From Post-Conditions to Post-Region Invariants: Deductive Verification of Hybrid Objects

Eduard Kamburjan
eduard@ifi.uio.no
University of Oslo and Technische Universität Darmstadt

ABSTRACT

We introduce a new system for object-oriented distributed hybrid systems to verify object invariants and method contracts. In a hybrid setting, the object invariant must not only be the post-condition of a method, but also has to hold in the *post-region* of a method, because the state of the object evolves according to continuous dynamics. The post-region describes all reachable states after method termination *before* another process runs. This set can be approximated using lightweight analysis of the class structure. The system naturally generalizes rely-guarantee reasoning of discrete object-oriented languages to hybrid systems and carries over its compositionality to hybrid systems: only one $d\mathcal{L}$-proof obligation is generated per method. By reasoning about the minimal size of the post-region, local Zeno behavior can also be analyzed. Our approach is implemented for the Hybrid Active Object language HABS.

CCS CONCEPTS

• **Theory of computation → Timed and hybrid models**; Distributed computing models; **Invariants**; **Pre- and post-conditions**; **Logic and verification**.

KEYWORDS

Hybrid Systems, Invariants, Deductive Verification, Active Objects

ACM Reference Format:
Eduard Kamburjan. 2021. From Post-Conditions to Post-Region Invariants: Deductive Verification of Hybrid Objects. In *Proceedings of HSCC '21.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Distributed cyber-physical systems are behind modern innovation drivers such as the Internet-of-Things. Support for formal verification of such systems, however, is lagging behind. Especially *deductive* verification of hybrid systems [28, 29], which does not suffer from the state-space explosion problem of model checking, has little compositionality mechanisms. This is in stark contrast to the situation for distributed discrete systems, where a rich theory of composition principles for object-oriented languages exists, e.g., method contracts [1, 26] and object invariants. Here, we show that this theory can be carried over to a hybrid setting.

HSCC '21, May 19–21, 2021, Nashville, TN

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Recently, *Hybrid Active Objects* (HAO) [22] have been proposed as an object-oriented formalism that combines the formal semantics and verification tools of low-level formalisms with the usability of mature programming languages. An HAO encapsulates continuous dynamics inside an object and defines methods to interact with it from the outside. HAOs use the Active Object concurrency model [6]: message passing concurrency with futures, as well as cooperative scheduling. With cooperative scheduling, only one process is active at each point in time (per object) and cannot be interrupted by other processes, unless it explicitly releases control over the object. Active Objects have been proven to be a suitable modeling language in industrial case studies in many domains [2, 4, 21].

Challenge. Rely-guarantee reasoning for object invariants is an established modularization technique in discrete systems. Every method may *rely* on the object invariants when it starts, but must also *guarantee* it to other methods whenever another methods runs. This results in proof obligations of the following form for each method m:

$$\mathsf{inv} \rightarrow [\mathsf{s_m}]\mathsf{inv}$$

This expresses, in dynamic logic [1, 13], that if the invariant inv holds before executing the method body $\mathsf{s_m}$, then it holds after the execution. If this is shown for all methods, every method can indeed safely assume the invariant upon its start. In hybrid systems this does not suffice, as the state evolves after $\mathsf{s_m}$ terminates.

The existing system to verify invariants of HAOs, thus, uses one proof obligation *per class*. This has several flaws that inhibit one from applying it to more realistic systems: (1) after changes in one method the whole class must be re-proven, (2) the proof obligation is exponential in the number of methods, and (3) two[1] more loop invariants must be inferred, which hampers automatization.

Overview. The main insight of this work is that the post-condition of a method is not just the object invariant inv, but inv and a term that inv holds *until the next method runs*. In HABS, the considered HAO language, every method has some condition when it can be scheduled. Additionally, we have knowledge about what methods may be scheduled upon termination of a method. Thus, we can compute the *post-region* pst of a method: a region where no method is *guaranteed* to run. The post-condition is then a term expressing that the invariant holds inside the post-region pst *while following the continuous dynamics* dyn. Such constraints are expressed with the formula $[\mathsf{dyn\&pst}]I$ in differential dynamic logic [31] ($d\mathcal{L}$). In the most simple, non-trivial case, the proof obligation is the following:

$$\mathsf{inv} \rightarrow \big[\mathsf{s_m}\big]\big(\mathsf{inv} \wedge \big[\mathsf{dyn\&pst}\big]\mathsf{inv}\big)$$

[1] One loop advances time, one loop models scheduling, as multiple methods can run without advancing time.

In the trivial case, pst is true, but by using the call structure in a program we can compute more precise post-regions. Method contracts are handled by adding them as pre- and post-conditions, independently of the post-region.

Contribution and Structure. Our main contribution is a natural generalization of rely-guarantee reasoning from discrete programming languages to hybrid object-oriented languages. The approach is implemented[2]. Additionally to object invariants we verify method contracts and, for a large subset of the considered language, local Zeno behavior. The main advantage of the system is its *modularity*: one proof obligation is used per method, changes of one method do not necessarily require to reprove the whole program.

Sec. 2 describes the HABS language and Sec. 3 the $d\mathcal{L}$ logic. Sec. 4 gives the verification system itself. Sec 5 introduces local Zeno behavior for hybrid active objects and gives an analysis for it. Sec. 6 describes the implementation and Sec. 7 gives related work and concludes.

2 HYBRID ACTIVE OBJECTS

In this section we present the *Hybrid Abstract Behavioral Specification* (HABS) language that implements Hybrid Active Objects. For brevity's sake we refrain from introducing full HABS and omit, e.g., inheritance. We refer to [22] for a complete description.

Example 2.1. Before introducing the language, we use the water tank given in Fig. 1 as an informal example. The tank keeps a water level between 3 and 10. The pictured class, `Tank` has one discrete field (`log`) and a **physical** block that introduces physical fields. A physical field is described by its initial value and an ODE. Here, `level` and `drain` model water level and drain of the tank. The drain is constant and the water level changes linear w.r.t. the drain. Additionally, an initialization block is provided, which calls the methods `up` and `down`. Each method starts with a statement that has as its guard the condition when the process will be scheduled (for `up`, at the moment the level reaches 10 while water rises). This is logged by calling the external object `log` on method `triggered`. This call is asynchronous, i.e., execution continues without waiting for it to finish. Then, the drain is adjusted and the methods calls itself recursively to react the next time.

Each object is *strongly* encapsulated. No other object can access the fields of an instance. Inside an object, only one process is active at a time. This process cannot be preempted by the scheduler — it must explicitly release control by either terminating or suspending (via **await**). These two properties make (hybrid) active object easy to analyze: approaches for sequential program analyses can be applied between two **await** statements (and method start and end).

Syntax. The syntax of HABS is given by the grammar in Fig. 2. Standard expressions `e` are defined over fields `f`, variables `v` and operators `!`, `|`, `&`, `>=`, `<=`, `+`, `-`, `*`, `/`. Differential expression `de` are expressions extended with a derivation operator `e'`. Types `T` are all class names, type-generic futures **Fut**<T>, `Real`, `Unit` and `Bool`.

A program consists of a main block and a set of classes. Each class may have a list of discrete fields that are passed as parameters on object creation, an optional physical block, a list of discrete

[2]Tools and examples available under `https://formbar.raillab.de/en/chisel/`.

```
1  class Tank(Log log){
2    physical{
3      Real level = 5; level' = drain;
4      Real drain = -1; drain' = 0;
5    }
6    { this!up(); this!down(); }
7    Unit down(){
8      await diff level <= 3 & drain < 0;
9      log!triggered(); drain =  1; this!down();
10   }
11   Unit up(){
12     await diff level >= 10 & drain > 0;
13     log!triggered(); drain = -1; this!up();
14   }
15 }
```

Figure 1: A water tank in HABS.

fields that are not passed as parameters, a set of methods and an initializing block that is executed after object creation.

The physical block is a list of field declarations followed by an ODE describing the dynamics of the declared fields. These fields are called *physical*. Methods, initializing and main block consist of statements. The non-standard statements are the asynchronous method calls (`e!m()`) described above and the following:

- The **duration**`(e)` statement advances time[3] by `e` time units. No other process may execute during this advance.
- The `e.`**get** statement reads from a future. A future is a container that is generated by an asynchronous call. Afterwards a future may be passed around. With the `statement` one can read the return values once the called process terminates. Until then, the reading process blocks and no other process can run on the object.
- The **await** `g` statement suspends the process until the guard `g` holds. A guard is either (1) a future poll `e?` that determines whether the call for the future in `e` has terminated, (2) a duration guard that advances time while *unblocking* the object or (3) a differential guard that holds once expression `e` holds.

We assume that all methods are suspension-leading, i.e., each method starts with an **await** statement. This is easily achieved by adding **await diff** `true` if a method is not suspension-leading without changing the behavior.

A useful pattern, already applied in Ex. 2.1, are controllers.

Definition 2.2 (Controller). A method is a *controller* if it (1) starts with an **await** (2) contains no other suspensions and neither **get** nor **duration** statements, (3) ends with a recursive call, (4) is called only from the initial block.

Because of (2), a controller runs instantaneous and because of (3) and (4) always potentially schedulable. Analyzing the leading guard of (1) allows one to precisely see under which conditions it can be scheduled.

