A Hybrid Programming Language for Formal Modeling and Verification of Hybrid Systems

Eduard Kamburjan[1,3], Stefan Mitsch[2], and Reiner Hähnle[3]

1 **Department of Informatics, University of Oslo, Norway**
`eduard@ifi.uio.no`

2 **Computer Science Department, Carnegie Mellon University, USA**
`smitsch@cs.cmu.edu`

3 **Department of Computer Science, Technische Universität Darmstadt, Germany**
`haehnle@cs.tu-darmstadt.de`

Abstract

Designing and modeling complex cyber-physical systems (CPS) faces the double challenge of combined discrete-continuous dynamics and concurrent behavior. Existing formal modeling and verification languages for CPS expose the underlying proof search technology. They lack high-level structuring elements and are not efficiently executable. The ensuing modeling gap renders formal CPS models hard to understand and to validate. We propose a high-level *programming*-based approach to formal modeling and verification of hybrid systems as a hybrid extension of an Active Objects language. Well-structured hybrid active programs and requirements allow automatic, reachability-preserving translation into differential dynamic logic, a logic for hybrid (discrete-continuous) programs. Verification is achieved by discharging the resulting formulas with the theorem prover KeYmaera X. We demonstrate the usability of our approach with case studies.

2012 ACM Subject Classification Distributed programming languages, Model verification and validation, Logic and verification, Timed and hybrid models

Keywords and phrases Active Objects, Differential Dynamic Logic, Hybrid Systems

Digital Object Identifier 10.4230/LITES.xxx.yyy.p

Received Date of submission. **Accepted** Date of acceptance. **Published** Date of publishing.

Editor LITES section area editor

1 Introduction

Networked cyber-physical systems (CPS) are a main driving force of innovation in computing, from manufacturing to everyday appliances. But to design and model such systems poses a double challenge: first, their *hybrid* nature, with both continuous physical dynamics and complex computations in discrete time steps. Second, their *concurrent* nature: distributed, active components (sensors, actuators, controllers) execute simultaneously and communicate asynchronously. It is notoriously difficult to get CPS models right. *Formal* modeling languages, including hybrid automata [5], hybrid process algebra [27], and logics for hybrid programs [65], can be used to formally verify properties of CPS. Contrary to simulation frameworks, such as Ptolemy [71] or Simulink, however, these languages were *designed for verification* and are based on concepts of the underlying verification technology: automata, algebras, formulas. Their minimalist syntax lacks standard structuring elements of programming languages such as types, scopes, methods, complex commands, futures, etc. Thus it is hard to adequately represent concurrently executing, communicating, hybrid components with *symbolic* data structures and computations, for example, servers or cloud applications.

Moreover, "low-level" models are hard to *validate*, i.e. to ensure that a CPS model reflects the designer's intention, because these formalisms are not (efficiently) executable. To bridge the modeling gap we propose a high-level *programming*-based approach to formal modeling and verification of hybrid systems.

The basis of our approach is an *Active Objects* (AO) language [29] called ABS [48]. AO languages combine OO programming with strong encapsulation as well as asynchronous, parallel execution. Their concurrency model permits to decompose concurrent execution into sequential execution in a compositional manner. We chose ABS for its formal semantics, its open source implementation tool chain, and its demonstrated scaling on massively distributed systems [75], but our approach is applicable to other AO languages. ABS is efficiently executable via compilation to Erlang and was used to model complex, real-world systems for cloud processing [3], virtualized services [49], data processing [56], and railway operations [53]. However, it lacks the capability to model hybrid systems. The *first main contribution* of this paper is the design of the *Hybrid ABS* (HABS) language, a conservative (syntax and semantics preserving) extension of ABS, generalizing the Active Objects paradigm to *Hybrid Active Objects* (HAO): AO with continuous dynamics. Obviously, it is necessary to accordingly extend the formal semantics of ABS and its runtime environment. This is our *second main contribution*. Our *third main contribution* is the implementation of HABS and a formal verification tool for it.

Our approach to formal verification of HABS programs is based on reachability-preserving translation into an existing verification formalism for hybrid programs. We choose differential dynamic logic ($d\mathcal{L}$) [66, 68, 69], as implemented in the KeYmaera X system [36], because it is based on an imperative programming language that is a good match for the sequential fragment of HABS and verification in $d\mathcal{L}$ has been demonstrated to scale to realistic systems (e.g., [47]). The translation from HABS to $d\mathcal{L}$ involves to decompose a given HABS verification problem into a set of independent *sequential* $d\mathcal{L}$ problems. This is possible, because we impose an interaction pattern for communication on HABS that is less restrictive than available component-based techniques [64], yet is general enough to permit intuitive and concise modeling of relevant case studies. The identification of this pattern, the generation of $d\mathcal{L}$ verification conditions, and a reachability preservation theorem constitute our *fourth main contribution*.

The overall approach is illustrated in Fig. 1: A CPS is modeled as an HABS program with the aim to analyze its properties statically. One formulates desired properties as invariants that are formally verified to hold under certain assumptions. Before verification is attempted, the model is *validated* by executing it in the runtime environment to ensure that it behaves as intended. A visualization component helps to analyze behavior over time. Subsequently, the verification claim is automatically decomposed and translated into a set of $d\mathcal{L}$ verification problems discharged in KeYmaera X (optionally, formally verified runtime monitors [63] and formally verified machine code is available from KeYmaera X through VeriPhy [18]). Both, unexpected runtime behavior and failed verification attempts, serve to fix the model and/or the claimed properties.

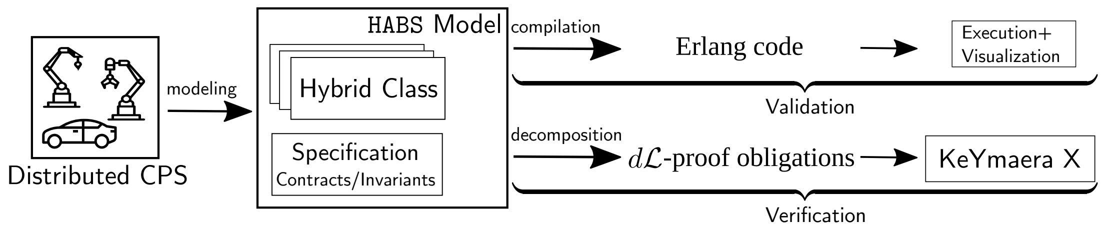


Figure 1 Structure of HABS workflow.

The paper is structured as follows. Sect. 2 gives an informal example of an HABS model with a distributed water tank controller. Sect. 3 formally defines syntax and semantics of HABS. Sect. 4 describes modeling patterns. Sect. 5 gives theoretical background on $d\mathcal{L}$, the translation into $d\mathcal{L}$, the decomposition theorem, and tells how to prove correctness. It also contains a distributed controller case study. Finally, Sect. 6 discusses related and future work and concludes.

2 Distributed Hybrid Systems by Example

Active Objects [29] are objects that realize actor-based concurrency [44] with futures [28] and cooperative scheduling: Active Objects communicate via asynchronous method calls. On the caller side, each method invocation generates a future as a handle to retrieve the call's result, once it is available. The caller may synchronize on that future, i.e. suspend and wait until it is resolved. At most one process is running on an Active Object at any time. That process suspends when it encounters the synchronization statement **await** on an unresolved future or a false Boolean condition. Once the guard becomes true, the process may be re-scheduled. All fields are strictly object-private.

Running a Hybrid Active Objects (HAO) model of a CPS can be pictured as follows: each object is capable of modeling a physical object, for example, a water tank. It may declare *physical* behavior via ordinary differential equations (ODEs) over "physical" fields, as well as *discrete* behavior via class and method declarations that can be used to control physical behavior. Once an HAO starts executing, the values of the physical fields evolve, governed by their ODEs, even when the controller is idle. This models the intuition that a physical system evolves independently of any observers and controllers.

Object orientation allows natural modeling of hybrid systems: continuous behavior is attached to an *object*, not a process. Processes realize discrete control behavior related to sensors and controllers. Specifically, the controller methods of an object may wait to execute until a certain physical state is reached (event-triggered control, for example, "tank is nearly full"). This "sensing" is modeled with getter methods of physical fields. Obviously, for validation the HABS runtime system must solve the differential equations in the physical model to determine the time point when such a waiting controller can start at the earliest; for verification, ODEs need not be solvable; they are analyzed with invariant-based techniques [67, 70]. Another communication pattern for controllers—time-triggered control—is provided by fixed sampling durations. More complex control patterns can be realized by waiting until the result of a subcomputation, i.e. a future, is ready.

Whenever a control process is activated, it can modify the physical state through actuators (for example, close a valve). In consequence, there are no timed race conditions, but the physical state might be changed by any process at the time it is scheduled. Actuation is modeled with setter methods of physical fields. Execution of control methods is assumed to take no physical time, unless explicitly modeled to do so.

Generally, a CPS can be modeled by several HAOs that communicate with each other via asynchronous method calls, for example, modeling a central controller. Often a controller object has no associated physical behavior; vice versa, an object that models physics, may not contain any control, but only sensor and actuator methods.

We demonstrate HAOs using three variants of water tank models. The first model, **TankMono**, is a single water tank that keeps its water level between two thresholds. It is modeled as a single object that integrates control and physics. The second model, **TankTick**, is also a single water tank, but it is modeled with two separate objects for tank and controller. The final model, **TankMulti**, is a distributed system of n **TankMono** tanks that, in addition to the local threshold, maintain a global threshold over the sum of all local water levels.

```
1 interface ISingleTank {
2  /*@ ensures 3 <= outLevel() <= 10 @*/
3  Real outLevel();
4  /*@ ensures −1/2 <= outDrain() <= 1/2 @
   */
5  Real outDrain();
6 }
7 /*@ requires 4 <= inVal <= 9 @*/
8 class CSingleTank(Real inVal)
9       implements ISingleTank {
10 /*@ invariant
11      3 <= level <= 10
12     & −1/2 <= drain <= 1/2
13     & (drain<0−>level>3)
14     & (drain>0−>level<10) @*/
15  physical {
16   Real level = inVal : level' = drain;
17   Real drain = -1/2  : drain' = 0;
18  }
19  Unit run() { this!ctrl(); }
20  Unit ctrl() {
21   await diff (level<=3 & drain<=0) | (level>=10 & drain>=0);
22   if (level <= 3) drain =  1/2;
23   else            drain = -1/2;
24   this.ctrl();
25  }
26  Real outDrain() { return this.drain; }
27  Real outLevel() { return this.level; }
28 }
```

Figure 2 TankMono: A water tank as a single HAO.

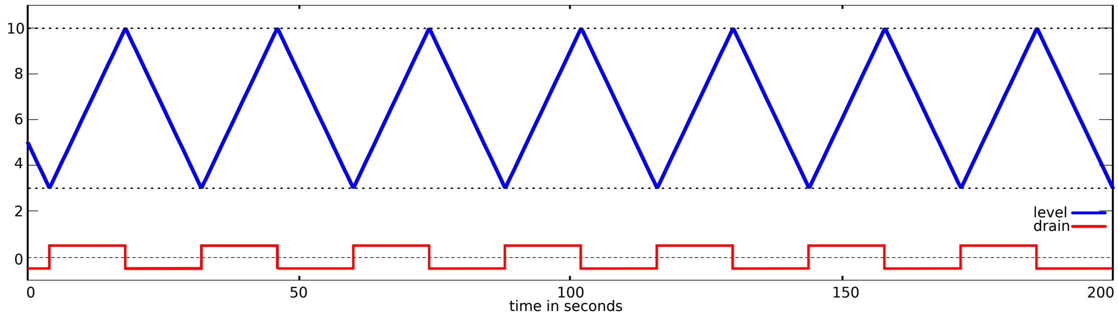


Figure 3 Simulation Output of **TankMono** with `inVal` = 5.

2.1 Base System: TankMono

Fig. 2 shows an HAO model of a water tank whose **physical** section makes it either fill with $\frac{1}{2}l/sec$ or drain at the same rate, according to the initial values and governing ODEs of the `level` and `drain` fields. Method `ctrl()` realizes a control loop that switches the `drain` field between those states so that the water level stays between $3l$ and $10l$. The controller `ctrl` waits until the water level reaches the upper or lower limit, i.e. until the condition in Fig. 2, Line 21 holds. Depending on the case, it changes the state and calls itself recursively.

The JML style [20] comments in Fig. 2 contain an assumption on the initial state of `inVal` and a conjectured safety invariant and conjectured output guarantees that, in this case, can be proven: if the initial level is between $4l$ and $9l$, then it always stays between $3l$ and $10l$. Note that Lines 13–14 express a safety invariant that must be *shown* to be true, rather than control conditions. Intuitively, Line 13 expresses the property that the tank won't drain below a threshold (`level` > 3) even if water is leaking from it (`drain` < 0). Similarly, Line 14 expresses that the tank won't overflow (`level` < 10) even if water is pumped into the tank (`drain` > 0). Prior to formal verification of this property one typically runs tests to see whether the model behaves as intended. Our implementation allows to simulate and visualize an HAO model. The graph in Fig. 3 shows the behavior of a `CSingleTank` object instantiated with `inVal` $= 5$. In Sect. 5 we show how the class is translated into $d\mathcal{L}$ and how to prove the safety invariant in KeYmaera X for *any* object created with a parameter that satisfies the precondition. The only methods exposed to clients in the interface are `outDrain()` and `outLevel()`.

2.2 Discrete Controller: TankTick

The `ctrl()` method in **TankMono** corresponds to a perfect sensor/controller that physically reacts to the water level and drain. **TankTick** splits controller and sensor into two objects and uses a clock to read the water level at certain intervals. This corresponds to a closed-loop control system with a discrete-time controller that samples the plant behavior.

```
interface Tank {
  /* requires -1/2 <= newD <= 1/2; */
  Unit inDrain(Real newD);
  /* ensures 3 <= outLevel() <= 10; */
  Real outLevel();
}
class CTank(Real inVal) implements Tank {
  physical {
    Real level = inVal : level' = drain;
    Real drain = -1/2 : drain' = 0;
  }
  Unit run() { }
  /* requires newD > 0 -> level <= 9.5 */
  /* requires newD < 0 -> level >= 3.5 */
  /* timed_requires inDrain < 1 */
  Unit inDrain(Real newD) { drain = newD; }
  Real outLevel() { return level; }
}
```

```
/* requires 0 < tick < 1 & inVal > 3.5*/
class FlowCtrl(Tank t, Real tick, Real inVal) {
/* invariant (drain > 0 -> level <= 9.5)
       & (drain < 0 -> level >= 3.5) */
  Real drain = -1/2;
  Real level = inVal;

  Unit run() { this!ctrlFlow(); }

  Unit ctrlFlow() {
    await duration(tick,tick);
    level = t.outLevel();
    if (level <= 3.5) drain =  1/2;
    if (level >= 9.5) drain = -1/2;
    t!inDrain(drain);
    this.ctrlFlow();
  }
}
```

Figure 4 **TankTick**: A water tank modeled as two HAOs. Invariant and precondition of `CTank` are as in Fig. 2.

Fig. 4 shows a water tank realized by a controller `FlowCtrl` and a `Tank` implementation `CTank`. The tank has an in-port (setter) method `inDrain()` and an out-port (getter) method `outLevel()`. It has no active *discrete* behavior on its own (the `run` method is empty), but its state changes nonetheless due to the *continuous* **physical** block. The `FlowCtrl` controller's fields `drain`, `level` are its *local copies* of the state of the tank: `CTank.drain`, `CTank.level` are different fields from `FlowCtrl.drain`, `FlowCtrl.level`, respectively, residing in different objects. The `ctrlFlow()` method first updates `level`, decides on the state of `drain`, then pushes the (possibly changed) state of `drain` to the tank. No time passes in the controller, which ensures that the copied fields are synchronized at the end of the round. As the `Tank`'s fields are not directly accessible by the `FlowCtrl` instance, it is not possible to wait on the `Tank`'s `level` with an **await diff** statement. Instead, the controller uses **await duration** to run every `tick` seconds: `tick` is the sampling time of the controller.

