Designing Distributed Control with Hybrid Active Objects

Eduard Kamburjan[1,2], Rudolf Schlatte[1],
Einar Broch Johnsen[1], and S. Lizeth Tapia Tarifa[1]

[1] University of Oslo, Oslo, Norway
{eduard,rudi,einarj,sltarifa}@ifi.uio.no
[2] Technische Universität Darmstadt, Darmstadt, Germany

Abstract. Models of distributed software systems extend naturally to cyber-physical systems "in the large"; i.e., systems of loosely coupled software components which interact with models of physical processes. But how do we model such combined systems? This paper discusses this problem from the perspective of active object systems. We attach different active objects to models of physical systems, but maintain the objects' actor-like decoupling of communication and synchronization. The result is a model of hybrid active objects. In this setting, we discuss different ways of modeling and controlling time advance and value propagation between components, which may be inside the model, controlled by the model, or controlling the model as a simulation unit. The patterns of on-demand value propagation as well as fixed- and variable-step time advance arise naturally from the semantics of hybrid active object models in HABS, a hybrid extension of the formal specification language ABS.

1 Introduction

Models of distributed systems extend naturally to cyber-physical systems "in the large"; i.e., loosely coupled distributed systems which include one or more cyber-physical components. Cyber-physical systems in the large describe both applications building on the Internet of Things and for Digital Twins in which various models of physical systems interact with a distributed software system. It is today a challenge to verify and even to validate such hybrid distributed system models [11].

This paper discusses a compositional approach to this challenge, based on formal semantics and executable models of loosely coupled distributed systems of so-called *hybrid active objects*. Hybrid active objects are a hybrid extension of active objects for modeling distributed cyber-physical systems. Active objects [5] are object-oriented systems based on the actor concurrency model [12] which decouple communication and synchronization through asynchronous message passing, allowing very flexible decentralized systems to be easily expressed. ABS [13] is a formally defined, executable modelling language based on active object concepts that has proven to be suited for intuitive and natural modeling, and

formal verification of complex systems in an industrial context and a multitude of domains, ranging from railway operations [17] to cloud-based systems [27]. Timed ABS adds a dense-time discrete event semantics to active objects. ABS also includes an interface for data acquisition from outside sources via http requests, called the Model API. To model hybrid active object systems, the *Hybrid Abstract Behavioral Specification language* (HABS) [15, 18] is designed to combine simulation and verification as a hybrid extension of the modeling language ABS.

In this paper, we explore concepts of *distributed control* in the setting of HABS. We discuss these concepts in the context of two simple water tank examples, the first one targeting distributed controllers and the second one targeting prediction capabilities. We examine their components to discuss the runtime structure of the HABS models. In particular, we distinguish between implicit (or declarative) and explicit (or operational) components of a model, and between messages needed for time advance and value-passing. A central consideration is the level of control in the HABS models. HABS differentiates between *control over time advance* and *control over value passing*. Time advance is specified by the active objects and regulated by a central orchestrator that computes the maximal possible time advance. This time-orchestrator cannot rewind time and can only send time advance messages to the objects. Control over value passing does not involve the time orchestrator; it is distributed and explicitly modeled in the active objects. This allows flexible and complex communication patterns to be expressed, that capture communication as it is happening in the modeled system. This decoupling may also make the system simulation more performant, because the time-orchestrator is no bottleneck for communication of values. This is critical if the modeled system itself is distributed and communication is more complex than simple value propagation, but triggers complex behavior in the hybrid active objects.

HABS relies on a white-box model of discrete controllers as well as continuous dynamics for verification. To bridge between HABS and possible white- and black-box components of a model, we explore how interfacing to the continuous dynamics can be used to integrate implemented systems with recreated live data, as functional mock-ups or complex simulators for simulation (as a white-box) while retaining the ability of formal reasoning. We also explore how to integrate recorded data (as a black-box), via the Model API, a REST interface, to explore what-if scenarios for predictions, in this case formal analysis can still be done using, e.g., runtime monitoring and assertion checking. These integrations provide two ways to link between HABS models and Digital Twins.

Related Work. Digital Twins are a very interesting application area for formal methods because the basis for Digital Twins is model-based analysis. To cover an asset life-cycle, a digital twin need to combine a declarative model of the design of the asset with an inductive model derived from data observations of the system in operation [8]. However, the richness of the models makes it hard to do full verification and analysis in a digital twin setting, particularly for hybrid models with discrete and dynamic components. Such analysis is often

based on simulation [22] or co-simulation [11]. A prominent example of work in this direction is INTO-CPS [30], which is pushing a co-simulation platform in the direction of Digital Twins based on FMI [3], an emerging standard for connecting simulators. In the case of ABS, the Model API allows data to be fed into a running model, thus integrating snapshots of industrial data with simulations in real time [27]. We have also used ABS to model a digital twin triggering software upgrades of connecting cars by sensor data [21, 26].

Digital Twins are closely related to cyber-physical systems, but typically consider only distributed cyber-physical systems which connect various cyber-physical components into a distributed system, which is a major challenge for formal methods [28]. We do not attempt to cover that large area here, but mention Platzer's work on verification of cyber-physical systems using differential dynamic logic $d\mathcal{L}$ [25], which has been a source of inspiration for our work on HABS as a hybrid extension of ABS. HABS is a formally defined, executable modeling language with a proof theory building on $d\mathcal{L}$. Formal support for co-simulation algorithms has been done by means of model checking [29] and verification of the simulation itself by means of basic contracts [10]. The use of Hybrid Active Objects complements this work by offering a rich toolkit for the analysis of expressive properties for distributed and hybrid systems, which hopefully suffice to verify models of distributed control. Whereas this paper hints at verification for such models, the verification of such algorithms is not attempted here.

Structure. In Sec. 2 we describe active objects and the ABS modelling language for distributed active objects, and in Sec. 3 the hybrid extension HABS. In Sec. 4 we give two water tank models to illustrate modeling with HABS showcasing different notions of distributed control. In Sec. 5 we relate these notions to co-simulation and digital twins. We conclude the paper in Sec. 6 and present future work.