[3]Time in HABS is global but symbolic, i.e., not wall time.

Prgm ::= $\overline{\texttt{CD}}$ {s} CD ::= **class** C $\left[(\overline{\texttt{T f}})\right]$ {[Phys] $\overline{\texttt{FD}}$ $\overline{\texttt{Met}}$ {s}} Met ::= T m($\overline{\texttt{T v}}$) {s;**return** e;} Programs, Classes, Methods

Phys ::= **physical** {$\overline{\texttt{DED}}$} DED ::= Real f = e : f' = de FD ::= T f[= e]; Physical Block, Fields

s ::= **while** (e) {s} | **if** (e) {s} [**else** {s}] | s;s | **await** g | [[T] e] = rhs | **duration**(e) Statements

g ::= e? | **duration**(e) | **diff** e rhs ::= e | **new** C($\overline{\texttt{e}}$) | e.**get** | e!m($\overline{\texttt{e}}$) Guards, RHS Expressions

Figure 2: HABS grammar. Notation $[\cdot]$ denotes optional elements and $\overline{\cdot}$ lists.

In Ex. 2.1 class Tank is controlled by up and down. Controllers are a natural modeling pattern: the largest Active Object case study (modeling railway operations [21] in ABS) uses controllers for almost all active classes.

Semantics are defined as structural operational semantics, i.e., rewrite rules on runtime syntax. Runtime syntax represents the state of a program. The HABS semantics are fairly standard, except that (1) we use the timed extension of Bjørk et al. [5], which allows us to keep track of time in the semantics, and (2) that time advance changes not only the time, but also the object state, because it evolves according to the **physical** block.

Runtime Syntax. Runtime syntax extends statements by the **suspend** statement, which deschedules a process. The semantics are based on Timed ABS [5].

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

$ob ::= (o, \rho, ODE, f, prc, q) \quad q ::= \overline{prc} \quad msg ::= \mathsf{msg}(o, \overline{\mathtt{e}}, fid)$

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

Figure 3: Runtime syntax of HABS.

Definition 2.3 (Runtime Syntax [5]). The runtime syntax of HABS is given by the grammar in Fig. 3. Let *fid* range over future names, o over object identities, and ρ, τ, σ over stores (i.e., maps from fields or variables to values). We use $\bar{\cdot}$ to denote lists.

A timed configuration *tcn* has a clock clock with the current time, as an expression of Real type and an object configuration *cn*. An object configuration consists of messages *msg*, futures *fut* and objects *ob*. A message $\mathsf{msg}(o, \overline{\mathtt{e}}, fid)$ contains callee o, passed parameters $\overline{\mathtt{e}}$ and the generated future *fid*. A future $\mathsf{fut}(fid, \mathtt{e})$ connects the future name *fid* with its return value e. An object $(o, \rho, ODE, f, prc, q)$ has an identifier o, an object store ρ, the current dynamic f, an active process *prc* and a set of inactive processes q. *ODE* is taken from the class declaration. A process is either terminated $\bot$ or has the form $(\tau, fid, \mathtt{rs})$: the process store τ with the current state of the local variables, its future identifier *fid*, and the statement rs left to execute. Composition $cn_1\ cn_2$ is commutative and associative.

Evaluation of Guards and Expressions. Additionally to the current store, expressions and guards are evaluated with respect to a given time and continuous dynamics. This is necessary to compute how much time may advance before a differential guard is evaluated to true. Given fixed initial values, the solution f of an ODE is unique and $f(0) = \sigma$ with $\mathbf{dom}(\sigma) = \mathbf{dom}(\rho) \cup \mathbf{dom}(\tau)$, $\forall x \in \mathbf{dom}(\rho).\ \sigma(x) = \rho(x)$ and $\forall x \in \mathbf{dom}(\tau).\ \sigma(x) = \tau(x)$. We denote this composition of functions with $\sigma = \rho \circ \tau$. Non-phyiscal fields do not evolve.

$$[\![\mathtt{g}]\!]^{f,0}_{\sigma} = \text{true} \iff mte^{f}_{\sigma}(\mathtt{g}) \leq 0$$

$$mte^{f}_{\sigma}(\textbf{duration}(\mathtt{e})) = [\![\mathtt{e}]\!]^{f,0}_{\sigma}$$

$$mte^{f}_{\sigma}(\mathtt{e?}) = \begin{cases} 0 & \text{if } [\![\mathtt{e}]\!] \text{ is resolved} \\ \infty & \text{otherwise} \end{cases}$$

$$mte^{f}_{\sigma}(\textbf{diff}\ \mathtt{e}) = \operatorname*{argmin}_{t \geq 0} \left([\![\mathtt{e}]\!]^{f,t}_{\sigma} = \text{true} \right)$$

Figure 4: Semantics of Guards

Definition 2.4 (Expressions). Let f be a continuous dynamics of class C, i.e., a mapping from $\mathbb{R}^+$ to stores and $\sigma = f(0)$ the current state. Let $\mathtt{f}_p$ be a physical field and $\mathtt{f}_d$ a non-physical field. The semantics of fields $\mathtt{f}_p$, $\mathtt{f}_d$, unary operators $\sim$ and binary operators $\oplus$ after t time units is defined as follows:

$$[\![\mathtt{f}_d]\!]^{f,t}_{\sigma} = \sigma(\mathtt{f}_d) \qquad [\![\mathtt{f}_p]\!]^{f,t}_{\sigma} = f(t)(\mathtt{f}_p)$$

$$[\![\sim\mathtt{e}]\!]^{f,t}_{\sigma} = \sim[\![\mathtt{e}]\!]^{f,t}_{\sigma} \qquad [\![\mathtt{e} \oplus \mathtt{e'}]\!]^{f,t}_{\sigma} = [\![\mathtt{e}]\!]^{f,t}_{\sigma} \oplus [\![\mathtt{e'}]\!]^{f,t}_{\sigma}$$

Evaluation of guards is defined in two steps: First, the *maximal time elapse* is computed. I.e., the maximal time that may pass without the guard expression evaluating to true. For differential guards **diff** e this is the minimal time until e becomes true. Then, the guard evaluates to true if no time advance is needed.

Definition 2.5 (Guards). Let f be a continuous dynamic of object o in state σ. The maximal time elapse (*mte*) and evaluation of guards is given in Fig. 4.

Discrete Transition System. Fig. 5 gives the most important rules for the semantics of a single object, the omitted rules are given in [5]. The rule (1) introduces a **suspend** statement in front of an **await** statement. Rule (2) consumes a **suspend** statement and moves a process into the queue of its object—at this point, the ODEs are translated into some dynamics with sol. Rule (3) activates a process with a following **await** statement, if its guard evaluates to true. An analogous rule (not shown in Fig. 5) activates a process with any other non-**await** statement. Rule (4) evaluates an assignment to a local variable. The rule for field is analogous. Rule (5) realizes a termination (with solutions of the ODEs) and (6) a future read. Finally, (7) is a method call that generates a message.

Continuous Transition System. For configurations, there are two rules, shown in Fig. 6. Rule (i) realizes a step of an object without advancing time. Only if (i) is not applicable, rule (ii) can be applied.

$$
\begin{aligned}
&\text{(1)}\ (o,\rho,ODE,f,(\tau,fid,\textbf{await}\ \texttt{g};\texttt{s}),q) &&\to (o,\rho,ODE,f,(\tau,fid,\textbf{suspend};\textbf{await}\ \texttt{g};\texttt{s}),q)\\
&\text{(2)}\ (o,\rho,ODE,f,(\tau,fid,\textbf{suspend};\texttt{s}),q) &&\to (o,\rho,ODE,\mathrm{sol}(ODE,\rho),\bot,q\circ(\tau,fid,\texttt{s}))\\
&\text{(3)}\ (o,\rho,ODE,f,\bot,q\circ(\tau,fid,\textbf{await}\ \texttt{g};\texttt{s})) &&\to (o,\rho,ODE,f,(\tau,fid,\texttt{s}),q) \qquad \text{if } [\![g]\!]^{f,0}_{\rho\circ\tau}=\text{true}\\
&\text{(4)}\ (o,\rho,ODE,f,(\tau,fid,\texttt{v = e;s}),q) &&\to (o,\rho,ODE,f,(\tau[\texttt{v}\mapsto[\![\texttt{e}]\!]^{f,0}_{\rho\circ\tau}],fid,\texttt{s}),q) \qquad \text{if e contains no call or } \textbf{get}\\
&\text{(5)}\ (o,\rho,ODE,f,(\tau,fid,\textbf{return}\ \texttt{e;}),q) &&\to (o,\rho,ODE,\mathrm{sol}(ODE,\rho),\bot,q)\ \mathrm{fut}(fid,[\![\texttt{e}]\!]^{f,0}_{\rho\circ\tau})\\
&\text{(6)}\ (o,\rho,ODE,f,(\tau,fid,\texttt{v = e.}\textbf{get};\texttt{s}),q)\ \mathrm{fut}(fid,\texttt{e'}) &&\to (o,\rho,ODE,f,(\tau,fid,\texttt{v = e';s}),q) \qquad \text{if } [\![\texttt{e}]\!]^{f,0}_{\rho\circ\tau}=fid\\
&\text{(7)}\ (o,\rho,ODE,f,(\tau,fid,\texttt{v = e!m(e}_1,\ldots\texttt{e}_n\texttt{;s}),q) &&\to (o,\rho,ODE,f,(\tau[\texttt{v}\mapsto f'],fid,\texttt{s}),q)\ \mathrm{msg}([\![\texttt{e}]\!]^{f,0}_{\rho\circ\tau},([\![\texttt{e}_1]\!]^{f,0}_{\rho\circ\tau},\ldots,[\![\texttt{e}_n]\!]^{f,0}_{\rho\circ\tau}),fid_2) \qquad \text{where } fid_2 \text{ is fresh}
\end{aligned}
$$

Figure 5: Selected rules for HABS objects.