The `Tank` interface specification declares an input requirement and a guarantee on returned values. The input requirement of the `inDrain()` specification is a constraint on the input parameter `newD`; specifically, it means that the tank can only be instructed to fill if there is sufficient capacity left (similar for draining). The initial requirement is sufficient to establish the controller's invariant, which in turn ensures that the tank's requirements are met. The `timed_requires` clause stipulates that `inDrain()` is called at least once per second, which suffices for the output guarantee. Fig. 5 shows example output. We stress that all calls to `Tank` methods are *asynchronous*.

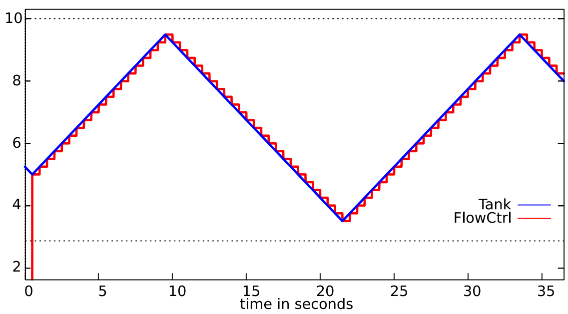


Figure 5 Simulation Output of **TankTick** with `inVal` $= 5$ for $30s$.

```
class CControl(List<ISingleTank> tanks,
              Real totalLower,
              Real totalHigher,
              Real tick)
implements IControl {
  Unit run() {
    await duration(tick, tick);
    Real total = 0;
    List<ISingleTank> lower  = list[];
    List<ISingleTank> higher = list[];
    foreach ( next in tanks ) {
      Real val = next.outLevel();
      Real dir = next.outDrain();
      if (dir < 0 && val > 3)
        lower  = Cons(next, lower);
      if (dir > 0 && val < 10)
        higher = Cons(next, higher);
      total = total + val;
    }
    if (total <= totalLower+1)
       foreach ( lnext in lower  )
         lnext!inDrain(1/2);
    if (total >= totalHigher-1)
       foreach ( hnext in higher )
         hnext!inDrain(-1/2);
    this.run();
  }
}
```

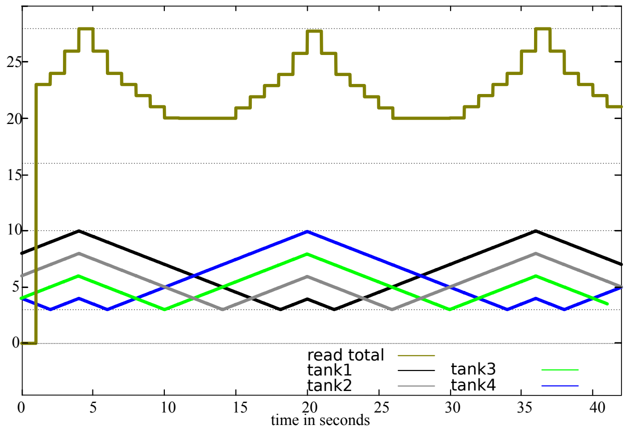


Figure 6 **TankMulti**: A controller for n **TankMono** instances and an example simulation output. Interface omitted.

2.3 Distributed Tank Control: TankMulti

Consider a system where n water tanks are monitored by a central controller that aims to keep the sum of all water levels between some thresholds. The code in Fig. 6 shows a controller that monitors a list of `ISingleTank` (Fig. 2) instances. Each `tick` seconds the central controller iterates over the list of tanks and if their combined level is almost at the upper threshold, the controller drains all water tanks with rising levels (analogously for the lower threshold). Single water tanks still ensure that their local thresholds are observed. To allow the `CControl` instance to manipulate the `ISingleTank` instances, we add the following method to `CSingleTank` (and an analogous method to the interface):

```
/* requires newD > 0 -> level < 10  */
/* requires newD < 0 -> level >  3  */
/* requires -1/2 <= newD <= 1/2 */
Unit inDrain(Real newD) { this.drain = newD; }
```

Contrary to the contract in **TankTick**, we do not need to specify how frequently the method is called, because this information is available in the guard of the `ctrl` method of the instances. The recursive call at the end of `ctrl` ensures that there is always one process executing `ctrl` for each instance of `FlowCtrl`.

The graph in Fig.6 shows the simulation output for four water tanks with different initial values. The upper thresholds are managed by the distributed controller and the water tanks cooperatively: Only tanks 1 and 4 reach their local upper thresholds, the others are drained by the distributed controller to maintain the global threshold. The lower local thresholds are managed locally, the lower global threshold is never reached.

2.4 Futures

Future-based communication allows to decouple the call of a method from retrieving its result. For example, consider the code in Fig. 7. Class `Node` can perform some complex and time consuming computations on behalf of class `Client`. To enable load balancing the client has only a reference to an interface `Server`, which relays its request. The `Server` performs basic load balancing by a round-robin scheduling on a list of nodes. It then returns to the issuing client the future of the relayed request *without having to wait* for the computation to finish (Line 17). The client can then retrieve the future (Line 7) to synchronize on it without blocking the interface server (Line 8).

```
1 class Node {
2    Real compute_internal(Real r1, Real r2, Real r3){ ...   }
3 }
4 class Client(Server s){
5    Unit run(){
6       Fut<Fut<Real>> ffr = s!compute(1,2);
7       Fut<Real> fr = ffr.get;
8       Real r = fr.get;
9       ...
10    }
11 }
12 class Server(Queue<Node> internal, Real param){
13    Fut<Real> compute(Real r1, Real r2){
14       Node n = internal.pop();
15       Fut<Real> fr = n!compute_internal(r1,r2,param);
16       internal.push(n);
17       return fr;
18    }
19 }
```

Figure 7 An example for load balancing using futures. Interfaces omitted.

3 Hybrid Active Objects

An informal description of the intended semantics of Hybrid Abstract Objects in the Hybrid Abstract Behavioral Specification (`HABS`) language was provided in Section 2. The present section gives a formal account of its syntax and semantics. `HABS` is an extension of the Active Object language `ABS` [48]. `ABS` itself extends standard OO concepts as follows:

Encapsulation. All fields are strictly object-private.
Cooperative Scheduling. Active Objects cannot be preempted: a process running in an object may not be interrupted by other processes, unless the active process suspends itself or terminates.
Asynchronous Calls, Futures. All method calls to other objects are asynchronous. Every call not only generates a process on the callee side, but a future that points to that process. A process may pass around a future or synchronize with it to read the return value of the associated process once it has terminated.

As a *Timed* Active Object language, `HABS` also features:

Simulation Time. `HABS` allows to manipulate *simulation time* by explicitly advancing (and reading) an internal clock with specific statements. Simulation time is independent of the wall time.

3.1 Syntax

The syntax of HABS is given by the grammar in Fig. 8 and explained in the following section. With `e` we denote standard expressions over fields `f`, variables `v` and operators `|`, `&`, `>=`, `<=`, `+`, `-`, `*`, `/`. Types `T` are all interface names, type-generic futures **Fut**`<T>`, lists `List<T>`, `Real`, `Int`, `Unit` and `Bool`. We also assume the usual functions for lists, etc.

Prgm ::= $\overline{\text{ID}}$ $\overline{\text{CD}}$ Main	ID ::= **interface** `I` [**extends** $\overline{\texttt{I}}$]?{$\overline{\text{MS}}$}	Programs, Interfaces
Main ::= {s?}		Main
CD ::= **class** `C` [**implements** $\overline{\texttt{I}}$]? [($\overline{\texttt{T f}}$)]?{Phys? $\overline{\text{FD}}$ $\overline{\text{Met}}$ Run?}		Classes
Run ::= `Unit run()` {s}	FD ::= `T f = e`	Run Method and Fields
Phys ::= **physical** {$\overline{\text{DED}}$}	DED ::= `Real f = e : f' = e`	Physical Block
MS ::= `T m(`$\overline{\texttt{T v}}$`)`	Met ::= MS {s;**return** `e`;}	Signatures, Methods
s ::= **while** `(e)` {s} | **if** `(e)` {s} [**else** {s}]? | s;s		
| **await** g | [`T`? `e`]? = rhs		Statements
g ::= **duration**`(e,e)` | **diff** `e` | `e?`		Guards
rhs ::= `e` | **new** `C(`$\overline{\texttt{e}}$`)` | `e.`**get** | `e!m(`$\overline{\texttt{e}}$`)`		RHS Expressions

Figure 8 HABS grammar. `T` ranges over types, `I` over interfaces and `C` over classes. Differential expression `de` are normal expressions extended with a derivation operator `e'`.

A program contains a main method Main, interfaces $\overline{\text{ID}}$ and classes $\overline{\text{CD}}$. Interfaces are standard, the main method contains a list of object creations. Classes can have parameters $\overline{\text{Tf}}$, these are fields being initialized during object creation. Classes have fields $\overline{\text{FD}}$, methods $\overline{\text{Met}}$, an optional run method Run to start a process, and an optional physical block Phys that declares physical fields. A declaration of a physical field is a field declaration followed by a differential equation. A differential equation is an equation between two differential expressions, which are standard expressions extended with a derivation operator `e'` for $\frac{de}{dt}$. HABS supports explicit autonomous differential equations. The differential expressions and the field initialization form an initialized ordinary differential equation, e.g., `Real f = 0: f' = 5-f`. Note that `f = 0` specifies the initial value of `f`, whereas the differential equation `f' = 5-f` is phrased in terms of the time-varying value of `f`, so models logarithmic growth towards `f = 5`.

Methods and statements are mostly standard, we focus on HAO-specific constructs. Methods are called asynchronously with `e!m(`$\overline{\texttt{e}}$`)`, i.e., after the call, the caller continues execution without waiting for the callee to finish. Instead, the caller generates a *future*. A future identifies the call and can be passed around by the caller. A process interacts in two ways with a future: either by awaiting its result with **await** `e?` on the guard `e?`, or by reading its value with `e.`**get**. Statements `e.`**get** block the reading *object*—no other process may run on it. In contrast, statements **await** `g` release the process control over the object while waiting for the guard `g` to hold. The guard is either a future guard `e?`, a differential guard **diff** `e`, or a timed guard **duration**`(e1,e2)`. The future guard `e?` awaits the result of future `e`, the differential guard **diff** `e` suspends the process until the expression `e` evaluates to true, and the timed guard **duration**`(e1,e2)` suspends the process for at least `e1` time units[1]. The notation `T v = o.m()` is short for **Fut**`<T> f = o!m(); T v = f.`**get**; (a call followed by a synchronization).

---

[1] The parameter `e2` is used by certain scheduling policies [16], and is not relevant for HABS.

3.2 Semantics of `HABS`

`HABS` extends the structural operational semantics (SOS) for Timed `ABS` [16] in three aspects: (i) it includes physical behavior in the object state; (ii) determines whether a differential guard holds and, if not, when it will at the earliest; (iii) updates the state whenever time passes. This affects only expression evaluation and auxiliary functions. *No new SOS rule is needed.* In the following we extend the core of the `ABS` SOS semantics [16] to hybrid systems.

3.2.1 States

The state of an object has three parts: (i) a store ρ that maps (physical and non-physical) fields to values, and the variables of the active process[2] to values; (ii) ODE, the differential equations from its physical block; (iii) F, the set of current solutions of ODE[3]. A solution f is a function from time to a store which only contains the physical fields. The set F may change, because the ODEs are solved as an initial-value problem with the current state of the physical fields as the initial values. For each $f \in F$ and each physical field $\mathtt{f}$ the following holds: $f(0)(\mathtt{f}) = \rho(\mathtt{f})$, i.e., the initial value $f(0)(\mathtt{f})$ of physical field $\mathtt{f}$ is the current value $\rho(\mathtt{f})$ in the store ρ. We denote the solutions of ODE with initial values from ρ by $\mathsf{sol}(ODE, \rho)$. We define runtime configurations formally:

$$tcn ::= \mathsf{clock}(\mathtt{e})\ cn \qquad cn ::= cn\ cn \mid fut \mid msg \mid ob$$

$$ob ::= (o, \rho, \underset{\cdots\cdots}{ODE, F}, prc, \overline{prc}) \qquad msg ::= \mathsf{msg}(o, \overline{\mathtt{e}}, f)$$

$$prc ::= (\tau, f, \mathtt{rs}) \mid \bot \qquad \mathtt{rs} ::= \mathtt{s} \mid \textbf{suspend};\mathtt{s} \qquad fut ::= \mathsf{fut}(f, \mathtt{e})$$

Figure 9 Runtime Syntax of `HABS`.

▶ **Definition 1** (Runtime Configuration [16])**.** The runtime syntax of `HABS` is summarized in Fig. 9: f ranges over future identities, o over object identities, ρ, τ over stores, i.e., assignments from fields or variables to values. A timed configuration has a clock clock with the current time, as an expression of `Real` type and an object configuration cn. An object configuration cn consists of messages msg, futures fut, objects ob, and can be composed $cn\ cn$ (as usual, composition is commutative and associative). A message $\mathsf{msg}(o, \overline{\mathtt{e}}, f)$ records callee o, passed parameters $\overline{\mathtt{e}}$ and the generated future f. A future configuration $\mathsf{fut}(f, \mathtt{e})$ connects the future f with its return value $\mathtt{e}$. An object $(o, \rho, F, ODE, prc, \overline{prc})$ has an identifier o, an object store ρ, the current solutions F, an active process prc and a queue of inactive processes. ODE is taken from the class declaration. A process is either terminated $\bot$ or has the form $(\tau, f, \mathtt{rs})$: the process store τ with current state of the local variables, its future f, and the statement $\mathtt{rs}$ left to execute. The runtime syntax also allows the **suspend** statement, which is used to deschedule a process. Dotted underlined elements are an extension of `HABS` relative to ABS (also in Fig. 10 below).

Given a process store τ and an object store ρ we use $\sigma = \rho \circ \tau$ to denote the state of both fields and local variables. We first define the evaluation of expressions and guards.

[2] Recall that the active process executes the ABS methods, it does not relate to physical behavior.
[3] The solutions computed relative to the initial values (state) at the last suspension.

3.2.2 Evaluation of Expressions

Expressions `e` are evaluated with a function $[\![\texttt{e}]\!]_\sigma^{F,t}$ over a store σ and a set of solutions F at t time units in the future. The semantics of expressions containing physical fields is as follows.

▶ **Definition 2** (Semantics of Expressions)**.** Let F be the set of solutions. Given a store σ, we can check whether F is a model of an expression `e` after t time units. Let $\texttt{f}_p$ be a physical field and $\texttt{f}_d$ a non-physical field of o. The semantics of fields $\texttt{f}_p$, $\texttt{f}_d$, unary operators $\sim \in \{\texttt{!},\texttt{-}\}$ and binary operators $\oplus \in \{\texttt{|},\texttt{\&},\texttt{>=},\texttt{<=},\texttt{+},\texttt{-},\texttt{*},\texttt{/}\}$ is defined as follows:

$$[\![\texttt{f}_d]\!]_\sigma^{F,t} = \sigma(\texttt{f}_d) \qquad [\![\texttt{f}_p]\!]_\sigma^{F,t} = \begin{cases} v & \text{if } \forall f \in F.\ v = f(t)(\texttt{f}_p) \\ \infty & \text{otherwise} \end{cases}$$

$$[\![\sim\texttt{e}]\!]_\sigma^{F,t} = \sim[\![\texttt{e}]\!]_\sigma^{F,t} \qquad [\![\texttt{e}_1 \oplus \texttt{e}_2]\!]_\sigma^{F,t} = [\![\texttt{e}_1]\!]_\sigma^{F,t} \oplus [\![\texttt{e}_2]\!]_\sigma^{F,t}$$

Outside differential guards, only the evaluation in the current state $[\![\texttt{e}]\!]_\sigma^{F,0}$ is needed, which is $\rho(\texttt{f}_p)$ from $f(0)(\texttt{f}_p)$ and this expression is never ∞. We identify $[\![\texttt{e}]\!]_\sigma^{F}$ and $[\![\texttt{e}]\!]_\sigma$with $[\![\texttt{e}]\!]_\sigma^{F,0}$.

