Software and Systems Modeling
https://doi.org/10.1007/s10270-021-00877-y

THEME SECTION PAPER

An actor-based framework for asynchronous event-based cyber-physical systems

Iman Jahandideh[1] · **Fatemeh Ghassemi**[1,2] · **Marjan Sirjani**[3,4]

Received: 27 February 2020 / Revised: 3 February 2021 / Accepted: 19 February 2021

Abstract
In cyber-physical systems like automotive systems, there are components like sensors, actuators, and controllers that communicate asynchronously with each other. The computational model of actors supports modeling distributed asynchronously communicating systems. We propose the Hybrid Rebeca language to support the modeling of cyber-physical systems. Hybrid Rebeca is an extension of the actor-based language Rebeca. In this extension, physical actors are introduced as new computational entities to encapsulate physical behaviors. To support various means of communication among the entities, the network is explicitly modeled as a separate entity from actors. We develop a tool to derive hybrid automata as the basis for the analysis of Hybrid Rebeca models. We demonstrate the applicability of our approach through a case study in the domain of automotive systems. We use the SpaceEx framework for reachability analysis of the case study. Compared to hybrid automata, our results show that for event-based asynchronous models hybrid Rebeca improves analyzability by reducing the number of real variables, and increases modularity and hence, minimizes the number of changes caused by a modification in the model.

Keywords Actor model · Cyber-physical systems · Hybrid automata

1 Introduction

Embedded systems consist of microprocessors which control a physical behavior. Ninety-eight percent of all microprocessors are manufactured as components of embedded systems [40]. In such *hybrid* systems, physical and cyber behaviors, characterized as continuous and discrete respectively, affect each other. Cyber-physical systems (CPSs) are heterogeneous systems with tight interactions between physical and software processes where components in the system usually communicate through the network. These systems are used in wide variety of safety-critical applications, from automotive and avionic systems to robotic surgery and smart grids. This makes verifying and analyzing CPSs one of the main concerns while developing such systems.

Model-based design is an effective technique for developing correct CPSs [11]. It relies on models specifying the behavior of the system often with informal notations. Using formal models instead of physical realizations of the system not only provides new insights early in the design process, but also enables analyzing the system behavior in many complex situations that can not easily be reproduced in its real environment. Furthermore, formal and extensive analysis of the model can provide more confidence in the correctness of the system. The heterogeneity of CPSs creates new modeling challenges that stem from interactions between different kinds of components. New theories and tools are needed to facilitate designing and analyzing CPSs. In [36], Sirjani argues that for dealing with complicated and heterogeneous systems, *friendliness* of our design tools can be as important as their *expressiveness*. Friendliness of a modeling language

Communicated by Eugene Syriani and Manuel Wimmer.

✉ Fatemeh Ghassemi
fghassemi@ut.ac.ir

Iman Jahandideh
jahandideh.iman@ut.ac.ir

Marjan Sirjani
marjan.sirjani@mdh.se

1 School of Electrical and Computer Engineering, College of Engineering, University of Tehran, Tehran, Iran

2 School of Computer Science, Institute for Research in Fundamental Sciences, P.O. Box 19395-5746, Tehran, Iran

3 School of Innovation, Design and Engineering, Mälardalen University, Västerås, Sweden

4 School of Computer Science, Reykjavik University, Reykjavik, Iceland

is evaluated by its *faithfulness* to the target system that is going to be modeled, and *usability* to the modeler.

Lee is one of the pioneers to address design challenges of cyber-physical systems in [25]. According to him for realizing the full potential of CPS, we will have to rebuild computing and networking abstractions. These abstractions will have to embrace physical dynamics and computation in a unified way. In addition to components for representing software and physical processes, we need components to model communication over a network. Modeling different types of communication over network are crucial in CPS. For instance in the domain of automotive systems, ECUs, sensors and actuators may be connected directly by wire or through a communication medium such as a serial bus. Moreover, to represent embedded systems and CPS we need models with a strong and rich support for time and timing constraints.

In this work, we propose *Hybrid Rebeca*, as an extension of Timed Rebeca [1,38]. Rebeca is an imperative actor-based language with formal semantics, Java-like syntax, and supported by model checking tool. Its timed extension supports modeling of the computation time, and network delay for message communication. Actors are suggested as excellent candidates for modeling CPS. Hewitt actors [2,16] provide a suitable level of abstraction to faithfully model distributed event-based asynchronously communicating systems. Actors are units of computation which can only communicate by asynchronous message passing. Each actor is equipped by a mailbox in which the received messages are buffered. This provides a faithful model for a networked system. We extend the actor model of Timed Rebeca to include physical actors with continuous behavior to support hybrid systems. We also introduce an explicit network entity to support various types of communication, namely wired connections with no delay, serial buses with deterministic behavior, and wireless communication among the actors. Without this explicit entity in the model, in order to model different behaviors of different network media, one has to model the network as a separate actor within a Timed Rebeca model.

Improving the level of abstraction by providing first-class entities for each modeling concept, reduces the number of errors introduced during the design process and improves the understandability of the model. Existing formal modeling languages for hybrid systems such as hybrid automata [4,15], hybrid process algebra [8], logic for hybrid programs [32], and hybrid Petri nets [9] are introduced to model CPSs. No high-level structuring elements such as modules, or modeling concepts for asynchronous communication are supported in these languages. These languages are not generally designed for modeling systems composed of many interacting heterogeneous entities communicating via network. The design is mostly based on the underlying formalism with the focus on formal verification rather than ease of modeling.

Hybrid Rebeca defines two types of classes, software and physical. Software classes are similar to reactive classes in the Rebeca language where the computational behaviors are defined by message servers. Physical classes in addition to message servers, can also contain different modes, where the continuous behaviors are specified. A physical actor (which is instantiated from a physical class) must always have one active mode. This active mode defines the runtime continuous behavior of the actor. By changing the active mode of a physical actor, it is possible to change the continuous behavior of the actor. The modes of physical classes are similar to the concept of locations in hybrid automata.

To be able to have a concrete example and clearer discussions, here we focus on automotive systems. In such systems, a Control Area Network (CAN) is defined as a network model for communications of the actors. Actors can communicate with each other either through the CAN network or directly by wire. Since CAN is a priority-based network, a priority must be assigned for the messages that are sent through CAN.

We define the semantic model of hybrid Rebeca in terms of hybrid automata. In other words, for a given hybrid Rebeca model, we derive a monolithic hybrid automaton. To analyze the resulting hybrid automaton, we provide a set of guideline how verification of timed properties can be reduced to reachability analysis of hybrid automaton, for which many verification algorithms and tools are available. We demonstrate the applicability of our approach through a case study on a simplified Brake-By-Wire system with Anti-lock Braking System. We use the SpaceEx framework for reachability analysis of the case study.

We evaluate the effectiveness of our approach by comparing our modeling framework to hybrid automata. To this aim, we show that by using our framework, the cost of improving and modifying models is vastly reduced compared to modeling in hybrid automata. We also show that modeling high level concepts of hybrid Rebeca like message passing and message buffering directly in hybrid automata can hugely decrease the analyzability of the models. Furthermore, the abstraction resulted from choosing actors as the basic units of computation, offers more friendliness toward cyber-physical systems compared to the low-level languages like hybrid automata. Our contributions are summarized as follows:

- Proposing the actor-based modeling language, Hybrid Rebeca, with first-class entities for modeling the major concepts in the domain of CPSs, i.e., physical and software actors, and different network models,
- Defining formal semantics rules to generate monolithic hybrid automata as the semantic model of hybrid Rebeca,
- Providing algorithms and tools for formal verification of Hybrid Rebeca models using existing tools for the reachability analysis of hybrid automata, and

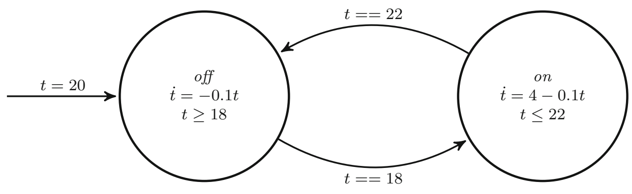


Fig. 1 A hybrid automaton for the thermostat controller

– Specifying compositional semantics of Hybrid Rebeca using hybrid automata.

This paper extends an earlier conference publication [17] by adding structural rules for operational formal semantics as a monolithic hybrid automata, proposing a compositional semantics, and providing precise guidelines for reducing the verification of time properties to reachability analysis of hybrid automata in order to use SpaceX tool.

The novelty of our modeling language can be explained in terms of our design decisions: separating physical rebecs from the software ones, explicit modeling of network, message-based communication between the software and physical rebecs (instead of using shared variables or ports).

2 Preliminaries

As we define the semantics of our framework based on hybrid automata, we first provide an overview on this model and its analysis approaches. Then, we explain the actor model and timed Rebeca.

2.1 Hybrid automata

Hybrid automata (HA) [4,15] is a formal model for systems with discrete and continuous behaviors. Informally a hybrid automaton is a finite state machine consisting of a set of real-valued variables, modes, and transitions. Each mode, which we also call *location*, defines a continuous behavior on the variables of the model. The continuous behaviors or *flows* are usually described by ordinary differential equations which define how the values of the variables change with time. Transitions act as discrete actions between continuous behaviors of the system, where the variables can change instantaneously.

Example 1 Assume a controller that turns on when the environment temperature is 18 and turns off the heater when the temperature is 22. The model of this controller is given in Fig. 1. The variable t represents the temperature of the environment. The locations named *off* and *on* define the continuous behavior of the temperature when the heater is off and on, respectively. For each location, the flow of the temperature is defined accordingly. The transition with the guard $t == 22$ states that when the temperature is equal to 22 the heater *can* be turned off. In hybrid automaton, the choice between staying in one location and taking an enabled transition is nondeterministic. To make the turning off behavior deterministic, the invariant $t \leq 22$ is defined in the *on* location. This invariant states that the heater can only stay in this location as long as the temperature is less than equal to 22. The turning on behavior of the heater is defined similarly. Initially the heater is off and the temperature is 20.

Let a valuation $v : V \rightarrow \text{Real}$ be a function that assigns a real value to each variable of V. We denote the set of valuations on the set of variables V as $\mathcal{V}(V)$.

Definition 1 A hybrid automaton is defined by the tuple (Loc, Var, Lab, $\Rightarrow$, Flws, Inv, Init) as follows:

- Loc is a finite set of locations,
- Var is a finite set of real-valued variables,
- Lab is a finite set of synchronization labels.
- $\Rightarrow$ is a finite set of transitions. A transition is a tuple $(l, a, \mu, l') \in \Rightarrow$ where $l \in$ Loc is the source location, $l' \in$ Loc is the destination location, $a \in$ Lab is a synchronization label and $\mu \in \mathcal{V}(\text{Var})^2$ is a transition relation on variables. The elements of $\mu = (v, v')$ represent the valuation of the variables before and after taking the transition. In some models, like in our example, this transition relation is represented with a guard and a set of assignments on the variables. The guard defines the valuation v and the assignments define the valuation v'.
- Flws is a labeling function that assigns a set of flows to each location $l \in$ Loc. Each flow is a function from $\mathbb{R}^{\geq 0} \rightarrow \mathcal{V}(\text{Var})$. Each flow specifies how the values of variables evolve over time. A flow is usually defined by a dotted variable $\dot{x}$ which represents the first derivative of variable x.
- Inv is a labeling function that assigns an invariant $\text{Inv}(l) \subseteq \mathcal{V}(\text{Var})$ to each location $l \in$ Loc.
- Init is a labeling function that assigns an initial condition $\text{Init}(l) \subseteq \mathcal{V}(\text{Var})$ to each location $l \in$ Loc.

In the example given in Fig. 1, the locations and the variables are defined as $Loc = \{on, off\}$ and $Var = \{t\}$ respectively. Since our example only consists of a single automaton, $Lab = \emptyset$.

The flows and the invariant of each location are defined on the location itself. The initial condition for location *off* is $Init(off) = \{t = 20\}$ and for location *on* is $Init(on) = \emptyset$. Note that in our language, we use x' instead of $\dot{x}$ to represent the first derivative of variable x.

2.2 Analysis of hybrid auotomata

In this section, we elaborate on the methods and tools for the analysis of hybrid automata. There are mainly two approaches for the analysis of hybrid automata: *reachability analysis* and *invariant computation*.

In the reachability analysis approach, given an undesired set of states, called unsafe states, for hybrid automaton, all reachable states of the model are computed to examine if they contain any unsafe state. As the problem of determining the reachability of a state for a hybrid automaton is not decidable [3], an over-approximation for the reachable state set is computed within a bounded time horizon. We can immediately infer a system is safe when its over-approximation does not contain any unsafe state. Otherwise, the safety is unknown. Different tools and approaches use different representations for over-approximating sets. In the invariant computation approach, one derives a system of constraints such that all system states satisfy them [35]. The system is safe if the constraints are inconsistent with the specification of unsafe set. Reachability analysis can be applied in a time-bounded analysis task with a much lower cost, while invariant computation work well on unbounded time horizons.

There are many tools supporting reachability analysis of hybrid automata. Among them, we can mention PHAVer [13], SpaceEx [14], and Flow* [6]. We derive hybrid automata from hybrid Rebeca models for reachability analysis. We use SpaceEx, supported by a user-friendly GUI as an example tool for handling our reachability analysis. Within this tool, we can assign a name to each location and define the unsafe condition in terms of the name of locations and real-valued variables. For example the unsafe condition "loc() == Fault" examines if the location named Fault is reachable.

2.3 Actor model and timed Rebeca

Actor model is used for modeling distributed systems. It was originally proposed by Hewitt [16]. In this model actors are self-contained and concurrent [2] and can be regarded as units of computation. Any communication is done through asynchronous message passing on a fair medium where message delivery is guaranteed but is not in-order. This model abstracts away the network effects like delays, message conflicts, and node crashes. Each actor can only communicate with its so-called *acquaintances*. In this model, each actor has an address and a mailbox which stores the received messages. The behavior of an actor is defined by its message handlers, called *methods*. The methods are executed by processing the messages.

To extend the actor model with hybrid concepts for specifying CPSs, we use Rebeca as our basis framework and hence, use the terms actor model and Rebeca interchangeably in this paper. Rebeca [38] is a formal actor-based modeling language and is supported by model checking tools to bridge the gap between formal methods and software engineering. Rebeca provides an operational interpretation of the actor model through a Java-like syntax. It also offers a compositional verification approach to deal with the state-space explosion problem in model checking. Because of its design principle, it is possible to extend the core language for a specific domain [37]. For example, different extensions have been introduced in various domains such as probabilistic systems [39], real-time systems [1], software product lines [34], and broadcasting environment [41,42].