It computes the global maximal time elapse and advances the time in the clock and all objects.

$$
\begin{aligned}
&\text{(i)}\ \mathrm{clock}(t)\ cn\ cn' \to \mathrm{clock}(t)\ cn''\ cn' \qquad \text{with } cn\to cn''\\
&\text{(ii)}\ \mathrm{clock}(t)\ cn \to \mathrm{clock}(t+t')\ adv(cn,t')\\
&\quad \text{if (i) is not applicable and } mte(cn)=t'\neq\infty
\end{aligned}
$$

Figure 6: Timed semantics of HABS configurations.

$$
\begin{aligned}
mte(cn\ cn') &= \mathbf{min}(mte(cn),mte(cn'))\\
mte(msg) &= mte(fut)=\infty\\
mte(o,\rho,ODE,f,prc,q) &= [\![\mathbf{min}_{q'\in q}(mte(q'),\infty)]\!]_\rho\\
mte(\tau,fid,\textbf{await}\ \texttt{g;s}) &= [\![mte(g)]\!]_\tau\\
mte(\tau,fid,\texttt{s}) &= \infty \text{ if s} \neq \textbf{await}\ \texttt{g;s'} \qquad mte(\textbf{duration}(\texttt{e})) = \texttt{e}\\
mte^f_\sigma(\textbf{diff}\ \texttt{e}) &= \mathop{\mathbf{argmin}}_{t\geq 0}\left([\![\texttt{e}]\!]^{f,t}_\sigma=\text{true}\right) \qquad mte(\texttt{e?})=\infty
\end{aligned}
$$

$$
\begin{aligned}
adv_{prc}((\tau,fid,\texttt{s}),f,t) &= (\tau,fid,\texttt{s}) \text{ if s} \neq (\textbf{await duration}(\texttt{e});\ \texttt{s'})\\
adv_{prc}((\tau,fid,\textbf{await duration}(\texttt{e});\texttt{s}),f,t) &=\\
&(\tau,fid,\textbf{await duration}(\texttt{e+}t);\texttt{s})
\end{aligned}
$$

$$
adv_{heap}(\rho,f,t)(\texttt{f}) = \begin{cases} \rho(\texttt{f}) & \text{if f is not physical}\\ f(t)(\texttt{f}) & \text{otherwise}\end{cases}
$$

Figure 7: Auxiliary functions.

Fig. 7 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. State change during a time advance is handled by a family of functions *adv* which are applied to all stores and objects. We only give two members of the family: adv_{heap} takes as parameter a store σ, the dynamics f and a duration t, adv_{prc} takes a process *prc*, the dynamics f and a duration t. Both advance its first parameter by t time units according to f. The state evolves according to the current dynamics and the guards of **duration** guards and statements are decreased by t (if the **duration** clause is part of the first statement of an unscheduled process). For non-hybrid Active Objects $adv_{heap}(\rho,t)=\rho$.

The semantics trigger the guards as soon as possible. This is crucial for the simulation capabilities of HABS. The more relaxed semantics of, e.g., hybrid automata, introduce additional non-determinism that is beneficial for verification but difficult to handle meaningfully for simulation. We remind that we only allow weak inqualities, i.e., an event boundary always exists and the minimum indeed exists.

Runs. The semantics of a programs are expressed as *run*, generated by the operational semantics, and for each run a set of traces, one for each object.

A trace θ is a mapping from $\mathbb{R}^+$ to states, meaning that at time t the state of the program is $\theta(t)$. A trace is extracted from a run by interpolating between two configurations resulting from discrete steps using the last solution. We say that $\mathrm{clock}(t_i)\ cn_i$ is the final configuration at t_i in a run, if any other timed configuration $\mathrm{clock}(t_i)\ cn'_i$ is before it.

Definition 2.6 (Traces). Let Prgm be a program. Its initial state configuration is denoted $\mathrm{clock}(0)\ cn_0$ [5]. A run of Prgm is a (possibly infinite) reduction sequence

$$\mathrm{clock}(0)\ cn_0 \to \mathrm{clock}(t_1)\ cn_1 \to \cdots$$

A run is *time-convergent* if it is infinite and $\lim_{i\mapsto\infty} t_i < \infty$. A run is *locally terminated* if every process within terminates.

For each object o occurring in the run, its *trace* is defined as θ_o is

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

We normalize all traces and let them start with 0 by shifting all states by the time the object is created.

Example 2.7. Consider Fig. 8 which illustrates the semantics of tank from Ex. 2.1, starting at time 1. The store is $\rho_1 = \{\texttt{level}\mapsto 4, \texttt{drain}\mapsto -1, \texttt{log}\mapsto l\}$ and two processes are suspended. The one for up is denoted *prc*, the one for down has the remaining statement $\texttt{s}^+_{\texttt{down}}$, which is the whole method body.

Nothing can execute without advancing time, so time is advanced by 1 time unit (using rule (ii)) until the store changes to $\rho_1 = \{\texttt{level}\mapsto 3, \texttt{drain}\mapsto -1, \texttt{log}\mapsto l\}$.

This enables rule (3) to schedule the process for down, where the **await** statement is removed: $\texttt{s}_{\texttt{down}}$ is the method body without the leading suspension. Finally, rule (7) is used to generate a message to call the other object.

$$\begin{aligned}
&\mathsf{clock}(1)\ \left(o, \rho_1, \mathit{ODE}, f, \bot, \left\{(\emptyset, \mathit{fid}_1, \mathsf{s}^{+}_{\mathsf{down}}), \mathit{prc}\right\}\right)\\
\xrightarrow{(ii)}\ &\mathsf{clock}(2)\ \left(o, \rho_2, \mathit{ODE}, f, \bot, \left\{(\emptyset, \mathit{fid}_1, \mathsf{s}^{+}_{\mathsf{down}}), \mathit{prc}\right\}\right)\\
\xrightarrow{(3)}\ &\mathsf{clock}(2)\ \left(o, \rho_2, \mathit{ODE}, f, (\emptyset, \mathit{fid}_1, \mathsf{s}_{\mathsf{down}}), \left\{\mathit{prc}\right\}\right)\\
\xrightarrow{(7)}\ &\mathsf{clock}(2)\ \left(o, \rho_2, \mathit{ODE}, f, (\emptyset, \mathit{fid}_1, \ldots), \left\{\mathit{prc}\right\}\right)\quad \mathsf{msg}(o, \mathit{log}, \mathit{fid}_2)
\end{aligned}$$

Figure 8: Example semantics of a tank from Ex. 2.1.

3 DIFFERENTIAL DYNAMIC LOGIC

Differential dynamic logic [30, 33] is a first-order dynamic logic implemented in the KeYmaera X tool [10] that embeds hybrid programs into its modalities. Hybrid programs are defined by a simple while-language. extended with a statement for ordinary differential equations (ODEs). Such a statement evolves the state according to some dynamics for a non-deterministically chosen amount of time.

Definition 3.1 (Syntax of $d\mathcal{L}$). Let p range over predicate symbols (such as $\doteq, \geq$), f over function symbols (such as $+, -$) and x over variables. Hybrid programs α, formulas φ and terms t are defined by the following grammar.

$$\varphi ::= p(\bar{t}) \mid \neg\varphi \mid \varphi \wedge \varphi \mid \exists \mathsf{x}.\ \varphi \mid [\alpha]\varphi$$
$$t ::= f(\bar{t}) \mid \mathsf{x} \qquad dt ::= f(\overline{dt}) \mid t \mid (t)'$$
$$\alpha ::= \mathsf{x} := t \mid \mathsf{x} := * \mid \alpha \cup \alpha \mid \alpha^* \mid ?\varphi \mid \alpha;\alpha \mid \{\alpha\} \mid \overline{\mathsf{x} = dt}\&\varphi$$

Modalities may be nested using the ? operator and all ODEs are autonomous. The semantics of hybrid programs is as follows: Program $\mathsf{x} := t$ assigns the value of t to x. Program $\mathsf{x} := *$ assigns a non-deterministically chosen value to x. Program $\alpha_1 \cup \alpha_2$ is a non-deterministic choice. Program α^* is the Kleene star. Program $?\varphi$ is a test or filter. It either discards a run (if φ does not hold) or performs no action (if φ does hold). Program $\alpha_1;\alpha_2$ is sequence and $\{\alpha\}$ is a block for structuring. Finally, the ODE $\overline{\mathsf{x} = dt}\&\varphi$ evolves the state according to the given ODE in evolution domain φ for some amount of time. The evolution domain describes where a solution is allowed to evolve, not the solution itself. The semantics of the first-order fragment is completely standard. The semantics of $[\alpha]\varphi$ is that φ has to hold in *every* post-state of α if α terminates. We stress that if α is an ODE, then this means that φ holds throughout the *whole* solution.