2 A short overview of ABS

ABS [13] is an actor-based, object-oriented language for modeling concurrent and distributed systems and supporting the design, verification, and execution of such systems. ABS has a purely *functional layer* of functions and algebraic datatypes, and an *object-oriented layer* for modeling Java-like objects and interfaces. Communication between objects is implemented via *asynchronous method calls*, which produce a *future* at the caller and a *new process* at the callee object. The caller continues execution immediately and only synchronizes with the future when the result of the call is needed. The callee schedules one of its pending processes; that process runs without preemption until it either finishes or *suspends*, e.g., when waiting for a future, at which point the object schedules another waiting process. These cooperative, explicit scheduling points are the foundation of the compositional proof system of distributed ABS models [6] and of its timed semantics which is explained later in this section.

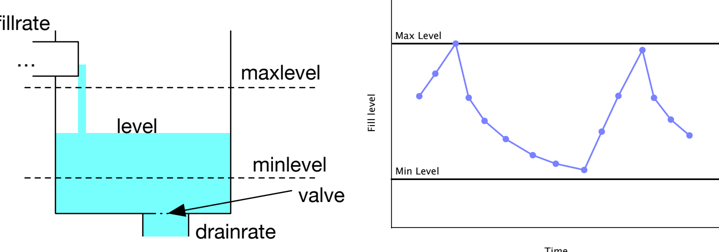

Fig. 1. The discrete-event watertank model and its behavior

Example 1. Consider a simple model of a `WaterTank`, which is filled from the top at a fixed `fillrate` and is drained (if the `valve` is open) from the bottom at a fixed `drainrate` (see Fig. 1 left for a basic sketch and Fig. 2 for its ABS implementation). We explain the basic concepts of asynchronous method calls and cooperative scheduling in ABS using an example from class `Controller` (Fig. 2, Line 27), which controls the water level of such tank. The execution of an asychronous call statement like `tank!valvecontrol(Open)` (Line 33) creates a new process in the object `tank` which will execute the method `valvecontrol` but *does not release the control* at the caller object, where the current process continues its execution. Since we do not store the return value of the call, this corresponds to a "pure" message sending operation.

If we want to synchronize and/or receive the result of a method call, we store its future (`f=o!m();`). This future `f` can be used to both *suspend* the current process until the callee process has finished (Syntax: `await f?;`) and to read the future's value (Syntax: `f.get`). Suspending the current process means that the object running the calling process can schedule another process from its own pool or stay idle; when awaiting on a future, the suspended process becomes eligible for scheduling once the future `f` contains a value (i.e., when its process has finished).

As a shorthand, the statement `level = await tank!getLevel()` (Fig. 2 Line 31) creates a new process in the object `tank`, suspends the calling process until the return value of that method has been received, and then assigns the value to the variable `level`. Figure 1 shows a sketch of the physical setup (left) and its expected behavior (during "a kind of" discrete event simulation) in terms of how the level of the water changes as time progress (right).

2.1 The time model of Timed ABS

The previous section has already hinted that an object in ABS can be either running or idle; the latter case occurs when its process queue is empty or all processes are waiting on some condition. Timed ABS, an extension of ABS, adds a global logical clock, which acts as a time orchestrator that advances once all objects are idle. All objects can specify their own time advance interval; the clock

```
1 data ValveCommand = Open | Close;
2
3 interface WaterTank {
4     Float getLevel();
5     Unit valvecontrol(ValveCommand command);
6 }
7
8 class WaterTank(Float fillrate, Float drainrate)
9 implements WaterTank {
10  Float level = 0.0;
11  Bool valve_open = False;
12
13  Float getLevel() { return level; }
14  Unit valvecontrol(ValveCommand command) {
15   valve_open = when command == Open then True else False;
16  }
17  Unit run() {
18   while (True) {
19    await duration(1, 1);
20    level = level + fillrate;
21    if (valve_open)
22     level = max(0.0, level - drainrate * level);
23   }
24  }
25 }
26
27 class Controller(WaterTank tank, Float minlevel,
28                  Float maxlevel) implements Controller {
29   Unit run() {
30     while (True) {
31       Float level = await tank!getLevel();
32       if (level >= maxlevel) {
33         tank!valvecontrol(Open);
34       } else if (level <= minlevel) {
35         tank!valvecontrol(Close);
36       }
37       await duration(1, 1);
38     }
39   }
40 }
```

Fig. 2. ABS (Non-hybrid) model of a water tank and controller

will advance by the maximum amount that satisfies all these conditions. Since the time domain is the rational numbers, dense time and thus Zeno behavior is supported.

For example, In Lines 19 and 37 of Figure 2, we see statements **await duration** `(1, 1);`. These statements suspend the running process until the global clock has advanced by `1` (which is both the minimum and maximum advance given in the statement).

Timed ABS thus allows models to incorporate explicit manipulation of time from a dense time domain, which can be used to represent execution time inside methods, which either block an object, using **duration**`(min, max);` or suspend a thread, using **await duration**`(min, max);` for a certain amount of time between `min` and `max`. This way of modelling timed behavior is well-known from, e.g., timed automata in UPPAAL [20]. By using this modelling of the passage of time, we abstract from the actual execution time and execution context of a thread (i.e., the computation time of the thread is independent of the cpu speed and the load on the server where it is executed).

2.2 The Model API

The Model API of ABS is a way to call methods on objects of a running model from the outside via http requests. The model annotates the objects and methods that should be callable, i.e., the model is not completely open via the Model API. In addition, the Model API provides some measure of control over the model time advance: As explained in Section 2.1, the clock advances by the maximum possible amount. The Model API can set a limit for the clock and then raise it, again controlled by a http request from an agent on the outside of the model.

Using the Model API transforms a model into an *open system*, where the behavior cannot be determined statically, although class invariants and method contracts hold as before. The Model API is *reactive*, i.e., it receives method calls and time advance triggers as external stimuli but cannot send stimuli to the outside of the model.

3 Hybrid Abstract Behavioral Specification Language

The Hybrid ABS (HABS) [18] language is an extension of Timed ABS for modeling Cyber-Physical systems. In this section we introduce the main features of HABS: (1) integrated continuous behavior declared in a special block in the class that contains ODEs, (2) special **await** statements to react to changes caused by this continuous behavior and (3) a transparent extension of the time model.