In Rebeca, actors are called rebecs and are instances of *reactive classes* defined in the model. Rebecs communicate with each other through asynchronous message passing and its mailboxes are modeled by message queues. A reactive class consists of *known rebecs* to specify its acquaintances, *state variables* to maintain the internal state, and *message servers* to define the reaction of the rebec on the received messages. The computation in a rebec takes place by removing a message from the message queue and executing its corresponding message server.

Timed Rebeca [1] is an extension of Rebeca for distributed and asynchronous systems with timing constraints. It adds the timing concepts *computation time*, *message delivery time* and *message expiration*. These concepts are materialized by the constructs: *delay*, *after*, and *deadline*. In timed Rebeca models, each rebec has its own local clock which can be considered as synchronized distributed clocks. The *delay* statement models the passage of time during the execution of a message server. Statements *after* and *deadline* are used in conjunction with send statements and specify the network delay and the message deadline, respectively. The syntax of timed Rebeca is given in Fig. 2.

Example 2 Consider a room in which a controller turns on/off the heater in terms of the room temperature. The temperature is sent to the controller periodically by a sensor. The model of a controller turning on/off a heater in a room, based on the received information from the sensor, is given in Fig. 3. It consists of two actors, *HeaterWithSensor* and *Controller*. In the class *HeaterWithSensor*, the room temperature is modeled by the discrete state variable tempr which is updated every one second by the message server update regarding the status of the *HeaterWithSensor*. When status holds, it means that the heater is on and the temperature is increased. Conversely, when the heater is off, the temperature is decreased. The periodic update behavior is modeled by using the *delay* statement and sending the update message to itself, using the self. It also periodically sends the current room temperature by sending the control message to *Controller* every 3 s through the message server sampleTemp. The delay of the network is modeled by the *after* statement; meaning that after

$$\begin{array}{rcl}
\text{Model} & ::= & \langle Class \rangle^{+}\ \text{Main} \\
\text{Main} & ::= & \mathsf{main}\ \{\text{InstanceDcl}^{*}\} \\
\text{InstanceDcl} & ::= & \text{C r } (\langle \text{r} \rangle^{*})\ :\ (\langle \text{c} \rangle^{*}) \\
\text{Class} & ::= & \mathsf{reactiveclass}\ \text{C}\ \{\mathsf{KnownRebecs\ Vars\ MsgSrv}^{*}\} \\
\text{KnowRebecs} & ::= & \mathsf{knownrebecs}\ \{\text{VarDcl}\} \\
\text{Vars} & ::= & \mathsf{statevars}\ \{\text{VarDcl}\} \\
\text{VarDcl} & ::= & \langle \text{T v} \rangle^{*}; \\
\text{MesgSrv} & ::= & \mathsf{msgsrv}\ \text{m (VarDcl) } \{\text{Stmt}^{*}\} \\
\text{Stmt} & ::= & \text{v} = \text{Expr};\ \mid \text{Call};\ \mid \mathsf{if}(\text{Expr})\ \text{MSt}\ [\mathsf{else}\ \text{MSt}] \mid \mathsf{delay}(\text{Expr}); \\
\text{Call} & ::= & \text{r.m}(\langle \text{Expr} \rangle^{*})[\mathsf{deadline}\ \text{Expr}][\mathsf{after}\ \text{Expr}] \\
\text{MSt} & ::= & \{\text{Stmt}^{*}\} \mid \text{Stmt} \\
\text{Expr} & ::= & c \mid v \mid \text{Expr}\ \mathsf{op}\ \text{Expr},\ \mathsf{op} \in \{+,-,*,\wedge,\vee,<,\leq,>,\geq\} \mid (\text{Expr}) \mid !(\text{Expr})
\end{array}$$

Fig. 2 Abstract syntax of Timed Rebeca. Angle brackets ⟨ ⟩ denotes meta parenthesis, superscripts + and ∗ respectively are used for repetition of one or more and repetition of zero or more times. Combination of ⟨ ⟩ with repetition is used for comma separated list. Brackets [ ] are used for optional syntax. Identifiers C, T, m, v, c, and r respectively denote class, type, method name, variable, constant, and rebec name, respectively.

2 s the message control will be delivered to *Controller*. The controller changes the status of the heater to on and off when the received room temperature is less than or equal to 18 and more than or equal to 22, respectively.

By model checking technique, we can verify the property that "The room temperature never reaches 15". Initially, the sensor has two messages update and sampleTemp in its queue. Upon handling sampleTemp by *HeaterWithSensor* at time = 1 and time = 4 s, *Controller* receives control(19) and control(16) messages at time = 3 s and time = 6 s, respectively. *Controller* sends a change message at time = 6 s to *HeaterWithSensor*, it will be delivered at time = 8 s. As the temperature decreased one by one until time = 8 s, the property is violated. We will replace the discrete variable tempr with a real one while its stepwise changes is governed by a dynamic differential equation in hybrid Rebeca.

3 The actor model for CPSs

Extending the actor model for modeling cyber-physical systems can be divided into two parts, *offering more concrete models for the network*, and *extending actors with physical behaviors*.

Rebeca offers a fair and nondeterministic network model. For many applications of CPSs this network model is too abstract or completely invalid. For example, control area network (CAN) [31] protocol is a dominant networking protocol in automotive industry, which cannot be faithfully modeled by Rebeca's network model because by using this protocol, messages are deterministically delivered to their receivers. Modeling the network as an explicit actor does not guarantee determinacy of message deliveries as the network actor is executed concurrently with other actors, therefore its determinacy is affected by the interleaving semantics. In other words, the message delivery sequence by the network actor depends on the execution order of sending actors. So, we model the network as a separate entity from the actors.

To extend the actor model with physical behaviors, we decided to separate physical actors from software actors. In this approach, software actors will be similar to Timed Rebeca actors and the physical behaviors are defined in separate physical actors. Physical actors are similar to a hybrid automaton in syntax and semantics. Like a hybrid automaton, each physical actor consists of a set of modes. Each mode defines its flows, invariant, guard, and a block statement. The statements of the block define the effect of the mode when the continuous behavior is finished. A physical actor can only be in one mode (characterizing a specific continuous behavior) at any moment. In this approach, the physical behavior of a system can be easily started, stopped or changed by changing the active modes of physical actors, either by the actor itself or by a request from another actor.

As we focus on automotive systems, to make the network specification more concrete, in the first step we consider the CAN protocol in our language. CAN is a serial bus network where nodes can send messages at any moment. When multiple nodes request to send messages at the same time, only the message with the highest priority is accepted and sent through the network. After a message is sent, the network chooses another message from the requested messages. The messages are sent through the network one by one. As messages in this protocol must have unique priorities, messages are deterministically communicated. Furthermore, we assume that all CAN nodes implement a priority-based buffer. This simplifies the network model which can be represented by a single global priority-based queue [10]. To implement this protocol, a unique priority must be assigned to each message and

```
reactiveclass  HeaterWithSensor
{
   knownrebecs{
      Controller  controller ;
   }
   statevars{
      bool status;
      int  tempr;
   }

   msgsrv  initial (int  t){
      tempr = t;
      status  = false;
      self .update();
      self .sampleTemp();
   }

   msgsrv change(bool s){
      status  = s;
   }

   msgsrv sampleTemp(){
      controller . control(tempr)
         after  2;
      self .sampleTemp()
         after  3;
   }

   msgsrv update(){
     if  (status)
        tempr=tempr+1;
     else  tempr=tempr−1;
     delay(1);
     self .update();
   }
}

reactiveclass  Controller
{
   knownrebecs {
      HeaterWithSensor hws;
   }
   statevars  {}
   msgsrv  initial (){}
   msgsrv control(int  tpr){
      if (tpr  >= 22)
        hws.change(false)
          after  2;
      if (tpr  <= 18)
        hws.change(true)
          after  2;
   }
}

main
{
   HeaterWithSensor hws
      ( controller ):(20);
   Controller  controller
      (hws):();
}
```

Fig. 3 A model of a room with a heater, sensor and controller, specified in timed Rebeca

the communication delay between each two communicating actors must be specified. These specifications can be defined outside of the actors so that actors become agnostic about the underlying network of the model. This will also make the model more modular, since it is easier to change the network of the system without modifying the actors. Not all the actors communicate through CAN. Some of the actors may be connected by wire and have direct communication with each other. In our language, both types of communication are considered, and actors can communicate with each other either via wire or CAN. All messages, irrespective of the communication medium, are eventually inserted to their receiver's message queue. If two or more simultaneous messages (from wire or CAN) be inserted into a message queue, the resulting ordering will be nondeterministic. Note that there can not be two simultaneous messages from CAN, since CAN is a serial bus. The resulting hybrid Rebeca model is illustrated in Fig. 4.

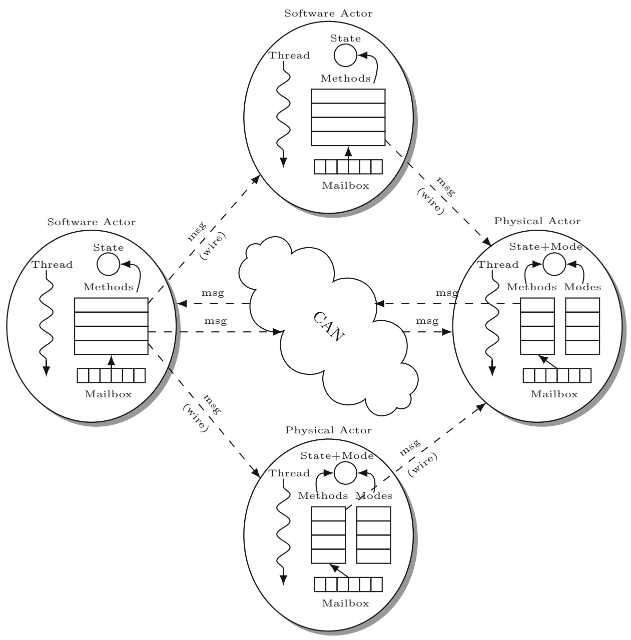


Fig. 4 Hybrid Rebeca model: each actor has its own thread of control, message queue, and ID. In addition to these, physical actors have modes that are defined by a guard, an invariant, and actions. Actors can communicate with each other either by sending messages through CAN or directly by wire

4 Hybrid Rebeca

Hybrid Rebeca is an extension of timed Rebeca to support physical behaviors. As we focus on the core functionality of hybrid Rebeca, we only consider the *delay* statement. Furthermore, we support connections with nonzero delays through CAN bus. So *after* statement is not needed anymore. Similar to timed Rebeca models, each rebec owns a local clock in hybrid Rebeca which can be considered synchronized with other rebec clocks. We do not support *deadline*, but it can be handled similar to the delay statement.

In Hybrid Rebeca we have two types of rebecs: software and physical. These rebecs communicate through asynchronous message passing. Each rebec has a queue for messages, and handles the message at the head of the queue by executing the corresponding message server for that message. Software rebecs are for modeling software (discrete) behaviors. These rebecs are reactive and self-contained, and they can have multiple message servers. Physical rebecs are for modeling physical (continuous) behaviors and the physical behaviors are defined by their modes. For physical rebecs a reserved message server is defined for changing the rebec's active mode.

Hybrid Rebeca has the concept of class for rebecs, and rebecs of the model are instantiated from these classes in the main block. In the instantiation phase the connection type of the rebecs with each other must be defined. For now our language supports only two types of connection: CAN and wire. When rebecs communicate through wire, the communication delay is considered to be zero. In contrast, CAN passes messages regarding their priorities by imposing delays depending on the communicating rebecs and messages. After instantiating rebecs, the CAN specification must be defined by expressing the delays and priority of messages.

4.1 Syntax

A hybrid Rebeca model definition consists of a set of *classes* and a *main* block, where classes define different types of rebecs and *main* specifies the initial configuration and CAN specification.

The syntax of hybrid Rebeca is presented in Fig. 5. The syntax of a software class is similar to a reactive class in timed Rebeca, which resembles a class definition in Java. A software class consists of a set of known rebecs, state variables and message servers. The known rebecs are the rebecs that an instance of this class can send message to. The syntax of message servers is like a method in object-oriented languages, except that they have no return value.

The core statements of our language are variable/mode assignment, conditional, delay, and method calls. An actor can send a message asynchronously to other rebecs through method calls.

A physical class is similar to a software class except that it also contains the definition for physical modes. The structure of a mode resembles a location in hybrid automata, and it consists of invariant, flows, guard and a block statement. The comparison expression of an invariant and a guard expression are specified by the reserved words inv and guard, respectively. The block statement following the guard expression, called *trigger* of the mode, defines the behavior of the rebec upon leaving the corresponding mode. We remark that the next entering mode is either explicitly defined by the user through the statement setmode in the statement block or the default mode none if it was not specified. Mode none is a special mode defined in all physical rebecs. This mode represents an idle behavior and its flows are defined as zero. Activating this mode can be interpreted as stopping the physical behavior of a physical rebec. Other rebecs can change the mode of a physical rebec by sending the message *setMode*.

Three primitive data types are available in Hybrid Rebeca: *int*, *real*, and *float*. Variables of types *int* are only allowed in software classes, variables of type *int* and *real* are only allowed in physical classes. Variables of type *float* can be used in both types of classes. Mathematically the float and real values are the same. However, to each real variable, a flow is assigned which determines how its value evolves over time. A float variable can be used to capture the value of a real variable in different snapshots. This can be used in communication with software rebecs. The value of a float variable can be changed only by assignment, but the value of a real variable can be changed by both assignment and the flow defined on the variable. The assignment of a real value to a float is managed implicitly in the semantics and no explicit casting is needed.

Every class definition must have at least one message server, named *initial*. In the initial state of the model, an *initial* message is put in all rebecs's message queue. The state variables and behavior of rebecs are initialized by executing this message server. The keyword self is used for sending a message to the rebec itself.