Example 3.2. The following formula expresses that the dynamics of bouncing ball with position x and velocity v are below 10 before the ball reaches the ground, when starting with a velocity of 0 and a height below 10.

$$0 \leq \mathsf{x} \leq 10 \wedge \mathsf{v} \doteq 0 \rightarrow [\mathsf{x}' = \mathsf{v}, \mathsf{v}' = -9.81\&\mathsf{x} \geq 0]\mathsf{x} \leq 10$$

Events can be expressed as usual by an event boundary created between a test and an evolution domain. The following program models that the ball repeatedly bounces back exactly on the ground.

$$\left(\{\mathsf{x}' = \mathsf{v}, \mathsf{v}' = -9.81\&\mathsf{x} \geq 0\}; ?\mathsf{x} \leq 0; \mathsf{v} := -\mathsf{v} * 0.9\right)^*$$

We identify HABS variables and fields with $d\mathcal{L}$ variables and denote with trans(e) the straightforward translation of HABS expressions into $d\mathcal{L}$ terms. Standard control flow (such as while and if) is encoded using the operators above [31].

4 REGION-BASED VERIFICATION

First, let us make more precise what we mean by object invariants. Classically, an object invariant has to hold whenever a method starts or ends. This is not sufficient for hybrid systems, because whenever a method terminates, the dynamics may change the state during a time advance and result in a state where the invariant does not hold. We, thus, introduce object invariants as a property that has to hold (1) whenever a method starts or ends and (2) whenever time advances. This also includes the case where time advances but a process is active. This ensures that an object is safe, even if its discrete part is deadlocked.

Having fixed the notion of object invariants, we can now examine how this effects traces. For simplicity, let us ignore **get** and **duration** statements for now. It is clear that we have to verify the state directly after a method releases control. Additionally, all the states that are part of the trace before the next process is scheduled. We must, thus, express that the invariant is preserved by the dynamics until the next process is scheduled.

To do so, it is critical to give an overapproximation of the states following a suspension/termination. If more about these states is known, then the verification becomes *more precise* but *less compositional*. We give three techniques, each with a different trade-off between compositionallity and precision.

Basic Regions As a baseline, we can use true to describe all states. We introduce this in Sec. 4.1 as *basic* regions. This technique is the least precise, but the most compositional one.

```
1 class C{
2  physical{ ... }
3  { this!c(); }
4  Unit m(){ ...; this!n();}
5  Unit n(){ await x >= 10; }
6  //controller
7  Unit c() { await duration(2); ...; this!c();}
8 }
```

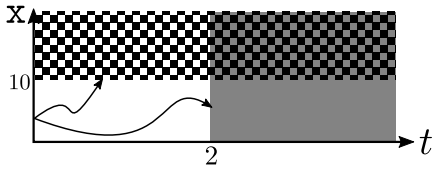

Figure 9: Post-region of C.m. The arrows are (partial) traces after C.m terminates. The checkered region is locally controlled by the call to C.n. The shaded region is struturally controlled by C.c. Both these areas are not relevant for verifition: the post-region is only the white region.

Locally Controlled Regions A more precise approximation is possible by analyzing the calls of a method. We remind

that all methods are suspending-leading. If we know that a method m is called before the last suspension/termination, then the guard g of m describes the state where the called process is scheduled at the latest. It suffices to show that the object remains save until g holds. The object may schedule a process earlier (e.g., if called from the outside), but this process may then also assume the invariant. We introduce this as *locally controlled* regions in Sec. 4.2. This technique is more precise than using basic regions, but eventually less compositional: if a method m is removed, all methods calling m must be repoven because their post-regions changes, even if m has no contract.

Structurally Controlled Regions Finally, if a controller is used in the class, then its suspension conditions also describe conditions when the next (controller) process is scheduled at the latest. We introduce this as *structurally controlled* regions in Sec. 4.3. This technique is even more precise than using locally controlled regions, but eventually even less compositional: removal of a controller requires to reprove all other methods, as all post-regions change.

Example 4.1. Consider the class and illustration in Fig. 9. After m terminates, safety has to be established for all states until the next process is scheduled. The upper (checkered) region cannot be part of these states — as m called n, at the latest when x >= 10 holds this process will run. This is the locally controlled region. Similarly, after at most 2 time units, the controller will run. This is the right (shaded) region, the structurally controlled region. Thus, any of the dynamics (pictured as lines) must only establish safety for the unshaded region. This is the post-region of m. Note that we do not verify the post-region itself, but all dynamics inside the post-region.

Specification. We specify objects with object invariants and methods with simple method contracts directly in HABS using expressions. Extracting post-conditions from futures is described in [20] and not specific to hybrid active objects.

Objects. Object invariants are specified as an annotation at the physical block. Additionally, we specify a creation condition that has to hold once an instance is created. Both creation condition and object invariant can only refer to the fields of the specified class, not to the fields of referenced objects. We refer to the creation condition of a class C with init_C and to the invariant with inv_C.

Methods. A method may be annotated with a pre-condition and a post-condition. The pre-condition is denoted pre_m and can only refer to the parameters of the method. The post-condition is denoted with post_m and may refer only to fields and the special symbol result for the return value. Fig. 11 gives an example.

4.1 Basic Regions

The proof obligation adds the translation of the method body into a surrounding logical formula. The translation itself is straightforward, but as $d\mathcal{L}$ allows only to verify post-conditions, we must encode suspensions, object creations and method calls differently. E.g., to deal with pre-conditions, we test whether it holds at the point the call is made. If it does, no further action is made. If it does *not* hold, we set a special variable cll to 1. The pre- and post-conditions encode that cll is always 0, i.e., all conditions hold.

$$
\begin{aligned}
\mathsf{trans}_b(s_1;s_2) &= \mathsf{trans}_b(s_1);\mathsf{trans}_b(s_2)\\
\mathsf{trans}_b(\textbf{if}(e)\{s_1\}\textbf{else}\{s_2\}) &= \mathsf{if}(\mathsf{trans}(e))\{\mathsf{trans}_b(s_1)\}\mathsf{else}\{\mathsf{trans}_b(s_2)\}\\
\mathsf{trans}_b(\textbf{while}(e)\{s_1\}) &= \mathsf{while}(\mathsf{trans}(e))\{\mathsf{trans}_b(s_1)\}\\
\mathsf{trans}_b(v = e) &= v := \mathsf{trans}(e)\\
\mathsf{trans}_b(\textbf{return}\ e) &= \texttt{result} := \mathsf{trans}(e)\\
\mathsf{trans}_b(v = e.\textbf{get}) &= \{\{?\psi\} \cup \{?\neg\psi;\mathsf{fail}\}\};v := *\\
\mathsf{trans}_b(\textbf{await}\ g) &= \{\{?\psi\} \cup \{?\neg\psi;\mathsf{fail}\}\};\mathsf{havoc};?(\mathsf{trans}(g) \wedge \mathsf{inv}_C)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{trans}_b(v = e!m(e_1,\dots)) =\\
&\qquad \{\{?\mathsf{pre}_m(e_1,\dots)\} \cup \{?\neg\mathsf{pre}_m(e_1,\dots);\mathsf{fail}\}\};\\
&\qquad v := *\\
&\mathsf{trans}_b(v = \textbf{new}\ C(e_1,\dots)) =\\
&\qquad \{\{?\mathsf{init}_C(e_1,\dots)\} \cup \{?\neg\mathsf{pre}_C(e_1,\dots);\mathsf{fail}\}\};\\
&\qquad v := *\\
&\mathsf{trans}_b(\textbf{duration}(e)) =\\
&\qquad t := 0;\{\{?\psi(e)\} \cup \{?\neg\psi(e);\mathsf{fail}\}\};\\
&\qquad t := 0;\{\mathsf{ode}, t' = 1 \& t \le \mathsf{trans}(e)\};?\mathsf{trans}(e)
\end{aligned}
$$

Figure 10: Translation of HABS-statements into $d\mathcal{L}$ programs.

Let $trig_m$ be the differential guard of m if such a guard exists and true otherwise (analogously for translation of guards trans(g)). Let s_m be the method body of m without the leading suspension. In the special case of init this is the initial block followed by all field initializations.

Basic regions are suited to verify objects which have to be stable without an explicit control loop or a specific order of method calls. They do not use any additional information about the program and can be a sound fallback if no better regions can be computed.