3.1 Syntax and Semantics

HABS extends ABS by two new constructs:

- A **physical** block in class definitions contains *physical fields* which define the continuous behavior over time of a part of the object state via (autonomous) ordinary first-order differential equations (ODE). In the definition of a physical field `f`, the notation `f'` is used for the first derivative of `f`. All physical fields have the type `Real`, while non-physical fields can have any type.

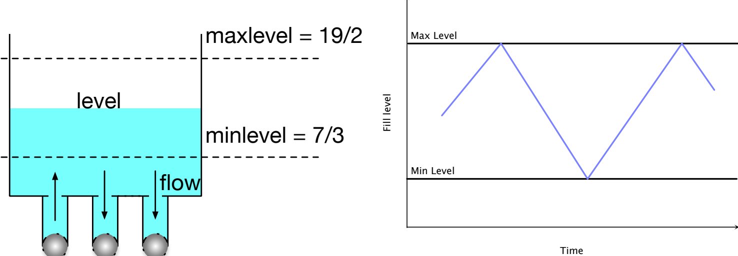

Fig. 3. The multi-controller watertank and its behavior

– A new suspension statement **await diff** `g` which suspends the running thread until `g` holds. The guard `g` is a quantifier-free formula over the object's physical fields and the local variables of the process. This statement plays a similar role as transition guards in hybrid automata [2] and allows one to model reaction to state change.

In HABS, an object that has a **physical** block is a *hybrid* object. If the object has no physical block, but uses **duration** statements in its methods, then it is a *timed* object. Otherwise it is a *discrete* object. Note that HABS does not model evolution domains (e.g., `level` ≥ 0, sometimes called mode invariants) directly, to give the user the possibility to implement how leaving the evolution domain shall influence the behavior of the object upon time advance of the system.

Example 2. Consider a model of a water tank, depicted in Fig. 3 left, and coded in Fig. 4. It declares an object with a physical field `level` to describe the water level of the tank, and three normal fields `flowA`, `flowB`, `flowC` that hold the status of three different valves of the tank, which let water either fill or drain depending on the direction of the flow. The field declaration in the physical block, in addition to specifying an initial value `inVal`, describes the dynamics of the `level` field via an ODE. We use `level' = flowA+flowB+flowC` to denote that the derivative of `level` (seen as a function over time) is equal to the sum of the three flows. The `setFlow` method, when called, sets any of the valves, and the `outLevel` method returns the current water level. Finally, the `acqLevel` method suspends until the water level falls below a certain level `r`.

The `acqLevel` method shows how continuous behavior in the **physical** block and discrete program in the rest of the object interact, by using a differential suspension **await diff** `g`. Semantically, this statement suspends the running process P until the earliest time t_d where `g` holds for the physical block. The solution F of the physical block can be computed over the time interval within `now()` and t_d, with the initial values (with underscore 0) provided by the current state. Upon reactivation, the values of the physical fields are updated according to $F(t_d)$. In our example, F for the variable `level` is the solution defined by:

$$F(t_d) = \texttt{level}_0 + \texttt{flowA}_0 \times t_d + \texttt{flowB}_0 \times t_d + \texttt{flowC}_0 \times t_d$$

```
1  class Tank(Rat inVal, Rat dIVal) implements ITank {
2    Real flowA = dIVal;
3    Real flowB = dIVal;
4    Real flowC = dIVal;
5    physical{
6      Real level = inVal : level' = flowA+flowB+flowC;
7    }
8    Unit setFlow(Real newFlow, Int id){
9      if(id == 0) flowA = newFlow;
10     if(id == 1) flowB = newFlow;
11     if(id == 2) flowC = newFlow;
12   }
13   Real outLevel(){ return level; }
14   Unit acqLevel(Real r){ await diff level <= r; }
15 }
```

Fig. 4. A water tank with three controllers in Hybrid ABS.

such that the solution over the time interval keeps the current values of `level`, `flowA`, `flowB` and `flowC` as constants.

Any other process that runs between `now()` and t_d, and thereby modifies the object state and the state of the physical variables, will cause the solution for `g` to be recomputed and the waiting time for P to be extended or shortened according to the new object state. For every blocking statement that might cause time to advance, such as **get** or **duration**, we similarly compute the solution to the continuous behavior in the current state and update the state once the passed time is known.

This is transparent to the time model of Timed ABS because the messages for suspension, possible time advance and reactivation do not change for HABS.

For simulation, HABS uses maxima [23] to (1) compute the solution to the initial value problem stated by the physical block and the current heap and (2) to compute t_d by minimization of time for the solution, constrained by the guard. For the scheduler of the active object in ABS, the extension is transparent: Processes in hybrid active objects are scheduled in the same way as for normal active objects and the time orchestrator handles t_d in the same way as timed duration statements. HABS and the tools in its backend, can handle differential equations beyond ODEs and linear dynamics.

3.2 Analyzability

The semantics of HABS take care of the problems that naturally occur in hybrid systems: solving ODEs and computing the next time advance, and keeping the state updated after discrete time steps. The model in Ex. 2 is, thus, more concise and natural than its discretized counterpart in Ex. 1. The update of state is

crucial for modeling distributed systems, as a hybrid object must be reactive to calls from the outside at any point in time.

Formal analysis and verification is also more simple than in the discrete case: the continuous behavior of each object is described directly and centralized in the physical block with a well-established mechanism. The physical block is an *abstraction* of the numerical and symbolic operations performed by the runtime (in HABS) or the explicit discretization (in ABS). Verification of Ex. 2 can use the ODEs to reason about continuous dynamics, while Ex. 1 requires an analysis of the implemented simulator. There are established tools for ODEs and the code in Ex. 2 can be automatically translated in proof obligations in differential dynamic logic [24] which can be checked with the KeYmaera X theorem prover [9].

A hybrid system in HABS contains hybrid objects and non-hybrid (timed and discrete) objects. Due to the strong encapsulation of the language semantics, properties of distributed systems that are not dependent on the hybrid objects can be analyzed by the rich toolkits [1] and logics [14] for discrete objects.