Rebecs are instantiated in the *main* block of the model. To instantiate a rebec, its known rebecs must be specified to be bound to the appropriate instances. Furthermore, for each known rebec, the connection type must also be specified, which can either be CAN or Wire. For example, by the statement $A \;\; a \;\; (@\text{Wire } b, \;\; @\text{CAN } c) : ()$, a rebec named a is instantiated from the class A that its known rebecs are b and c while the communication from a to b is through wire and a to c through CAN. We remark that the connection type between two rebecs can be different for each communication direction. The pair of parenthesizes () after the colon represents the parameters of the *initial* message server (which is empty in this case). After instantiation, the *CAN* specification is defined on the messages that may be transmitted through CAN. This specification consists of two parts. First the *priorities* of these messages must be specified. To this aim, a unique priority is assigned to a message. For example the statement $a \;\; b.m \;\; 1$; means that a message sent from rebec a to rebec b containing the message server name m has a priority of 1. A lesser number indicates a higher priority. After the priorities, the network *delays* of CAN communications are specified. For instance the statement $a \;\; b.m$ 0.01; expresses that the communication delay of sending a message from a to b containing the message server name m is 0.01.

4.1.1 Well-formedness

An hybrid Rebeca model is *well-formed* if all classes, rebecs and all variables, modes, messages and message parameters inside a rebec have unique names. All send statements are specified on defined message servers where actual parameters conform the formal parameters in both length and type compatibility. In instantiation, the list of actual known rebecs and actual parameters conforms to the list of formal known rebecs and formal parameters in both length and type. All expressions are well-typed and the conditions in conditional

```
        Model ::=  ⟨SClass | PClass⟩+ Main
         Main ::=  main {InstanceDcl* CANSpec?}
  InstanceDcl ::=  C r (⟨ @CAN | @Wire  r⟩*) : (⟨c⟩*)
      CANSpec ::=   CAN {Priorities Delays}
   Priorities ::=   priorities {⟨r r.m c;⟩*}
       Delays ::=   delays {⟨r r.m c;⟩*}
       SClass ::=  softwareclass C {KnownRebecs Vars MsgSrv*}
       PClass ::=  physicalclass C {KnownRebecs Vars MsgSrv* Mode*}
   KnowRebecs ::=  knownrebecs {VarDcl*}
         Vars ::= statevars {VarDcl*}
       VarDcl ::= T ⟨v⟩+;
      MesgSrv ::= msgsrv m (⟨T v⟩*) {Stmt*}
         Mode ::=  mode m {inv(e) {⟨v′ = e;⟩+} guard(e) {Stmt*}}
         Stmt ::=  v = Expr; | Call; | if(Expr) MSt [else MSt] | delay(Expr); | setmode(m);
         Call ::=  r.m(⟨Expr⟩*) |  r.setMode(m)
          MSt ::=  {Stmt*} | Stmt
         Expr ::=  c | v | Expr op Expr, op ∈ {+, −, ∗, ∧, ∨, <, ≤, >, ≥} | (Expr) | !(Expr)
```

Fig. 5 Abstract syntax of Hybrid Rebeca. The main differences in syntax compared to Timed Rebeca, are highlighted with color green. Identifiers C, T, m, $\mathfrak{m}$, v, c, and r respectively denote class, type, method name, mode name, variable, constant, and rebec name (color figure online)

```
physicalclass  HeaterWithSensor
{
   knownrebecs{
      Controller  controller ;
   }
   statevars{
       real  tempr;
       real  timer;
   }

  msgsrv  initial ( float  t){
       tempr  =  t;
       timer  =  0;
       setmode(Off);
   }

  mode  On{
    inv(timer<=0.05){
      timer'=  1;
      tempr'=4−0.1∗tempr;
    }
    guard(timer==0.05){
       timer  =  0;
       controller . control(tempr);
     }
   }

   mode  Off{
     inv(timer<=0.05){
       timer'=  1;
       tempr'=−0.1∗tempr;
     }
    guard(timer==0.05){
       timer  =  0;
       controller . control(tempr);
     }
   }
}

softwareclass  Controller
{
   knownrebecs  {
     HeaterWithSensor  hws;
   }
   statevars  {}

   msgsrv  initial (){  }

   msgsrv  control( float  tpr)
   {
    if (tpr  >=  22)
        hws.SetMode(Off);
    if (tpr  <=  18)
        hws.SetMode(On);
   }
}

main
{
   HeaterWithSensor  hws
     (@Wire  controller ):(20);
   Controller  controller
     (@Wire  hws):();
}
```

Fig. 6 A model of a room with a heater, sensor and controller, specified in Hybrid Rebeca

statements, invariants and guards are of type Boolean. The flows in each mode are defined in the from of $v' = e$ where v is a real variable and e is an expression and at most one flow is defined for each real variable within a mode.

Example 3 We revise the model given in Example 2 for a room with a heater, sensor and controller. We replace the discrete variable tempr with a real one in the class *Heater-WithSensor*. We consider two modes for the actor (instead of the boolean variable status). The stepwise changes of temperature is specified by two dynamic differential equations for each mode in Hybrid Rebeca as shown in Fig. 6. In each mode, a control message is sent to *Controller* periodically each 0.05 s by using a local variable timer which is updated with the rate of 1. In this model the controller and sensor are connected through a wire, and hence any message communicated between these two actors will be immediately delivered.

4.2 Operational semantics

To formally define the semantics of our language, we first derive an abstract model, called *Hybrid Rebeca model*, from the specification model specified by the Hybrid Rebeca language. To this aim, we represent statements with abstract notations. Then, we define the semantics of our language using the abstract Hybrid Rebeca model and abstract notions.

4.2.1 Notations and auxiliary functions

Given a set $\mathbb{A}$, the set $\mathbb{A}^*$ is the set of all finite sequences over elements of $\mathbb{A}$. For a sequence $a \in \mathbb{A}^*$ of length n, the symbol a_i denotes the ith element of the sequence, where $1 \leq i \leq n$. The empty sequence is denoted by ϵ and $\langle h|T \rangle$ represents a sequence whose first element is $h \in \mathbb{A}$ and $T \in \mathbb{A}^*$ is the sequence of remaining elements. For two sequences a and a' over $\mathbb{A}$, $a \oplus a'$ is the sequence obtained by appending a' to the end of a. For a function $f : X \to Y$, we use the notation $f[\mathfrak{x} \mapsto \mathfrak{y}]$ to denote updating the function f by mapping $\mathfrak{x}$ to $\mathfrak{y}$, where $\mathfrak{x} \in X$, and $\mathfrak{y} \in Y$.

A record type Record(T) is defined by $(\text{name}_1 : T_1, \ldots, \text{name}_n : T_n)$ where $T = T_1 \times \ldots \times T_n$. For each record $\mathfrak{t} = (e_1, e_2, \ldots, e_n)$ of type Record(T), where $e_i \in T_i$, we use $\mathfrak{t}.\text{name}_i$ to denote the i^{th}-element of the tuple called name$_i$, i.e., e_i. Furthermore, we use $\mathfrak{t}[e'/\text{name}_i]$ as symbolic representation for the tuple achieved by replacing the element $\mathfrak{t}.\text{name}_i$ of $\mathfrak{t}$ by e' in the record $\mathfrak{t}$:

$$\mathfrak{t}[e'/\text{name}_i].\text{name}_j = \mathfrak{t}.\text{name}_j, \quad \text{if} \quad i \neq j$$
$$\mathfrak{t}[e'/\text{name}_i].\text{name}_i = e'$$

We assume that each rebec has a unique identifier and *Id* is the set of identifiers, ranged over by x and y. Let Name be the set of all valid names for variables, message servers, and modes, *Stmt* be the set of statements, and *Expr* be the set of expressions. For readability, the conditional and send statements are represented abstractly by the notations if(e, σ, σ') and (x, m) in *Stmt*, respectively. A conditional statement with no *else* part is represented as if(e, σ, ϵ).

Each message server is abstractly denoted by the tuple $(m, b) \in \text{Name} \times \text{Stmt}^*$ where m is the name of the message server and b is the body of the message server which is a sequence of statements. For simplicity we ignore the message server parameters here. Each mode $\mathfrak{M}$ is defined by the tuple $(\mathfrak{m}, i, f, g, a) \in \text{Name} \times \text{Expr} \times \text{Expr} \times \text{Expr} \times \mathit{Stmt}^*$ where $\mathfrak{m}$ is the name of the mode, i, f, g and a are respectively invariant, flows, guard and trigger of the mode. We use the notations $\mathit{invariant}(r_p, \mathfrak{M})$, $\mathit{flows}(r_p, \mathfrak{M})$, $\mathit{guard}(r_p, \mathfrak{M})$ and $\mathit{trigger}(r_p, \mathfrak{M})$ to respectively denote the invariant, flows, guard and trigger of the mode $\mathfrak{M}$ in the physical rebec r_p. Let *Msg* denote the set of messages communicated among rebecs. Each message $M \in \mathit{Msg}$ is a triple $(\text{sender}, m, \text{receiver}) \in \mathit{Id} \times \text{Name} \times \mathit{Id}$ where sender is the sending rebec, m is the name of a message server of the receiver, and receiver is the receiving rebec.

4.2.2 Hybrid Rebeca model

A hybrid Rebeca model consists of the rebecs of the model and the network specification. A software rebec consists of the definitions of its variables, message servers and known rebecs. A physical rebec is defined like a software rebec plus the definitions of its modes. The network specification consists of the communication types of rebecs, which can be either CAN or wire, the message priorities and message delivery delays.

Definition 2 (*Hybrid Rebeca model*) A Hybrid Rebeca model is defined by the tuple (R_s, R_p, N) where R_s and R_p are the set of software and physical rebecs in the model, respectively, and N is the network specification. The set $R = R_s \cup R_p$ denotes the set of all rebecs in the model.

A software rebec $r_s \in R_s$ and physical rebec $r_p \in R_p$ with a unique identifier i, are defined by tuples $(i, V_i, \mathit{msgsrvs}_i, K_i)$ and $(i, V_i, \mathit{msgsrvs}_i, \mathit{modes}_i, K_i)$, respectively, where V_i is the set of its variables, $\mathit{msgsrvs}_i$ is the set of its message servers, $K_i \subseteq R$ is the set of its known rebecs, and modes_i is the set of modes.

A network specification is defined as a tuple $N = (\text{conn}, \text{netPriority}, \text{netDelay})$ where conn is a partial function $\mathit{Id} \times \mathit{Id} \to \{\text{Wire}, \text{CAN}\}$ which defines the one way connection type from a rebec to another rebec, $\text{netPriority} : \text{Msg} \to \mathbb{N}$ and $\text{netDelay} : \text{Msg} \to \mathbb{R}$ define the priority and the network delivery delay for a message, respectively. A lower value indicates a higher priority.

We use the notation $\mathit{body}(y, M)$ to express the body of the corresponding message server for message M defined for the rebec r with the identifier y, which is a sequence of statements:

$$\mathit{body}(y, M) = b, \quad \text{where} \quad M = (x, m, y) \wedge r = (y, V, \mathit{msgsrvs}, K) \wedge (m, b) \in \mathit{msgsrvs}.$$

4.2.3 Formal semantics of hybrid Rebeca models

Rebecs respond to expiration of a physical mode or taking a message from their message queues. A physical mode expires when its guard holds, then the trigger of the physical mode is executed. Upon taking a message, the rebec processes it by executing its corresponding message server. The execution of all the statements except the delay statement is instantaneous. To model communication via CAN, a network entity is considered in the semantics which buffers the messages from the rebecs and delivers them one-by-one to the respective receivers based on the messages' priorities and delays specified in the model. For communication via wire, the message is directly inserted into the receiver's message queue instantaneously. The operational semantics of a Hybrid Rebeca model is defined as a monolithic hybrid automaton.

Definition 3 (*Hybrid automaton for a Hybrid Rebeca model*) Given a Hybrid Rebeca model $\mathfrak{H} = (R_s, R_p, N)$, its formal semantics based on hybrid automata is defined by $H_{\mathfrak{H}} = (Loc, Var, Lab, \Rightarrow, Flws, Inv, Init)$, where *Var* is the set of all continuous variables in the model (variables of types *float* or *real*), *Lab* is the set of labels which is empty as we generate a monolithic hybrid automaton. The set of locations *Loc*, transitions $\Rightarrow$, flows *Flws*, invariants *Inv*, and initial conditions *Init* are defined in the following.

Locations

Each location is defined by four entities, denoted by the record $(ss : SS, ps : PS, ns : NS, es : ES)$. The entity *ss* defines the states of software rebecs by mapping a given software rebec with the identifier x to its state. Similarly *ps* maps a physical rebec with the identifier x to its state. The third entity *ns*, defines the network state and the forth entity *es* represents the sequence of pending events. We define each entity in the following.

Definition 4 (**State of a rebec**) The state of a software rebec is denoted by the tuple (v, q, σ) where v is the valuation of its discrete variables, $q \in \text{Msg}^*$ is the message queue of the rebec, and $\sigma \in Stmt^*$ denotes the sequence of statements that the rebec is going to execute. The state of a physical rebec is a tuple of the form $(\mathfrak{M}, q, \sigma)$ where $\mathfrak{M}$ is the active mode and q and σ are defined as in the software rebec's state.

We remark that a software rebec has the notion of being suspended (due to the execution of a delay statement). The suspension status is maintained by a reserved variable in the valuation of the rebec. As delay statements are not allowed in physical rebecs, they do not have such a reserved variable. The state of a physical rebec does not contain any valuation since discrete variables are not defined for physical rebecs and the continuous variables are handled in the hybrid automaton.

The network state, which is the state of the CAN network, consists of the buffered messages in the network and the status of the network which indicates that the network is busy by sending a message or is ready to send one.

Definition 5 (*State of the network*) The network state is defined by the pair (b, r), where $b \in \text{Msg}^*$ is the network buffer and the boolean flag r indicates the status of the network, which can be ready or busy.

Events are used for time consuming actions: executing delay statements or transferring messages via a CAN bus. Upon executing a time consuming action, an event is stored in *es* to be triggered at the time that the delay of the action is over. Two types of events are defined in the semantics of Hybrid Rebeca: *Resume* and *Transfer*. Let Event denote the set of all events, specified by $\{Transfer(x, M), Resume(x) \mid x \in Id,\ M \in \text{Msg}\}$. A pending event with event *Resume*, parameterized by a rebec identifier, is generated and inserted into the pending event list when a delay statement is executed in the Rebeca model, and the corresponding rebec is suspended. To model the passage of time for the delay statement, a timer is assigned to the pending event. After the specified delay has passed, the event is triggered, and consequently the behavior of the given rebec is resumed by updating the suspension status of the rebec. The pending event $Transfer(x, M)$ is generated when a message from the network buffer is chosen to be delivered to its receiver. A timer is assigned to model the message delivery delay, and the pending event is inserted into the pending event list. Upon triggering of a *Transfer* event, the specified message is enqueued in the receiver's message queue, and the network status is set to ready which means the network is ready to send another message. The effect of each event trigger on a given location is formally defined by a function $effect : \text{Event} \times Loc \rightarrow Loc$ which is presented in Table 1.

