable at the current state. This means, that source hybrid programs should be structured such that functionality that is important to achieve desired goals occur on the left-hand side branches of nonderministic choices (e. g., robot favors following remote control input over emergency stopping).

An integral feature of KeYmaera X is that tactics and proofs operate symbolically on terms, formulas, and programs and can therefore be used to transform their shape in a provably correct way. Here, we prepare the code transformation with a tactic that proves that executing eagerly is a safe implementation of a nondeterministic choice as follows: $[\alpha \cup \beta]P \rightarrow [(\alpha; s := 1 \cup s := 0);\ \text{if } (\neg(s = 1))\ \beta]P$. The success indicator s is a fresh variable mentioned neither in programs α, β nor the condition P. Note that the semantics of dL "discards" unsuccessful executions (e. g., if a test fails), which we implement by remembering the starting state, tracking the success of statements within programs, and resetting the state upon unsuccessful execution in the following translation template for the hybrid program $(\alpha; s := 1 \cup s := 0);\ \text{if } (\neg(s = 1))\ \beta$.

```
{ state reset = result.state;
  C(a);
  if (!result.success) result.state = reset;
}
if (!result.success) {
  state reset = result.state;
  C(b);
  if (!result.success) result.state = reset;
}
```

Nondeterministic repetition. Similar to nondeterministic choice, we execute a nondeterministic repetition a^* until the first time the loop body program a is executed successfully: `while (!result.success) { C(a) }`.

Differential equations. Differential equations are implemented by handing the current state to actuators, so just `actuate(result); result.success = true;`. In doing so, we exploit the structure of component contract compliance [14], which allows a single differential equation at a specific position in the component contract and hence guarantees that backtracking occurs only locally in control code and in the communication between sub-components, but never undoes the effect of differential equations.

4.2 Monitoring

In a readily composed system, only a subset of the components may describe system functionality and can be implemented by generating control code from the component model with above methods. Other components may describe environment behavior or agents (e. g., obstacles) that we cannot control. Nevertheless, the composed system makes crucial safety-relevant assumptions about the behavior of such environment components. Safety is provably guaranteed through ModelPlex [13] when we monitor the environment behavior for compliance with their model. The necessary monitoring conditions in dL are generated by proof [13]. The methods introduce above are applicable to generate executable monitor code. Here, we briefly discuss the nature of monitor conditions in C for debugging purposes and readability. The generated monitor conditions test for compliance between component environment model and true environment. Provably correct monitoring is available in our VeriPhy pipeline, which generates provably correct machine code [4] that witnesses compliance between a model and reality.

The monitor conditions generated by ModelPlex are formulas that describe how a current state of the program is related to the next state through the program statements. A monitor condition $P(x, x^+)$ compares two states for compliance with the program statements: variables in the previous state are identified by the program variable name x, whereas those in the next state are marked x^+. For example, the robot controller ctrl_r (if (Drive) $s_r := \hat{d}$ else $s_r := 0$) is transformed by proof into a monitor condition $(\text{Drive} \wedge s_r^+ = \hat{d}) \vee (\neg\text{Drive} \wedge s_r^+ = 0)$ and represents assignments, tests, and nondeterministic choices as described below.

Assignments and tests. Some of the statements are unambiguous computations that represent points in the state space: for example, assignment $s_r := 0$ only admits a specific new speed (0), and therefore the monitor condition will test speed with the formula $s_r^+ = 0$. Others represent regions in the state space and may refer to both states (e. g., testing whether stopping is necessary with Drive).

Nondeterministic choices. Execution in a hybrid program splits into several paths at each nondeterministic choice, which result in disjunctions in the generated monitoring conditions (e. g., one disjunct describing how the robot chooses speed when following the remote control command, another disjunct for stopping). For example, the robot controller ctrl_r results in the disjunction $(\text{Drive} \wedge s_r^+ = \hat{d}) \vee (\neg\text{Drive} \wedge s_r^+ = 0)$, which means that the monitor is satisfied either when the robot chooses a new speed $s_r^+ = \hat{d}$ when also the condition Drive was satisfied, or else chooses a new speed $s_r^+ = 0$. For provable runtime safety, the main result of ModelPlex [13] guarantees that the system at runtime enjoys the guarantees of the offline proof when the monitor conditions evaluate to true at runtime. The ModelPlex monitor conditions can be translated into C code using the control code transformations introduced above.