However, execution (simulation) becomes more complex, as the abstraction provided by the physical block does not help with simulation. While simulation is not an analysis per se, it plays an important role during development by using testing as a complement to verification. HABS uses **assert** statements for runtime verification. HABS can be executed using a simple simulator, but increasing its performance by connecting it to external tools is an open challenge.

The model API and HABS. The model API allows one to connect a HABS model to external tools at runtime. This makes the model open and has two main consequences: On one hand, continuous behavior which is too complex to be described in the **physical** block can interact with the model. On the other hand, static analysis becomes difficult and must possibly be mixed with runtime monitoring with **assert** statements. The reason for this are two-fold: (1) there is no abstraction of the external continuous behavior that can be used for verification and (2) the order of send messages cannot be analyzed a priori.

4 Models for Distributed Control

This section explores the design of modular cyber-physical systems in HABS. Two HABS models of distributed water tanks show different possible runtime structures, showing in particular the issue of *control*: which parts of the HABS code are models of physical devices and which parts are modeling their control.

- The first HABS model has a tank with three controllers that must coordinate the three valves, as in Fig. 3, to keep the tank level in a safe zone. The three controllers read the level of the tank and use a simple round-robin leader election mechanism to make decisions. The behavior of the water tank model is passive.
- The second HABS model has a different scenario: the model API is used to communicate the current level to the HABS model and the water tanks are used to *predict* the optimal control of a single valve.

```
1  class FlowCtrl(ITank tt, Real tick, Int id)
2         implements FlowCtrl{
3  // We elide details on initialization of these attributes
4  Bool isLeader = False;
5  FlowCtrl ctrlA; FlowCtrl ctrlB;
6
7  Unit ctrlFlow(){
8    while True {
9     await duration(tick,tick);
10    if (isLeader) {
11     Real level = tt.outLevel();
12     if(level <= 7/2 ){
13      tt!setFlow(-1/2,id); ctrlA!act(1/2); ctrlB!act(1/2);
14     }
15     if (level >= 19/2) {
16      tt!setFlow(1/2,id); ctrlA!act(-1/2); ctrlB!act(-1/2);
17     }
18     ctrlA!makeLeader();
19     isLeader = False;
20    }
21   }
22 }
23 Unit makeLeader() { isLeader = True; }
24 Unit act(Rat flow) { tt!setFlow(flow, id); }
25 }
```

Fig. 5. Distributed control with *pull* communication from physical device to controllers.

4.1 Internal Control

The first model uses the tank from Fig. 4 and adds three controllers, one for each valve. The controllers read the value of the current level and coordinate the new state of the valves accordingly. Time advances in constant intervals of size `tick` via an `await duration(tick,tick);` statement in each controller.

Example 3. Consider the tank in Fig. 4, with its three controllers. Each controller can control the flow of up to $0.5\frac{l}{s}$ water in or out of the tank, but the tank is only considered safe if the total flow is between $-1\frac{l}{s}$ and $1\frac{l}{s}$, hence the controllers need to coordinate.

The code in Fig. 5 shows one controller implementing the following behavior: After every `tick` seconds (Line 9), the controller checks whether it is the current leader (Line 10) and if so, reads the current water level (Line 11). If the level is below a certain threshold, it sets its own valve to $-0.5\frac{l}{s}$ and orders the other controllers to set theirs to $0.5\frac{l}{s}$ (Line 13). This realizes the requirement above that it is not safe to add/remove water with more than $1\frac{l}{s}$ at every valve. The inverse happens if the water level is above a certain threshold (Line 16). Finally,

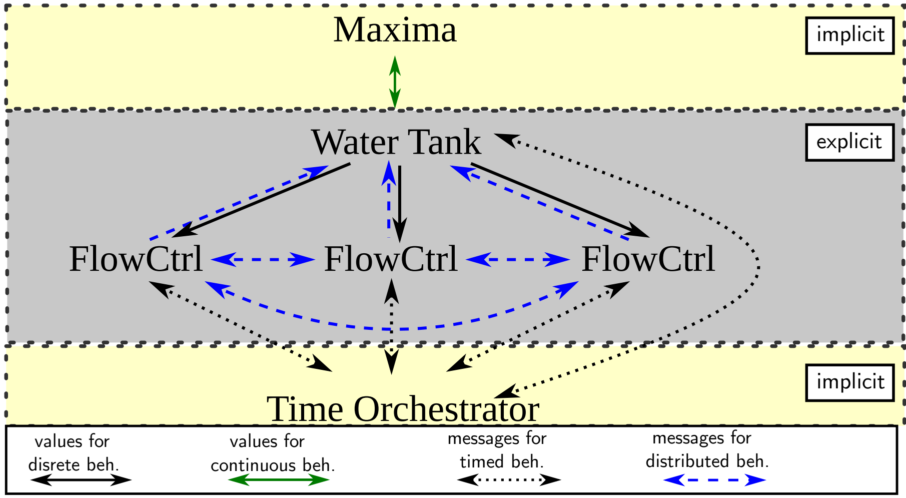

Fig. 6. Structure of the compiled HABS model of Fig, 5.

the controller passes leadership to its left controller (Lines 18–19). Figure 3 shows a sketch of the physical setup (left) and its expected behavior (right) in terms of how the level of the water changes as time progress (during "a kind of" continuous time simulation). In the sketch we are depicting a state in which the first controller has a positive flow (filling water into the tank), while the other two have a negative flow (draining water from the tank).

Runtime Structure. The structure of the compiled model of Example 3 has the following components:

Water Tank Object. This object is hybrid, because it has a **physical** block.

Three Valve Controller Objects. These objects are timed, because they have a **duration** guard but no **physical** block. The messages send around in the distributed system are *only* between the three controller and the water tank objects.

Maxima. A maxima instance computes the continuous dynamics of the physical block of the water tank instance on-demand when asked by it.

Time Orchestrator. The global scheduler of ABS coordinates the global time advance over all objects. It is not responsible for passing values and scheduling messages. Maxima, in essence the “main” simulator in this system, is controlled by its hybrid object.