Definition 6 (*Pending event*) A pending event is a tuple (d, e, t) where d is the delay of the event e and t is a timer that is assigned to this event. The event e can either be a *Resume* or *Transfer* event. The timer is a real variable used for defining the timing behavior for the delay of the pending event. The event is triggered (and executed) after d units of time after the pending event is created.

Table 1 Formal definition of the function *effect*

Resume	$\dfrac{l = (ss, ps, ns, es) \wedge ss(x_s) = (v, q, \sigma)}{effect(Resume(x_s), l) = l[ss[x_s \mapsto (v[suspended \mapsto false], q, \sigma)]/SS]}$
Transfer(Software)	$\dfrac{l = (ss, ps, ns, es) \wedge ss(x) = (v, q, \sigma)\ \wedge\ ns = (b, false)}{effect(Transfer(x, M), l) = l[ss[x \mapsto (v, q \oplus M, \sigma)]/SS, (b, true)/NS]}$
Transfer(Physical)	$\dfrac{l = (ss, ps, ns, es) \wedge ps(x) = (\mathfrak{M}, q, \sigma)\ \wedge\ ns = (b, false)}{effect(Transfer(x, M), l) = l[ps[x \mapsto (\mathfrak{M}, q \oplus M, \sigma)]/PS, (b, true)/NS]}$

Transitions

We define two general types of transitions: urgent and non-urgent transitions. An urgent transition must be taken immediately upon entering its source location. We further divide urgent transitions into message, statement and network transitions which are respectively shown as $\Rightarrow_m$, $\Rightarrow_s$ and $\Rightarrow_n$. The non-urgent transitions are shown as $\Rightarrow_N$. These transitions indicate the passage of time. We use these types of transitions to differentiate between different types of actions. Message transitions are only for taking a message. Statement transitions are for executing the statements. A network transition chooses a message from the buffer of the network to be sent.Non-urgent transitions are used to model the passage of time. These transitions include the behaviors of physical rebecs' active modes and pending time of events since they are time consuming.

The ordering $\Rightarrow_m = \Rightarrow_s > \Rightarrow_n > \Rightarrow_N$ is considered among the transitions. Whenever a higher-order transition is enabled in a location, no lower order transition can be taken in that location. As explained in [10], a CAN bus is modeled by a single global priority-based queue. To deterministically deliver messages to their recipients according to their priorities, all messages inserted into CAN at the same time should be defined first and then the highest priority message among them is selected. For this aim, we consider the effect of the network entity in the operational semantics when no rebecs can progress instantaneously, resulting $\Rightarrow_m = \Rightarrow_s > \Rightarrow_n$. The semantics of actions in hybrid Rebeca are defined using these transitions in the locations. In the following, we define these transitions.

Message transitions Message transitions define the act of taking a message. A rebec takes a message from the head of its message queue, whenever the rebec has no statement to execute. For a software rebec, the rebec should not be also suspended. A rebec is suspended when it executes a delay statement. When a message is taken, the body of its corresponding message server is added to the execution queue of the rebec, and the message is removed from the message queue.

$$\frac{l = (ss, ps, ns, es) \wedge ss(x) = (v, \langle M|q\rangle, \epsilon) \wedge \neg v(suspended)}{l \overset{\tau}{\Rightarrow}_m l[ss[x \mapsto (v, q, \text{body}(x, M))]/SS]}$$

$$\frac{l = (ss, ps, ns, es) \wedge ps(x) = (\mathfrak{M}, \langle M|q\rangle, \epsilon)}{l \overset{\tau}{\Rightarrow}_m l[ps[x \mapsto (\mathfrak{M}, q, \text{body}(x, M))]/PS]}$$

Statement transitions Statement transitions define the act of executing the statements. Like message transitions, a statement transition can take place by a software rebec whenever the rebec is not suspended. Consider the tuples (v, q, σ) and $(\mathfrak{M}, q, \sigma)$ as the states of a software rebec and a physical rebec respectively. The statement transitions include the followings:

– *Assignment statement* This statement has two rules. When assigning to a discrete variable, the value of the variable is updated in the valuation of the rebec. When assigning to a continuous variable, since its value is not determined (it may depend on the continuous behaviors), the assignment is transferred over to the transition to be handled by the resulting hybrid automaton.

$$\frac{l = (ss, ps, ns, es) \wedge ss(x) = (v, q, \langle \text{dvar} = e|\sigma\rangle) \wedge \neg v(suspended)}{l \overset{\tau}{\Rightarrow}_s l[ss[x \mapsto (v[\text{dvar} \mapsto \text{eval}(e, v)], q, \sigma)]/SS]}$$

$$\frac{l = (ss, ps, ns, es) \wedge ps(x) = (\mathfrak{M}, q, \langle \text{cvar} = e|\sigma\rangle)}{l \xRightarrow{\text{cvar:=eval}(e,v)}_s l[ps[x \mapsto (\mathfrak{M}, q, \sigma)]/PS]}$$

– *Conditional statement* This statement has three rules. If the value of the condition is evaluated to true, the statements of the true block are added to the tail of the execution queue. Similarly, if the value of the condition is evaluated to false, the statements of the false block are used. If the value of the condition is not determined (because of continuous variables used in the condition), both possible paths are considered by creating two separate transitions. The condition and its negation act as the guards for these transitions.

$$\frac{\begin{array}{c} l = (ss, ps, ns, es) \wedge ss(x) = (v, q, \langle \text{if}(e, \sigma, \sigma')|\sigma''\rangle) \\ \wedge eval(e, v) = true \wedge \neg v(suspended) \end{array}}{l \overset{\tau}{\Rightarrow}_s l[ss[x \mapsto (v, q, \sigma \oplus \sigma'')]/SS]}$$

$$\frac{\begin{array}{c} l = (ss, ps, ns, es) \wedge ss(x) = (v, q, \langle \text{if}(e, \sigma, \sigma')|\sigma''\rangle) \\ \wedge eval(e, v) = false \wedge \neg v(suspended) \end{array}}{l \overset{\tau}{\Rightarrow}_s l[ss[x \mapsto (v, q, \sigma' \oplus \sigma'')]/SS]}$$

$$\frac{l = (ss, ps, ns, es) \wedge ps(x) = (\mathfrak{M}, q, \langle \text{if}(e, \sigma, \sigma')|\sigma''\rangle)}{l \overset{e}{\Rightarrow}_s l[ps[x \mapsto (\mathfrak{M}, q, \sigma \oplus \sigma'')]/PS]}$$
$$l \overset{!e}{\Rightarrow}_s l[ps[x \mapsto (\mathfrak{M}, q, \sigma' \oplus \sigma'')]/PS]$$

– *Send statement* This statement, depending on the communication type, has two rules. When the communication is via wire, the message is directly added to the receiver's message queue. When the communication is via CAN, the message is added to the CAN buffer to be handled by the CAN behavior. The same rules hold for physical rebecs with the difference that the condition $\neg v(suspended)$ is not needed.

$$\frac{\begin{array}{c} l = (ss, ps, ns, es) \wedge ss(x) = (v_x, q_x, \langle (y, m)|\sigma_x\rangle) \\ \wedge \neg v_x(suspended) \wedge \\ ss(y) = (v_y, q_y, \sigma_y) \wedge \text{conn}(x, y) = Wire \end{array}}{l \overset{\tau}{\Rightarrow}_s l[ss[x \mapsto (v_x, q_x, \sigma_x), y \mapsto (v_y, q_y \oplus (x, m, y), \sigma_y)]/SS]}$$

$$\frac{\begin{array}{c} l = (ss, ps, ns, es) \wedge ss(x) = (v, q, \langle (y, m)|\sigma\rangle) \wedge \\ ns = (b, r) \wedge conn(x,y)=CAN \wedge \neg v(suspended) \end{array}}{l \overset{\tau}{\Rightarrow}_s l[ss[x \mapsto (v, q, \sigma)]/SS, (b \oplus (x, m, y), r)/NS]}$$

– *Delay statement* This statement suspends the software rebec by updating the corresponding variable (suspension status) in the valuation of the rebec and creates a pending event $(d, Resume, t)$ for resuming the rebec after d units of time. The value d is the delay specified in the delay statement and t is a fresh timer acquired from the pool of timers by calling acquire_timer().

$$\frac{l = (ss, ps, ns, es) \wedge ss(x) = (v, q, \langle delay(d)|\sigma\rangle) \wedge \neg v(suspended) \wedge t = \text{acquire\_timer}()}{l \overset{\tau}{\Rightarrow}_s l[ss[x \mapsto (v[\text{suspended} \mapsto \text{true}], q, \sigma)]/SS, es \oplus (d, Resume(x), t)/ES]}$$

– *Set mode statement* This statement changes the active mode $\mathfrak{M}$ of the physical rebec to the specified mode.

$$\frac{l = (ss, ps, ns, es) \wedge ps(x) = (\mathfrak{M}, q, \langle setmode(\mathfrak{M}')|\alpha\rangle)}{l \overset{\tau}{\Rightarrow}_s l[ps[x \mapsto (\mathfrak{M}', q, \sigma)]/PS]}$$

Network transitions These transitions define the behavior of the CAN network which only includes the behavior of choosing a message from the network buffer to be sent. Since network transitions have a lower priority than message and statement transitions, this makes the choosing behavior to happen only when no rebec can progress instantaneously. Let (b, r) be the network state. For this behavior, the message with the highest priority is removed from the network buffer b by calling highest_priority(b), a pending event of *Transfer* type with the delay d for sending the message is created, and the flag r of the network is updated to indicate that the network is busy. The delay d for the created pending event is the network delay for the message determined in the network specification.

$$\frac{\begin{array}{c} l = (ss, ps, ns, es) \wedge ns = (b, true) \wedge b \\ = b_1 \oplus (x, m, y) \oplus b_2 \wedge l \not\Rightarrow_m \wedge l \not\Rightarrow_s \wedge \\ (x, m, y) = \text{highest\_priority}(b) \wedge d = \text{netDelay}(x, m, y) \\ \wedge t = \text{acquire\_timer}() \end{array}}{l \overset{\tau}{\Rightarrow}_n l[(b_1 \oplus b_2, \text{false})/NS, es \oplus (d, Transfer(y, (x, m, y)), t)/ES]}$$

Non-urgent transitions Non-urgent transitions are used to define the end of active physical modes and triggering pending events. These transitions are defined only when no urgent transition is possible. These transitions are as follows:

– *End of an active mode* For a physical rebec $(\mathfrak{M}, q, \sigma)$ if $\mathfrak{M}$ is not none, the guard of the active mode $\mathfrak{M}$ is transferred to the transition, and the trigger of the active mode is added to the execution queue, and the active mode is set to none.

$$\frac{l = (ss, ps, ns, es) \wedge ps(x) = (\mathfrak{M}, \epsilon, \epsilon) \wedge \mathfrak{M} \neq \text{none} \wedge l \not\Rightarrow_m \wedge l \not\Rightarrow_s \wedge l \not\Rightarrow_n}{l \xRightarrow{guard(x,\mathfrak{M})}_N l[ps[x \mapsto (\text{none}, \epsilon, trigger(x, \mathfrak{M}))]/PS]}$$

– *Triggering of an event* For a pending event (d, e, t), the guard $t == d$ is defined on the transition where t and d are the timer and the delay of the pending event respectively. The event e is executed as a result of this transition and the pending event is removed from the pending event list. The effects of the execution of an event is defined by *effect* function given in Table 1.

$$\frac{l = (ss, ps, ns, es) \wedge es = es_1 \oplus (d, e, t) \oplus es_2 \wedge l \not\Rightarrow_m \wedge l \not\Rightarrow_s \wedge l \not\Rightarrow_n}{l \xRightarrow{t==d}_N effect(e, l[es_1 \oplus es_2/ES])}$$

Example 4 Consider the model of a heater, sensor, and controller, given in Fig. 6. Assume the resulting location after the rebecs have handled their initial messages. In this location, called ℓ_1, the message queues of hws and controller are empty and they have no statement to execute. As controller has no discrete variable (except suspended) and no delay statement within its message handler, we abstractly represent its valuation by an empty set. The rebec hws is in the mode Off. We represent the local state of controller as $c : (\emptyset, \epsilon, \epsilon)$ and of hws as $h : (\text{Off}, \epsilon, \epsilon)$. As there is no message to handle and statement to execute, no urgent transition is derived. Only one non-urgent transition is derived by using the rule *End of an Active Mode*:

$$\frac{\ell_1 = (c : (\emptyset, \epsilon, \epsilon), h : (\text{Off}, \epsilon, \epsilon)) \wedge ps(h) = (\text{Off}, \epsilon, \epsilon) \wedge \ell_1 \not\Rightarrow_m \wedge \ell_1 \not\Rightarrow_s, \wedge \ell_1 \not\Rightarrow_n}{\ell_1 \xRightarrow{\text{timer}==0.05}_N (c : (\emptyset, \epsilon, \epsilon), h : (\text{none}, \epsilon, \langle \text{timer} = 0\,(\text{controller}, \text{control})\rangle))}$$

A number of reachable locations from ℓ_1 are shown in Fig. 7. As rebecs are connected via wire and no delay statement exists in the model, for the sake of brevity, we have removed the local state of network and event list from the locations. In the resulting location ℓ_2, an urgent transition is derived as the consequence of assignment leading to the location ℓ_3. Next, an urgent transition is generated as the consequence of a send statement; the message control is inserted into the queue of controller. We will explain the flow and invariants of the location in Example 5.