Theorem 4.2 (Basic Regions). *Let* C *be a class with dynamics* ode. *Let*

$$\psi = \mathsf{inv}_C \wedge [\mathsf{ode}\&\mathsf{true}]\mathsf{inv}_C$$
$$\psi(e) = \mathsf{inv}_C \wedge [\mathsf{ode}, t' = 1 \& t \le e]\mathsf{inv}_C$$
$$\mathsf{fail} = \mathtt{cll} := 1 \quad \mathsf{havoc} = f_1 := *; \dots f_n := *; \quad \textit{for all fields } f_i$$

Translation trans_b *of statements is given in Fig. 10. Class* C *is safe if the formula*

$$\mathsf{init}_C \wedge \mathtt{cll} \doteq 0 \rightarrow \left[\mathsf{trans}_b(s_{\mathsf{init}})\right](\mathtt{cll} \doteq 0 \wedge \psi)$$

is valid and for each method m *in* C *the following formula is also valid:*

$$\mathsf{inv}_C \wedge \mathsf{pre}_m \wedge \mathtt{cll} \doteq 0 \rightarrow \left[?trig_m;\mathsf{trans}_b(s_m)\right](\mathsf{post}_m \wedge \mathtt{cll} \doteq 0 \wedge \psi)$$

If all classes are safe and the following formula for main block s_0 *is valid,*

$$\mathtt{cll} \doteq 0 \rightarrow \left[\mathsf{trans}_b(s_0)\right]\mathtt{cll} \doteq 0$$

then inv_D *holds in every state of every locally terminating, time-divergent trace of every object* o *realizing any class* D*, whenever (1)* o *is inactive or (2) time advances. The pre-condition of a method holds in every prestate and the post-condition in every poststate.*

Proof Idea. Let θ be a trace as required by the theorem. We need to show that inv_C holds at all $\theta(x), x \ge 0$ and that the invariant

holds whenever a process is descheduled (note that not all these states may occur in the trace, as they can be not final).

We may assume each pre-condition, as they are part of discrete part of HABS: their soundness is shown in [20] and requires the third proof obligation.

For $x = 0$, the statement follows from the validity of the first proof obligation. Let D be the set of time points where a discrete rule was applied. Let $\mathrm{succ}(i)$ be the successor of $i \in D$ in D. Now, we chop the trace θ into segments $\theta_i = [\theta(i), \theta(\mathrm{succ}(i))]$. I.e., subtraces that model exactly the states between two discrete steps. For this to work, we observe that (1) we only consider time-divergent traces, so D is countable and (3) we do not allow strong inequalities in guards, so the interval is always well-defined.

The rest of the proof is an induction on D. The base case is the run of the first process, it may assume the invariant (as it starts at $x = 0$) and the translation ensures that once it hits the first suspension the invariant holds (as otherwise the proof would fail). We must now show that the invariant holds in $[\theta(\min D), \theta(\mathrm{succ}(\min D))]$. We have just shown $\theta(\min D)$. We observe that $\mathrm{succ}(\min D)$ must be reached following the class dynamics — this is exactly the second proof obligation. The induction step is analogous.

The core of the translation is the formula ψ. It checks that the invariant holds in the current state and does so forever. Its variation $\psi(\mathtt{e})$ checks that the invariants holds for the next e time units.

The first formula expresses that the object is created in a state such that it remains safe. The formulas for methods express that if the method is run in a safe state, then the state is safe afterwards and remains safe. The formula for the main block only checks that pre-conditions are adhered to.

The translation of sequence, branching, iteration and side-effect free assignment is straightforward. Calls and creation realize the aforementioned pattern and set `cll` to 1 when the pre-condition is not adhered to. Terms $\mathrm{init}_{\mathtt{C}}(\dots)$ and $\mathrm{pre}_{\mathtt{m}}(\dots)$ are the pre-condition instantiated for the concrete call parameter expressions. Translation of duration first checks that the invariant holds when the state evoles for the given amount of time and then advances the state for the rest of the method. Translation of a **get** checks for ψ, as time may advance until the future is resolved. In the translation of **await**, which works as the one for **get**, we also remove all information about the fields except that the guard and the invariant holds using havoc. This is needed because the state may have evolved.

It is standard to consider only time-divergent runs. The use of locally terminating runs corresponds to partial correctness and is necessary because the box modality only verifies its post-condition in case of termination. The post-condition of the method contract only has to hold in the post-state itself.

Example 4.3. Fig. 11 describes a simple class where the field `v` is specified to follow limited growth with limit `bnd` and growth rate `rate`. It is specified that the value of `v` is always between 0 and `bnd` and the rate is between 0 and 1. Both rate and bound can be reset, but the bound can only be increased. The rate has to observe the specified interval. The bound is set after a dynamic check, because the field is not visible from the outside, while the new growth rate is specified with a method pre-condition. Note that safety relies on the fact that the method contract is adhered to. The four proof

```
1  interface Element {
2   Unit inBound(Real nB);
3   [HybridSpec: Requires(nR > 0 && nR < 1)]
4   Unit inRate(Real nR);
5   [HybridSpec: Ensures(0 < result)]
6   Real outV();
7  }
8
9  [HybridSpec:
10  Requires(inV > 0 && inB > inV && inR < 1 && inR > 0)]
11 [HybridSpec:
12  ObjInv(v > 0 && bnd > v  && rate < 1 && rate > 0)]
13 class Element(Real inV, Real inR, Real inB) implements
     Element{
14  physical{
15   Real rate  = inR : rate'  = 0;
16   Real bnd   = inB : bnd' = 0;
17   Real v     = inV : v'     = rate*(bnd-v);
18  }
19  Unit inBound(Real nB){ if(nB >= bnd) bnd = nB; }
20  Unit inRate(Real nR){ this.rate = nR; }
21  Real outV(){ return this.v; }
22 }
```

Figure 11: Controlled limited growth.

obligations are as follows:

$$init \wedge \mathtt{cll} \doteq 0 \rightarrow \left[?\mathsf{true}; \mathtt{bnd} := \mathtt{inB}; \mathtt{rate} := \mathtt{inR}; \mathtt{v} := \mathtt{inV}\right] phy \quad (1)$$

$$I \wedge \mathtt{cll} \doteq 0 \rightarrow \left[?\mathsf{true}; \mathtt{if}(\mathtt{nB} \geq \mathtt{bnd})\mathtt{bnd} := \mathtt{nB}\right] phy \quad (2)$$

$$I \wedge \mathtt{cll} \doteq 0 \rightarrow \left[?\mathsf{true}; \mathtt{result} := \mathtt{v}\right] (phy \wedge 0 < \mathtt{result}) \quad (3)$$

$$I \wedge 0 < \mathtt{nR} < 1 \wedge \mathtt{cll} \doteq 0 \rightarrow \left[?\mathsf{true}; \mathtt{rate} := \mathtt{nR}\right] phy \quad (4)$$

Where

$$I \equiv \mathtt{v} > 0 \wedge \mathtt{bnd} > \mathtt{v} \wedge \mathtt{rate} < 1 \wedge \mathtt{rate} > 0$$

$$init \equiv \mathtt{inV} > 0 \wedge \mathtt{inB} > \mathtt{inV} \wedge \mathtt{inR} < 1 \wedge \mathtt{inR} > 0$$

$$phy \equiv \big((I \wedge \mathtt{cll} \doteq 0 \wedge [\mathtt{rate}' = 0, \mathtt{bnd}' = 0, \mathtt{v}' = \mathtt{rate} * (\mathtt{bnd} - \mathtt{v})\&\mathsf{true}] I\big)$$

The above proof obligations can be automatically closed by `KeYmaera X`. We can improve the analysis slightly by excluding methods which obviously do not influence the invariant. This is a simple version of frames [12].

Lemma 4.4. *Lem. 4.2 holds also if no proof obligations are generated for methods that (1) do not assign to any field that is read in the invariant, (2) make no method calls, (3) create no object, and (4) have no* `Ensures` *specification.*

Basic regions are more compositional than the component-based system of [22], where any change to the class requires the whole class to be reproven. For basic regions, only an added or modified method needs to be (re)proven.

Lemma 4.5. *Let* `C` *be a safe class according to Thm. 4.2. Let* $\mathtt{C}^-$ *be* `C` *with some method removed*[4] *and* $\mathtt{C}^+$ *be* `C` *with some method added.*

[4] I.e., it is removed from the class and all calls to it are replaced by **skip**.

```
1 class Tank {
2     [HybridSpec: ObjInv("x >= 3 & x <= 10")]
3     physical{ Real x = 5: x' = v; Real v = -1; v' = 0;
  }
4     { this!down(); }
5     Unit down(){ await x <= 3; v = 1; this!up(); }
6     Unit up(){ await x >= 10; v = -1; this!down(); }
7 }
```

Figure 12: Water Tank with Local Control

(1) $\mathtt{C}^-$ is safe.
(2) To verify $\mathtt{C}^+$, only the proof obligation of the new method has to be shown.

If another class calls the added method, only the proof obligation of the calling methods has to be shown to make the calling class safe.

4.2 Locally Controlled Regions

Next, we use the call structure for more precision: Methods called within m are guaranteed to be in the process pool afterwards — the negation of their condition can be added to the post-region. We call such regions locally controlled, as each process locally starts another process to limit its post-region.

Example 4.6. Consider the class in Fig. 12. It models a water tank with two event boundaries, one for the upper limit and one for the lower limit. The invariant holds, as whenever the water is rising, method up is active and will react before the tank overflows and analogously for down.

We use again a special variable t to model time advance, and some auxiliary structures. Function trans_u translates statements but also keeps track of the functions which are guaranteed to be schedulable at a suspension. It is defined in Fig. 13.