The communication is pictured in Fig. 6. Communication here is pull communication, because the valve controllers pull the values of the tank object, which is passive otherwise. The main control lies with the three controllers, which also execute the leader election.

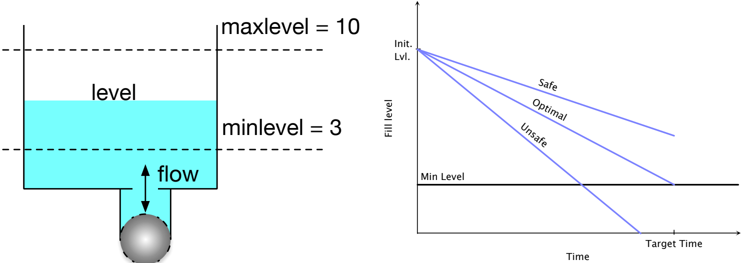

Fig. 7. The predictive watertank and its behavior

As for modeling, each object is a representation of a real world structure, while the implicit components (time orchestrator, maxima) are not exposed in the model. The model is more intuitive, as it requires no knowledge about the details of time advance or synchronization and, thus, is more easy to validate with domain experts. Having the implicit components fixed in the language also simplifies analysis.

We discussed the water tank from the modeling point of view. From a co-simulation point of view we make three observations.

Observation 1 *The maxima instance for a* **physical** *block of the water tank object is essentially a simulation unit, controlled by the* ABS *water tank object (for values) and the orchestrator (for time). Its ODEs can be seen twofold: as configuration of the simulation unit or as the behavioral interface of the continuous behavior used for verification.*

Observation 2 *The* ABS *objects without the* **physical** *block can be seen as a (value and time) orchestrator that propagates and computes values of the sole continuous simulation unit.*

Observation 3 *The controllers can be seen as simulation units themselves.*

From the model and the observations we can derive three points to connect HABS with co-simulation:

1. One can connect the co-simulation engine to the **physical** block, i.e., extend the **physical** block with a more generic interface to the outside. The enveloping object would play the role of an orchestrator. From this view, the water tank object is a co-simulation engine with one simulator (a maxima script).
2. One can connect the HABS-orchestrator with the orchestrator of a co-simulation engine as a sub-orchestrator. However, this would require that the orchestrator can propagate values to the objects.
3. One can connect a co-simulation engine with a designated object, that shares values with other objects and time advances with the HABS-orchestrator, but is transparent to the time semantics of HABS.

We examine these possibilities in detail in Sec. 5.

4.2 Predictive Control

The second model illustrates the use a physical tank model to simulate “what-if” scenarios in order to optimize a model parameter.

Example 4. Consider the water tank depicted in Fig. 7 and coded in Fig. 8. Its `outResult` methods simulates the water tank, returning the final water level by adjusting the flow at the given rate `flowrate` for `wait` time units. The initial water level is always reset to `inVal`. This model can therefore be used for *predictive* control by simulating possible settings for adjusting the `flow` and choosing an appropriate one.

```
1  class Tank(Rat inVal) implements ITank {
2      Real flow = 0;
3      physical{ Real level = inVal : level' = flow; }
4      Real outResult(Real flowrate, Real wait) {
5        level = inVal; flow = flowrate;
6        await duration(wait, wait);
7        flow = 0;
8        return level;
9      }
10 }
```

Fig. 8. Water tank for predictive control.

Figure 9 shows a controller that calculates the largest flow rate that keeps the tank level within bounds for 1 time unit. A call to `go` returns the maximal or minimal setting (depending on whether the water should rise or fall depending on the negative or positive value) for `flow`. To do so, starting from the most extreme setting (determined by `pivot`), the potential flow is changed by $1/10$ until the simulated water level is within bounds for 1 time units. Figure 7 shows a sketch of the physical setup (left) and its expected predicted behavior (right) in terms of how the chosen value for the flow can predict an optimal level in the water for a targeted time.

For the sake of brevity, we refrain from using the three-valve model of Sec. 4.1 here. In brief, extending the model with these cooperating controllers can be done by extending the method `outResult` to take three objects as parameters and simulating, e.g., pull control for one iteration.

Note that in Fig. 9, the controller itself is exposed via the model API. This means that the main control and the parallelism lie *outside* the model. Since this model simulates alternative scenarios, it will typically run under a temporary time orchestrator that does not influence the surrounding model.

Observation 4 *The previous model discussed the HABS runtime as a co-simulation framework. This model exposes that we can see the HABS model as a simulation unit in itself.*

```
1 interface FlowCtrl{
2     [HTTPCallable] Pair<Bool, Rat>  go(Int level);
3 }
4 class FlowCtrl(Real pivot) implements FlowCtrl{
5  Pair<Bool, Rat>  go(Int level){
6    Bool stop = False;
7    Pair<Bool, Rat> lastDecision = Pair(False,0);
8    ITank predict = new Tank(level);
9    Real flow = 0;
10   if(level <= pivot){
11     flow = -2;
12     while(!stop && flow <= 2){
13       Real sim = await predict!outResult(flow, 1);
14       if(sim <= 3) flow = flow + 1/10;
15       else         stop = True;
16     }
17   } else if(level > pivot){
18     flow = 2;
19     while(!stop && flow >= -2){
20       Real sim = await predict!outResult(flow, 1);
21       if(sim >= 10) flow = flow -1/10;
22       else           stop = True;
23     }
24   }
25   return Pair(stop,flow);
26  }
27 }
```

Fig. 9. Predictive controller.

Thus, we can connect co-simulation to HABS at a forth point:

4. One can connect the co-simulation engine via the model API with a special object (here: the `FlowCtrl` instance) as the interface.

Compared to the first model, predictive control is more easy to connect to techniques like co-simulation and digital twins, especially since the model API can be integrated into a live system, but less *natural*: the parts of the water tank do no longer correspond directly to some physical equivalent in the physical system. The `outResult` method, with its discrete change of the water level, is only relevant for simulation. Naturalness is a critical property of models because it influences the communication with domain experts [19].