3.2.3 Evaluation of Guards

The semantics of an **await** `g` statement is to suspend until the guard holds, i.e. until $[\![\texttt{g}]\!]_\sigma^{F}$ evaluates to true. For example, a duration guard **duration**`(e1,e2)` evaluates to true if $[\![\texttt{e1}]\!]_\sigma^{F} \leq 0$. Defining the semantics of guards requires two operations: An extension of the *evaluation function* that returns true if the guard holds and the *maximal time elapse* mte_σ^F returning the time t that may elapse before the guard evaluates to true, or ∞ if it never does.

First we define $mte(\texttt{e})$: the *maximal* time that may elapse without missing an event is the *minimal* time needed by the system to evolve into a state where the guard is guaranteed to hold. This yields also the semantics of the guard itself.

▶ **Definition 3** (Semantics of Differential Guards)**.** Let F be the set of solutions of object o in state σ. Then we define:

$$mte_\sigma^F(\textbf{diff}\ \texttt{e}) = \underset{t \geq 0}{\textbf{argmin}}\ \left([\![\texttt{e}]\!]_\sigma^{F,t} = \text{true}\right)$$

diff `e` is evaluated to true if no time advance is needed:

$$[\![\textbf{diff}\ \texttt{e}]\!]_\sigma^{F,0} = \text{true} \iff mte_\sigma^F(\textbf{diff}\ \texttt{e}) = 0$$

If `e` contains no continuous variable then the differential guard semantics and the evaluation of expressions in Def. 2 coincides with condition synchronization and expression evaluation in the standard `ABS` semantics [48].

3.2.4 Transition System

Fig. 10 gives the most important rules for the semantics of a single object, the omitted rules are given in [16]. Rules (1)–(3) define the semantics of process suspension. An **await** statement suspends the current process and gives other processes in the queue q a chance to run, even if its guard is evaluated to true. Suspension is modeled in rule (1) simply by introducing a **suspend** statement in front of the **await**.[4] Rule (2) realizes a **suspend** statement by moving the current

[4] We follow the original ABS semantics, where suspension is handled with a separate **suspend** statement for reasons of uniformity—in principle, rules (1)+(2) could be combined.

$$\textbf{(1)} \quad \big(o,\rho,\underline{ODE,F},(\tau,f,\textbf{await}\ \texttt{g;s}),q\big) \;\rightarrow\; \big(o,\rho,\underline{ODE,F},(\tau,f,\textbf{suspend};\textbf{await}\ \texttt{g;s}),q\big)$$

$$\textbf{(2)} \quad \big(o,\rho,\underline{ODE,F},(\tau,f,\textbf{suspend};\texttt{s}),q\big) \;\rightarrow\; \big(o,\rho,\underline{ODE,\mathsf{sol}(ODE,\rho)},\bot,q\circ(\tau,f,\texttt{s})\big)$$

$$\textbf{(3)} \quad \big(o,\rho,\underline{ODE,F},\bot,q\circ(\tau,f,\textbf{await}\ \texttt{g;s})\big) \;\rightarrow\; \big(o,\rho,\underline{ODE,F},(\tau,f,\texttt{s}),q\big) \quad \text{if } [\![\texttt{g}]\!]_{\rho\circ\tau} = \text{true}$$

$$\textbf{(4)} \quad \big(o,\rho,\underline{ODE,F},(\tau,f,\texttt{v = e;s}),q\big) \;\rightarrow\; \big(o,\rho,\underline{ODE,F},(\tau[\texttt{v}\mapsto[\![\texttt{e}]\!]_{\rho\circ\tau}],f,\texttt{s}),q\big) \quad \text{if } \texttt{e} \text{ contains no call or } \textbf{get}$$

$$\textbf{(5)} \quad \big(o,\rho,\underline{ODE,F},(\tau,f,\textbf{return}\ \texttt{e;}),q\big) \;\rightarrow\; \big(o,\rho,\underline{ODE,\mathsf{sol}(ODE,\rho)},\bot,q\big)\ \mathsf{fut}\big(f,[\![\texttt{e}]\!]_{\rho\circ\tau}\big)$$

$$\textbf{(6)} \quad \big(o,\rho,\underline{ODE,F},(\tau,f,\texttt{v = e}_1\texttt{.}\textbf{get}\texttt{;s}),q\big)\ \mathsf{fut}\big(f,\texttt{e}_2\big) \;\rightarrow\; \big(o,\rho,\underline{ODE,F},(\tau,f,\texttt{v = e}_2\texttt{;s}),q\big) \quad \text{if } [\![\texttt{e}_1]\!]_{\rho\circ\tau} = f$$

$$\textbf{(7)} \quad \big(o,\rho,\underline{ODE,F},(\tau,f,\texttt{v = e!m(e}_1,\ldots\texttt{e}_n\texttt{;s}),q\big) \rightarrow \big(o,\rho,\underline{ODE,F},(\tau[\texttt{v}\mapsto\tilde{f}],f,\texttt{s}),q\big)\ \mathsf{msg}\big([\![\texttt{e}]\!]_{\rho\circ\tau},([\![\texttt{e}_1]\!]_{\rho\circ\tau},\ldots,[\![\texttt{e}_n]\!]_{\rho\circ\tau}),\tilde{f}\big) \quad \text{where } \tilde{f} \text{ is fresh}$$

Figure 10 Selected Rules for `HABS` objects.

process to the object's queue. As explained in Sect. 3.2.3, upon reactivation of a suspended process we must ensure its guard to be true, relative to the solution of *ODE* with *initial values at suspension time*. Therefore, rule **(2)** also recomputes the solutions F. Rule **(3)** can then re-activate a process beginning with an **await** statement, simply by checking whether its guard evaluates to true at current time (advancing time in timed configuration is explained below). An analogous rule (not shown in Fig. 10) activates a process with any other non-**await** statement. Rule **(4)** evaluates an assignment to a local variable. The rule for fields is analogous. Rule **(5)** realizes a termination (with solutions of the ODEs) and **(6)** a future read. Finally, **(7)** is a method call, the rule for transforming a message into a process is straightforward.

For configurations, there are two rules, shown in Fig. 11. Rule **(i)** realizes a step of some object without advancing time, Only if **(i)** is not applicable, i.e. all ABS processes are blocked, rule **(ii)** can be applied. It computes the global maximal time elapse *mte* and advances the time in the clock and all objects. In particular, it decreases syntactically the timed guards.

$$\textbf{(i)} \quad \mathsf{clock}(t)\ \ cn\ \ cn_1 \rightarrow \mathsf{clock}(t)\ \ cn_2\ \ cn_1 \qquad \text{with } cn \rightarrow cn_2$$

$$\textbf{(ii)} \quad \mathsf{clock}(t)\ \ cn \rightarrow \mathsf{clock}(t+\tilde{t})\ \ adv(cn,\tilde{t}) \qquad \text{if } \textbf{(i)} \text{ is not applicable and } mte(cn) = \tilde{t} \neq \infty$$

Figure 11 Timed Semantics of `HABS` configurations.

Fig. 12 shows the auxiliary functions and includes the full definition of *mte*. Note that *mte* is not applied to the currently active process, because, when **(1)** is not applicable, it is currently blocking and, thus, cannot advance time. *The characteristic feature of hybrid objects is that their physical state changes when time advances, even when no process is active.* This is expressed in the semantics by a function $adv(\sigma,t)$ which takes a state σ, a duration t, and advances σ by t time units. For non-hybrid Active Objects $adv(\sigma,t) = \sigma$. There, the function is needed only to modify the process pool of an object for scheduling, not its state, and is used exactly as in [16].

The *adv* auxiliary function handles uniqueness w.r.t. the solutions of the ODE at the points in

$$
\begin{aligned}
mte(cn_1\ \ cn_2) &= \mathbf{min}(mte(cn_1), mte(cn_2)) && mte(msg) = mte(fut) = \infty \\
mte(o, \rho, ODE, F, prc, q) &= [\![\mathbf{min}(mte(q), \infty)]\!]_\rho && mte(\tau, f, \textbf{await}\ \texttt{g;s}) = [\![mte(\texttt{g})]\!]_\tau \\
mte(\tau, f, \texttt{s}) &= \infty \text{ if } \texttt{s} \neq \textbf{await}\ \texttt{g;}\tilde{\texttt{s}} && mte(\textbf{duration}(\texttt{e}_1,\texttt{e}_2)) = \texttt{e}_1 \\
mte^F_\sigma(\textbf{diff}\ \texttt{e}) &= \underset{t\geq 0}{\mathbf{argmin}}\ \left([\![\texttt{e}]\!]^{F,t}_\sigma = \text{true}\right) && mte(\texttt{e?}) = \infty
\end{aligned}
$$

$$
\begin{aligned}
&adv(cn_1\ \ cn_2, t) = adv(cn_1, t)\ adv(cn_2, t) \\
&adv(msg, F, t) = msg \qquad adv(fut, F, t) = fut \\
&adv((o, \rho, ODE, F, prc, q), F, t) = (o, adv(\rho, t), ODE, F, adv(prc, F, t), adv(q, F, t)) \\
&adv(\bot, F, t) = \bot \\
&adv((\tau, f, \texttt{s}), F, t) = (\tau, f, \texttt{s}) \text{ if } \texttt{s} \neq \textbf{await}\ \textbf{duration}(\texttt{e}_1,\texttt{e}_2)\texttt{;}\tilde{\texttt{s}} \\
&adv((\tau, f, \textbf{await}\ \textbf{duration}(\texttt{e}_1,\texttt{e}_2)\texttt{;s}), F, t) = (\tau, f, \textbf{await}\ \textbf{duration}(\texttt{e}_1\texttt{+}t,\texttt{e}_2\texttt{+}t)\texttt{;s}) \\
&adv(\sigma, t)(\texttt{f}) = \begin{cases} \sigma(\texttt{f}) & \text{if } \texttt{f} \text{ is not physical} \\ v & \text{if } \forall f \in F.\ v = f(t)(\texttt{f}) \end{cases}
\end{aligned}
$$

Figure 12 Auxiliary functions. Lifting to lists is not shown.

time where the solutions are accessed: Note that the solutions are handled as a set F: at time t function adv checks that all solutions coincide *at this point in time*. If this is not the case, or if no solution can be found by the implementation, a runtime error is thrown. Also, all solutions are computed without restrictions on the time domain (e.g., for how long they exits) because it is not known for how long the dynamics are followed at this point. Alternatively, one could either impose restrictions on the ODE to enforce uniqueness or non-deterministically choose one of the solutions.

We can now define *traces* of programs and objects.

▶ **Definition 4** (Traces). Given a program Prgm, we denote with $\mathsf{clock}(0)\ cn_0$ the initial state configuration [16]. A run of Prgm is a (possibly infinite) reduction sequence

$$\mathsf{clock}(0)\ cn_0 \rightarrow \mathsf{clock}(t_1)\ cn_1 \rightarrow \cdots$$

The trace θ_o of an object o in a run is an assignment from the dense time domain $\mathbb{R}^+$ to states. We say that $\mathsf{clock}(t_i)\ cn_i$ is the final configuration at t_i in a run, if any other timed configuration $\mathsf{clock}(t_i)\ \tilde{cn}_i$ is before it. Fig. 13 gives a formal definition.

$$
\theta_o(x) = \begin{cases}
\textit{undefined} & \text{if } o \text{ is not created yet} \\
\rho & \text{if } \mathsf{clock}(x)\ cn \text{ is the final configuration at } x \\
 & \quad \text{and } \rho \text{ is the state of } o \text{ in } cn \\
adv(\rho, F, x - y) & \text{if there is no configuration at } \mathsf{clock}(x) \\
 & \text{and the last configuration was at } \mathsf{clock}(y) \\
 & \text{with state } \rho \text{ and solutions } F
\end{cases}
$$

Figure 13 Extraction of a trace θ_o for an object o from a given run.

For any point in time x, the state of o is taken from the run, if a reduction step was made at x and o was already created. The third case in the definition is illustrated in Fig. 14: At time

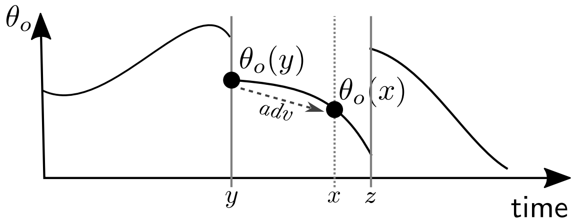


Figure 14 Illustration of the state at time x and two discrete states with clock(y) and clock(z).

points y and z, discrete steps are done, but none at x. The state $\theta_o(x)$ is extrapolated from the state $\theta_o(y)$ by following solutions from the last step at point y, if o is created.

3.3 The Component Fragment

We define a sublanguage of `HABS` called *Component `HABS`* (`CHABS`) to model component-style architectures with in- and out-ports, as well as dedicated controllers with a read-evaluate-write cycle. Syntactically, a class is a *component* if it can be derived from the syntax in Fig. 8 with the rule for Met replaced by the following:

```
  Met ::= MS [OPort | IPort | Ctrl]
OPort ::= {return this.f;}      IPort ::= {this.f = v; return Unit;}
 Ctrl ::= {sa; si; sc; so; this.m();}
   sa ::= await duration(e,e) | await diff e
   si ::= this.f = e.m() | si;si
   sc ::= while (e) {sc} | if (e) {sc} [else {sc}]? | sc;sc | T? e = e | e!m(ē)
   so ::= e!m(this.f) | so;so
```

Additionally, we demand that the only numerical data types used are `Int`, `Real`. Out-ports return the value of a field and in-ports copy a method parameter into a field. A controller method Ctrl has a timed or differential guard sa, followed by reads si from the out-port methods of other objects (recall that `this.f = e.m()` is a shortcut for an asynchronous call followed by a read, not a synchronous call), computations sc, and writes so to the in-ports of other objects. In the component fragment, we realize a component-based controller with a read-compute-write loop by restricting the `run` method of Fig. 8 to start a controller with an asynchronous call to an object's own controller method Ctrl and each controller ends with a recursive call to itself. The **TankMono** and **TankTick** models are `CHABS` models, the central controller in **TankMulti** is not. A controller method with a differential guard is an event-triggered controller, a controller with a timed guard a time-triggered controller.

We model instantaneous controllers in `CHABS`: once controller is scheduled (i.e., after its guards evaluates to true) no time can pass because all calls in Ctrl are to port methods that cannot block the caller and neither suspensions nor future reads are allowed.

3.4 Simulation

The implementation of `HABS` extends the `ABS` compiler [81] to compute solutions for differential guards, time elapse, and state advance. To compile differential guards correctly, it needs to compute mte^F_σ(**diff** `e`) (Def. 3).

The ODEs of a class cannot be changed at runtime and are, therefore, represented as a string in the class table. The simulator uses an external solver to solve initial value problems and minimize/maximize duration between events.

Solutions To compute solutions F, the ODEs and the current state of the physical fields are passed to Maxima [61] as an *initial value problem*. The solution is an equation system or an error. In its default setting, the simulator neither supports non-unique solutions nor non-solvable ODEs. The simulator, however, has the infrastructure to use solvers other than Maxima. This allows us to handle non-linear ODEs: by prefixing the **physical** block with `[1]`, the modeler can select the solver `ic1` (instead of the default `desolve`), which can handle non-linear systems.

Time elapse After solving the initial value problem, Maxima is invoked with a *minimization problem*: it minimizes the time t with the equation system representing F as the constraints (this corresponds to eager mode switching in a hybrid automaton). The result is then handled in the same way as a parameter to a timed guard by the runtime system. Once time has passed and the suspended process is reactivated, the physical fields are updated according to F. This uses the Maxima function `fmin_cobyla`.

State advance To implement the advance function *adv*, if the state of the object changes any physical field, the procedure used to compute time elapse is repeated for every currently suspended differential guard to accumulate the result.

The output files used to visualize a program execution are of the form $t_1, F_1, t_1, F_2, t_2, \ldots, F_n, t_n$. Here t_i are the points in time where the object schedules a process and F_i the function describing its physical behavior in the previous suspended state. Each time a differential guard is reactivated, not only its state is updated, but the solution F_{i+1} and the reactivation time t_{i+1} are written to the output. Each object has its own output file.