We remark that time is advanced implicitly by the kernel of the generated hybrid automaton through the use of timers and their corresponding flows. In the following, we will assign flows to the timers which make timers advance when there

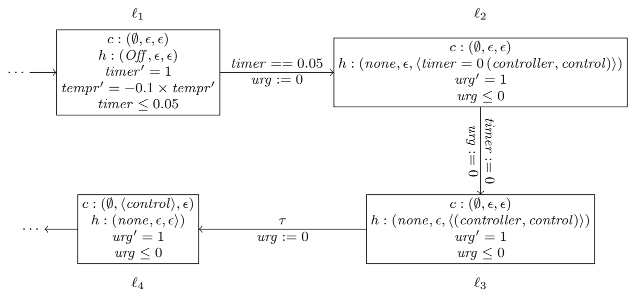

ℓ_1
$c : (\emptyset, \epsilon, \epsilon)$
$h : (Off, \epsilon, \epsilon)$
$timer' = 1$
$tempr' = -0.1 \times tempr'$
$timer \leq 0.05$
$timer == 0.05$
$urg := 0$
ℓ_2
$c : (\emptyset, \epsilon, \epsilon)$
$h : (none, \epsilon, \langle timer = 0\,(controller, control)\rangle)$
$urg' = 1$
$urg \leq 0$
$urg := 0$
$timer := 0$
ℓ_3
$c : (\emptyset, \epsilon, \epsilon)$
$h : (none, \epsilon, \langle (controller, control)\rangle)$
$urg' = 1$
$urg \leq 0$
τ
$urg := 0$
ℓ_4
$c : (\emptyset, \langle control\rangle, \epsilon)$
$h : (none, \epsilon, \epsilon\rangle)$
$urg' = 1$
$urg \leq 0$

Fig. 7 Partial hybrid automaton derived for the system of a controller, sensor, and heater

is no statement to be executed, no message to handle, and no message to transfer.

Flows and invariants

To define flows and invariants for each location, we need to consider continuous and instantaneous behaviors separately. There are two kinds of continuous behaviors in the model, behaviors regarding physical rebecs' modes and behaviors regarding pending events. Physical modes have all the necessary information in themselves and the pending events have simple timing behaviors. Instantaneous behaviors, i.e., urgent transitions, should be executed without allowing the time passage. So time should not be passed when the system resides in the source locations of such transitions, called *urgent* locations. Any location with an instantaneous behavior is considered as an urgent location.

Definition 7 (*Urgent location flow and invariant*) A possible implementation for an urgent location is $\mathrm{urg}' = 1$ as its flow and $\mathrm{urg} \leq 0$ as its invariant, where urg is a specific variable. Note that in this method, this new variable must be added to the set *Var* of the hybrid automaton. Also the assignment $\mathrm{urg} = 0$ must be added to all incoming transitions to an urgent location. The defined invariant prevents the model from staying in the location as the value of urg will be increased by the defined flow.

If a location is urgent, the urgency flow, as defined above, should be set as its flows. In case a location is not urgent, it inherits the flows of all physical rebecs' active modes, denoted by *ModeFlows*, the flows for timers of pending events, denoted by *EventFlows*, and a flow of zero for each float variable to freeze its value, denoted by *FloatFlows*. The flow of a pending event is simply defined as $t' = 1$ where t is the timer variable of the pending event. The *Flws* function of the hybrid automaton is defined as:

$$\begin{aligned}
&\mathit{Flws}(l)\\
&= \begin{cases} \mathit{urg}' = 1, & \text{if } l \text{ is urgent}\\ \mathit{ModeFlows}(l) \bigcup \mathit{EventFlows}(l) \bigcup \mathit{FloatFlows}(l), & \text{otherwise} \end{cases}
\end{aligned}$$

$$\begin{aligned}
\mathit{ModeFlows}(l) &= \bigcup_{x \in R_p} \mathit{flows}(x, \mathfrak{M}) \text{ where } l.PS(x) = (\mathfrak{M}, q, \sigma)\\
\mathit{EventFlows}(l) &= \bigcup_{(d,e,t) \in l.ES} t' = 1\\
\mathit{FloatFlows}(l) &= \bigcup_{v \in \text{float variables}} v' = 0
\end{aligned}$$

Similarly, if a location is urgent, its invariant is set to urgency invariant, otherwise it inherits the invariants of all physical rebecs' active modes, denoted by *ModeInvs* and the invariants for corresponding pending events' timers, denoted by *EventInvs*. The invariant of a pending event is defined as $t \leq d$ where t and d are the timer variable and the delay of the pending event, respectively. The *Inv* function of the hybrid automaton is defined as:

$$\begin{aligned}
\mathit{Inv}(l) &= \begin{cases} \mathit{urg} \leq 0, & \text{if } l \text{ is urgent}\\ \mathit{ModeInvs}(l) \wedge \mathit{EventInvs}(l), & \text{otherwise} \end{cases}\\
\mathit{ModeInvs}(l) &= \bigwedge_{x \in R_p} \mathit{invariant}(x, \mathfrak{M}) \text{ where } l.PS(x) = (\mathfrak{M}, q, \sigma)\\
\mathit{EventInvs}(l) &= \bigwedge_{(d,e,t) \in l.ES} t \leq d
\end{aligned}$$

Example 5 The location ℓ_1 in Fig. 7 is not urgent as it has no outgoing urgent transition. So the flow and invariant of the mode Off are added to this location. However, all the remaining locations ℓ_2, ℓ_3, and ℓ_4 are urgent. We make these locations urgent by using the specific variable urg as explained in Definition 7.

Initial location and initial condition

For the initial location l_0, we initialize all discrete variables of rebecs to the value zero. Furthermore, the initial message for each instantiated rebec, is put into its message queue. We also set the value of all continuous variables to zero in the initial condition of the initial location. The initial location l_0 and the function *Init* of the hybrid automaton are defined as below.

$$\begin{aligned}
&l_0 = (ss_0, ps_0, ns_0, es_0)\\
&ss_0(r_s) = (v_{r_0}, \langle (r_s, \text{initial}, r_s)\rangle, \epsilon)\\
&\text{where } v_{r_0}(x) = 0 \text{ and } x \text{ is a discrete variable of } r_s\\
&ps_0(r_p) = (\text{none}, \langle (r_p, \text{initial}, r_p)\rangle, \epsilon)\\
&ns_0 = (\epsilon, \mathit{true})\\
&es_0 = \epsilon\\
&\mathit{Init}(l) = \begin{cases} \bigcup_{v \in \text{continuous variables}} v = 0 & \text{if } \; l = l_0\\ \emptyset, & \text{otherwise} \end{cases}
\end{aligned}$$

4.3 Technical details

For simplicity, some details were omitted from our semantics. Here, we describe these details informally.

Limited size for message queues In the semantics of hybrid Rebeca, the message queues of rebecs are considered unbounded. To derive a hybrid automaton with a finite state space, a specific size must be specified for message queues of rebecs. We remark that a hybrid automaton can be derived only for those hybrid Rebeca models that generate a finite number of locations, which are the models with bounded queues and limited possible values for rebec variables.

Message arguments To incorporate message arguments, we must consider discrete and continuous arguments separately. For discrete arguments, since their values are known in the state, that value is included in the message and when the message is taken, its arguments are added to the rebec's valuation. When the execution of the message is finished, the arguments are removed from the valuation. For continuous arguments, the values are not generally determined so it is not possible to send the value within the message. To this aim, a non-evolving auxiliary variable is used. Before sending a message, each continuous argument is assigned to an auxiliary variable (by using continuous variable assignment). Then, a reference to the auxiliary variable is included in the message. When the message is taken, for each continuous argument, an assignment from the auxiliary variable to its respective parameter variable is implicitly executed.

Continuous variable pools When creating a new event (for a delay statement or for sending a message from CAN), a new timer is assigned to each event. But in hybrid automata all continuous variables must be defined statically. To handle this, variable pools with fixed sizes are used. There are two variable pools in our semantics, one for timer variables and one for the auxiliary variables of message arguments (as mentioned above). The size of the variable pools affects the behavior of the model. A small size will lead to an incomplete model, and a large size will lead to a huge model which can not be easily analyzed.

We consider a specific location, called Fault, for faulty situations like queue overflow, running out of the pooled variables, and having messages with the same priority in the CAN buffer. We assume the message with the highest priority must always be unique in CAN buffer. We generate a transition to this specified location Fault upon occurrences of the mentioned faults.

5 Compositional hybrid automata model and hybrid Rebeca

The semantics of our language is defined as a monolithic hybrid automaton. In this section, we define a compositional hybrid automata for a Hybrid Rebeca model. In the compositional semantics we provide a mapping from constituent elements of actor models to hybrid automata constructs and templates. We can consider these constructs and templates as a generic mechanism for customizing hybrid automata for modeling event-based asynchronous communicating distributed systems. We illustrate how working at the level of Hybrid Rebeca instead of hybrid automata increases modularity and improves analyzability; a Hybrid Rebeca model makes the modification and maintenance of a model easier, while its monolithic semantics results in a more feasible model for analysis. Throughout this section, we use the definition included in Definition 7 for the meaning of urgent locations.

5.1 Mapping

Due to the complexity of defining the algorithm for finding the highest priority message in the network buffer at the hybrid automata level, we only focus on a subset of Hybrid Rebeca language, namely specifications in which rebecs are connected by wire. To this aim, we only provide mapping for software rebecs as the structure of physical rebecs is similar to hybrid automata.

5.1.1 Software rebec

To define a software rebec $r_{s_i} = (i, V_i, msgsrvs_i, K_i)$, we break it to three hybrid automata namely, *MsgQueue*, *MsgServers* and *ConsVars*. The automaton *MsgQueue* is the message queue of the rebec. Other rebecs can only interact with this component. This component enqueues the received messages and executes the appropriate message server in *MsgServers* when it is ready. The automaton *MsgServers* contains the execution logic of all message servers of the rebec r_{s_i}. The automaton *ConsVars* is an auxiliary component for variables that do not evolve over time. As the only available variable type for hybrid automata is real, we consider a real variable for each variable of the type int and float. In the following we explain these components in more details.

MsgQueue: This component has two responsibilities, queuing the received messages and executing the corresponding message server of the head of the queue when *MsgServers* is ready. We use variables to represent queue elements. A queue element consists of a variable for message ID and some variables for parameters of messages. For simplicity, here we only focus on messages ID. The automaton *MsgQueue* has $C + 1$ queue elements where C is the capacity of the queue. The extra queue element is used as a temporary buffer. Other rebecs put their message ID in the buffer element and *MsgQueue* will put the buffered message ID into the appropriate element. This approach encapsulates the inner working of the queue from other rebecs. We use a circular queue algorithm to handle the message queue. This automaton has variables for the queue head, tail and size, and based on these variables, it decides in which element it should put the buffered message. This decision is implemented as branching transitions. The guard of each transition is a comparison of the tail variable with the indexes of the queue elements. In the assignment of each transition, the buffered message ID is assigned to the message ID of the corresponding element, the value of the tail variable is incremented and buffer flag is marked as empty. We remark that the queue state of *MsgQueue* could be unfolded into the locations similarly to the monolithic semantics, but this extremely complicates the mapping, as for all possible permutations of the queue, a location should be considered.

To start the execution of the head element, *MsgQueue* first checks whether the queue is not empty and *MsgServers* is idle. The idleness is checked by means of a shared variable between *MsgQueue* and *MsgServers*. To start the execution, we use a synchronization label between *MsgQueue* and *MsgServers*. Each message has a unique synchronization label and based on the message ID of the head element, a transition with the appropriate label becomes enabled. To this aim, the head of the queue is first determined based on the head variable by using branching transitions. Then each branch is further divided to more branches to determine the correct synchronization label based on the message ID of the chosen element. After synchronization, *MsgQueue* will update its queue state accordingly. The size of these branches is an order of the capacity of the queue times the number of message servers. The overall behavior of *MsgQueue* is implemented as a loop. After completing one of its behaviors, the component goes back to its initial location. All the locations except the initial location are urgent, so execution of each behavior happens instantaneously. The schematic of a *MsgQueue* with the queue size of 2 and two message servers is represented in Fig. 8.

MsgServers This component merely consists of the execution logic of all message servers. The overall behavior is to wait for a synchronization label from the *MsgQueue*. Then, it marks itself as "busy" and executes the specified message server statements. At the end of execution, it marks itself as "idle" and goes to the initial location again.

Since a message server is a sequence of statements, to translate a message server we convert its statements to a sequence of locations and transitions. Translating most statements to hybrid automata is trivial. An assignment statement is translated to one urgent location. The assignment itself will be over the outgoing transition of the location. A delay statement is translated to one location. The delay behavior is defined as a simple timing behavior on the location and its outgoing transition. A send statement is translated to one urgent location. In the outgoing transition we check the emptiness of the destination queue buffer (which is a shared variable) and if empty, we put the corresponding message ID in the buffer's message ID. Conditional statement can be regarded as a composite statement. It consists of two paths based on its condition and each path again can be translated by the mentioned translations recursively. An overview of these translations are presented in Fig. 9. The translation of two consecutive statements is achieved by merging the outgoing transition of the first statement with the incoming transition of the second statement.

ConsVars This component has only one location with no transition. For each variable in the rebec, a flow of zero is defined in the location to prevent its value from changing over time. Furthermore, a flow of zero is also defined for method parameters, queue variables and the variable for the readiness of the *MsgServers*. It should be noted that in our monolithic approach, we have generated a location for each valuation of discrete variables and the local queues of actors.

5.1.2 Model

A model $H = (R_s, \emptyset, N)$ is the parallel composition of the translation of the rebecs in R_s according to the mentioned rules. To translate the initial messages, we only need to initialize the queue variables of the corresponding rebecs with appropriate values. Note that the set of physical rebecs is empty and we have assumed only wire communication among rebecs in the network specification N.

5.2 Comparison

In this section, we first compare the monolithic semantics to the compositional approach in terms of modularity and then analyzability

Modularity Providing a formal measurement for modularity is beyond the scope of this paper. Here we adopt a simple measure for modularity, which is the amount of changes required in a model to accommodate with the changes in the design. We present a list of possible changes in a hybrid Rebeca model and give an informal measure of required changes in the defined compositional hybrid automata, from now on denoted by CHA.

Adding a new message server without parameter In CHA we add the translation of the message server to *MsgServers* with a fresh synchronization label for that message, and in the *MsgQueue* we must integrate the execution start of the message server. As explained before this is done by first determining the head element and then determining the corresponding synchronization label based on the message ID. The cost of this integration is an order of C (capacity of queue) because for each queue element we need to add the synchronization label.