Observation 5 *If the data is input from a live system, then the overall model is a predictive model of a digital twin.*

From an analysis point of view there are 3 challenges: 1. The model is open and thus harder to analyze statically. 2. The model is harder to specify because

some properties (here: whether the water tank empties completely) are violated by design *without consequence*. 3. The model is harder to specify as parts of the system are outside the model, yet influence its behavior.

5 Discussion: Hybrid Active Objects in Relation to Co-Simulation and Digital Twins

As we have shown in the last section, HABS models have some similarities to co-simulation and digital twins. This raises some questions on the relation between these concepts and how they would benefit from an integration.

5.1 Relation to Co-Simulation

The distributed control patterns discussed for HABS models have some similarity with co-simulation frameworks. Co-simulation consists of techniques to enable global simulation of a coupled system via the composition of simulators [11]. Each simulator can be seen as a black-box capable of exhibiting behavior, consuming inputs, and producing outputs. In discrete event simulations, a simulation unit synchronizes with the environment at some specific timestamps to exchange values. If two events happen at the same time, both are processed before the simulated time progresses. In continuous time simulation (e.g., for cyber-physical systems), state evolves continuously, which introduces flexibility in the step size of the time synchronization. For co-simulation with hybrid simulation units, the units cannot be coupled together by simply connecting input to output ports. Instead the orchestrator needs to reconcile the different assumptions about the inputs and output of each unit, to make sure that the properties of the constituent systems are retained. In the large majority of co-simulators which couple such hybrid simulation units, the coupling is done ad-hoc [11], which makes the co-simulation systems increasingly complex and error-prone.

HABS is expressive enough to model a (distributed) co-simulation framework and reason about the correctness of the framework. We have seen that loosely coupled orchestration can be modeled, separating data exchange from time synchronization. More tightly coupled orchestration can be modeled as a special case in which, e.g., all data pass through a central object. ABS has a developed, tool-supported theory of reproducibility [31] and search through reachable states, which may be used to reason about determinism and confluence for parallel orchestrators, a recognized challenge in discrete event simulation [11]. The hybrid extension of HABS does not affect the semantics of parallelism underlying this theory. In a hybrid co-simulation, the orchestrator needs to ensure the validity of the simulation in terms of the assumptions about parameter values to the different simulation units. Although the time semantics of HABS makes it difficult to directly model roll- back and similar direct manipulation of time inside the executable model, but we believe HABS can be used to reason about such time manipulation by branching executions.

Aspect	Physical block	Orchestrator	Model API	Interface objects
Runtime	−	+	+	+
Naturalness	+	+	−	−/+
Analyzability	+	−	−−	−/+
Relevant example	Sec. 4.1	Sec. 4.1	Sec. 4.2	Sec. 4.2

Table 1. Connection between internal and external components.

HABS models can also be part of a co-simulation framework. Section 4.1 illustrates a HABS model as a decentralized orchestrator, which needs to obtain consensus on the actions of the different pumps. Section 4.2 illustrates a HABS model as a simulation unit in a co-simulation setting, which receives parameter values from the outside. Our work so far does not point to a clear "best way" to connect HABS to a co-simulation framework, as this depends on the modeling style used for a system.

The observations made in Sec. 4 suggest what we can expect from the different integration concepts we have considered between HABS and a co-simulation framework. The benefit of such an integration would be an enhancement of simulation capabilities for HABS and an enhancement of analyzability of the co-simulation framework. Such an integration would be an important step to simulate large and hybrid HABS systems. There are three main questions, which we now discuss in more detail:

I **Runtime:** how to technically connect external units to HABS entities?
II **Naturalness:** how to preserve the naturalness of a HABS model when connecting external units?
III **Analyzability:** how to keep parallelism analyzable and retain tools developed for closed HABS/ABS systems?

Table 1 summarizes the expected consequences of using a particular connection method with respect to runtime integration (i.e., how easy is it to exchange data with external entities), naturalness of the model and model analyzability. We discuss the table from left to right.

– Interfacing to external components using the **physical** block preserves the naturalness of the model and, if a proper abstraction can be provided, we expect the model to be analyzable in the sense that the semantics of parallelism does not change and the system is still closed (because the object containing the **physical** block retains control). However, this control over the **physical** block limits the kind of simulation one can do at runtime because the semantics do not allow for complex message exchanges.
– Using the orchestrator, we expect a better connection at runtime but a worse analyzability, because the control is now *outside* of the formal semantics. Naturalness again depends on how the interface is modeled, but as the connection is to an implicit runtime component, we expect only limited effects.

- Using the model API moves more control to the outside of the formal semantics. This gives one most freedom about the connection to an external orchestrator, but will lead to less natural models and lower analyzability.
- We can, however, internalize the model API to retain its benefits of easy integration while limiting its impact on analyzability and naturalness. This can be done by means of *designated interface objects*. which are the only objects exposed in the model API. These objects could extend the current Model API to support not only receiving messages but may also sending messages to the outside. In future work, we intend to define the design space for interface objects such that they comply with the compositional time semantics of HABS, to preserve the analyzability. Both timed and functional behavior should be specifiable, e.g., by behavioral types that specify communication, time and state [4,16]. This would allow specified assumptions about external behavior to be monitored at runtime, such that static checking of internal behavior remains feasible. We expect such specifications to improve the naturalness of models with interface objects.

In summary, the model API appears as a difficult choice because we lose the benefits of using a formal modeling language, but it gives the best prospect for simulation. The other discussed options give possible trade-offs between simulation, naturalness and analyzability. Using a special interface object that serve to control the external interface may be the best compromise if external behavior can then be encapsulated with specifications which allow static intra-model reasoning combined with runtime monitoring of the interface behavior. However, this approach does impose restrictions on the control of an external orchestrator.

5.2 Relation to Digital Twins

A digital twin (DT) is a digital replica of an underlying system, often called the physical twin (PT). The digital twin is connected to its physical twin in real-time through sensor measurements at different locations and by other ways of collecting data. This turns the DT into a *live replica* of the PT, with the purpose of providing insights into its behavior, and clearly distinguishes a DT from, e.g., a standard simulation model.