Additionally, $\mathit{ttrig}_{\mathtt{g}}$ is the external trigger of g. If g is a differential guard, then $\mathit{ttrig}_{\mathtt{g}}$ is the translation of the guard expression. If g is a duration guard with parameter e then

$$\mathit{ttrig}_{\mathtt{g}} = \mathtt{t} \geq \mathrm{trans}(\mathtt{e})$$

Otherwise, $\mathit{ttrig}_{\mathtt{g}} = \mathrm{false}$. We denote the external trigger of the leading guard of m with $\mathit{ttrig}_{\mathtt{m}}$. We use weak negation to preserve event boundaries. It is defined as normal negation, except for weak inequalities:

$$(\widetilde{\neg} t_1 \leq t_2) = t_1 \geq t_2 \qquad (\widetilde{\neg} t_1 \geq t_2) = t_1 \leq t_2$$

The major difference to Thm. 4.2 is that the translation keeps track of the methods called so far and adds their extended trigger to the controlled region at the termination and suspension of a method. To do so, ψ is extended with a parameter CM, the set of called methods that are guaranteed to be considered for scheduling at the next suspension. This set is computed on-the-fly during translation.

Theorem 4.7 (Locally Controlled Regions). *Let* C *be a class with dynamics* ode. *Let* t *be a fresh variable. Let*

$$\psi(\mathrm{CM}, \mathtt{g}) = \mathrm{inv}_{\mathtt{C}} \wedge \left[\mathtt{t}:=0; \mathrm{ode}, \mathtt{t}' = 1 \,\&\, \left(\widetilde{\neg}\mathit{ttrig}_{\mathtt{g}} \wedge \bigwedge_{\mathtt{m}\in\mathrm{CM}} \widetilde{\neg}\mathit{ttrig}_{\mathtt{m}}\right)\right] \mathrm{inv}_{\mathtt{C}}$$

$$\psi(\mathrm{CM}) = \mathrm{inv}_{\mathtt{C}} \wedge \left[\mathtt{t}:=0; \mathrm{ode}, \mathtt{t}' = 1 \,\&\, \left(\bigwedge_{\mathtt{m}\in\mathrm{CM}} \widetilde{\neg}\mathit{ttrig}_{\mathtt{m}}\right)\right] \mathrm{inv}_{\mathtt{C}}$$

C *is safe (with locally controlled regions) if the formula*

$$\mathrm{pre}_{\mathtt{C}} \wedge \mathtt{cll} \doteq 0 \rightarrow \left[\widehat{\mathtt{s}}\right]\left(\mathtt{cll} \doteq 0 \wedge \psi(\mathrm{CM})\right)$$

(with $\mathrm{trans}_u(\mathtt{s}_{\mathtt{init}}, \emptyset) = (\widehat{\mathtt{s}}, \mathrm{CM})$*) and for each method* m *in* C *the formula*

$$\mathrm{inv}_{\mathtt{C}} \wedge \mathrm{pre}_{\mathtt{m}} \wedge \mathtt{cll} \doteq 0 \rightarrow \left[?\mathit{trig}_{\mathtt{m}}; \widehat{\mathtt{s}_{\mathtt{m}}}\right]\left(\mathrm{post}_{\mathtt{m}} \wedge \mathrm{inv}_{\mathtt{C}} \wedge \psi(\mathrm{CM}) \wedge \mathtt{cll} \doteq 0\right)$$

(with $\mathrm{trans}_u(\mathtt{s}_{\mathtt{m}}, \emptyset) = (\widehat{\mathtt{s}_{\mathtt{m}}}, \mathrm{CM})$*) is valid. If all classes in a program are safe and the following formula for the main block is valid,*

$$\mathtt{cll} \doteq 0 \rightarrow \left[\mathrm{trans}_b(\mathtt{s}_{\mathbf{main}})\right]\mathtt{cll} \doteq 0$$

then $\mathrm{inv}_{\mathtt{D}}$ *holds in every state of every locally terminating, time-divergent trace of every object o realizing any class* D*, whenever (1) o is inactive or (2) time advances. The pre-condition of a method holds in every prestate and the post-condition in every poststate.*

$\mathrm{trans}_u(\mathtt{s}_1; \mathtt{s}_2, \mathrm{CM}) = (\widehat{\mathtt{s}_1}; \widehat{\mathtt{s}_2}, \mathrm{CM}_2)$
where $\mathrm{trans}_u(\mathtt{s}_1, \mathrm{CM}) = (\widehat{\mathtt{s}_1}, \mathrm{CM}_1)$ *and* $\mathrm{trans}_u(\mathtt{s}_2, \mathrm{CM}_1) = (\widehat{\mathtt{s}_2}, \mathrm{CM}_2)$

$\mathrm{trans}_u(\mathbf{if}(\mathtt{e})\{\mathtt{s}_1\}\mathbf{else}\{\mathtt{s}_2\}, \mathrm{CM}) = (\mathrm{if}(\mathrm{trans}(\mathtt{e}))\{\widehat{\mathtt{s}_1}\}\mathrm{else}\{\widehat{\mathtt{s}_2}\}, \mathrm{CM}_1 \cap \mathrm{CM}_2)$
where $\mathrm{trans}_u(\mathtt{s}_1, \mathrm{CM}) = (\widehat{\mathtt{s}_1}, \mathrm{CM}_1)$ *and* $\mathrm{trans}_u(\mathtt{s}_2, \mathrm{CM}_1) = (\widehat{\mathtt{s}_2}, \mathrm{CM}_2)$

$\mathrm{trans}_u(\mathbf{while}(\mathtt{e})\{\mathtt{s}\}, \mathrm{CM}) = (\mathrm{while}(\mathrm{trans}(\mathtt{e}))\{\widehat{\mathtt{s}}\}, \mathrm{CM}_1 \cap \mathrm{CM})$
where $\mathrm{trans}_u(\mathtt{s}, \emptyset) = (\widehat{\mathtt{s}}, \mathrm{CM}_1)$

$\mathrm{trans}_u(\mathtt{v} = \mathtt{e}, \mathrm{CM}) = (\mathtt{v} := \mathrm{trans}(e), \mathrm{CM})$

$\mathrm{trans}_u(\mathtt{v} = \mathtt{e}.\mathbf{get}, \mathrm{CM}) = \left(\left\{\{?\psi(\emptyset)\} \cup \{?\neg\psi(\emptyset); \mathrm{fail}\}\right\}; \mathtt{v} := *, \mathrm{CM}\right)$

$\mathrm{trans}_u(\mathbf{await}\ \mathtt{g}, \mathrm{CM}) = \left(\left\{\{?\psi(\mathrm{CM}, \mathtt{g})\} \cup \{?\neg\psi(\mathrm{CM}, \mathtt{g}); \mathrm{fail}\}\right\}; \mathrm{havoc}; ?\mathrm{trans}(\mathtt{g}), \emptyset\right)$

$\mathrm{trans}_u(\mathtt{v} = \mathtt{e}!\mathtt{m}(\mathtt{e}_1, \ldots), \mathrm{CM}) =$
$\left(\left\{\{?\mathrm{pre}_{\mathtt{m}}(\mathtt{e}_1, \ldots)\} \cup \{?\neg\mathrm{pre}_{\mathtt{m}}(\mathtt{e}_1, \ldots); \mathrm{fail}\}\right\};\right.$
$\left.\mathtt{v} := *, \mathrm{CM}\right)$ *if* $\mathtt{e} \neq \mathbf{this}$

$\mathrm{trans}_u(\mathtt{v} = \mathbf{this}!\mathtt{m}(\mathtt{e}_1, \ldots), \mathrm{CM}) =$
$\left(\left\{\{?\mathrm{pre}_{\mathtt{m}}(\mathtt{e}_1, \ldots)\} \cup \{?\neg\mathrm{pre}_{\mathtt{m}}(\mathtt{e}_1, \ldots); \mathrm{fail}\}\right\};\right.$
$\left.\mathtt{v} := *, \mathrm{CM} \cup \{\mathtt{m}\}\right)$

$\mathrm{trans}_u(\mathtt{v} = \mathbf{new}\ \mathtt{C}(\mathtt{e}_1, \ldots), \mathrm{CM}) =$
$\left(\left\{\{?\mathrm{init}_{\mathtt{C}}(\mathtt{e}_1, \ldots)\} \cup \{?\neg\mathrm{pre}_{\mathtt{C}}(\mathtt{e}_1, \ldots); \mathrm{fail}\}\right\};\right.$
$\left.\mathtt{v} := *, \mathrm{CM}\right)$

$\mathrm{trans}_u(\mathbf{duration}(\mathtt{e})) =$
$\mathtt{t} := 0; \left\{\{?\psi(\mathtt{e})\} \cup \{?\neg\psi(\mathtt{e}); \mathrm{fail}\}\right\};$
$\mathtt{t} := 0; \{\mathrm{ode}, \mathtt{t}' = 1 \& \mathtt{t} \leq \mathrm{trans}(\mathtt{e})\}; ?\mathrm{trans}(\mathtt{e})$

Figure 13: Translation of HABS-statements into $d\mathcal{L}$ programs.

Proof Idea. The proof is analogous, except that we use that the reachable states from a state in D cannot be models for any guard of a called method, except on the event boundary.