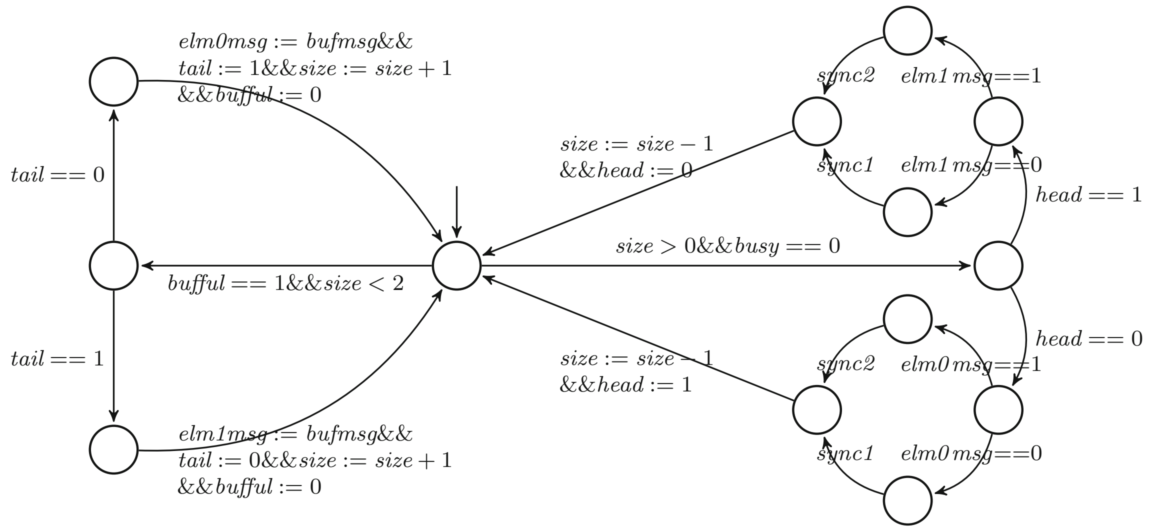


Fig. 8 Schematic for a *MsgQueue* with a capacity of 2 and two message servers. *head*, *tail* and *size* are the index of the head element, the index of tail element and the size of the circular queue, respectively. *sync1* and *sync2* are the corresponding synchronization labels for the two message servers. *buffull* is a flag which determines whether the buffer element is full or not. *elem0msg*, *elem1msg* and *bufmsg* are the id of the message stored in the first element, the second element and the buffer element, respectively

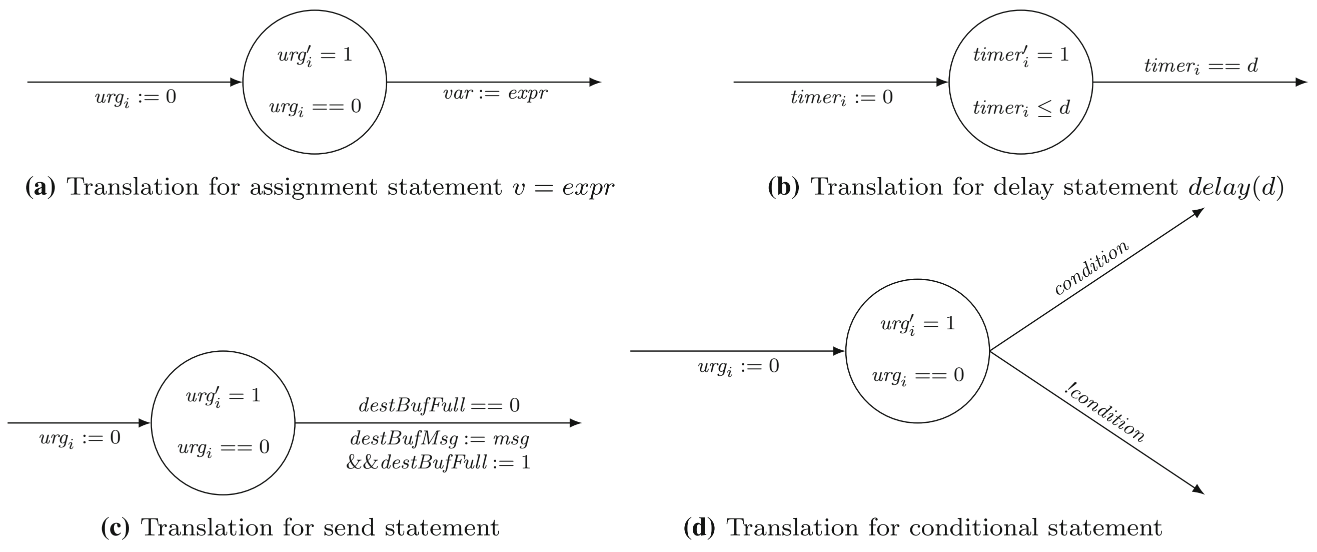


Fig. 9 An overview of statement translations for compositional hybrid automata

Adding a new statement to a message server To add a new statement we simply add the translation of that statement to the corresponding message server in the *MsgServers*.

Adding a new variable We introduce the new variable as a shared variable between *MsgServers* and the *ConsVars* while the *ConsVars* is modified by adding a flow of zero for the new variable.

Increasing rebec's queue size Since inner workings of the queue is encapsulated from other rebecs, we only need to change the *MsgQueue* of the rebec. We need to change two places for each added queue element. First we add one branch for processing the buffer element (queuing the buffer element in the tail element) and then we add a new branch for processing the head of the queue. Note that this branch must again be further divided for each message server. So the cost of this change is in order of the number of the message servers.

Changing the known rebecs bindings The binding is done by sharing the buffer element of the target rebec with the original rebec. To change a known rebec binding in a rebec, we only need to change this sharing.

Adding a new rebec Three automata, based on our mapping approach, will be added while the names of its variables are different from previous instances of the same class. However, in most tools, the concept of template allows to have several instances of an automaton. So in practice, adding a new rebec is trivial.

Table 2 Comparison summary for the number of continuous variables for a single rebec in the two approaches. Here dvar and cvar denote the number of the rebec's discrete variables and continuous variables, respectively

	MNO	CHA
Defined variables	cvar	dvar + cvar
Communication variables	$\leq C \times \text{param}_i^{\max}$	$\geq (C+1) \times (1 + \text{param}_i^{\max})$
Message execution variables	$\leq \text{param}_i^{\max}$	$\geq \text{param}_i^{\max}$
Other	1	6

Analyzability For analyzability, we focus on the number of continuous variables in models, since often there is an exponential relation between the number of variables and the analysis time of a model. Here, we compare the number of continuous variables in a software rebec i in our monolithic approach (MNO) compared to the CHA approach. In the MNO, only the defined continuous variables of the rebec are translated as continuous variables in the final hybrid automata, but in the CHA both continuous and discrete variables are translated. As mentioned before, in MNO for sending continuous parameters a variable pool is used, so at worst case $C \times \text{param}_i^{\max}$ continuous variables are needed for a rebec, where C is the queue size and $\text{param}_i^{\max}$ is the maximum number of parameters of the rebec's message servers. In CHA, for each queue element and the buffer, along side the message ID, at least $\text{param}_i^{\max}$[1] variables must be considered for the message parameters. So the total number of variables for messages is at least $(C+1) \times (1 + \text{param}_i^{\max})$. In CHA, at least another $\text{param}_i^{\max}$ continuous variables are needed in the *MsgServers* for the execution of statements to handle parameters. However, in MNO at most $\text{param}_i^{\max}$ continuous variables are needed, since only float parameters are translated to continuous variables. In MNO, at most one extra variable is needed for the rebec's delay timer. In CHA, in addition to the delay timer, the variables for the queue head, queue tail, queue size, buffer fullness flag and message server busyness flag are needed. The summary of this comparison is given in Table 2.

From the modularity standpoint, the main drawback of CHA is the translation of decisions encoded by the *MsgQueue*. In the above mapping, these decisions lead to inserting the buffered message into the appropriate queue element and calling the corresponding message server for the head of the message queue. The former behavior depends on the queue size and the latter depends on both the queue size and the number of the message servers of the rebec. These dependencies make extending or modifying CHA cumbersome compared to Hybrid Rebeca.

[1] The exact number depends on the implementation. However since all the parameters of the message server which has the most parameters must be considered, at least this number of variables are needed.

From analyzability standpoint, as it was shown, the number of continuous variables in the CHA approach is higher than the MNO approach. This higher number of variables may increase the analysis time of models.

6 Guideline for verification of hybrid Rebaca models

As we focus on verification of asynchronous event-based systems using the notion of a single global clock, the relative intervals are used to naturally specify the timimg behavior of such systems. *Metric Temporal Logic* (MTL) [21] is an extension of linear time properties by adding optional real-time constraints to the temporal operators, suitable for specifying relative intervals. We explain how verification of MTL safety properties can be reduced to reachability analysis by using monitor classes.

Given a set P of atomic propositions, the formulas of MTL are built from P using Boolean connectives, and time-constrained versions of the until operator U as follows:

$$\phi ::= p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \, U_I \, \phi,$$

where $I \subseteq (0, \infty)$ is an interval over the non-negative reals with endpoints in $\mathbb{N} \cup \{\infty\}$. The constrained eventually operator $F_I \, \phi \equiv \top U_I \phi$ and constrained always operator $G_I \, \phi \equiv \neg F_I \neg\phi$ can be defined.

We explain the two timed property patterns that we use for the analysis of our case study:

- *Maximal distance between two events* specified by the formula $G \, a \Rightarrow F_{[0,\max]} \, b$ where the events a and b are communication events, either sending/handling a message. This property expresses that whenever the event a occurs, after at most max time units the event b will happen.
- *Zero distance between two constraints on variables* specified by $G \, c_1 \Rightarrow F_{[0,0]} \, c_2$ where c_1 and c_2 are constraints on variables. This property is helpful to inspect the immediate effect of variables on each other.

```
physicalclass Monitor{
  knownrebecs { }
  statevars{real timer; float r;}

  msgsrv initial(){
   setMode(Running);
   r = 0;}

  msgsrv start(){
   timer = 0;
   r = 1;}

  msgsrv stop(){r = 0;}

  mode Running{
   inv(true){timer'=r;}
   guard(false){}
  }
}
```

Fig. 10 The definition of monitor class

We explain how such patterns can be verified by using a monitor class. To measure the amount of time between two events, a monitor class is added to the Hybrid Rebeca model to measure the time between two events. The monitor class is a simple physical class with one physical mode and two message servers, namely *start* and *stop*. The physical mode tracks the time with the real variable *timer* and the message servers are used to start and stop the tracking. The definition of the monitor class is presented in Fig. 10. Note that in the start message server *timer* is reset.

For inspecting the first pattern $G\,a \Rightarrow F_{[0,\max]}b$, one instance of Monitor class is added to the model under consideration. We revise the code by sending a *start* message to the monitor in the code after the statement triggering the event a. We also send a *stop* message to the monitor in the code after the statement triggering the event b. A sending event, shown as **send**(m), is triggered by the statement $y.m()$ while the handle event, denoted by **handle**(m), is triggered by executing the first statement of the message server of m. The monitor class should be wired to the rebecs that their class definitions have been revised. To enforce that the distance between two events should be less than max in SpaceEx, the unsafe condition for reachability analysis is *timer* $>$ max where *timer* is the name of the monitor's timer.

For verifying the second pattern $G\,c_1 \Rightarrow F_{[0,0]}\,c_2$, we almost act with the same fashion. Sending *start* and *stop* messages are conditioned to c_1 and c_2, respectively. In revising the code, after any statement that assigns to the variables involved in c_1, we add a conditional statement "if (c_1) monitor.start()". Similarly, after any statement that assigns to the variables involved in c_2, we add a conditional statement "if (c_2) monitor.stop()". To enforce that the distance is zero, the unsafe condition for reachability analysis is *timer* > 0.

7 Case study

We demonstrate the applicability of our language on a simplified Brake-by-Wire (BBW) system with Anti-lock Braking System (ABS) [12,20,28]. In a BBW system instead of using mechanical parts, braking is managed by electronic sensors and actuators. In ABS, the safety is increased by releasing the brake based on the slip rate to prevent uncontrolled skidding.

In this system, the brake pedal sensor calculates the brake percentage based on the position of the brake pedal. A global brake controller computes the brake torque and sends this value to each wheel controller in the vehicle. Each wheel controller monitors the slip rate of its controlled wheel and releases the brake if the slip rate is greater than 0.2. There is a nonlinear relationship between the friction coefficient of the wheel and the slip rate. When the slip rate is between zero and around 0.2, any increase in the slip rate increases the friction coefficient, but after 0.2, further increase in the slip rate, leads to a reduction in the friction coefficient. For this reason when the slip rate is greater than 0.2, no brake will be applied to the wheel. In this system, each pair of wheel and its wheel controller are connected directly by wire. The brake pedal sensor sends the brake percentage value to the global brake controller through wire. All other communications are done through a shared CAN network. For simplicity, we have considered two wheels in the model. In the following sections, we explain in detail the behavior of its modules by Hybrid Rebeca language.

7.1 Model definition

The model consists of four major classes: *WheelWithSensor* and *WheelCtlr* classes, specified in Fig. 11, *BrakeWithSensor* and *BrakeCtrl* classes, specified in Fig. 12. The main block, defining the configuration of the system and CAN network is given in Fig. 13.

The *WheelWithSensor* class models the sensors and actuators of the wheel. This class has only one active mode *Rolling*. In this mode, it periodically sends the speed of the wheel to its wheel controller and applies the effect of braking on the wheel speed. The *WheelCtlr* class defines the behavior of the wheel controller. It monitors the slip rate of the wheel and decides to apply the brake based on its value.

BrakeWithSensor class defines the behavior for the brake pedal. Here we assume a simple behavior where the brake

```
physicalclass  WheelWithSensor{
   knownrebecs {WheelCtlr ctlr;}
   statevars  {float  trq;  real  spd;  real  t;}
   msgsrv  initial (float  spd_){
      spd = spd_;
      setmode(Rolling);
   }
   msgsrv setTrq(float  trq_){
      trq  = trq_;
   }
   mode Rolling{
      inv(t  <= 0.05){
        t'  = 1;
        spd'  = −0.1−trq;
       }
      guard(t == 0.05){
        t  = 0;
        ctlr .setWspd(spd);
        if(spd > 0)  setmode(Rolling);
}}}
```

```

softwareclass  WheelCtlr{
   knownrebecs {
      WheelWithSensor w;
      BrakeCtlr bctlr;}
   statevars  {int  id;  float  wspd; float  slprt;}
   msgsrv  initial (int  id_){
      id  = id_;
   }
   msgsrv setWspd(float wspd_){
      wspd = wspd_;
      bctlr .setWspd(id,wspd);
   }
   msgsrv applyTrq(float reqTrq,  float  vspd){
      if(vspd == 0) slprt  = 0;
      else
        slprt=(vspd−wspd∗WRAD)/vspd;
      if  (slprt>0.2)     w.setTrq(0);
      else   w.setTrq(reqTrq);
}}
```

Fig. 11 Specification of *WheelWithSensor* and *WheelCtlr* classes in Hybrid Rebeca. WRAD is a constant value of 0.3 for the wheels' radius

percentage is increased by a constant rate until it reaches a predefined max percentage. The class has one known rebec *bctrl* which is the global brake controller. It defines four state variables *bprcnt*, *mxprcnt*, *t* and *r* which are the brake percentage, maximum brake percentage, an auxiliary timer variable and a variable that defines the rate for the brake percentage, respectively. In the *initial* message server, the values of the initial and maximum brake percentage are initialized with the given values and the rate variable is set to 1 and the active mode of the rebec is set to *Braking*. This mode defines a periodic behavior where the value of *bprcnt* is sent to *bctrl* and the brake percentage is increased by the rate defined by *r*. In the actions of this behavior, if the brake percentage is equal or greater than *mxprcnt*, the rate variable *r* is set to zero to stop the brake percentage from changing by time.