4.3 Using the Generated Code

The generated control code implements a control step that reads from sensors and writes to actuators, which are attached through hooks (signature repeated here for easy reference):

```
state ctrl(state const* const current,
           parameters const* const params,
           Sensors sense,
           Actuators actuate)
```

In the following code snippet, we implement the `Sensors` and `Actuators` callback functions to read from and modify the state of the system components according to a manually implemented simple simulator of the obstacle and robot motion, and a random desired speed suggestion of the remote control.

```
parameters sysParams = { .ep = 0.5L, .S=3.0L, .D=5.0L };
rcState rc           = { .dr = 0.0L };
obsState obs         = { .po = 10.0L, .so = 0.0L };
state robot = {
  .pr = 0.0L, .sr = 0.0L, .t = 0.0L,
  .por0 = 10.0L, .dr0 = 0.0L, .t0 = 0.0L
};

inputs readRobotInputs() {
  return ((inputs){ .po = obs.po, .dr = rc.dr });
}

void actuateRobotOutputs(state const* const out) {
  /* random time st in [0..ep] */
  long double st = randomDouble(0.0L, sysParams.ep);
  /* simulate robot */
  robot.sr = out->sr;
  robot.t = st;
  robot.pr = robot.pr + robot.sr*st;
  /* simulate obstacle: random so in [0..S] */
  obs.so = randomDouble(0.0L, sysParams.S);
  obs.po = obs.po + obs.so*st;
  /* simulate remote control: random dr in [0..D] */
  rc.dr = randomDouble(0.0L, sysParams.D);
}
```

In a real system, the sensor and actuator callback functions are used to interact with the system hardware through sensor and actuator drivers.

5 Related Work

Formal verification tools based on hybrid-automata, like SpaceEx [6], allow verification of parallel composed hybrid I/O automata, but either rely on soundness-critical extensions to support compositional reasoning or analyze a readily composed system in a monolithic fashion, which may lead to state space explosion. Modeling and simulation tools, such as Ptolemy [5] and Modelica [7], support component-based modeling of hybrid systems, but do not support verification or exploit the modular structure of the system for simulation. The model-driven development tool Simulink/Stateflow comes with a design verifier that allows a model analysis, but does not create proofs like KeYmaera X. Additional approaches allow formal verification of Simulink/Stateflow by translation into a other formalisms (e. g., timed automata [18], hybrid automata [1]), but thus rely on soundness-critical extensions. Conversely, [2] allow transformation of formally verified hybrid automata to Simulink/Stateflow, with follow-up code generation through (unverified) syntactic transformations. The CyPhyML paradigm [10], developed as part of the OpenMETA tool chain, supports component-based modeling of CPS. However, for analysis purposes (e. g., simulation, verification), the models must be translated to other formalisms, like Modelica.

In summary, the component-based verification and code-generation functionality of related tools extend the soundness-critical core of those tools. We, in contrast, analyze components separately and prove system safety from component safety in tactics [8] outside the soundness-critical core. For code generation, we follow a pragmatic approach similar to existing tools and use systematic transformations, but we strive for performing program transformations with proofs when possible: we exploit tactics to adapt the shape of hybrid programs from their nondeterministic nature to a deterministic implementation with proofs to prepare for emitting C code. The imperative nature of hybrid programs then makes inspecting the remaining syntactic transformations straightforward.

6 Conclusion

This paper demonstrates a tool chain that exploits the strict separation of tactics from the soundness-critical core in KeYmaera X in order to build component-based verification techniques. The input language, definitions, and lemma mechanism of KeYmaera X are a useful basis when structuring models into separate components and to combine separate component safety and connection compatibility proofs into a safety proof of a readily composed system.

For implementation of components and easy integration with embedded and cyber-physical systems, we take a pragmatic approach by generating C code with informally inspected program transformations. These transformations are designed to emit control code for components that describe implementable system functionality, and monitor code to monitor for violation of the assumptions that models make about environment behavior or agents outside the control of our own system. The transformations also address the gap between the nondeterministic operators of hybrid programs and the deterministic implementations in a way that preserves safety. We demonstrate how tactics can help implement these transformations by proof on the level of hybrid programs to prepare for emitting C code, so that only the final transformation step from hybrid programs into C is unverified. Fully verified machine code with a chain of theorem provers is available in KeYmaera X with VeriPhy [4].