A Python script translates output files into a discrete dynamic graph in Maxima format which in turn calls `gnuplot` that is responsible for creating the graph. The graphs in this work are slightly beautified outputs.

4 Modeling with HABS

We give more examples of `HABS` models and discuss some design decisions in the language, as well as modeling patterns in `HABS` for common phenomena in hybrid system control.

4.1 Non-Linear Dynamics

`HABS` can handle non-linear ODEs and non-linear dynamics to the extent the backends support it. For an example, consider a resistor attached to an alternating current source that produces a sine-formed current. This is described by the class in Fig 15.

We use the non-linear solver of Maxima (by annotating `[1]`). This solver requires the input to satisfy certain syntax constraints, which entail the slightly awkward specification `r' = 0*t`. We must give an explicit ODE for each non-constant variable for KeYmaera X and as `HABS` requires an autonomous system, we add a clock variable `time` to express sine and cosine.

The example has a `run` method that illustrates validation. We check whether our simple model is in fact a resistor and adheres to the law $R = I/V$: Even before visualization, we can use simple command line output to check I/V by sampling every 1 second. The output for an instance `Resistor(5)` is shown in Fig. 15, where `Time(n)` is the symbolic time at the point of time when `now()` is evaluated. In the example this corresponds to seconds. As a next step, we can use the visualization to observe longer trends in Fig. 16, again for a `Resistor(5)`.

```
class Resistor(Real init) {
  [1] physical {
    /* format expected by Maxima */
    Real t = 0:    t' = 1;
    Real r = init: r' = 0*t;
    Real i = 0:    i' = cos(t);
    Real v = 0:    v' = r*cos(t);
  }
  Unit run() {
    await duration(1,1);
    println("step: " + toString(now()) +
      " with " + toString(v/i));
    if (timeValue(now()) < 60) this!run();
  }
}
```

```
step: Time(1) with 5
step: Time(2) with 4286450913523623 /
    ↪ 857290182704725
step: Time(3) with 1319812111494398 /
    ↪ 263962422298881
step: Time(4) with 1313376056981147 /
    ↪ 262675211396229
step: Time(5) with 295788950328081 /
    ↪ 59157790065616
step: Time(6) with 723097187038613 /
    ↪ 144619437407721
step: Time(7) with 758118670875062 /
    ↪ 151623734175013
step: Time(8) with 5
step: Time(9) with 5
...
```

Figure 15 A resistor attached to an AC-circuit and its sine-formed current

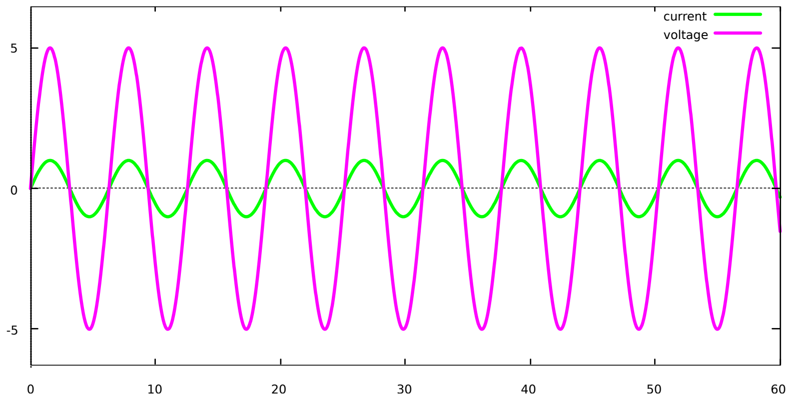


Figure 16 Example simulation output of a `Resistor(5)`

Finally, we can formally verify the behavior with our translation approach to KeYmaera X by removing the `run` method and, thus, transforming it into a `CHABS` component.

4.2 Delays and Imprecision

Communication is imperfect in realistic models. We demonstrate how to model two such imperfections, delays and imprecision, in `HABS`. We use a simple platooning example, where a follower car wants to follow a lead car at a certain distance. Follower cars are modeled in the `CHABS` class `FollowerCar` in Fig. 17. For simplicity, the minimal (`minDist`) and maximal distance (`maxDist`) to the lead car are independent of the speed and the controller sampling frequency, which means the follower car will not provably stay in the desired distance interval. The time consuming statement **await duration** can be used to model two kinds of delays:

1. Complex computations that take some time to finish.
2. Latency: By adding a time consuming statement as the last statement of a method before the return, one can model delays in a network.

```
class FollowerCar
  (Real inita, Real start,
   Real tick, Real minDist,
   Real maxDist, ICar leadCar)
   implements ICar {
     Real next = start + minDist;
     physical {
       Real a = inita : a' = 0;
       Real v = 0     : v' = a;
       Real x = start : x' = v;
     }
     Unit run() {
       this!ctrlObserve();
     }

     Unit ctrlObserve() {
       await duration(tick, tick);
       next = leadCar.getPosition();
       if(next - x <= minDist) a = a/2;
       if(next - x >= maxDist) a = a*2;
       this.ctrlObserve();
     }

     Real getPosition() {
       return x;
     }
   }
```

Figure 17 Simple platooning example for a follower car following safely behind a lead car

For example, we extend `getPosition()` in `FollowerCar` to model sensing latency as follows:

```
Real getPosition() {
  Real oldVal = x;
  await duration(1/10, 1/10);
  return oldVal;
}
```

Like `ABS`, `HABS` has access to a (uniformly distributed) random number generator. There are functions to generate other statistical distributions. This allows to model imprecision/uncertainty. The following method adapts `getPosition()` to model sensor uncertainty:

```
Real getPosition() {
  Real imp = (random(11) + 95)/100; // number between 0.95 and 1.05
  return this.level * imp;
}
```

4.3 Variability Modeling

One of the main advantages of using a mature programming language as a host for hybrid behavior is that we can use its structuring elements and concepts: `HABS` inherits the module system with import/export clauses[5], as well as the delta-oriented [73], feature-oriented [14] *product line* [8, 74] (DFPL) mechanisms of `ABS` [25] to model variability.

DFPLs define not a single model, but a set of models which are variants of each other. From a given *core* model, so-called code *deltas* define variants based on syntactic operations: removal, modification and addition of classes, methods and fields. A variant is obtained from the core model by applying modifications specified by the deltas to it.

To determine the relevant deltas, each delta has a set of features that activate its application. A feature of a variant corresponds roughly to one implemented feature of the modified model. A set of features is called a *product*. After selecting a product, the corresponding deltas are computed and applied, resulting in an `HABS` model without variability.

[5] Omitted from the language syntax in Sec. 3 for brevity.

```
delta Delay;
modifies class Cars.FollowerCar {
  modifies Real getPosition() {
    Real old = original();
    await duration(1/10,1/10);
    return old;
  }
}
delta Imprecision;
modifies class Cars.FollowerCar {
  modifies Real getPosition() {
    return original()*(random(11)+95)/100;

  }
}
```

```
delta CruiseControl;
modifies class Cars.FollowerCar {
  adds Real ccTick = this.tick*2;
  adds Unit cruise() {
    await duration(ccTick, ccTick);
    if ((v >= 5 || v <= 0) && a != 0) {
      a = 0;
    }
    this.cruise();
  }
  modifies Unit run() {
    original();
    this!cruise();
  }
}
```

```
productline PL1;
features FDelay, FImprecision, FCruiseControl;
delta CruiseControl when FCruiseControl;
delta Delay when FDelay;
delta Imprecision after Delay when FImprecision;
```

Figure 18 Product line based on Fig. 17 for variability in position readings and cruise control

We refrain from introducing the whole variability layer of `ABS` and refer to [25] for a detailed and formal introduction. Instead, we use the platooning example in Fig. 17 to demonstrate variability modeling in practice. The changes for imprecision and delay, as well as adding a cruise control system can be modeled as a product line. This allows to select the appropriate car product for a concrete system, as summarized in Fig. 18. The product line consists of three deltas (`Delay`, `Imprecision` and `CruiseControl`), three features (`FDelay`, `FImprecision` and `FCruiseControl`) and a knowledge base that defines which features select which delta (**delta** `D` **when** `F`) and in which order deltas are applied if they modify the same method (**delta** `D` **after** `D2`).

The delta `Delay` modifies class `Cars.FollowerCar`[6] and its method `getPosition()`. The modified method first calls the existing variant of the method via **original** and then waits before returning the value. Delta `Imprecision` is similar. Both deltas modify the same method. There are numerous desirable properties, and to make the product line outcome deterministic, we must fix the order in which methods are applied that modify the same method. Here, we demand that `Imprecision` is applied after `Delay`. Delta `CruiseControl` adds a field and method implementing a simple cruise control system. Deltas may also remove methods and fields (not shown here). In our example we represent each delta as a feature, and so any product that refers to a feature invokes its assigned delta. The deltas are applied *syntactically* before type checking. As a result, a standard `HABS` program is created. For example the product {`FDelay`} results in the code below.

452
```
class FollowerCar (...) implements ICar {
  ... // as above
  Real getPosition_core() { return x; }
  Real getPosition() { return this.getPosition_core()*(random(11) + 95)/100; }
}
```

[6] `Cars` is the module.

5 Formal Verification of HABS Models

As a prerequisite for formal verification of HABS, we briefly review *differential dynamic logic* ($d\mathcal{L}$) [68, 69] as implemented in the hybrid systems theorem prover KeYmaera X [36]. We then discuss translation from HABS to $d\mathcal{L}$, and sketch formal verification in $d\mathcal{L}$ with sequent proofs.

5.1 Background: Differential Dynamic Logic

Differential dynamic logic expresses the combined discrete and continuous dynamics of hybrid systems in a sequential imperative programming language called *hybrid programs*. Its syntax and informal semantics are in Table 1.

Table 1 Hybrid programs in $d\mathcal{L}$

Program	Informal semantics
$?\varphi$	Test whether formula φ is true, abort if false
$x := \theta$	Assign value of term θ to variable x
$x := *$	Assign any (real) value to variable x
$\{x' = \theta \ \& \ H\}$	Evolve ODE system $x' = \theta$ for any duration $t \geq 0$ with evolution domain constraint H true throughout
$\alpha; \beta$	Run α followed by β on resulting state(s)
$\alpha \cup \beta$	Run either α or β non-deterministically
α^*	Repeat α n times, for any $n \in \mathbb{N}$

Hybrid programs provide the usual discrete statements: assignment ($x := \theta$), non-deterministic assignment ($x := *$), test ($?\varphi$), non-deterministic choice ($\alpha \cup \beta$), sequential composition ($\alpha; \beta$), and non-deterministic repetition (α^*). A typical modeling pattern combines non-deterministic assignment and test (e.g., "$x := *; ?H$") to choose any value subject to a $d\mathcal{L}$ constraint H. Standard control structures are expressible, for example: (i) **if** H **then** α **else** $\beta \equiv (?H; \alpha) \cup (?\neg H; \beta)$, (ii) **if** H **then** $\alpha \equiv (?H; \alpha) \cup (?\neg H)$, (iii) **while** (H) $\alpha \equiv (?H; \alpha)^*; ?\neg H$.

For continuous dynamics, the notation $\{x' = \theta \,\&\, H\}$ represents an ODE system (derivative x' in time) of the form $x'_1 = \theta_1, \ldots, x'_n = \theta_n$. Any behavior described by the ODE stays inside the evolution domain H, i.e. the ODE is followed for a non-deterministic, non-negative period of time, but stops before H becomes false. For example, a basic model of the water level x in a tank draining with flow $-f$ is given by the ODE $\{x' = -f \,\&\, x \geq 0\}$, where the evolution domain constraint $x \geq 0$ means the tank will not drain to negative water levels. With a careful modeling pattern, ODEs can be governed by H so that one can react to events, without restricting or influencing the continuous dynamics modeled in the ODE [72]: The pattern $\{x' = \theta \,\&\, H\} \cup \{x' = \theta \,\&\, \widetilde{H}\}$ permits control intervention to achieve different behavior triggered by an event H. $\widetilde{H}$ is the *weak* complement of H: they share exactly their *boundary* from which both behaviors are possible. For example, $H \equiv x \leq 0$, $\widetilde{H} \equiv x \geq 0$.

The $d\mathcal{L}$-formulas φ, ψ relevant for this paper are propositional logic operators $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \rightarrow \psi$, $\neg\varphi$ and comparison expressions $\theta \sim \eta$, where $\sim \in \{<, \leq, =, \neq, \geq, >\}$ and θ, η are real-valued terms over $\{+, -, \cdot, /\}$. In addition, there is the $d\mathcal{L}$ modal operator $[\alpha]\varphi$. The $d\mathcal{L}$-formula $[\alpha]\varphi$ is true iff φ holds in all states reachable by program α. The formal semantics of $d\mathcal{L}$ [68, 69] is a Kripke semantics in which the states of the Kripke model are the states of the hybrid system. The semantics of a hybrid program α is a relation $[\![\alpha]\!]$ between its initial and final states. Specifically, $\nu \models [\alpha]\varphi$ iff $\omega \models \varphi$ for all states $(\nu, \omega) \in [\![\alpha]\!]$, so all runs of α from ν are safe relative to φ.

Proofs in $d\mathcal{L}$ are sequent calculus proofs on the basis of $d\mathcal{L}$ axioms. For example, validity of the $d\mathcal{L}$ formula $x \geq 0 \rightarrow [x := x + 1 \cup x := 2; \{x' = 3\}]x \geq 1$ over a simple program that either increments the value of x or continuously evolves x with a constant slope $x' = 3$ after setting the initial value of the differential equation with $x := 2$ is shown in the sequent proof below:

$$
\dfrac{
  \dfrac{\dfrac{*}{x \geq 0 \vdash x+1 \geq 1}\,\text{QE}}{x \geq 0 \vdash [x := x+1]x \geq 1}\,[:=]
  \qquad
  \dfrac{\dfrac{\dfrac{*}{x = 2 \vdash [\{x' = 3\}]x \geq 1}\,\text{dI}}{x \geq 0 \vdash [x := 2][\{x' = 3\}]x \geq 1}\,[:=],\text{hideL}}{x \geq 0 \vdash [x := 2; \{x' = 3\}]x \geq 1}\,[;]
}{
  \dfrac{x \geq 0 \vdash [x := x+1 \cup x := 2; \{x' = 3\}]x \geq 1}{\vdash x \geq 0 \rightarrow [x := x+1 \cup x := 2; \{x' = 3\}]x \geq 1}\,\rightarrow_R
}\,[\cup],\wedge_R
$$

Sequent proofs proceed bottom-up but validity transfers top-down, i.e., from the subgoals above the horizontal bar, the axiom or proof rule annotated to the left of the bar implies the sequent below the horizontal bar. In each step, assumptions are listed to the left of the $\vdash$, and the alternatives to prove to the right of it. The proof starts with step $\rightarrow_R$ to make the left-hand side $x \geq 0$ of the implication available as an assumption. Next, the non-deterministic choice step $[\cup]$ means that both choices must ensure the postcondition $x \geq 1$, so with conjunction splitting $\wedge_R$ we get two subgoals: a left subgoal for the increment program $x := x + 1$ and a right subgoal for the differential equation program. On the increment program branch, we execute the assignment in step $[:=]$ and the result follows by real arithmetic in step QE. On the differential equation branch, step $[;]$ splits the sequential composition into nested box modalities, and then step $[:=]$, hideL executes the assignment and weakens the now obsolete assumption $x \geq 0$. The branch closes by differential induction dI (intuitively, the dI step expresses that $x \geq 1$ stays true along the flow of the differential equation, see [67]). This concludes the example proof.