A DT is commonly seen as an architecture with three layers: the *data layer* with, e.g., CAD drawings and sensor data, an *information layer*, which turns these raw data into structures, and an *insight layer*, which applies different analysis and visualization techniques to these data structures. The analysis techniques of the insight layer can be classified as follows: The DT is typically able to compute an approximation of how the PT acts in a given scenario (**simulation** or "what-happened" scenarios), or to estimate how the PT will behave in the future based on historical and current data (**prediction** or "what-may-happen" scenarios). By configuring the parameters of the different models, the DT may analyze the consequence of different options on future behavior (**prescription** or "what-if" scenarios).

When modeling human-built artifacts, the physical twin can be divided into a *cyber-physical system* (CPS) and its *physical environment*. The latter is a mix of physical conditions (e.g., temperature or fluid pressure) and modeled conditions (e.g., motion tracking devices or computerized decision rules). In the DT, the CPS model may consist of several sub-models reflecting the different parts of the CPS. Similarly, the environment model comprises many models capturing the relevant dynamics of the operational environment. These smaller, targeted models are typically created by experts in the respective fields.

HABS is an expressive modeling language, which includes abstractions of cyber-physical systems. Although in theory an entire physical twin can be captured in a huge model, we do not think this is the way to go for complex, industrial cases. The digital twin as a "stack of models" seems more realistic, and a HABS model could be situated at many layers of such a stack. The compositional treatment of time in HABS suggests that models can contain components at different levels of abstraction, synchronized through the timed semantics. The information layer of a digital twin may quickly become very large and contain a lot of unnecessary information for a given analysis. HABS is not made for modeling complex data relations and large sets of data, and we propose to use complementary techniques to connect the executable models to static data (see, e.g., recent work on combining semantic ontologies with formal models [7]). We have shown how HABS models may both receive such data from the outside (Sect. 4.2). and call external simulators such as maxima (Sect. 4.1). Section 4.2 has shown by example how HABS models can integrate real-time data series through the Model API [27], which enables real-time simulation in a DT context of sensor data. This example also suggests through a simple example how different, alternative parameter settings can be used to explore what-if scenarios for prescriptive analysis.

6 Conclusion

This paper considers decentralized orchestration of cyber-physical systems "in the large". We have explored different ways of connecting a model of distributed active objects with cyber-physical components to model different forms of orchestration.

We did so from an active object perspective, a concurrency model that decouples communication from synchronization. This makes control-flow of active object systems flexible and enables complex orchestration patterns which depend on the exchange of values between different active objects. Our hybrid active object model HABS, a hybrid extension of ABS, is executable, which makes the language well-suited for model simulation, yet it has a formal, compositional semantics which makes the language well-suited for verification and validation.

We discuss different solutions to the modeling of distributed cyber-physical systems which combine a HABS model with external simulators for natural systems from the perspective of both analysis, simulation and naturalness. The resulting hybrid multi-models give rise to a co-simulation problem. Our exam-

ples show that it is challenging to preserve both support for simulation and for compositional reasoning which relies on the formal language semantics of HABS. The examples allow us to observe that we profit from built-in time orchestration in HABS which facilitates composition, but that we suffer from external simulation units which reduce that value of a composition formal semantics for reasoning about the orchestration model.

Despite the challenges, we identify interface objects with formal specifications as a promising solution that may provide a reasonable compromise between flexible simulation and orchestration, and restricting on the level of control to which an external unit may subject the HABS model. The work reported in this paper points to several interesting directions for further work.

Formally Analyzable Co-Simulation. By implementing orchestrators as models in HABS that interact with the external simulation units through (possibly extended) physical blocks, we plan to study correctness properties for orchestration algorithms through formal analysis, including safety proofs for value propagation.

Formally Analyzable Digital Twins. By implementing designated interface objects, we plan to study the combination of compositional reasoning and simulation for predictive and prescriptive analysis in a digital twin setting.

References

1. Albert, E., Arenas, P., Flores-Montoya, A., Genaim, S., Gómez-Zamalloa, M., Martin-Martin, E., Puebla, G., and Román-Díez, G. SACO: static analyzer for concurrent objects. In *Tools and Algorithms for the Construction and Analysis of Systems, 20th Intl. Conf., TACAS, Grenoble, France* (2014), E. Ábrahám and K. Havelund, Eds., vol. 8413 of *LNCS*, Springer, pp. 562–567.
2. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T. A., Ho, P., Nicollin, X., Olivero, A., Sifakis, J., and Yovine, S. The algorithmic analysis of hybrid systems. *Theor. Comput. Sci. 138*, 1 (1995), 3–34.
3. Blochwitz, T., Otter, M., Åkesson, J., Arnold, M., Clauss, C., Elmqvist, H., Friedrich, M., Junghanns, A., Mauss, J., Neumerkel, D., Olsson, H., and Viel, A. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In *Proceedings of the 9th International Modelica Conference* (2012), The Modelica Association, pp. 173–184.
4. Bocchi, L., Murgia, M., Vasconcelos, V. T., and Yoshida, N. Asynchronous timed session types - from duality to time-sensitive processes. In *Proc. 28th European Symposium on Programming (ESOP 2019)* (2019), L. Caires, Ed., vol. 11423 of *Lecture Notes in Computer Science*, Springer, pp. 583–610.
5. Boer, F. D., Serbanescu, V., Hähnle, R., Henrio, L., Rochas, J., Din, C. C., Johnsen, E. B., Sirjani, M., Khamespanah, E., Fernandez-Reyes, K., and Yang, A. M. A survey of active object languages. *ACM Comput. Surv. 50*, 5 (Oct. 2017), 76:1–76:39.
6. Din, C. C., and Owe, O. Compositional reasoning about active objects with shared futures. *Formal Asp. Comput. 27*, 3 (2015), 551–572.