References

1. Agrawal, A., Simon, G., Karsai, G.: Semantic translation of Simulink/Stateflow models to hybrid automata using graph transformations. Electr. Notes Theor. Comput. Sci. **109**, 43–56 (2004)
2. Bak, S., Beg, O.A., Bogomolov, S., Johnson, T.T., Nguyen, L.V., Schilling, C.: Hybrid automata: from verification to implementation. STTT (2017)
3. Bohrer, B., Rahli, V., Vukotic, I., Völp, M., Platzer, A.: Formally verified differential dynamic logic. In: Bertot, Y., Vafeiadis, V. (eds.) Certified Programs and Proofs - 6th ACM SIGPLAN Conference, CPP 2017. pp. 208–221. ACM, New York (2017)

4. Bohrer, B., Tan, Y.K., Mitsch, S., Myreen, M.O., Platzer, A.: VeriPhy: Verified controller executables from verified cyber-physical system models. In: Grossman, D. (ed.) Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018. pp. 617–630. ACM (2018)
5. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S.R., Xiong, Y.: Taming heterogeneity - the Ptolemy approach. Proceedings of the IEEE **91**(1), 127–144 (2003)
6. Frehse, G., Guernic, C.L., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011. Proceedings. LNCS, vol. 6806, pp. 379–395. Springer (2011)
7. Fritzson, P., Engelson, V.: Modelica - A unified object-oriented language for system modelling and simulation. In: Jul, E. (ed.) Object-Oriented Programming, 12th European Conference. LNCS, vol. 1445, pp. 67–90. Springer Berlin Heidelberg (1998)
8. Fulton, N., Mitsch, S., Bohrer, B., Platzer, A.: Bellerophon: Tactical theorem proving for hybrid systems. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) ITP. LNCS, vol. 10499, pp. 207–224. Springer (2017)
9. Fulton, N., Mitsch, S., Quesel, J.D., Völp, M., Platzer, A.: KeYmaera X: An axiomatic tactical theorem prover for hybrid systems. In: Felty, A., Middeldorp, A. (eds.) International Conference on Automated Deduction, CADE'15, Berlin, Germany, Proceedings. LNCS, vol. 9195, pp. 527–538. Springer, Berlin (2015)
10. Lattmann, Z., Nagel, A., Levendovszky, T., Bapty, T., Neema, S., Karsai, G.: Component-based modeling of dynamic systems using heterogeneous composition. In: Hardebolle, C., Syriani, E., Sprinkle, J., Mészáros, T. (eds.) Proceedings of the 6th International Workshop on Multi-Paradigm Modeling, MPM@MoDELS 2012. pp. 73–78. ACM (2012)
11. Loos, S.M., Platzer, A.: Differential refinement logic. In: Grohe, M., Koskinen, E., Shankar, N. (eds.) LICS. pp. 505–514. ACM, New York (2016)
12. Mitsch, S., Platzer, A.: The KeYmaera X proof IDE: Concepts on usability in hybrid systems theorem proving. In: Dubois, C., Mery, D., Masci, P. (eds.) 3rd Workshop on Formal Integrated Development Environment. EPTCS, vol. 240, pp. 67–81. Open Publishing Association (2016)
13. Mitsch, S., Platzer, A.: ModelPlex: Verified runtime validation of verified cyber-physical system models. Form. Methods Syst. Des. **49**(1-2), 33–74 (2016), special issue of selected papers from RV'14
14. Müller, A., Mitsch, S., Retschitzegger, W., Schwinger, W., Platzer, A.: Tactical contract composition for hybrid system component verification. STTT **20**, 615âĂŞ643 (2018), special issue for selected papers from FASE'17
15. Platzer, A.: Differential dynamic logic for hybrid systems. J. Autom. Reas. **41**(2), 143–189 (2008)
16. Platzer, A.: A complete uniform substitution calculus for differential dynamic logic. J. Autom. Reas. **59**(2), 219–265 (2017)
17. Platzer, A.: Logical Foundations of Cyber-Physical Systems. Springer, Switzerland (2018)
18. Yang, Y., Jiang, Y., Gu, M., Sun, J.: Verifying simulink stateflow model: timed automata approach. In: Lo, D., Apel, S., Khurshid, S. (eds.) Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016. pp. 852–857. ACM (2016)