5.2 Formal Verification of Components

To establish system-wide properties, hybrid active objects must be shown to satisfy their class *invariants*, provided that the constraints expressed in the preconditions are met. We make this precise now. A *class specification* is a tuple (inv, pre, TReq, Req, Ens), where inv is the class invariant (annotated */* invariant ... */*, see Fig. 19, lines 20–21), a $d\mathcal{L}$ formula over the fields and parameters of the class; pre is the precondition (annotated to class declarations with */* requires ... */*, see Fig. 19, line 17), a $d\mathcal{L}$ formula over the initial values of fields and class parameters. TReq is the set of timed input requirements for in-port methods (annotated with */* timed_requires ... */*, see Fig. 19, line 12): $d\mathcal{L}$ formulas over a dedicated program variable with the method's name. Req is the set of input requirements for in-port methods (annotated with */* requires ... */*, see Fig. 19, lines 2, 10–11): $d\mathcal{L}$ formulas over fields and method parameters. Ens is the set of output guarantees for out-port methods (annotated with */* ensures ... */*, see Fig. 19, line 4): $d\mathcal{L}$ formulas over a dedicated program variable with the method's name.

```
1  interface Tank {
2   /* requires -1/2 <= newD <= 1/2; */
3   Unit inDrain(Real newD);
4   /* ensures 3 <= outLevel <= 10; */
5   Real outLevel();
6  }
7
8  class CTank(Real inVal)
9       implements Tank {
10  /* requires newD > 0 -> level <= 9.5  */
11  /* requires newD < 0 -> level >= 3.5 */
12  /* timed_requires inDrain < 1 */
13  Unit inDrain(Real newD) { ... }
14  ...
15 }
16
17 /* requires 0 < tick < 1 & inVal > 3.5*/
18 class FlowCtrl(Tank t, Real tick,
19               Real inVal) {
20 /* invariant (drain > 0 -> level <= 9.5)
21           & (drain < 0 -> level >= 3.5) */
22 ...
23 }
```

Figure 19 Annotations in the TankTick model, repeated from Fig. 4

To verify a class C against a class specification, both are translated into $d\mathcal{L}$-formula (1) that expresses safety.

$$\texttt{assumptions}_{\texttt{C}} \rightarrow \left[(\texttt{code}_{\texttt{C}};\ \texttt{plant}_{\texttt{C}})^{*}\right] \texttt{safety}_{\texttt{C}} \tag{1}$$

The placeholders $\texttt{assumptions}_\texttt{C}$, $\texttt{code}_\texttt{C}$, $\texttt{plant}_\texttt{C}$, and $\texttt{safety}_\texttt{C}$ (defined formally in Sect. 5.3 below) encode class C and its specification ($\mathsf{inv}, \mathsf{pre}, \mathsf{TReq}, \mathsf{Req}, \mathsf{Ens}$) as follows: The formula $\texttt{assumptions}_\texttt{C}$ is the conjunction of pre and conditions on variables that keep track of time. As usual in controller verification, the program repeats a control part $\texttt{code}_\texttt{C}$ followed by the continuous behavior $\texttt{plant}_\texttt{C}$. The condition $\texttt{safety}_\texttt{C}$ must hold after an arbitrary number of iterations. It combines inv with input requirements of in-port methods of referred objects and guarantees of own out-port methods.

Even though formula (1) $\texttt{safety}_\texttt{C}$ is a postcondition that must hold only in the final states of the system, we stress that this means at *every real time point* during the continuous dynamics, because ODEs advance for a non-deterministic duration while discrete statements take no time. The modality, therefore, expresses that whenever $\texttt{code}_\texttt{C}$ executes completely, the invariant holds. In particular, the invariant holds at the beginning of and throughout the evolution of the continuous dynamics in $\texttt{plant}_\texttt{C}$. Thus, validity of formula (1) expresses safety of every correctly created object (with respect to its specification).

The following translation of an HABS class and its specification defines formally how the placeholders are composed. The translation is fully automatic and verification is compositional: only classes whose code changed explicitly need re-verification, not the whole system.

5.3 Translation from CHABS to $d\mathcal{L}$

We use two operations on sets of programs P. Operation $\sum P$ constructs a program that non-deterministically executes one of the elements. Operation $\prod P$ constructs all permutations of sequential element-wise execution. Let $|P| = n$:

$$\sum P = \sum\{p_1, \ldots, p_n\} \;=\; p_1 \cup p_2 \cup \cdots \cup p_n$$
$$\prod P = \{p_1; \ldots; p_n \mid \forall i, j \leq n.\; p_i, p_j \in P \wedge (i \neq j \rightarrow p_i \neq p_j)\}$$

We translate classes C with the following design restrictions: (1) All controllers update their local caches of other objects before providing information to those objects (for example, read the current water level before instructing the tank to drain or fill); local caches, once updated, are not modified later. (2) In-port methods with a timed input requirement are only called from timed controllers (for example, a tank that expects to be filled every $5\,s$ is governed by a controller running at a corresponding frequency). (3) Duration statements are exact (have two identical parameters). (4) Local variable names are unique. The first two constraints fix the interaction pattern between components, the last two simplify the presentation. For classes following these restrictions, the translation has four phases, each discussed in detail in subsequent paragraphs: (i) provision of program variables, (ii) generation of assumptions and safety condition, (iii) control code generation, (iv) provision of ODEs and constraints.

5.3.1 Program Variables

For each field, parameter, and local variable in C we create a program variable with the same name. For each method m we create a time variable $t_\texttt{m}$, for each in-port method m a tick variable $tick_\texttt{m}$, both type **Real**; $tick_\texttt{m}$ models the unknown time when an in-port method is *called* next. Time variables are local time for each method and determine when a time-triggered controller or an in-port is *executed* the next time. We denote the set of all tick variables with Tick and the set of all time variables with Time.

$$\left.\begin{array}{r}\mathsf{trans}(\mathtt{f}) \equiv f \text{ , where } f \text{ is a } d\mathcal{L} \text{ variable representing field } \mathtt{f}\\ \mathsf{trans}(\mathtt{v}) \equiv v \text{ , where } v \text{ is a } d\mathcal{L} \text{ variable representing variable } \mathtt{v}\\ \mathsf{trans}(\mathtt{e}_1 \ op\ \mathtt{e}_2) \equiv \mathsf{trans}(\mathtt{e}_1)\ op\ \mathsf{trans}(\mathtt{e}_2)\end{array}\right\}\text{ expressions } \mathtt{e}$$

$$\left.\begin{array}{r}\mathsf{trans}(\mathbf{if}(\mathtt{e})\{\mathtt{s}\}[\mathbf{else}\ \{\mathtt{s}\}]) \equiv \mathbf{if}\ (\mathsf{trans}(\mathtt{e}))\ \mathbf{then}\ \mathsf{trans}(\mathtt{s})[\mathbf{else}\ \mathsf{trans}(\mathtt{s})]\\ \mathsf{trans}(\mathbf{while}(\mathtt{e})\{\mathtt{s}\}) \equiv \mathbf{while}(\mathsf{trans}(\mathtt{e}))\mathsf{trans}(\mathtt{s}) \qquad \mathsf{trans}(\mathtt{s}_1;\mathtt{s}_2) = \mathsf{trans}(\mathtt{s}_1);\mathsf{trans}(\mathtt{s}_2)\\ \mathsf{trans}(\mathtt{[T]\ v\ =\ e}) \equiv \mathsf{trans}(\mathtt{v}) := \mathsf{trans}(\mathtt{e}) \qquad \mathsf{trans}(\mathtt{f\ =\ e}) \equiv \mathsf{trans}(\mathtt{f}) := \mathsf{trans}(\mathtt{e})\\ \mathsf{trans}(\mathtt{e!m()}) \equiv ?\mathit{true} \qquad \mathsf{trans}(\mathtt{f\ =\ e.m()}) \equiv \mathsf{trans}(\mathtt{f}) := *;?\varphi_{\mathtt{m}}\\ \text{where } \varphi_{\mathtt{m}} \text{ is the postcondition of } \mathtt{m}\text{, with the method name replaced by } \mathsf{trans}(\mathtt{f})\end{array}\right\}\text{ statements } \mathtt{s}$$

Figure 20 Translation of expressions `e` and statements `s`

5.3.2 Assumptions and Safety Condition

The formula $\mathtt{assumptions}_\mathtt{C}$ (2) is `C`'s precondition pre plus all initializations init plus conditions on the time and tick variables: in the beginning, each time variable starts at zero and the tick variables have an unknown positive value. Each tick variable *tick* has a method $\mathtt{m}_{tick}$ that is responsible for its generation. We refer to the timed input requirement of this method with $\psi(tick)$, where the method name $\mathtt{m}_{tick}$ has been replaced with *tick*. The initial value of the tick variable is also described by the timed input requirement and describes when the method is issued for the first time at the latest.

$$\mathtt{assumptions}_\mathtt{C} \equiv \mathsf{pre} \wedge \bigwedge_{\varphi \in \mathsf{init}} \varphi \wedge \bigwedge_{t \in \mathrm{Time}} t \doteq 0 \wedge \bigwedge_{tick \in \mathrm{Tick}} \big(0 < tick \wedge \psi(tick)\big) \tag{2}$$

The formula $\mathtt{safety}_\mathtt{C}$ (3) captures the guarantees of class `C`: we need to show that `C` (i) preserves its own invariant inv; (ii) provides guarantees Ens about own out-port methods (shows what others can rely on); (iii) respects timed preconditions TReq^s; and, (iv) when writing to in-port methods of callees, respects their input requirements Req^s. If class `C` comes with a time-triggered controller with guard **duration**`(e,e)`, technical constraint (1) above ensures that at the moment the controller calls an in-port of another object, it has a correct copy of the callee state. Req^s are input requirements of used in-port methods of other classes than `C`, where the method parameter is replaced by the field passed to it. Ens are guarantees of all out-port methods of `C`. Some special care needs to be taken for timed input requirements. With TReq^s, we denote the set of timed input requirements (constructed over *tick*, as above) of all called in-ports where such a clause is given.

$$\mathtt{safety}_\mathtt{C} \equiv \mathsf{inv} \wedge \bigwedge_{\varphi \in \mathsf{Req}^s} \varphi \wedge \bigwedge_{\tau \in \mathsf{TReq}^s} \tau \wedge \bigwedge_{\psi \in \mathsf{Ens}} \psi \tag{3}$$

The safety condition expresses that the controllers of class `C` respect the input requirements when writing to the in-port methods *of other components* and call in-port methods with a timed input requirement sufficiently open. The structure of controllers in `CHABS` per Sect. 3.3 enforces that these calls occur last in the controller bodies.

5.3.3 Control Code

The translation of `ABS` statements to hybrid programs is defined in Fig. 20. We discuss the non-obvious rules: Calls `e!m()` to in-port methods of other objects are mapped to $?\mathit{true}$ (i.e. skip), because there is no effect on the caller object. A read `f=e.m()` from an out-port method is mapped to $\mathsf{trans}(f) := *;?\varphi_{\mathtt{m}}$: a non-deterministic assignment, restricted with a subsequent test for the guarantee of the called out-port method.

The translation of ports and control methods has the *general form* **if** (check) **then** {exec; cleanup}. This pattern is instantiated per method type as follows:

- Time-triggered controller `m` with method body `await duration(e,e); s; this.m()`: check makes sure the correct duration elapsed and cleanup resets time, so check $\equiv t_{\mathtt{m}} \doteq \mathsf{trans}(\mathtt{e})$, exec $\equiv \mathsf{trans}(\mathtt{s})$, cleanup $\equiv t_{\mathtt{m}} := 0$.
- Event-triggered controller `m` with body `await diff e; s; this.m()`: check tests the guard, so check $\equiv \mathsf{trans}(\mathtt{e})$, exec $\equiv \mathsf{trans}(\mathtt{s})$, cleanup $\equiv ?\mathit{true}$.
- In-port method `m` with body `this.f = v`, input requirement φ and timed input requirement ψ: check ensures the correct duration elapsed, so check $\equiv t_{\mathtt{m}} \doteq \mathit{tick}_{\mathtt{m}}$; exec chooses a value consistent with φ, so exec $\equiv f := *;?\varphi$; finally, cleanup does the same for a new duration consistent with ψ (method name replaced by $\mathit{tick}_{\mathtt{m}}$), so cleanup $\equiv \mathit{tick}_{\mathtt{m}} := *;?\mathit{tick}_{\mathtt{m}} > 0;?\psi; t_{\mathtt{m}} := 0$.
- Out-port methods and the `run` method are not translated. Out-port methods have no effect on object state and their guarantees (included in (1) in $\mathtt{safety}_{\mathtt{C}}$) must be shown to hold throughout plant execution. The `run` method initializes the system and ensures that every controller can run once before the first plant execution, which is guaranteed in (1) through sequential composition of $\mathtt{code}_{\mathtt{C}}$; $\mathtt{plant}_{\mathtt{C}}$.

Let M be the set of all translations of in-port methods and controllers, then:

$$\mathtt{code}_{\mathtt{C}} \equiv \left(\sum \prod M\right); \left(\sum M\right)^* \tag{4}$$

The controller $\mathtt{code}_{\mathtt{C}}$ first executes all controllers in a non-deterministically chosen order $(\sum \prod M)$, then allows each controller/in-port to repeat $(\sum M)^*$. The latter replicates eager `ABS` behavior on satisfied guards: when an event-triggered controller is triggered and its guard still holds after its execution, then in `ABS` the controller is run again.

Note that $(\sum M)^*$ safely overapproximates all possible orders, including the behavior of the first part $\sum \prod M$. However, including $\sum \prod M$ in $\mathtt{code}_{\mathtt{C}}$ simplifies practical proofs, because in typical models that disable the check guards at the end of control and in-port method bodies (e.g., a time-triggered controller that resets time in cleanup so that it becomes re-enabled only after some time passes), every method is executed at most once before time advances. The structure of the controller $\mathtt{code}_{\mathtt{C}}$ mirrors this with the first part $\sum \prod M$ to simplify practical proofs as follows: (i) the proof obligations of enabled control and in-port methods (i.e., whose check is true) are easier because the outer loop is dropped, and additionally the proof obligations of all the disabled control and in-port methods can be easily disposed of by contradiction with their check guards; (ii) finding a loop invariant for the second part $(\sum M)^*$ is easy when no method is executed twice before time advances: in that case, the loop invariant for $(\sum M)^*$ must simply imply that none of the check guards holds. Further note that $\sum \prod M$ does not exclude runs, because the general form **if** (check) **then** {exec; cleanup} of control methods and ports in M ensures that there is progress through the implicit **else** $?\mathit{true}$ even if all controllers and in-ports are disabled.

5.3.4 Plant

The plant of a class `C` has the form

$$\mathtt{plant}_{\mathtt{C}} \equiv \sum \{(\mathtt{ode}, \mathtt{ode}_t \,\&\, c) \mid c \in \mathcal{C}\} \ , \tag{5}$$

where `ode` is the ODE from its physical block, $\mathtt{ode}_t$ describes the time variables, and the constraints $c \in \mathcal{C}$ partition the domain of the physical fields. The boundaries of the subdomains overlap exactly where the differential guards hold.[7] This models guards as events in $d\mathcal{L}$, following the modeling pattern described in Sect. 5.1. To ensure that no differential guard is omitted, it is necessary that no two differential guards share a program variable. This is not a restriction, as two controllers can be merged with a disjunction: see the guard in Fig. 2.

To define $\mathcal{C}$ let $e_1, \ldots, e_m$ be the translations of differential guards in the class and $\tilde{e}_i$ the weak complement of e_i. Let $t_1, \ldots, t_l$ be all time variables introduced for time-triggered controllers with e_{t_i} the expression in the **duration** statement. Let $pt_1, \ldots, pt_k$ be all time variables introduced for in-port methods and $tick_{pt_i}$ the associated tick variable. We set $\mathtt{ode}_t \equiv \{t_1' = 1, \ldots, t_l' = 1, pt_1' = 1, \ldots, pt_k' = 1\}$ and define:

$$\mathcal{C} \equiv \big(\{e_1, \tilde{e}_1\} \times \{e_2, \tilde{e}_2\} \times \cdots \times \{e_m, \tilde{e}_m\}\big)$$
$$\cup \{t_1 \leq e_{t_i}\}_{i \leq l} \cup \{t_1 \geq e_{t_i}\}_{i \leq l} \cup \{pt_i \leq tick_{pt_i}\}_{i \leq n} \cup \{pt_i \geq tick_{pt_i}\}_{i \leq n}$$

5.3.5 On the Random Number Generator

We do not translate the `random(i)` expression from `HABS` to $d\mathcal{L}$, because its semantics is that it returns an *integer* below `i`. However, integer arithmetic is undecidable, which is the reason why $d\mathcal{L}$ opts to embed its modality into a decidable first-order logic over the reals [66]. A straightforward overapproximation with a translation to a variation of `random` that returns a real value is:

$$\mathsf{trans}\big(\texttt{f = random(r)}\big) \equiv \mathsf{trans}(\texttt{f}) := *; ?\big(0 \leq \mathsf{trans}(\texttt{f}) < \mathsf{trans}(\texttt{r})\big)$$

5.4 Compositional Verification

We can now state our main theorem: If we can prove safety of all classes, i.e., close all proof obligations, then the whole system is safe, i.e., every class indeed preserves its invariant. Verification is compositional: if we change the code or invariant of one class, only the proof obligation of this class has to be reproven. If we change a method precondition, additionally the proof obligations of all calling classes have to be reproven.