7. Dubslaff, C., Koopmann, P., and Turhan, A. Ontology-mediated probabilistic model checking. In *Proc. 15th Intl. Conf. on Integrated Formal Methods (IFM 2019)* (2019), W. Ahrendt and S. L. Tapia Tarifa, Eds., vol. 11918 of *Lecture Notes in Computer Science*, Springer, pp. 194–211.
8. Fitzgerald, J. S., Larsen, P. G., and Pierce, K. G. Multi-modelling and co-simulation in the engineering of cyber-physical systems: Towards the digital twin. In *From Software Engineering to Formal Methods and Tools, and Back - Essays Dedicated to Stefania Gnesi on the Occasion of Her 65th Birthday* (2019), vol. 11865 of *Lecture Notes in Computer Science*, Springer, pp. 40–55.
9. Fulton, N., Mitsch, S., Quesel, J., Völp, M., and Platzer, A. Keymaera X: an axiomatic tactical theorem prover for hybrid systems. In *CADE'25* (2015), A. P. Felty and A. Middeldorp, Eds., vol. 9195 of *LNCS*, Springer, pp. 527–538.
10. Gomes, C., Lúcio, L., and Vangheluwe, H. Semantics of co-simulation algorithms with simulator contracts. In *MODELS Companion* (2019), IEEE, pp. 784–789.
11. Gomes, C., Thule, C., Broman, D., Larsen, P. G., and Vangheluwe, H. Co-simulation: A survey. *ACM Comput. Surv. 51*, 3 (2018), 49:1–49:33.
12. Hewitt, C., Bishop, P., and Steiger, R. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (San Francisco, CA, USA, 1973), IJCAI'73, Morgan Kaufmann Publishers Inc., pp. 235–245.
13. Johnsen, E. B., Hähnle, R., Schäfer, J., Schlatte, R., and Steffen, M. ABS: A core language for abstract behavioral specification. In *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)* (2011), B. Aichernig, F. S. de Boer, and M. M. Bonsangue, Eds., vol. 6957 of *Lecture Notes in Computer Science*, Springer, pp. 142–164.
14. Kamburjan, E. Behavioral program logic. In *Automated Reasoning with Analytic Tableaux and Related Methods - 28th International Conference, TABLEAUX 2019, London, UK, September 3-5, 2019, Proceedings* (2019), S. Cerrito and A. Popescu, Eds., vol. 11714 of *Lecture Notes in Computer Science*, Springer, pp. 391–408.
15. Kamburjan, E. From post-conditions to post-region invariants: Deductive verification of hybrid objects. In *HSCC'21, to appear* (2021).
16. Kamburjan, E., and Chen, T. Stateful behavioral types for active objects. In *Integrated Formal Methods - 14th International Conference, IFM 2018, Maynooth, Ireland, September 5-7, 2018, Proceedings* (2018), C. A. Furia and K. Winter, Eds., vol. 11023 of *Lecture Notes in Computer Science*, Springer, pp. 214–235.
17. Kamburjan, E., Hähnle, R., and Schön, S. Formal modeling and analysis of railway operations with active objects. *Sci. Comput. Program. 166* (2018), 167–193.
18. Kamburjan, E., Mitsch, S., Kettenbach, M., and Hähnle, R. Modeling and verifying cyber-physical systems with hybrid active objects. *CoRR abs/1906.05704* (2019).
19. Kamburjan, E., and Stromberg, J. Tool support for validation of formal system models: Interactive visualization and requirements traceability. In *Proceedings Fifth Workshop on Formal Integrated Development Environment, F-IDE@FM 2019, Porto, Portugal, 7th October 2019* (2019), R. Monahan, V. Prevosto, and J. Proença, Eds., vol. 310 of *EPTCS*, pp. 70–85.
20. Larsen, K. G., Pettersson, P., and Yi, W. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer 1*, 1–2 (1997), 134–152.

21. Lin, J., Mauro, J., Røst, T. B., and Yu, I. C. A model-based scalability optimization methodology for cloud applications. In *2017 IEEE 7th International Symposium on Cloud and Service Computing (SC^2 2017)* (2017), IEEE Computer Society, pp. 163–170.
22. Margaria, T., and Schieweck, A. The digital thread in industry 4.0. In *Integrated Formal Methods - 15th International Conference, IFM 2019, Bergen, Norway, December 2-6, 2019, Proceedings* (2019), vol. 11918 of *Lecture Notes in Computer Science*, Springer, pp. 3–24.
23. Maxima Development Group. *Maxima Manual*, 5.43.0 ed., 2019. maxima.sourceforge.net.
24. Platzer, A. The complete proof theory of hybrid systems. In *LICS* (2012), IEEE, pp. 541–550.
25. Platzer, A. *Logical Foundations of Cyber-Physical Systems*. Springer, 2018.
26. Røst, T. B., Seidl, C., Yu, I. C., Damiani, F., Johnsen, E. B., and Chesta, C. Hyvar - scalable hybrid variability for distributed evolving software systems. In *Advances in Service-Oriented and Cloud Computing* (2017), vol. 824 of *Communications in Computer and Information Science*, Springer, pp. 159–163.
27. Schlatte, R., Johnsen, E. B., Mauro, J., Tarifa, S. L. T., and Yu, I. C. Release the beasts: When formal methods meet real world data. In *It's All About Coordination - Essays to Celebrate the Lifelong Scientific Achievements of Farhad Arbab* (2018), F. S. de Boer, M. M. Bonsangue, and J. Rutten, Eds., vol. 10865 of *Lecture Notes in Computer Science*, Springer, pp. 107–121.
28. Seshia, S. A. New frontiers in formal methods: Learning, cyber-physical systems, education, and beyond. *CSI Journal of Computing 2*, 4 (June 2015), R1:3–R1:13.
29. Thule, C., Gomes, C., DeAntoni, J., Larsen, P. G., Brauer, J., and Vangheluwe, H. Towards the verification of hybrid co-simulation algorithms. In *STAF Workshops* (2018), vol. 11176 of *Lecture Notes in Computer Science*, Springer, pp. 5–20.
30. Thule, C., Lausdahl, K., Gomes, C., Meisl, G., and Larsen, P. G. Maestro: The INTO-CPS co-simulation framework. *Simul. Model. Pract. Theory 92* (2019), 45–61.
31. Tveito, L., Johnsen, E. B., and Schlatte, R. Global reproducibility through local control for distributed active objects. In *Proc. 23rd International Conference on Fundamental Approaches to Software Engineering (FASE 2020)* (2020), H. Wehrheim and J. Cabot, Eds., vol. 12076 of *Lecture Notes in Computer Science*, Springer, pp. 140–160.