An important detail is that the translation of uncovered **get** statements uses True as the uncontrolled region. As the future access may deadlock, this ensures that in this case the object the object is in a safe state and remains so forever. The main block is translated using basic regions, as it cannot contain calls to its own (implicit) object.

Contrary to basic regions, we cannot use a single modality by:

$$[\mathsf{s}](\mathsf{inv}_\mathsf{C} \wedge [\mathsf{ode\&true}]\mathsf{inv}_\mathsf{C}) \iff [\mathsf{s};\mathsf{ode\&true}]\mathsf{inv}_\mathsf{C}$$

because the post-region may be empty. In this case, e.g., if a method with a leading guard `true` is called, the second modality simplifies to true. The used obligation still ensures the invariant in the poststate. Removal of a method requires to reprove all calling methods.

Lemma 4.8. *Let* C *be a safe class according to Thm. 4.7. Let* $\mathtt{C}^-$ *be* C *with some method* $\mathtt{m}^-$ *removed and* $\mathtt{C}^+$ *be* C *with some method added.*

(1) To verify $\mathtt{C}^-$*, only the proof obligations of methods calling* $\mathtt{m}^-$ *must be closed.*

(2) To verify $\mathtt{C}^+$*, only the proof obligation of the new method has to be shown.*

If another class calls $\mathtt{m}^+$*, only the calling methods have to be reproven.*

Beyond mutually recursive structures, locally controlled regions can verify classes with a single controller and no further methods.

Example 4.9. The controller in Fig. 14 checks the water level of a tank every $\frac{1}{2}$ seconds, so every process has to establish safety only for that time frame.

```
1  [HybridSpec: Requires("3.5<=inVal<=9.5")]
2  class TankTick(Real inVal){
3    [HybridSpec: ObjInv("3<=x<=10 & -1<=v<=1")]
4    physical{
5      Real x = 5: x' = v;
6      Real v = -1; v' = 0;
7    }
8    { this!ctrl(); }
9    Unit ctrl(){
10     await duration(1/2);
11     if(x <= 3.5) v = 1;
12     if(x >= 9.5) v = -1;
13     this!ctrl();
14   }
15 }
```

Figure 14: Timed water tank.

4.3 Structurally Controlled Regions

Locally controlled regions are not a sufficiently precise proof principle in general. Consider Ex. 2.1. The method for the lower boundary, down, does not establish safety by its recursive call, as the level may rise above the upper boundary. Similarly, if a method is added to Ex. 4.9, it cannot use the information from `ctrl`. To amend this, we use that in controlled classes the controllers can always be scheduled if their condition holds. Their conditions, thus, can be used in a more precise computation of the post-region.

Intuitively, the following theorem extends Thm. 4.7 to make use of the fact that controllers are always running. This is done by extending $\psi(\mathsf{CM})$ and $\psi(\mathsf{CM}, \mathsf{g})$ to add the regions defined by the controllers into account.

Theorem 4.10 (Structurally Controlled Regions). *Let all variables range as in Thm. 4.7. Let* Ctrl_C *be the set of controllers in* C*. Thm. 4.7 holds if* $\psi(\mathsf{CM})$ *is replaced by the following definition:*

$$\psi(\mathsf{CM}) = \mathsf{inv}_\mathsf{C} \wedge \left[\mathsf{t}:=0;\mathsf{ode},\mathsf{t}'=1\&\left(\bigwedge_{\mathsf{m}\in\mathsf{Ctrl}_\mathsf{C}\cup\mathsf{CM}} \widetilde{\neg}\mathit{ttrig}_\mathsf{m}\right)\right]\mathsf{inv}_\mathsf{C}$$

And analogously for $\psi(\mathsf{CM}, \mathsf{g})$*.*

While this increases precision, modularity suffers: removal of a controller requires to reprove all methods in the class. This is to be expected, as a controller influences every post-region.

Lemma 4.11. *Let* C *be a safe class with structurally controlled regions. Let* $\mathtt{C}^-$ *be* C *with some method* $\mathtt{m}^-$ *removed and* $\mathtt{C}^+$ *be* C *with some method added.*

(1) If $\mathtt{m}^-$ *is a controller then all methods of* $\mathtt{C}^-$ *have to be reproven.*

(2) If $\mathtt{m}^-$ *is not a controller then Lem. 4.8 holds.*

5 ZENO BEHAVIOR

Post-regions allow us to analyze another property than safety: Local Zeno behavior. Under local Zeno behavior we understand that a class exhibits Zeno behavior when run in isolation, i.e., infinitely many discrete steps in finite time. We can exclude such behavior if the post-region of every method (and every suspension) has some constant minimal size — this means that after a process terminates, the object advances some minimal time.

Definition 5.1. Let χ be a time-convergent run and o an object within. We say that o is *locally modified* by a transition if (1) some process is scheduled and (2) this process is not resulting from a message from another object. Object o is locally Zeno within χ if o is modified only locally by infinitely many transitions in χ.

Global Zeno can be defined by dropping condition (2). Global Zeno is a property of the overall program, while local Zeno is an inherent design error in a single class.

Example 5.2. The bouncing ball in Ex. 2.1 is locally Zeno. The timed water tank in Ex. 4.9 is not locally Zeno.

The above definition only considers scheduling events for Zeno to exclude non-termination: E.g., we do not consider a non-terminating loop without **await** and **duration** statements Zeno. The reason for this is that we consider non-termination and Zeno as different phenomena that express different underlying problems: Non-termination is a purely discrete behavior that happens in a hybrid context, while Zeno behavior is inherently hybrid and happens when discrete and continuous behavior interact.

To analyze local Zeno, we need to check at least the controllers to ensure that after a controller terminates, always some minimal times passes before the next controler runs. However, this is not

sufficient, e.g., for mutual recursion of other methods. But even general cycle detection in the call graph (in the style of Non-Zeno loops for hybrid automata [35]) does not allow us to detect all interactions between internal **await** **diff** statements.

To check local Zeno, we, thus, check that for every suspension there is a lower bound t_0 such that after every execution of m at least t_0 time units pass where the dynamics are inside the (uncontrolled) post-region. For brevities sake, we give the proof obligation only for methods without internal **await**.

Intuitively, each proof obligation expresses that at least t_0 seconds may pass without a discrete step being taken.

Theorem 5.3 (Local Zeno Freedom). *Let* C *be a safe class (acc. to Thm. 4.10) where every* **await** *is leading. Let* $\mathsf{trans}_u(\mathsf{s_m}, \emptyset) = (\widehat{\mathsf{s_m}}, \mathsf{CM})$. *If for every method* m *the following formula is valid, then there is no object of* C *in any program (that adheres to creation and method pre-conditions) that is locally Zeno.*

$$\mathsf{inv_C} \wedge \mathsf{pre_m} \rightarrow$$
$$\exists t_0.\ \Big(t_0 > 0 \wedge \big[?trig_{\mathsf{m}}\widehat{\mathsf{s_m}}; t := 0; \mathsf{ode}, t' = 1 \& t \leq t_0;\big] \bigwedge_{\mathsf{m} \in \mathsf{Ctrl_C} \cup \mathsf{CM}} \neg ttrig_{\mathsf{m}}\Big)$$

6 IMPLEMENTATION

We implemented the procedures in Sec. 4 and Sec. 5 in the Chisel tool[5]. Chisel reuses the HABS parser. The tool parses the input file, checks which class is controlled and which methods are open. The user can select to verify all classes, only one class, only one method or only the main block. Additionally, the user may select which algorithm for region computation is to be used. It then translates a class into a set of KeYmaera X proof obligations and automatically tries to verify them. Failed proof attempts can be retrieved by the user and Chisel allows to annotate proof scripts to manually apply a tactic instead of the automatic procedures: Such tactics are output by KeYmaera X after a proof is closed.

```
[HybridSpec: Tactic("expandAllDefs; master; DEs(1)")]
Unit method(){ ... }
```

All examples in this work are uploaded with the source code of the tool and can be closed fully automatically. Reverification of the water tank shows that the new approach is more suited for automated proving: The previous system required interaction to close Ex. 2.1[22, Fig. 2] but our new tool Chisel can verify it fully automatically. The reason is that the concurrency model and scheduling is now implicit for the prover, while it was encoded directly in the $d\mathcal{L}$ formula before and required to retrieve two additional loop invariants. While the previous system only handled controlled classes where **await** is only allowed as the first statement of a method, Chisel covers the the complete language (except additional data types).

Chisel can be used in combination with Crowbar, the verification tool for ABS implementing the Behavioral Program Logic [19]: While Chisel verifies hybrid classes, Crowbar verifies non-hybrid classes and allows one to use arbitrary data types and function definitions in non-hybrid classes. Both tools support method contracts as *cooperative contracts* [20] and interactions are handled automatically by this mechanism.

[5] Additionally to the compiled tool linked above, the source code is available under https://github.com/Edkamb/chisel-tool/.

7 RELATED WORK AND CONCLUSION

This is the first work to successfully generalize specification and verification principles for object-oriented language to a hybrid setting. Instead of verifying an object invariant by using it as a post-condition of a method, we instead use it as an invariant for the post-region: the states reachable from the post-state method when following the given dynamics. Verification of hybrid systems is hard, but we show that by using an object-oriented programming languages as a host for hybrid behavior, it is possible to use the additional structure provided by the language to improve compositionality of verification.