▶ **Theorem 5.** *Let* P *be a set of classes, with each* $\mathtt{C} \in \mathsf{P}$ *associated with* $\varphi_{\mathtt{C}}$ *per formula* (1). *If all the* $\varphi_{\mathtt{C}}$ *are valid, then for every main block that creates objects satisfying* $\mathsf{pre}_{\mathtt{C}}$ *all reachable states of all objects satisfy* $\mathsf{inv}_{\mathtt{C}}$.

Proof Sketch. Recall that the trace of an HAO is an assignment of time to stores (Def. 4). For the proof, each store is indexed by its time and the trace starts with 0 (i.e., the possible offset caused by the delayed object creation is removed):

$$\theta_o(t) = (\rho_t)_{t \in \mathbb{R}^+} = \rho_0 \cdots$$

We are going to use that there are only countably many discrete steps in a run and partition the trace into countably many subtraces. Then we show by induction on these discrete steps that the invariant is always preserved.

Let D be the set of all time points with discrete steps of o in the run that generates θ_o. Note that $0 \in D$ and that $\theta_o(d)$ is the last store defined by the SOS semantics, if several such stores share the same time; further note that this is reflecting the reachability relation of $d\mathcal{L}$.

We define θ_o^d as the subtrace of θ_o starting with d and ending at the next time point of a discrete step. Let $\mathsf{next}(d)$ be the next time point of a discrete step after d, if such a time point

[7] Expressions contain only `>=`, `<=`, so weak complement ensures a boundary overlap.

exists, and ∞ otherwise:

$$\mathbf{dom}\left(\theta_o^d\right) = \left[d..\mathsf{next}(d)\right] \qquad \text{with} \qquad \theta_o^d(t) = \theta_o(t)$$

We observe that each state in the HABS semantics is also a state in the Kripke structure of the semantics if all class parameters are removed. We show that trans preserves reachability: if from a state ρ state ρ' is reachable by an HABS statement s in the HABS semantics, then state ρ' is reachable from state ρ by trans(s). This is justified as follows:

1. The $d\mathcal{L}$ program omits no events, because each event is at a boundary of two evolution domain constraints on a variable and no two events share a variable (each controller has its own time variable).
2. The evolution domain constraints cover all possible states, so no run is rejected for a domain being too small.
3. Each test in $d\mathcal{L}$ formula φ_{C} that discards runs does so using a condition that is provably guaranteed by other objects. For example, the test that discards all runs of an in-port method for inputs not satisfying its input requirements is safe, because on the caller side this condition is part of the safety condition (3).
4. The observation also relies on technical constraint (1) above and the recursive call being at the end of a controller. Together, this guarantees that *at that moment* the caller copy of the callee's state is consistent with the callee's actual state.

Let $D = (d_i)_{i\in\mathbb{N}}$ be an enumeration of the discrete time points and $\hat{\theta}_o^{d_i}$ the union of all subtraces of θ_o up to d_i:

$$\mathbf{dom}\left(\hat{\theta}_o^{d_i}\right) = \bigcup_{j\leq i} \mathbf{dom}\left(\theta_o^{d_j}\right) \qquad \text{with} \qquad \hat{\theta}_o^{d_i}(t) = \theta_o(t)$$

We show by induction on i that every state in $\hat{\theta}_o^{d_i}$ is safe, i.e., a model for the invariant $\mathsf{inv}_{\mathsf{C}}$.

Induction Base: $i = 0$. It is explicitly checked that $\theta_o^{d_0}$ is safe. By assumption, the object is created in a state $\theta_o^{d_0}$ such that the precondition $\mathsf{pre}_{\mathsf{C}}$ holds. From axiom I of $d\mathcal{L}$ [68] we know that the safety condition must be true in the beginning of the loop, thus validity of φ_{C} implies validity of $\mathsf{pre}_{\mathsf{C}} \rightarrow \mathsf{inv}_{\mathsf{C}}$. Since all the formulas φ_{C} are proved in isolated component proofs, we conclude $\mathsf{inv}_{\mathsf{C}}$ holds for all reachable states of all objects as by the correctness argument reachability is preserved.

Induction Step. $i > 0$. This is analogous to the base case, but instead of an explicit check that d_i is safe, we use the induction hypothesis that every state in $\hat{\theta}_o^{d_{i-1}}$ is safe and that the statement for d_i is executed in a state at time $t \in \mathbf{dom}\left(\hat{\theta}_o^{d_{i-1}}\right)$. ◀

▶ Remark. The theorem states soundness of safety properties in $d\mathcal{L}$ proof obligations and does not prove semantic equivalence between the contained $d\mathcal{L}$-program and the HABS class. This approach stands in the tradition of modular deductive verification of object-oriented software, in particular, it follows the structure of systems for distributed object-oriented programs [52]. The main reason to pursue this approach is that the form of proof obligations and the translation of statements cannot be disentangled: the translation of method calls includes the postcondition of the called methods: soundness of the translation relies on the fact that all other proof obligations can be established. This is already the case for discrete, sequential languages [41]. Note that this is *not* circular. As the proof of Theorem 5 shows, we can order all method executions in a run such that we have a well-founded induction on them. The first method execution in every object relies only on the state precondition which is guarenteed at creation. These in turn are guaranteed in the main block, which has no assumptions. Another reason is that each $d\mathcal{L}$ proof obligation corresponds to the (symbolic) execution of one *object* in a class. To model all permissible evolutions of several method executions in a proof, therefore, it is necessary to encode the scheduler. This requires a form of proof obligation that assumes the object invariant (which contains scheduling constraints). This effect is well-known in deductive verification of distributed programs [31, 32, 52].

5.5 Case Study

We illustrate the `HABS`-to-KeYmaera X translation defined above with the **TankTick** system in Fig. 4. The example, the implementation of the translation and the simulation, as well as the mechanical proofs of the translation are available in the supplementary material.[8] We start with the two-object water tank, whose behavior for an initial level of $5\,l$ is plotted in Fig. 5.

5.5.1 Class `CTank`

The in-port method `inDrain()` of the `CTank` class gives rise to a time variable $t_{\texttt{inDrain}}$ and a tick variable $tick_{\texttt{inDrain}}$. Following (2), $\mathsf{assumptions}_{\texttt{Tank}}$ is:

$$\begin{aligned}\mathsf{assumptions}_{\texttt{Tank}} \equiv\ & 4 \leq \texttt{inVal} \leq 9\\ & \land t_{\texttt{inDrain}} \doteq 0 \land 0 < tick_{\texttt{inDrain}}\\ & \land \texttt{level} \doteq \texttt{inVal} \land \texttt{drain} \doteq -1/2\end{aligned} \tag{6}$$

The safety condition says the tank level stays within its limits and that `level` adheres to its contract which happen to be identical. No in-port methods of other classes are used, hence:

$$\mathsf{safety}_{\texttt{Tank}} \equiv 3 \leq \texttt{level} \leq 10\ . \tag{7}$$

The `CTank` class has no controller method, so the `inDrain` method, which has a timed input requirement, per (4) results in $\mathsf{code}_{\texttt{Tank}}$ below

$$\mathsf{code}_{\texttt{Tank}} \equiv \mathsf{p}; (\mathsf{p})^* \tag{8}$$

where $\mathsf{p} \equiv \mathsf{trans}(\texttt{inDrain})$ below is translated from Fig. 4 using the translation of Fig. 20:

$$\begin{aligned}\mathsf{p} \equiv\ & \textbf{if}\ (t_{\texttt{inDrain}} \doteq tick_{\texttt{inDrain}})\ \textbf{then}\\ & \quad \texttt{drain} := *;\\ & \quad ?{-1/2} \leq \texttt{drain} \leq 1/2 \land (\texttt{drain} < 0 \rightarrow \texttt{level} \geq 3.5)\\ & \qquad\qquad \land (\texttt{drain} > 0 \rightarrow \texttt{level} \leq 9.5);\\ & \quad tick_{\texttt{inDrain}} := *;\ ?0 < tick_{\texttt{inDrain}} < 1;\ t_{\texttt{inDrain}} := 0\end{aligned}$$

The plant $\mathsf{plant}_{\texttt{Tank}}$, following shape (5), is based on the physical block and the new clock variable (there are no differential guards), with the evolution domain constraint split along the new time variable $t_{\texttt{inDrain}}$. ODEs of the form $v' = 0$ are default and omitted.

$$\begin{aligned}&\mathsf{plant}_{\texttt{Tank}} \equiv \mathsf{plant}^{\leq}_{\texttt{Tank}} \cup \mathsf{plant}^{\geq}_{\texttt{Tank}}\\ &\mathsf{plant}^{\leq}_{\texttt{Tank}} \equiv \{\texttt{level}' = \texttt{drain}, t'_{\texttt{inDrain}} = 1\ \&\ t_{\texttt{inDrain}} \leq tick_{\texttt{inDrain}}\}\\ &\mathsf{plant}^{\geq}_{\texttt{Tank}} \equiv \{\texttt{level}' = \texttt{drain}, t'_{\texttt{inDrain}} = 1\ \&\ t_{\texttt{inDrain}} \geq tick_{\texttt{inDrain}}\}\end{aligned} \tag{9}$$

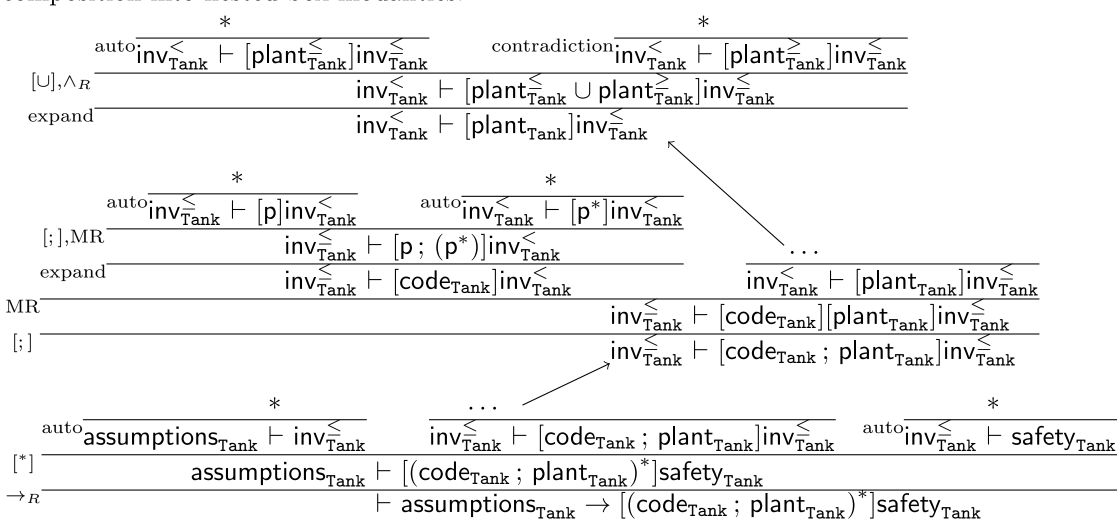
[8] `https://doi.org/10.5281/zenodo.5973904`

▶ **Lemma 6.** *Class* `Tank` *is safe, i.e., formula* $\varphi_{\texttt{Tank}}$ *—obtained per* (1) *referring to tank assumptions* $\mathsf{assumptions}_{\texttt{Tank}}$ (6)*, postcondition* $\mathsf{safety}_{\texttt{Tank}}$ (7)*, code* $\mathsf{code}_{\texttt{Tank}}$ (8)*, and plant* $\mathsf{plant}_{\texttt{Tank}}$ (9)*—is valid.*

$$\varphi_{\texttt{Tank}} \equiv \mathsf{assumptions}_{\texttt{Tank}} \rightarrow \left[(\mathsf{code}_{\texttt{Tank}};\ \mathsf{plant}_{\texttt{Tank}})^*\right] \mathsf{safety}_{\texttt{Tank}}$$

Proof. See KeYmaera X proofs in the supplementary material. The proof sketch here serves as an illustration of how sequent proofs in KeYmaera X systematically use the invariant annotations in `HABS`. In the proof, we show the inductive loop invariant $\mathsf{inv}^{\leq}_{\texttt{Tank}}$, which expresses that the level always stays within limits and that the next input will be supplied before exceeding the timed input requirement as follows: $3 \leq \texttt{level} \leq 10 \wedge -1/2 \leq \texttt{drain} \leq 1/2 \wedge 3 \leq \texttt{level} + \texttt{drain}(\mathsf{tick}_{\texttt{inDrain}} - t_{\texttt{inDrain}}) \leq 10 \wedge \mathsf{tick}_{\texttt{inDrain}} \leq t_{\texttt{inDrain}}$.

The proof starts in step $\rightarrow_R$ to make the left-hand side $\mathsf{assumptions}_{\texttt{Tank}}$ of the implication available as assumptions. Next, $[^*]$ uses the loop invariant $\mathsf{inv}^{\leq}_{\texttt{Tank}}$ for induction: the base case in the left-most subgoal and the use case in the right-most subgoal follow by real arithmetic automation; the induction step in the middle subgoal continues with $[;]$ to split the sequential composition into nested box modalities.