The *BrakeCtrl* class is the global brake controller and has the responsibility of delegating the brake torque to wheel controllers. It defines two known rebecs for each wheel controller named *wctlrR* and *wctlrL*. This class has five state variables for the speed of the right and left wheels, the brake percentage from the brake pedal, the global torque calculated from the brake percentage and the estimated vehicle speed. In the message server *control*, first the speed of the vehicle is estimated based on the speed of individual wheels and the desired brake torque is calculated based on the brake percentage. Here, we simply assume that the brake percentage is equal to the brake torque. Then, the estimated speed and global torque are sent to each wheel controller via the CAN network. The message servers *initial*, *setWspd* and *setBprcnt* are omitted for brevity. The message server *setWspd* updates the current wheel speed variable based on the input identifier. The message server *control* must be executed periodically, so an auxiliary *Clock* class is used to periodically send a *control* message to *BrakeCtrl*.

In the main block, all necessary rebecs are instantiated. The wheels *RightWheel* and *LeftWheel* are wired to their respecting wheel controllers by using the tag @Wire. Both wheels are initialized with the speed of 1.[2] The wheel controllers *RightWCtlr* and *LeftWCtlr* are connected to their corresponding wheels by wire and are connected to the global brake controller through CAN by using the tag @CAN. Identifiers of 0 and 1 are given to rebecs *RightWCtlr* and *LeftWCtlr* as initial parameters, respectively. The brake controller *bctlr* is connected to both wheel controllers through the CAN network and the brake *brake* is initialized with the brake percent 60 and maximum brake percent of 65. Both brake *brake* and clock *clock* are connected to *bctlr* by wire. There are four CAN messages in the model. The brake controller *bctlr* sends *applyTrq* message to the wheel controllers *RightWCtlr* and *LeftWCtlr*. The wheels *RightWheel* and *LeftWheel* send *setWspd* message to *RightWCtlr* and *LeftWCtlr* respectively. A higher priority is defined for *applyTrq* messages. Note that a lower number indicates a higher priority. The network delay of all four CAN messages is specified as 10 ms.

7.2 Analysis and verification

For the analysis of this model the queue size of *bctlr* is set to 4, the queue sizes of both *RightWCtlr* and *LeftWCtlr* is set to 2, and for other rebecs the queue size is set to 1. The size of timer variable pool is set to 1 and the size of arguments

[2] As the properties to be verified do not depend on the value of the speed, to minimize the analysis time, this value has been chosen.

```
physicalclass BrakeWithSensor{
  knownrebecs {BrakeCtlr bctlr;}
  statevars {real bprcnt; real t;
      float mxprcnt; float r}
  msgsrv initial(float b_, float mx_){
    bprcnt = b_;
    mxprcnt = mx_;
    r = 1;
    setmode(Braking);
  }
  mode Braking{
    inv(t <= 0.05){
      t' = 1;
      bprcnt' = r;
    }
    guard(t == 0.05){
      t = 0;
      bctrl.setBprcnt(bprcnt);
      if(bprcnt>=mxprcnt)
        r = 0;
      setmode(Braking);
}}}

softwareclass BrakeCtlr{
  knownrebecs{
    WheelCtlr wctlrR;WheelCtlr wctlrL;}
  statevars {float wspdR;float wspdL;
      float bprcnt;float gtrq; float espd;}
  msgsrv control(){
    espd = (wspdR + wspdL)/2;
    gtrq = bprcnt;
    wctlrR.applyTrq(gtrq, espd);
    wctlrL.applyTrq(gtrq, espd);
  }
  // Setters for wspdR, wspdL and bprcnt
  ...
}

physicalclass Clock{
  knownrebecs {BrakeCtlr bctlr;}
  statevars {real t;}
  msgsrv initial(){
    setmode(Running)
  }
  mode Running{
    inv(t <= 0.05){
      t' = 1;
    }
    guard(t == 0.05){
      t = 0;
      bctlr.control();
      setmode(Running);
}}}
```

Fig. 12 Specification of *BrakeWithSensor* and *BrakeCtrl* in Hybrid Rebeca

```
main {
  WheelWithSensor RightWheel
    (@Wire RightWCtlr):(1);
  WheelWithSensor LeftWheel
    (@Wire LeftWCtlr):(1);
  WheelCtlr RightWCtlr
    (@Wire RightWheel, @CAN bctlr):(0);
  WheelCtlr LeftWCtlr
    (@Wire LeftWheel, @CAN bctlr):(1);
  BrakeCtlr bctlr
    (@CAN RightWCtlr,@CAN LeftWCtlr):();
  BrakeWithSensor brake
    (@Wire bctlr):(60,65);
  Clock clock(@Wire bctlr):();

   CAN{
     priorities {
        bctlr  RightWCtlr.applyTrq     1;
        bctlr  LeftWCtlr.applyTrq      2;
        RightWCtlr bctlr.setWspd       3;
        LeftWCtlr  bctlr.setWspd       4;
     }
     delays{
        bctlr  RightWCtlr.applyTrq     0.01;
        bctlr  LeftWCtlr.applyTrq      0.01;
        RightWCtlr bctlr.setWspd       0.01;
        LeftWCtlr  bctlr.setWspd       0.01;
}}}
```

Fig. 13 Main block of the brake-by-wire model in Hybrid Rebeca

variable pool is set to 11. The hybrid automaton derived from the model consists of 10,097 locations and 25,476 transitions, derived automatically by our tool.[3] We use SpaceEx [14] tool to verify our model. We simplified the nonlinear equation of the slip rate for analysis as it is not supported by SpaceEx. By specifying a set of forbidden states, safety properties can be verified by the reachability analyzer of SpaceEx. We verified three properties for our case study:

1. The first property is "design-fault freedom".
2. The second property is a timing constraint. This property states that the time between the transmission of the brake percentage from the brake pedal and its reaction by wheel actuators, must not exceed 0.2 s.
3. The third property states that whenever the slip rate of a wheel exceeds 0.2, the brake actuator of that wheel must be immediately released.

[3] The tool converting a hybrid Rebeca model to a hybrid automaton, as an input of SpaceEX is available at http://rebeca-lang.org/allprojects/HybridRebeca. The tool handles models specified in an intermediate
Footnote 3 continued
format very close to Hybrid Rebeca. This format is suitable for translation into hybrid automata. The process of translating a Hybrid Rebeca model to the intermediate format is currently manual.

By design-fault we mean a fault caused by following situations: exceeding the capacity of a message queue, running out of the pooled variables, and having messages with the same priority in the CAN buffer. We assume the message with the highest priority must always be unique in CAN buffer.

For the first property, the verified forbidden condition in SpaceEx for this property is loc() == Fault, where the term loc() specifies the current location in SpaceEx and *Fault* is the specific location we considered for faulty situations as we explained in Sect. 4.3.

The second property inspects the time between two events: the transmission of the brake percentage indicates sending the message *setBprcnt* to *bctrl* by the instance of the class *BrakeWithSensor*. The reaction of the wheel actuator denotes receiving the message *setTrq* by the instances of *WheelWithSensor*. This property is specified as "$G\ \mathbf{send}(setBprcnt) \Rightarrow F_{[0,0.2]}\ \mathbf{handle}(setTrq)$". According to our guideline in Sect. 6, one monitor rebec is instantiated and is wired to the brake pedal and one of the wheels. A *start* message is sent in the trigger of the *Braking* mode of the *BrakeWithSensor* class after the brake percentage is sent to the brake controller and a *stop* message is sent by *setTrq* message server of the wired wheel.

The third property enforces that when the slip rate is greater than 0.2, then immediately the brake is released, i.e., the wheel torque becomes 0. This property is specified as "$G\ slprt > 0.2 \Rightarrow F_{[0,0]}\ \mathrm{trq} = 0$)". For this property, a monitor rebec is wired to one of the wheels and its wheel controller. A *start* message is sent from the wheel controller after it computes the slip rate when the slip rate is greater than 0.2 in the message handler *applyTrq* and a *stop* message is sent by *setTrq* message server after assigning to the variable trq.

The resulted hybrid automaton for the first property has 10,097 locations and 25,476 transitions, which is huge for verification purposes. This huge size stems from the fine-grained semantics of our language. But most of these locations are urgent locations where time does not advance and can be aggregated for the verification of the properties. Our tool automatically aggregates the urgent locations of the hybrid automaton derived from the given model. For aggregation, the tool removes each urgent location by combining its incoming and outgoing transitions. This aggregation is sound as we focus on properties that do not depend on aggregated locations. So, our aggregation preserves deadlock and timed properties expressing the time distance between two events/value updates in our analyses.

After aggregating these urgent locations, the size of the resulting hybrid automaton is reduced to 21 locations and 1148 transitions. The three properties are verified on their respective reduced hybrid automaton. The verification results of these properties are provided in Table 3.

8 Related work

There are some frameworks for modeling and analyzing cyber-physical systems. Some of these frameworks rely on simulation for analysis and others offer formal verification.

Application of the actor model as the basis for modeling CPSs has been followed in Ptolemy [33] and development tools such as Theatre [7] which use simulation for analyzing the systems. These tools help engineers to get insight of systems by restricting their behavior to be deterministic. With the aim to synthesis codes from models, non-determinism is not supported by these frameworks as deployed systems are deterministic. Hybrid Rebeca allows non-determinism inherent in concurrent and distributed systems, e.g., in the case of simultaneous arrival of messages (and no explicit priority-based policy to choose one over the other). We can analyze possible implementations of systems (due to possible resolution of non-determinisms) by application of reachability analysis to get insight into the system properties. Although by Ptolemy models we get precise insights into deterministic scenarios for a specific implementation, by more abstract Hybrid Rebeca models, we can derive formal analysis about the possible implementations of a system.

Ptolemy II is a framework that uses the concept of *model of computation* (MoC) which defines the rules for concurrent execution of components and their communications. Ptolemy supports many models of computation like process networks, discrete events, dataflow and continuous time. Heterogeneous model can be made by nesting these models of computations in a hierarchical structure. As far as we know there is no formal semantics for the hybrid models of Ptolemy framework to enable formal verification. Our hybrid Rebeca models can be considered as an extension of discrete events model of computation in Ptolemy.

In [7], an agent-based and control centric methodology, called Theatre, is presented for development of CPSs. This approach includes all development stages of a system from analysis by simulation to the execution of the final system. For the modeling phase concepts like actors, message, actions, processing units and environmental gateway are presented in this methodology. The message passing among actors is asynchronous and the computations of the model take place in the actions that are submitted to the processing units by the actor for execution. The environmental gateway is used for abstracting the physical processes where in later stages is replaced by the real entities. This approach relies on simulation to analyze a system, and no formal analysis is supported. Theatre is extended with continuous behaviors in [30] for modeling and property checking of CPSs. As mentioned by the authors in the paper the work is inspired from [17]. For continuous behaviors, the concept of *mode* is introduced which is similar

Table 3 Verification result of the case study

Property	Derived HA		Generation time (s)	Reduced HA		Verification result	Verification duration (s)
Design-fault freedom	10,097	25,476	12	21	1148	Passed	3705
Reaction time	16,317	42,976	20	21	1168	Passed	7521
Brake release	54,097	175,036	64	21	1168	Passed	3541

Legends: Property: verified property, Derived HA: derived hybrid automaton size where the first and second columns are the number of locations and transitions, respectively, Generation time: duration of hybrid automaton generation in seconds, Reduced HA: reduced hybrid automaton size, Verification result: result of verified property, Verification duration: duration of verification in seconds

to the concept of modes in Hybrid Rebeca, but they are defined as separate entities from the actors. Hybrid Theatre models are reduced into the stochastic timed and hybrid automata of Uppaal Statistical Model Checker for an approximate analysis based on simulation. In [29], a modular approach for specifying and validating CPSs using rewriting logic-based technique is proposed. In this work a CPS is described as a linear hybrid automata in rewriting logic where the components of the system communicate asynchronously. Timed hybrid Petri nets [9] can also be used to model hybrid systems and CPSs. For analysis of these hybrid Petri nets in [9] a translation to hybrid automata is presented. However, Petri-net-based approaches prohibit modular specification of systems. The framework of [22] provides a hybrid process calculus tailored for modeling CPSs and analyzing their security properties [23,24]. Differential dynamic logic (dL) [32] specifies hybrid systems in a sequential imperative programming language. This hybrid logic, not suitable for modeling distributed and concurrent hybrid systems, supports verification via its implementation in theorem prover tools. The Active Object, objects realizing actor-based concurrency, language ABS [18] has been extended with Hybrid Active Objects for modeling CPSs, called Hybrid ABS (HABS) [19]. In this formalism, object actives can have real-valued fields that their evolution over time is expressed by ODEs. Hybrid active objects are time-deterministic as opposed to Hybrid Rebeca and communicate via ports. Their verification approach is based on translation from HABS models following certain communication patterns to dL. In all mentioned approaches network governing the interactions between physical and cyber entities is not addressed.

9 Discussion

In Sect. 3, we presented our extended actor model for cyber-physical systems. In our model, the software and physical actors are separated and modes are added to physical actors for specifying the continuous behaviors. The separation of software and physical rebecs prevents the interference of continuous behaviors with discrete behaviors. In Rebeca, each actor has only one thread of execution and its local state is encapsulated from other actors. This greatly simplifies the interactions between actors. But having both continuous and discrete behaviors in one actor, can be considered as having multiple threads of execution in the actor; each continuous behavior is governed by a thread. Since these threads share the same variables, this approach is inconsistent with Rebeca and can surprise the modeler. A simple example to highlight this issue is to consider the following code segment of a message server:

```
a = k;
delay (2);
b = a + c;
```

The constant k is assigned to the variable a. The delay statement is used to abstractly model the computation time of complex computations. After the specified delay, the value of variable a is used to update the variable b. Assume that the rebec has a continuous behavior and during the execution of the delay statement, the continuous behavior is finished and changes the value of variable a in its actions. This affects the value of the variable b when the delay statement is over and can lead to a faulty behavior. The separation of software and physical actors solves this issue. Note that the delay statement is not allowed in the physical actors.