Related Work. There are only few programming languages for hybrid systems with full-fledged formal semantics. The best examined one is the algebraic language of $d\mathcal{L}$ [31–33]. Its minimal structure requires that compositionality is encoded in elaborate proof structures [23, 24, 27]. $\texttt{While}^{\texttt{dt}}$ [34] is a hybrid while-language with a verification condition generator [14], but compositionality is not examined. For other languages, such as HybCore [11], verification is not considered. Most hybrid languages, e.g., Hybrid Rebeca [17] have as semantics only translations into hybrid automata.

For hybrid automata, composition of *semantics* has been investigated since the beginning by parallel composition [3] or by marking some variable as input or output ports, e.g., in hybrid I/O automata [25], which can be used to build a hierarchical model made of components [7] for model checkers. Composition has also been investigated for model checking hybrid automata by assume-guarantee reasoning to decompose systems [9, 16]. Such approaches are similar to method contracts, but rely-guarantee reasoning for hybrid automata does not encapsulate of data and method calls, but abstractes of components executed in parallel.

Future Work. Concerning the Zeno analysis, possible future work includes a global Zeno notion and analysis for Hybrid Active Objects, as well as a precise characterization of both local and global Zeno. Besides the Zeno analyses, we plan to investigate the generalization of program analyses for object-oriented and distributed models, e.g., resource [8] or deadlock analyses [15, 18] for more precise post-regions. A deadlock and a global Zeno analysis would allow to reason about total correctness of HABS programs. A further principle from discrete languages, *dynamic* frames [12], shows premise to generalize well and significantly increase precision. We are developing a post-region based verification system for hybrid automata and integrating Modelica into the **physical** block.

7.1 Acknowledgments.

The author thanks Stefan Mitsch for fruitful discussions and the anonymous reviewers of HSCC for constructive feedback that helped improving the presentation.

REFERENCES

[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). 2016. *Deductive Software Verification - The*

KeY Book - From Theory to Practice. LNCS, Vol. 10001. Springer. https://doi.org/10.1007/978-3-319-49812-6

[2] Elvira Albert, Frank S. de Boer, Reiner Hähnle, Einar Broch Johnsen, Rudolf Schlatte, Silvia Lizeth Tapia Tarifa, and Peter Y. H. Wong. 2014. 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 (2014), 323–339.

[3] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. 1995. The Algorithmic Analysis of Hybrid Systems. *Theor. Comput. Sci.* 138, 1 (1995), 3–34. https://doi.org/10.1016/0304-3975(94)00202-T

[4] Nikolaos Bezirgiannis, Frank S. de Boer, Einar Broch Johnsen, Ka I Pun, and Silvia Lizeth Tapia Tarifa. 2019. Implementing SOS with Active Objects: A Case Study of a Multicore Memory System. In *FASE 2019 (LNCS, Vol. 11424)*, Reiner Hähnle and Wil M. P. van der Aalst (Eds.). Springer, 332–350.

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

[6] 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. 2017. A Survey of Active Object Languages. *Comput. Surveys* 50, 5 (2017), 1–39.

[7] Alexandre Donzé and Goran Frehse. 2013. Modular, hierarchical models of control systems in SpaceEx. In *ECC 2013*. IEEE, 4244–4251.

[8] Antonio Flores-Montoya. 2016. Upper and Lower Amortized Cost Bounds of Programs Expressed as Cost Relations. In *FM 2016 (LNCS, Vol. 9995)*, John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou (Eds.). 254–273. https://doi.org/10.1007/978-3-319-48989-6_16

[9] G. Frehse, Zhi Han, and B. Krogh. 2004. Assume-guarantee reasoning for hybrid I/O-automata by over-approximation of continuous interaction. In *CDC 2004*, Vol. 1. 479–484 Vol.1.

[10] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. 2015. KeYmaera X: An Axiomatic Tactical Theorem Prover for Hybrid Systems. In *CADE'25 (LNCS, Vol. 9195)*. Springer, 527–538.

[11] Sergey Goncharov and Renato Neves. 2019. An Adequate While-Language for Hybrid Computation. *CoRR* abs/1902.07684 (2019).

[12] Daniel Grahl, Richard Bubel, Wojciech Mostowski, Peter H. Schmitt, Mattias Ulbrich, and Benjamin Weiß. 2016. Modular Specification and Verification. In *The KeY Book*. LNCS, Vol. 10001. Springer.

[13] David Harel, Jerzy Tiuryn, and Dexter Kozen. 2000. *Dynamic Logic*. MIT Press.

[14] Ichiro Hasuo and Kohei Suenaga. 2012. Exercises in Nonstandard Static Analysis of Hybrid Systems. In *CAV 2012 (LNCS, Vol. 7358)*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer, 462–478. https://doi.org/10.1007/978-3-642-31424-7_34

[15] Ludovic Henrio, Cosimo Laneve, and Vincenzo Mastandrea. 2017. Analysis of Synchronisations in Stateful Active Objects. In *Integrated Formal Methods*, Nadia Polikarpova and Steve Schneider (Eds.). Springer, 195–210.

[16] Thomas A. Henzinger, Marius Minea, and Vinayak S. Prabhu. 2001. Assume-Guarantee Reasoning for Hierarchical Hybrid Systems. In *HSCC 2001 (LNCS, Vol. 2034)*, Maria Domenica Di Benedetto and Alberto L. Sangiovanni-Vincentelli (Eds.). Springer, 275–290. https://doi.org/10.1007/3-540-45351-2_24

[17] Iman Jahandideh, Fatemeh Ghassemi, and Marjan Sirjani. 2019. Hybrid Rebeca: Modeling and Analyzing of Cyber-Physical Systems. *CoRR* abs/1901.02597 (2019). arXiv:1901.02597

[18] Eduard Kamburjan. 2018. Detecting Deadlocks in Formal System Models with Condition Synchronization. *ECEASST* 76 (2018).

[19] Eduard Kamburjan. 2019. Behavioral Program Logic. In *TABLEAUX 2019 (LNCS, Vol. 11714)*, Serenella Cerrito and Andrei Popescu (Eds.). Springer, 391–408.

[20] Eduard Kamburjan, Crystal Chang Din, Reiner Hähnle, and Einar Broch Johnsen. 2020. Behavioral Contracts for Cooperative Scheduling. In *Deductive Verification: The State of the Future*, Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, and Mattias Ulbrich (Eds.). LNCS, Vol. 12345. Springer.

[21] Eduard Kamburjan, Reiner Hähnle, and Sebastian Schön. 2018. Formal modeling and analysis of railway operations with active objects. *Sci. Comput. Program.* 166 (2018), 167–193. https://doi.org/10.1016/j.scico.2018.07.001

[22] Eduard Kamburjan, Stefan Mitsch, Martina Kettenbach, and Reiner Hähnle. 2019. Modeling and Verifying Cyber-Physical Systems with Hybrid Active Objects. *CoRR* abs/1906.05704 (2019).

[23] Simon Lunel, Benoît Boyer, and Jean-Pierre Talpin. 2017. Compositional Proofs in Differential Dynamic Logic dL. In *ACSD'17*. IEEE Computer Society, 19–28.

[24] Simon Lunel, Stefan Mitsch, Benoît Boyer, and Jean-Pierre Talpin. 2019. Parallel Composition and Modular Verification of Computer Controlled Systems in Differential Dynamic Logic. In *FM (LNCS, Vol. 11800)*, Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira (Eds.). Springer, 354–370.

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

[26] Bertrand Meyer. 1992. Applying "Design by Contract". *IEEE Computer* 25, 10 (Oct. 1992), 40–51.

[27] Andreas Müller, Stefan Mitsch, Werner Retschitzegger, Wieland Schwinger, and André Platzer. 2018. Tactical Contract Composition for Hybrid System Component Verification. *STTT* 20, 6 (2018), 615–643.

[28] André Platzer. 2010. Differential-algebraic Dynamic Logic for Differential-algebraic Programs. *J. of Logic and Computation* 20, 1 (2010), 309–352.

[29] André Platzer. 2010. Quantified Differential Dynamic Logic for Distributed Hybrid Systems. In *CSL 2010 (LNCS, Vol. 6247)*. Springer, 469–483.

[30] André Platzer. 2012. A Complete Axiomatization of Quantified Differential Dynamic Logic for Distributed Hybrid Systems. *LMCS* 8, 4 (2012), 1–44.

[31] André Platzer. 2012. The Complete Proof Theory of Hybrid Systems. In *LICS*. IEEE, 541–550.

[32] André Platzer. 2017. A Complete Uniform Substitution Calculus for Differential Dynamic Logic. *J. Automated Reasoning* 59, 2 (2017), 219–265.

[33] André Platzer. 2018. *Logical Foundations of Cyber-Physical Systems*. Springer.

[34] Kohei Suenaga and Ichiro Hasuo. 2011. Programming with Infinitesimals: A While-Language for Hybrid System Modeling. In *ICALP (2) (LNCS, Vol. 6756)*. Springer, 392–403.

[35] Stavros Tripakis. 1999. Verifying Progress in Timed Systems. In *ARTS (Lecture Notes in Computer Science, Vol. 1601)*. Springer, 299–314.