The main insight now is that $\mathsf{code}_{\texttt{Tank}}$ reacts at the latest when $\mathsf{tick}_{\texttt{inDrain}} = t_{\texttt{inDrain}}$ and will reset the timer using $\mathsf{tick}_{\texttt{inDrain}} := 0$, so that the timing requirement $\mathsf{tick}_{\texttt{inDrain}} \leq t_{\texttt{inDrain}}$ can be strengthened to a strict inequality $\mathsf{tick}_{\texttt{inDrain}} < t_{\texttt{inDrain}}$ in the inductive loop invariant. The resulting intermediate condition $\mathsf{inv}^{<}_{\texttt{Tank}}$ is used in step MR to split into two subgoals: in the left subgoal of MR, we show that $\mathsf{code}_{\texttt{Tank}}$ guarantees the intermediate condition $\mathsf{inv}^{<}_{\texttt{Tank}}$. In the right subgoal of MR we show that $\mathsf{plant}_{\texttt{Tank}}$ preserves the loop invariant from that intermediate condition: the plant listens for the event $\mathsf{tick}_{\texttt{inDrain}} = t_{\texttt{inDrain}}$ with a choice between two differential equations, whose evolution domain constraints exactly overlap at the event. On evolution domain $\mathsf{tick}_{\texttt{inDrain}} \leq t_{\texttt{inDrain}}$ in $\mathsf{plant}^{\leq}_{\texttt{Tank}}$, the differential equation preserves the loop invariant, whereas on evolution domain $\mathsf{tick}_{\texttt{inDrain}} \geq t_{\texttt{inDrain}}$ in $\mathsf{plant}^{\geq}_{\texttt{Tank}}$ the contradiction shows that the controller reacts such that the plant can never enter this unsafe behavior. ◀

5.5.2 Time-Triggered Controller `FlowCtrl`

Assumptions $\mathsf{assumptions}_{\texttt{FlowCtrl}}$ of `FlowCtrl` constructed per (2) and plant $\mathsf{plant}_{\texttt{FlowCtrl}}$ constructed per (5) are straightforward. The latter is created for the sake of observing time events, even though no physical block is present:

$$\mathsf{assumptions}_{\texttt{FlowCtrl}} \equiv 0 < \texttt{tick} < 1 \tag{10}$$

$$\mathsf{plant}_{\texttt{FlowCtrl}} \equiv \{t'_{\texttt{ctrlFlow}} = 1 \,\&\, t_{\texttt{ctrlFlow}} \geq \texttt{tick}\} \cup \{t'_{\texttt{ctrlFlow}} = 1 \,\&\, t_{\texttt{ctrlFlow}} \leq \texttt{tick}\} \tag{11}$$

The safety condition $\mathsf{safety}_{\texttt{FlowCtrl}}$ constructed per (3) is the timed input requirement of the called `inDrain` method and the class invariant (subsumed by the input requirement of `inDrain`):

$$\begin{aligned}\mathsf{safety}_{\texttt{FlowCtrl}} \equiv\ & -1/2 \leq \texttt{drain} \leq 1/2 \wedge \texttt{tick} < 1 \\ & \wedge (\texttt{drain} < 0 \rightarrow \texttt{level} \geq 3.5) \\ & \wedge (\texttt{drain} > 0 \rightarrow \texttt{level} \leq 9.5)\end{aligned} \tag{12}$$

Finally, the code $\mathsf{code}_{\texttt{FlowCtrl}}$ is translated as

$$\mathsf{code}_{\texttt{FlowCtrl}} \equiv \mathsf{q}; (\mathsf{q})^* \tag{13}$$

with

$$\begin{aligned}\mathsf{q} \equiv\ & \textbf{if } (t_{\texttt{ctrlFlow}} \doteq \texttt{tick}) \textbf{ then} \\ & \quad \texttt{level} := *;\ ?3 \leq \texttt{level} \leq 10; \\ & \quad \textbf{if } (\texttt{level} \leq 3.5) \textbf{ then } \{\texttt{drain} := 1/2\}; \\ & \quad \textbf{if } (\texttt{level} \geq 9.5) \textbf{ then } \{\texttt{drain} := -1/2\}; \\ & \quad t_{\texttt{ctrlFlow}} := 0\end{aligned}$$

▶ **Lemma 7.** *Class* `FlowCtrl` *is safe, i.e., formula* $\varphi_{\texttt{FlowCtrl}}$*—obtained per* (1) *referring to assumptions* $\mathsf{assumptions}_{\texttt{FlowCtrl}}$ (10)*, postcondition* $\mathsf{safety}_{\texttt{FlowCtrl}}$ (12)*, code* $\mathsf{code}_{\texttt{FlowCtrl}}$ (13)*, and plant* $\mathsf{plant}_{\texttt{FlowCtrl}}$ (11)*—is valid.*

$$\varphi_{\texttt{FlowCtrl}} \equiv \mathsf{assumptions}_{\texttt{FlowCtrl}} \rightarrow \left[(\mathsf{code}_{\texttt{FlowCtrl}};\ \mathsf{plant}_{\texttt{FlowCtrl}})^*\right] \mathsf{safety}_{\texttt{FlowCtrl}}$$

Proof. See KeYmaera X-proofs in the supplementary material. ◀

5.5.3 Event-Triggered Controller `CSingleTank`

Translation of class `CSingleTank` from Fig. 2 illustrates the handling of event-triggered controllers. The plant and code interact. The plant separates the evolution domain into two parts, with the guard of the event-triggered controller (the white areas in Fig. 21) defining their boundary. The gray areas are *larger* than the safe region defined by `3 <= level <= 10`. This is necessary to avoid Zeno behavior in the eager execution semantics of `HABS`: If we used simply the weak complement of the safe region `level <= 3 | level >= 10` as a guard and happen to be in a program state at the boundary (the lower of the states indicated with a star in Fig. 21), then the controller changes the state as shown by the arrow. But if the next state is again *on the boundary*, which is the case when the safe region is too small, then the guard is triggered, the controller loops back to the first state, etc., without physical time being able to advance. The guard in Fig. 2 ensures that after the controller has run, the state is

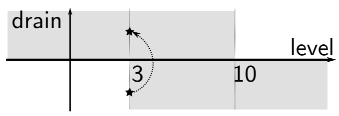


Figure 21 Avoiding Zeno-behavior in **TankMono**.

not on the boundary anymore. This behavior is exhibited by our implementation, see Fig. 3. The code $\mathsf{code}_{\texttt{CSingleTank}}$ has the form $\mathsf{r};(\mathsf{r})^*$ with r being:

$$
\begin{aligned}
\mathsf{r} \equiv\ & \textbf{if}\ (\texttt{level}{\leq}3 \wedge \texttt{drain}{\leq}0) \vee (\texttt{level}{\geq}10 \wedge \texttt{drain}{\geq}0)\ \textbf{then} \\
& \quad \textbf{if}\ (\texttt{level} \leq 3)\ \textbf{then}\ \texttt{drain} := 1/2\ \textbf{else}\ \texttt{drain} := -1/2
\end{aligned}
$$

The plant of `CSingleTank` with sufficiently large regions is as follows:

$$
\begin{aligned}
& \mathsf{plant}_{\texttt{CSingleTank}} \equiv \\
& \quad \{\texttt{level}' = \texttt{drain} \,\&\, (\texttt{level} \leq 3 \wedge \texttt{drain} \leq 0) \vee (\texttt{level} \geq 10 \wedge \texttt{drain} \geq 0)\} \\
& \cup \{\texttt{level}' = \texttt{drain} \,\&\, (\texttt{level} \geq 3 \vee \texttt{drain} \geq 0) \wedge (\texttt{level} \leq 10 \vee \texttt{drain} \leq 0)\}
\end{aligned}
$$

5.6 On Translation into $d\mathcal{L}$

`HABS` programs can be tested and validated, but the programmer needs to avoid writing programs that are **1.** inherently difficult to interpret and **2.** have a high degree of non-determinism. Both are good programming and software engineering practices, of course, and the fact that `HABS` is a *programming language* enables one to apply standard techniques for discrete programs.

A back-translation from $d\mathcal{L}$ to `HABS` would provide meaningful validation only for deterministic $d\mathcal{L}$ models. While being possible even in the general case, two traits of $d\mathcal{L}$ programs prohibit easy interpretation and simulation:

Highly Non-Deterministic Structure Additionally to non-deterministic assignment, branching and repetition are both non-deterministic: the—rather non-intuitive—representation of $(\texttt{s})^*$ in `HABS` is a loop that non-deterministically chooses to break out.

```
while(True) {
    Int i = random(2);
    if ( i == 1 ) break;
    s;
}
```

This loop may never terminate, while the semantics of $d\mathcal{L}$ loops defines an arbitrary but countable number of repetitions. A similar pattern has to be employed for branching.

Tests The test $?\varphi$ discards a run based on a $d\mathcal{L}$-guard. Translation would require **1.** to evaluate $d\mathcal{L}$ formulas, as opposed to Boolean expressions, and **2.** a mechanism to abort the program. This can be emulated by exceptions, but it obfuscates the semantics.

6 Related & Future Work, Conclusion

6.1 Related Work

Previous work on hybrid programming concentrated on purely sequential languages: `HybCore` [39] is a while-language with hybrid behavior and a simulator [40], but lacks formal verification techniques. Its extensional semantics is not able to express the timed properties needed for our distributed controller. $\texttt{While}^{\texttt{dt}}$ [77] is also a while-language and uses infinitesimals instead of ODEs to model continuous dynamics. It has a simple verification system based on Hoare triples [42], but is not executable.

Hybrid Rebeca (HR) [46] proposes to embed hybrid automata directly into the actor language Rebeca. In contrast to `HABS`, no simulation is available and verification is not object-modular: the whole model is translated to a single monolithic hybrid automaton. Because of this, a number of boundedness constraints have to be imposed. The translation is also the semantics: HR has no semantics beyond this translation and is mainly a frontend for Hybrid Automata tools. The verification backend of HR does not support non-linear ODEs (our examples are linear, but `HABS`, KeYmaera X, and Maxima, support non-linear ODEs; `HABS` models with non-linear ODEs are found in the online supplement).

Recent efforts [58, 64] split the verification task in $d\mathcal{L}$ into manageable pieces by modularizing deductive hybrid systems verification with component-based modeling and verification techniques, but impose strict structural requirements on components and communication. The Sphinx modeling tool [62] for $d\mathcal{L}$ represents non-distributed hybrid programs with UML class and activity diagrams, but for verification purposes it translates these model artifacts into a single monolothic hybrid program.

The Architecture Analysis and Design Language (AADL), a language to model hardware and software components in embedded systems, has a hybrid extension [2], which uses the HHL [80] theorem prover as its verification backend [1]. HHL is based on Hoare triples over hybrid CSP programs and duration calculus formulas [57]. Hybrid AADL offers structuring elements for components and their connections on the architecture level. The semantics of hybrid AADL is given as a translation of the *synchronous* fragment of AADL into hybrid CSP, while we extend the semantics of the actor-based programming language `ABS` to combine reasoning about the asynchronous behavior of communicating components in `ABS` with reasoning about the internal combined discrete and continuous component behavior in differential dynamic logic. As a side effect, the extended semantics enables proving the correctness of the translation to differential dynamic logic, as well as translating `HABS` to other formal languages.

A similar approach based on Stateflow/Simulink is implemented in the MARS toolkit [22]. The MARS approach is orthogonal to `HABS`: MARS connects a verification toolkit around a simulation language (which is a daunting task given the missing formal semantics of Stateflow/Simulink), while `HABS` is designed specifically to enable verification and simulation through its languages features. This is reflected in the soundness proof, which is based on a *bi*directional translation.

Another approach based on CSP and the duration calculus combines these formalisms with Object-Z [45]. This enables model-checking for real-time systems (clocks with resets), while we support hybrid systems theorem proving with (non-linear) differential equations. A further integration of Object-Z and (Timed) CSP was investigated by Mahony & Dong [60].

Hybrid Event-B [12, 13] extends Event-B refinement reasoning with continuous behavior between the usual discrete Event-B events. A more lightweight approach [76, 21] models hybrid systems in an abstract way as action systems without differential equations directly in Event-B, and complements analysis in Event-B with simulation in Matlab. Similarly, Dupont et al. [34] use Event-B for a correct-by-construction approach to hybrid systems. They embed the ODEs used for continuous modeling by declaring them as a special theory within Event-B instead of extending the core language itself.

Integrated tools such as Ptolemy [71], Stateflow/Simulink except the aforementioned MARS toolkit, and Modelica, all emphasize simulation, reachability analysis (e.g., CHARON [6, 7], Ariadne [15]), or testing (e.g., [30]). As supporting techniques, they provide modeling notation for timing aspects, signals, and data flow between heterogeneous models. Formal verification of hybrid systems with reachability analysis and model checking tools (SpaceEx [35], CORA [4], Flow* [23]) support modularity [33] based on hybrid I/O automata [59], assume-guarantee reasoning [17, 43], and hybridization [24]. However, they work best for finite-horizon analysis and finite regions (because over-approximations stay tight only for bounded time and from small starting regions). Similar restrictions apply to dReal/dReach [37, 55].

Dynamic I/O automata [9] for modeling dynamic systems introduce a notion of externally visible behavior, the ability to create and destroy automata and change their signature dynamically; those features are all naturally available in our object-oriented approach and do not need special extension like automata-based modeling tools. Our work contrasts with all mentioned simulation and verification approaches by providing a uniform modeling language, validation by simulation, modular infinite-horizon and infinite-region theorem proving through translation from HABS to $d\mathcal{L}$.

Translation among hybrid system languages so far centers around hybrid automata as a unifying concept [11, 79]. Others focus on the discrete fragment [38]. Our translation from HABS to $d\mathcal{L}$ translates complete hybrid system models written in a *programming language*, including annotations (preconditions, invariants, etc.). It is sound relative to the formal semantics of HABS and $d\mathcal{L}$.

Hybrid systems validation through simulation is addressed with translation to Stateflow/Simulink [10]; with a combination of discrete-event and numerical methods [19]; and with co-simulation between control software and dedicated physics simulators [26, 78, 82]. Here, we focus on safety verification, the distributed aspect of HABS models, and take a pragmatic first step for simulating continuous models.

In summary, HABS is designed for modular deductive verification (unlike simulation-centric tools), infinite-horizon analysis on infinite regions (unlike reachability analysis and model checking tools), without sacrificing high-level programming language features (unlike hybrid systems modularization techniques and assume-guarantee reasoning).

6.2 Future Work

The present work lifts the research on formal semantics of programming languages for hybrid systems from verification-centric minimalistic languages to distributed object-oriented languages. Carrying over techniques, ideas, and analyses from programming language research to hybrid systems programming, presents an intriguing research direction. Our ongoing work on larger case studies with HABS, in particular in connection with co-simulation [54], is expected to reveal additional challenges.

We plan to combine the verification of CHABS presented here with the more modular approach based on post-regions [51], which does not support timed input requirements yet. Future research avenues include investigating how the static analyses for ABS, in particular the deadlock analysis for boolean guards [50], can be extended for HABS, extending approximate simulation of non-solvable differential equations, experimenting with various computer algebra systems, and supporting guards with non-urgent semantics.

6.3 Conclusion

Distributed hybrid systems are not only difficult to *verify* formally, it is equally hard to *validate* a formal model of them, especially with components using symbolic computations, such as servers. Both activities have conflicting demands, so we propose a translation-based approach: modeling is guided by patterns over hybrid programs and class specifications in HABS, a hybrid extension of the concurrent active-object language ABS. These are automatically decomposed and translated (Thm. 5) into sequential proof obligations of the verification-oriented differential dynamic logic $d\mathcal{L}$ and discharged by the hybrid theorem prover KeYmaera X.

We illustrated the viability of our approach by a case study that features many complications: concurrent behavior, possible non-termination, correctness depending on timing constants, multi-dimensional domain, time lag in sensing, etc.

Acknowledgments

This work is partially supported by the FormbaR project, part of AG Signalling/DB RailLab in the Innovation Alliance of Deutsche Bahn AG and TU Darmstadt. This material is based upon work supported by AFOSR grant FA9550-16-1-0288.

References

1 Ehsan Ahmad, Yunwei Dong, Shuling Wang, Naijun Zhan, and Liang Zou. Adding formal meanings to AADL with hybrid annex. In Ivan Lanese and Eric Madelaine, editors, *Formal Aspects of Component Software - 11th International Symposium, FACS 2014, Bertinoro, Italy, September 10-12, 2014, Revised Selected Papers*, volume 8997 of *Lecture Notes in Computer Science*, pages 228–247. Springer, 2014.

2 Ehsan Ahmad, Brian R. Larson, Stephen C. Barrett, Naijun Zhan, and Yunwei Dong. Hybrid annex: an AADL extension for continuous behavior and cyber-physical interaction modeling. In Michael Feldman and S. Tucker Taft, editors, *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, pages 29–38. ACM, 2014.

3 Elvira Albert, Frank S. de Boer, Reiner Hähnle, Einar Broch Johnsen, Rudolf Schlatte, Silvia Lizeth Tapia Tarifa, and Peter Y. H. Wong. Formal modeling and analysis of resource management for cloud architectures: an industrial case study using real-time ABS. *Service Oriented Computing and Applications*, 8(4):323–339, 2014.

4 M. Althoff. An introduction to CORA 2015. In *Proc. of the Workshop on Applied Verification for Continuous and Hybrid Systems*, 2015.

5 Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, volume 736 of *LNCS*, pages 209–229, Berlin, Heidelberg, 1993. Springer.

6 Rajeev Alur, Thao Dang, Joel M. Esposito, Rafael B. Fierro, Yerang Hur, Franjo Ivancic, Vijay Kumar, Insup Lee, Pradyumna Mishra, George J. Pappas, and Oleg Sokolsky. Hierarchical hybrid modeling of embedded systems. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Embedded Software, First International Workshop, EMSOFT 2001, Tahoe City, CA, USA, October, 8-10, 2001, Proceedings*, volume 2211 of *Lecture Notes in Computer Science*, pages 14–31. Springer, 2001.