To analyze a Hybrid Rebeca model, we are constrained to first derive its corresponding monolithic hybrid automaton. This derivation is only possible for Rebeca models that generate a finite number of locations. However, this process may be time-consuming for the models with large bounded queues and large variations of values for rebec variables. The analysis of some properties may only need a partial generation of the automaton. To avoid building the whole automaton, in another project, we define our semantics in terms of timed transition systems and integrate an existing approximation algorithm for the reachability analysis of hybrid automata into our state-space generation process. As we explained in Sect. 2.2, the reachability analysis of hybrid automata is not decidable and approximation algorithms are used. We remark that if the over-approximating set contains an unsafe state, we cannot decide on the safety of the system.

10 Conclusion and future work

In this paper, we presented an extended actor model for modeling hybrid systems and CPSs, where both continuous and discrete processes can be defined. In this actor model, two kinds of actors are defined: software actors and physical actors. The software actors contain the software behaviors of the model and similarly, physical actors contain the physical behaviors. We also introduced a network entity to the actor model for modeling the behavior of the network of the model. We implemented this extended actor model in Hybrid Rebeca language. This language is an extension of Timed Rebeca language and allows defining classes to make models modular and reusable. The semantics of the language is defined based on hybrid automata allowing formal verification of models. Since our focus was automotive domain, CAN network is modeled in this version of the language. To show the applicability of our language, we modeled and analyzed a Brake-by-Wire system. For the verification of the model, three safety properties were considered. We used SpaceEx framework to verify these properties. It was shown that for some properties new entities were needed to make the verification feasible.

We also demonstrated how our modeling framework is useful. To this aim, we first defined a translation for hybrid rebeca models to compositional hybrid automata, then we compared these approaches from modularity and analyzability standpoints. We showed that changing and modifying a model is much easier in our language. Also, we showed that the compositional hybrid automata model contains more continuous variables compared to the hybrid automaton resulted from our language semantics. The number of continuous variables can have exponential effect on the verification time of a model.

Since we focused on the automotive domain, only CAN network was defined in our current version of the language. Other network models are needed for different applications of CPSs. Instead of defining multiple network models, it must be possible to allow user-defined network models. Providing a set of basic functionalities for defining most network models is among of our future work. Furthermore, defining multiple instances of a network model (e.g., multiple CAN networks) may be needed in some systems. Like network models which dispatch messages among rebecs and may resolve the non-determinism due to simultaneous arrival of messages, it must be possible to define internal message schedulers for rebecs to resolve the non-determinism due to simultaneous arrival of messages maybe from different network models.

Concerning deterministic models, a deterministic software model ensures that a model starting from a given initial state and with certain inputs always behaves the same (i.e., generates always the same output). In a timed and hybrid model, receiving physical triggers at different times are considered as different inputs. By definition, “inputs” are not controlled by the model and therefore cannot be determined by the model. Still there are ways to handle this so-called “nondeterminism in the inputs” which refers to the fact that inputs are not controlled by the model. Simultaneous occurrences of different events, of both cyber and physical types, can be handled in different ways. Hybrid Rebeca allows nondeterministic handling of such events, and hence a nondeterministic behavior in such situations. Some solutions to keep the behavior deterministic, even in such situations, are offered by synchronous languages [5], and some are recently proposed in Lingua Franca language [26,27].

Providing more patterns of MTL and reducing their verification to reachability analysis are among of our future work.

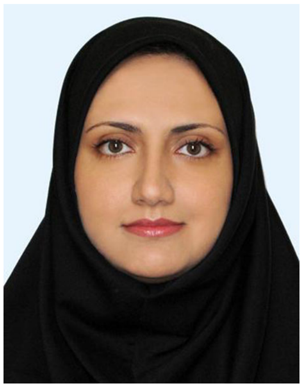
Acknowledgements We would like to thank Edward Lee for his support and patient guidance on modeling and analyzing CPSs, Tom Henzinger for his fruitful discussion on the extended actor model, and MohammadReza Mousavi and Ehsan Khamespanah for their useful contributions.

References

1. Aceto, L., Cimini, M., Ingólfsdóttir, A., Reynisson, A.H., Sigurdarson, S.H., Sirjani, M.: Modelling and simulation of asynchronous real-time systems using timed rebeca. In: 10th International Workshop on the Foundations of Coordination Languages and Software Architectures. EPTCS, vol. 58, pp. 1–19 (2011)
2. Agha, G.A.: ACTORS: A Model of Concurrent Computation in Distributed Systems. MIT Press Series in Artificial Intelligence, MIT Press, Cambridge (1986)
3. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T., Ho, P., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theor. Comput. Sci. **138**(1), 3–34 (1995)
4. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theor. Comput. Sci. **138**(1), 3–34 (1995)
5. Berry, G., Gonthier, G.: The Esterel synchronous programming language: design, semantics, implementation. Sci. Comput. Program. **19**(2), 87–152 (1992). https://doi.org/10.1016/0167-6423(92)90005-V
6. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: an analyzer for non-linear hybrid systems. In: 25th International Conference on Computer Aided Verification. LNCS, vol. 8044, pp. 258–263. Springer (2013)
7. Cicirelli, F., Nigro, L., Sciammarella, P.F.: Model continuity in cyber-physical systems: a control-centered methodology based on agents. Simul. Model. Pract. Theory **83**, 93–107 (2018)
8. Cuijpers, P., Reniers, M.A.: Hybrid process algebra. J. Log. Algebr. Program. **62**(2), 191–245 (2005)
9. David, R., Alla, H.: On hybrid petri nets. Discrete Event Dyn. Syst. **11**(1–2), 9–40 (2001)

10. Davis, R.I., Burns, A., Bril, R.J., Lukkien, J.J.: Controller area network (CAN) schedulability analysis: refuted, revisited and revised. Real Time Syst. **35**(3), 239–272 (2007)
11. Derler, P., Lee, E.A., Sangiovanni-Vincentelli, A.L.: Modeling cyber-physical systems. Proc. IEEE **100**(1), 13–28 (2012)
12. Filipovikj, P., Mahmud, N., Marinescu, R., Seceleanu, C., Ljungkrantz, O., Lönn, H.: Simulink to UPPAAL statistical model checker: Analyzing automotive industrial systems. In: 21st International Symposium on Formal Methods. LNCS, vol. 9995, pp. 748–756 (2016)
13. Frehse, G.: Phaver: algorithmic verification of hybrid systems past hytech. In: Morari, M., Thiele, L. (eds) 8th International Workshop on Hybrid Systems: Computation and Control. LNCS, vol. 3414, pp. 258–273. Springer (2005)
14. Frehse, G., Guernic, C.L., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: Spaceex: scalable verification of hybrid systems. In: 23rd International Conference on Computer Aided Verification. LNCS, vol. 6806, pp. 379–395. Springer (2011)
15. Henzinger, T.A.: The theory of hybrid automata. In: 11th Annual IEEE Symposium on Logic in Computer Science, pp. 278–292. IEEE Computer Society (1996)
16. Hewitt, C.: Description and theoretical analysis (using schemata) of planner: a language for proving theorems and manipulating models in a robot. Technical Report on Massachusetts Institute of Technology, Artificial Intelligence Laboratory (1972)
17. Jahandideh, I., Ghassemi, F., Sirjani, M.: Hybrid rebeca: Modeling and analyzing of cyber-physical systems. In: 8th International Workshop on Model-Based Design of Cyber Physical Systems. LNCS, vol. 11615, pp. 3–27. Springer (2018)
18. Johnsen, E., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: 9th International Symposium on Formal Methods for Components and Objects. LNCS, vol. 6957, pp. 142–164. Springer (2010)
19. Kamburjan, E., Mitsch, S., Kettenbach, M., Hähnle, R.: Modeling and verifying cyber-physical systems with hybrid active objects. arXiv:1906.05704 (2019)
20. Kang, E., Enoiu, E.P., Marinescu, R., Seceleanu, C.C., Schobbens, P., Pettersson, P.: A methodology for formal analysis and verification of EAST-ADL models. Reliab. Eng. Syst. Saf. **120**, 127–138 (2013)
21. Koymans, R.: Specifying real-time properties with metric temporal logic. Real Time Syst. **2**(4), 255–299 (1990)
22. Lanotte, R., Merro, M.: A calculus of cyber-physical systems. In: Language and Automata Theory and Applications: 11th International Conference. LNCS, vol. 10168, pp. 115–127 (2017)
23. Lanotte, R., Merro, M., Muradore, R., Viganò, L.: A formal approach to cyber-physical attacks. In: 30th IEEE Computer Security Foundations Symposium, pp. 436–450. IEEE Computer Society (2017)
24. Lanotte, R., Merro, M., Tini, S.: Towards a formal notion of impact metric for cyber-physical attacks. In: 14th International Conference on integrated Formal Methods (2018) (to appear)
25. Lee, E.A.: Cyber physical systems: Design challenges. In: 11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2008), pp. 363–369. IEEE Computer Society (2008)
26. Lohstroh, M., Romeo, Í.Í., Goens, A., Derler, P., Castrillón, J., Lee, E.A., Sangiovanni-Vincentelli, A.L.: Reactors: a deterministic model for composable reactive systems. In: 9th International Workshop on Model-Based Design of Cyber Physical Systems. Lecture Notes in Computer Science, vol. 11971, pp. 59–85. Springer
27. Lohstroh, M., Schoeberl, M., Goens, A., Wasicek, A., Gill, C., Sirjani, M., Lee, E.A.: Actors revisited for time-critical systems. In: Proceedings of the 56th Annual Design Automation Conference, p. 152. ACM (2019)
28. Marinescu, R., Mubeen, S., Seceleanu, C.: Pruning architectural models of automotive embedded systems via dependency analysis. In: 42th Euromicro Conference on Software Engineering and Advanced Applications, pp. 293–302. IEEE Computer Society (2016)
29. Metelo, A., Braga, C., Brandão, D.N.: Towards the modular specification and validation of cyber-physical systems: a case-study on reservoir modeling with hybrid automata. In: 18th International Conference on Computational Science and Its Applications, Part I. LNCS, vol. 10960, pp. 80–95. Springer (2018)
30. Nigro, L., Sciammarella, P.F.: Statistical model checking of cyber-physical systems using hybrid theatre. In: Proceedings of SAI Intelligent Systems Conference, pp. 1232–1251. Springer (2019)
31. Pfeiffer, O., Ayre, A., Keydel, C.: Embedded Networking with CAN and CANopen, 1st edn. Copperhill Media Corporation, Greenfield (2008)
32. Platzer, A.: Differential-algebraic dynamic logic for differential-algebraic programs. J. Log. Comput. **20**(1), 309–352 (2010)
33. Ptolemaeus, C. (ed.): System Design, Modeling, and Simulation using Ptolemy II. Ptolemy.org (2014)
34. Sabouri, H., Khosravi, R.: Delta modeling and model checking of product families. In: 5th International Conference on Fundamentals of Software Engineering. LNCS, vol. 8161, pp. 51–65. Springer (2013)
35. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constructing invariants for hybrid systems. Formal Methods Syst. Des. **32**(1), 25–55 (2008)
36. Sirjani, M.: Power is overrated, go for friendliness! expressivness versus faithfulness and usability in modeling-actor experience. In: Edward A. Lee Festschrift, LNCS, pp. 1–21. Springer (2018)
37. Sirjani, M., Jaghoori, M.M.: Ten years of analyzing actors: Rebeca experience. In: Formal Modeling: Actors, Open Systems, Biological Systems—Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday. LNCS, vol. 7000, pp. 20–56. Springer (2011)
38. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using Rebeca. Fundam. Inform. **63**(4), 385–410 (2004)
39. Varshosaz, M., Khosravi, R.: Modeling and verification of probabilistic actor systems using prebeca. In: 14th International Conference on Formal Engineering Methods. LNCS, vol. 7635, pp. 135–150. Springer (2012)
40. Wolf, W., Madsen, J.: Embedded systems education for the future. Proc. IEEE **88**(1), 23–30 (2000)
41. Yousefi, B., Ghassemi, F., Khosravi, R.: Modeling and efficient verification of broadcasting actors. In: 6th International Conference on Fundamentals of Software Engineering. LNCS, vol. 9392, pp. 69–83. Springer (2015)
42. Yousefi, B., Ghassemi, F., Khosravi, R.: Modeling and efficient verification of wireless ad hoc networks. Formal Asp. Comput. **29**(6), 1051–1086 (2017)

Iman Jahandideh received his M.Sc. degree in Software Engineering from University of Tehran in 2018. His researching focus was formal methods in Cyber-Phyiscal systems. His main interest is bridging the gap between game development and software engineering principles and practices.



Fatemeh Ghassemi has received her Ph.D in Software Engineering from Sharif university of technology in 2011, and in Computer Science from Vrije Universiteit of Amsterdam in 2018. She is an assistant professor at University of Tehran since 2012, supervising the Formal Methods laboratory. Her research interest includes formal methods in software engineering, protocol verification, model checking, process algebra, and software testing. Her research is now focused on combining machine learning and automata learning approaches. Fatemeh has been the PC member of several international conferences including iFM, Coordination, FM, and FSEN.

Marjan Sirjani Marjan Sirjani is a Professor and chair of Software Engineering at Mälardalen University, and the leader of Cyber-Physical Systems Analysis research group. Her main research interest is applying formal methods in Software Engineering. She works on modeling and verification of concurrent, distributed, timed, and self-adaptive systems. Marjan and her research group are pioneers in building model checking tools, compositional verification theories, and state-space reduction techniques for actor-based models. She has been working on analyzing actors since 2001using the modeling language Rebeca (http://www.rebeca-lang.org). Her research is now focused on safety assurance and performance evaluation of cyber-physical and autonomous systems. Marjan has been the PC member and PC chair of several international conferences including SEFM, iFM, Coordination, FM, FMICS, SAC, FSEN. She is an editor of the journal of Science of Computer Programming.