7 Rajeev Alur, Radu Grosu, Yerang Hur, Vijay Kumar, and Insup Lee. Modular specification of hybrid systems in CHARON. In Nancy A. Lynch and Bruce H. Krogh, editors, *Hybrid Systems: Computation and Control, Third International Workshop, HSCC 2000, Pittsburgh, PA, USA, March 23-25, 2000, Proceedings*, volume 1790 of *Lecture Notes in Computer Science*, pages 6–19. Springer, 2000.

8 Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation.* Springer, 2013.

9 Paul C. Attie and Nancy A. Lynch. Dynamic input/output automata: A formal and compositional model for dynamic systems. *Inf. Comput.*, 249:28–75, 2016.

10 Stanley Bak, Omar Ali Beg, Sergiy Bogomolov, Taylor T. Johnson, Luan Viet Nguyen, and Christian Schilling. Hybrid automata: from verification to implementation. *STTT*, 21(1):87–104, 2019.

11 Stanley Bak, Sergiy Bogomolov, and Taylor T. Johnson. HYST: a source transformation and translation tool for hybrid automaton models. In Antoine Girard and Sriram Sankaranarayanan, editors, *HSCC'15*, pages 128–133. ACM, 2015.

12 Richard Banach, Michael J. Butler, Shengchao Qin, Nitika Verma, and Huibiao Zhu. Core hybrid Event-B I: single hybrid Event-B machines. *Sci. Comput. Program.*, 105:92–123, 2015.

13 Richard Banach, Michael J. Butler, Shengchao Qin, and Huibiao Zhu. Core hybrid Event-B II: multiple cooperating hybrid Event-B machines. *Sci. Comput. Program.*, 139:1–35, 2017.

14 Don S. Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Software Eng.*, 30(6):355–371, 2004.

15 Luca Benvenuti, Davide Bresolin, Pieter Collins, Alberto Ferrari, Luca Geretti, and Tiziano Villa. Assume–guarantee verification of nonlinear hybrid systems with Ariadne. *International Journal of Robust and Nonlinear Control*, 24(4):699–724, 2014.

16 Joakim Bjørk, Frank S. de Boer, Einar Broch Johnsen, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, 9(1):29–43, 2013.

17 Sergiy Bogomolov, Goran Frehse, Marius Greitschus, Radu Grosu, Corina S. Pasareanu, Andreas Podelski, and Thomas Strump. Assume-guarantee abstraction refinement meets hybrid systems. In Eran Yahav, editor, *Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings*, volume 8855 of *Lecture Notes in Computer Science*, pages 116–131. Springer, 2014.

18 Brandon Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and André Platzer. VeriPhy: Verified controller executables from verified cyber-physical system models. In Dan Grossman, editor, *PLDI*, pages 617–630. ACM, 2018.

19 Christopher X. Brooks, Edward A. Lee, David Lorenzetti, Thierry S. Nouidui, and Michael Wetter. CyPhySim: a cyber-physical systems simulator. In Antoine Girard and Sriram Sankaranarayanan, editors, *HSCC'15*, pages 301–302. ACM, 2015.

20 Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.

21 Michael J. Butler, Jean-Raymond Abrial, and Richard Banach. Modelling and refining hybrid systems in Event-B and Rodin. In Luigia Petre and Emil Sekerinski, editors, *From Action Systems to Distributed Systems - The Refinement Approach*, pages 29–42. Chapman and Hall/CRC, 2016.

22 Mingshuai Chen, Xiao Han, Tao Tang, Shuling Wang, Mengfei Yang, Naijun Zhan, Hengjun Zhao, and Liang Zou. MARS: A toolchain for modelling, analysis and verification of hybrid systems. In Michael G. Hinchey, Jonathan P. Bowen, and Ernst-Rüdiger Olderog, editors, *Provably Correct Systems*, NASA Monographs in Systems and Software Engineering, pages 39–58. Springer, 2017.

23 Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 258–263. Springer, 2013.

24 Xin Chen and Sriram Sankaranarayanan. Decomposed reachability analysis for nonlinear systems. In *2016 IEEE Real-Time Systems Symposium, RTSS 2016, Porto, Portugal, November 29 - December 2, 2016*, pages 13–24. IEEE Computer Society, 2016.

25 Dave Clarke, Radu Muschevici, José Proença, Ina Schaefer, and Rudolf Schlatte. Variability modelling in the ABS language. In *FMCO*, volume 6957 of *Lecture Notes in Computer Science*, pages 204–224. Springer, 2010.

26 Fabio Cremona, Marten Lohstroh, David Broman, Edward A. Lee, Michael Masin, and Stavros Tripakis. Hybrid co-simulation: it's about time. *Software and Systems Modeling*, 18(3):1655–1679, 2019.

27 P.J.L. Cuijpers and M.A. Reniers. Hybrid process algebra. *J. of Logic and Algebraic Programming*, 62(2):191–245, 2005.

28 Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2007.

29 Frank S. de Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. A survey of active object languages. *ACM Computíng Surveys*, 50(5):1–39, 2017.

30 Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. Compositional programming and testing of dynamic distributed systems. *Proc. ACM Program. Lang.*, 2(OOPSLA):159:1–159:30, 2018.

31 Crystal Chang Din, Reiner Hähnle, Einar Broch Johnsen, Ka I Pun, and Silvia Lizeth Tapia Tarifa. Locally abstract, globally concrete semantics of concurrent programming languages. In Renate A. Schmidt and Cláudia Nalon, editors, *Automated Reasoning with Analytic Tableaux and Related Methods - 26th International Conference, TABLEAUX 2017, Brasília, Brazil, September 25-28, 2017, Proceedings*, volume 10501 of *Lecture Notes in Computer Science*, pages 22–43. Springer, 2017.

32 Crystal Chang Din and Olaf Owe. Compositional reasoning about active objects with shared futures. *Formal Asp. Comput.*, 27(3):551–572, 2015.

33 Alexandre Donzé and Goran Frehse. Modular, hierarchical models of control systems in SpaceEx. In *European Control Conference, ECC 2013, Zurich, Switzerland, July 17-19, 2013*, pages 4244–4251. IEEE, 2013.

34 Guillaume Dupont, Yamine Aït Ameur, Neeraj Kumar Singh, and Marc Pantel. Event-B hybridation: A proof and refinement-based framework for modelling hybrid systems. *ACM Trans. Embed. Comput. Syst.*, 20(4):35:1–35:37, 2021.

35 Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. SpaceEx: Scalable verification of hybrid systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 379–395. Springer, 2011.

36 Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. KeYmaera X: an axiomatic tactical theorem prover for hybrid systems. In Amy P. Felty and Aart Middeldorp, editors, *CADE'25*, volume 9195 of *LNCS*, pages 527–538. Springer, 2015.

37 Sicun Gao, Soonho Kong, and Edmund M. Clarke. dReal: An SMT solver for nonlinear theories over the reals. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 208–214. Springer, 2013.

38 Luis Garcia, Stefan Mitsch, and André Platzer. HyPLC: hybrid programmable logic controller program translation for verification. In *ICCPS'19*, pages 47–56, 2019.

39 Sergey Goncharov and Renato Neves. An adequate while-language for hybrid computation. *CoRR*, abs/1902.07684, 2019.

40 Sergey Goncharov, Renato Neves, and José Proença. Implementing hybrid semantics: From functional to imperative. In Violet Ka I Pun, Volker Stolz, and Adenilso Simão, editors, *Theoretical Aspects of Computing - ICTAC 2020 - 17th International Colloquium, Macau, China, November 30 - December 4, 2020, Proceedings*, volume 12545 of *Lecture Notes in Computer Science*, pages 262–282. Springer, 2020.

41 Daniel Grahl, Richard Bubel, Wojciech Mostowski, Peter H. Schmitt, Mattias Ulbrich, and Benjamin Weiß. Modular specification and verification. In Wolfgang Ahrendt, Bernhard Beckert, Richard

Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors, *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*, pages 289–351. Springer, 2016.

42 Ichiro Hasuo and Kohei Suenaga. Exercises in nonstandard static analysis of hybrid systems. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 462–478. Springer, 2012.

43 Thomas A. Henzinger, Marius Minea, and Vinayak S. Prabhu. Assume-guarantee reasoning for hierarchical hybrid systems. In Maria Domenica Di Benedetto and Alberto L. Sangiovanni-Vincentelli, editors, *Hybrid Systems: Computation and Control, 4th International Workshop, HSCC 2001, Rome, Italy, March 28-30, 2001, Proceedings*, volume 2034 of *Lecture Notes in Computer Science*, pages 275–290. Springer, 2001.

44 Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI'73*, pages 235–245, 1973.

45 Jochen Hoenicke and Ernst-Rüdiger Olderog. Combining specification techniques for processes, data and time. In Michael Butler, Luigia Petre, and Kaisa Sere, editors, *Integrated Formal Methods*, pages 245–266, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

46 Iman Jahandideh, Fatemeh Ghassemi, and Marjan Sirjani. Hybrid Rebeca: modeling and analyzing of cyber-physical systems. *CoRR*, abs/1901.02597, 2019.

47 Jean-Baptiste Jeannin, Khalil Ghorbal, Yanni Kouskoulas, Aurora Schmidt, Ryan Gardner, Stefan Mitsch, and André Platzer. A formally verified hybrid system for safe advisories in the next-generation airborne collision avoidance system. *STTT*, 19(6):717–741, 2017.

48 Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *FMCO'10*, volume 6957 of *LNCS*, pages 142–164. Springer, 2010.

49 Einar Broch Johnsen, Ka I Pun, and Silvia Lizeth Tapia Tarifa. A formal model of cloud-deployed software and its application to workflow processing. In Dinko Begusic, Nikola Rozic, Josko Radic, and Matko Saric, editors, *SoftCOM'17*, pages 1–6. IEEE, 2017.

50 Eduard Kamburjan. Detecting deadlocks in formal system models with condition synchronization. *ECEASST*, 76, 2018.

51 Eduard Kamburjan. From post-conditions to post-region invariants: Deductive verification of hybrid objects. In *HSCC*. ACM, 2021.

52 Eduard Kamburjan, Crystal Chang Din, Reiner Hähnle, and Einar Broch Johnsen. Behavioral contracts for cooperative scheduling. In *20 Years of KeY*, volume 12345 of *Lecture Notes in Computer Science*, pages 85–121. Springer, 2020.

53 Eduard Kamburjan, Reiner Hähnle, and Sebastian Schön. Formal modeling and analysis of railway operations with Active Objects. *Science of Computer Programming*, 166:167–193, November 2018.

54 Eduard Kamburjan, Rudolf Schlatte, Einar Broch Johnsen, and S. Lizeth Tapia Tarifa. Designing distributed control with hybrid active objects. In *ISoLA*, volume 12479 of *LNCS*. Springer, 2020.

55 Soonho Kong, Sicun Gao, Wei Chen, and Edmund M. Clarke. dReach: δ-reachability analysis for hybrid systems. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*, pages 200–205. Springer, 2015.

56 Jia-Chun Lin, Ingrid Chieh Yu, Einar Broch Johnsen, and Ming-Chang Lee. ABS-YARN: A formal framework for modeling hadoop YARN clusters. In Perdita Stevens and Andrzej Wasowski, editors, *FASE'16*, volume 9633 of *LNCS*, pages 49–65. Springer, 2016.

57 Jiang Liu, Jidong Lv, Zhao Quan, Naijun Zhan, Hengjun Zhao, Chaochen Zhou, and Liang Zou. A calculus for hybrid CSP. In Kazunori Ueda, editor, *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings*, volume 6461 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2010.

58 Simon Lunel, Stefan Mitsch, Benoît Boyer, and Jean-Pierre Talpin. Parallel composition and modular verification of computer controlled systems in differential dynamic logic. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods, The Next 30 Years, Third World Congress, FM, Porto, Portugal*, volume 11800 of *LNCS*, pages 354–370. Springer, 2019.

59 Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. Hybrid I/O automata. *Inf. Comput.*, 185(1):105–157, 2003.

60 Brendan P. Mahony and Jin Song Dong. Deep semantic links of TCSP and Object-Z: TCOZ approach. *Formal Aspects Comput.*, 13(2):142–160, 2002.

61 *Maxima Manual*, 5.43.0 edition, 2019. `maxima.sourceforge.net`.

62 Stefan Mitsch, Grant Olney Passmore, and André Platzer. Collaborative verification-driven engineering of hybrid systems. *Math. Comput. Sci.*, 8(1):71–97, 2014.

63 Stefan Mitsch and André Platzer. ModelPlex: Verified runtime validation of verified cyber-physical system models. *Form. Methods Syst. Des.*, 49(1):33–74, 2016.

64 Andreas Müller, Stefan Mitsch, Werner Retschitzegger, Wieland Schwinger, and André Platzer. Tactical contract composition for hybrid system component verification. *STTT*, 20(6):615–643, 2018. Special issue for selected papers from FASE'17.

65 André Platzer. Differential-algebraic dynamic logic for differential-algebraic programs. *J. of Logic and Computation*, 20(1):309–352, 2010.

66 André Platzer. The complete proof theory of hybrid systems. In *LICS*, pages 541–550. IEEE, 2012.
67 André Platzer. The structure of differential invariants and differential cut elimination. *Logical Methods in Computer Science*, 8(4):1–38, 2012.
68 André Platzer. A complete uniform substitution calculus for differential dynamic logic. *J. Automated Reasoning*, 59(2):219–265, 2017.
69 André Platzer. *Logical Foundations of Cyber-Physical Systems*. Springer, 2018.
70 André Platzer and Yong Kiam Tan. Differential equation invariance axiomatization. *J. ACM*, 67(1):6:1–6:66, 2020.
71 Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.
72 Jan-David Quesel, Stefan Mitsch, Sarah Loos, Nikos Aréchiga, and André Platzer. How to model and prove hybrid systems with KeYmaera: A tutorial on safety. *STTT*, 18(1):67–91, 2016.
73 Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *SPLC*, volume 6287 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2010.
74 Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. Software diversity: state of the art and perspectives. *STTT*, 14(5):477–495, 2012.
75 Rudolf Schlatte, Einar Broch Johnsen, Jacopo Mauro, Silvia Lizeth Tapia Tarifa, and Ingrid Chieh Yu. Release the beasts: When formal methods meet real world data. In *It's All About Coordination*, volume 10865 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2018.
76 Wen Su, Jean-Raymond Abrial, and Huibiao Zhu. Formalizing hybrid systems with Event-B and the Rodin platform. *Sci. Comput. Program.*, 94:164–202, 2014.
77 Kohei Suenaga and Ichiro Hasuo. Programming with infinitesimals: A while-language for hybrid system modeling. In *ICALP (2)*, volume 6756 of *Lecture Notes in Computer Science*, pages 392–403. Springer, 2011.
78 Casper Thule, Kenneth Lausdahl, Cláudio Gomes, Gerd Meisl, and Peter Gorm Larsen. Maestro: The INTO-CPS co-simulation framework. *Simulation Modelling Practice and Theory*, 92:45–61, 2019.
79 D. A. van Beek, Michel A. Reniers, Ramon R. H. Schiffelers, and J. E. Rooda. Foundations of a compositional interchange format for hybrid systems. In Alberto Bemporad, Antonio Bicchi, and Giorgio C. Buttazzo, editors, *HSCC'07*, volume 4416 of *LNCS*, pages 587–600. Springer, 2007.
80 Shuling Wang, Naijun Zhan, and Liang Zou. An improved HHL prover: An interactive theorem prover for hybrid systems. In Michael Butler, Sylvain Conchon, and Fatiha Zaïdi, editors, *Formal Methods and Software Engineering*, pages 382–399, Cham, 2015. Springer International Publishing.
81 Peter Y. H. Wong, Elvira Albert, Radu Muschevici, José Proença, Jan Schäfer, and Rudolf Schlatte. The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. *STTT*, 14(5):567–588, 2012.
82 Zhenkai Zhang, Emeka Eyisi, Xenofon D. Koutsoukos, Joseph Porter, Gabor Karsai, and Janos Sztipanovits. A co-simulation framework for design of time-triggered automotive cyber physical systems. *Simulation Modelling Practice and Theory*, 43:16–33, 2014.