Hybrid Rebeca: Modeling and Analyzing of Cyber-Physical Systems

Iman Jahandideh[1] and Fatemeh Ghassemi[1] Marjan Sirjani[2,3]

[1] School of Electrical and Computer Engineering, University of Tehran
{jahandideh.iman,fghassemi}@ut.ac.ir
[2] School of Innovation, Design and Engineering, Mälardalen University, Västerås, Sweden
marjan.sirjani@mdh.se
[3] School of Computer Science, Reykjavik University, Reykjavik, Iceland

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 actor supports modeling distributed asynchronously communicating systems. We propose Hybrid Rebeca language to support modeling of cyber-physical systems. Hybrid Rebeca is an extension of actor-based language Rebeca. In this extension, physical actors are introduced as new computational entities to encapsulate the physical behaviors. To support various means of communication among the entities, network is explicitly modeled as a separate entity from actors. We derive hybrid automata as the basis for analysis of Hybrid Rebeca models. We demonstrate the applicability of our approach through a case study in the domain of automotive systems. We use SpaceEx framework for the analysis of the case study.

Keywords: Actor model · Cyber-physcial 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 [25]. In such *hybrid* systems, physical and cyber behaviors, characterized as continuous and discrete respectively, affect each other. Cyber-physical sytems (CPSs) are heterogeneous systems with tight interactions between physical and software processes where components in the system usually communicate through 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 [7]. It relies on models specifying the behavior of the system often in a formal way. Using models instead of physical realizations of the system, beside reducing the costs of the development, can provide new insights early in the design process and enable 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. Furthermore, for dealing with such systems with complicated and heterogeneous components, besides expressiveness

power, a level of *friendliness* is appealing in design tools. This friendliness can be as important as expressiveness [21]. Friendliness is evaluated by its *faithfulness* to the system it is modeling, and *usability* to the modeler.

Existing modeling frameworks for hybrid systems such as hybrid automata [3,10] and hybrid Petri nets [5] can be used to model CPSs. The former has higher analysis power while the latter can be more easily used for modeling event-based systems [5]. Due to the existence of network in CPSs, the provided modeling power in these frameworks is not satisfactory for systems composed of many interacting heterogeneous entities. In the domain of automotive, ECUs, sensors and actuators may be connected directly by wire or through a communication media such as a serial bus. Improving the level of abstraction is beneficial to reduce errors introduced during the design process and improve perception of the model.

The computation model of actors provides a suitable level of abstraction to faithfully model distributed asynchronously communicating systems [2,11]. 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. We extend the actor-based language Rebeca, with physical behavior to support hybrid systems. Additionally, we need to support various types of communication, namely wired connections with no delay, serial buses with deterministic behavior, and wireless communication among the actors. So, we decided not to model the behavior of the network as an actor within a model, and instead model it as a separate entity.

To implement the extended actor model, we propose *Hybrid Rebeca*, as an extension of (Timed) Rebeca [23,1]. Rebeca provides an operational interpretation of actor model through a Java-like syntax. Its timed extension supports modeling of the computation time, and network delay for message communication. Hybrid Rebeca, extends Timed Rebeca with continuous behaviors based on our extended actor model.

Hybrid Rebeca defines two types of classes, software and physical. Software classes are similar to reactive classes in 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's possible to change the continuous behavior of the actor. In this version, CAN network is defined as 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. Real-valued variables are added on which continuous behaviors are defined. The modes of physical classes are similar to the concept of locations in hybrid automata, and to solve these behaviors, the semantics of Hybird Rebeca is defined as a hybrid automaton, for which many verification algorithms and tools are available.

The main contribution of the paper can be summarized as providing an actor-based formalism that supports "friendliness" with small number of primitive concepts. In particular it distinguishes between software and physical actors and supports two types of connections among actors (in principle one could have more types). A tool which automatically derives a hybrid automaton from a given model is implemented, which is suitable for formal reachability analysis. The rest of the paper is structured as follows. The next section defines hybrid automata, actor model and Rebeca language. Section 3 presents our extended actor model for modeling CPSs. In section 4 the syntax and semantics of Hybrid Rebeca language is defined. Section 5 presents our case study and its

results. In section 6 we briefly mention some related works. In section 7 we discuss one of our design decisions for our extend actor model. The conclusion is presented in section 8.

2   Preliminaries

As we define the semantics of our framework based on hybrid automata, we first provide an overview on this model and then explain actor model and Timed Rebeca.

2.1   Hybrid Automata

Hybrid automata (HA) [3,10] 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. In Fig. 1 a hybrid automaton for a simple heater model is presented. 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 22. The turning on behavior of the heater is defined similarly. Initially the heater is off and the temperature is 20.
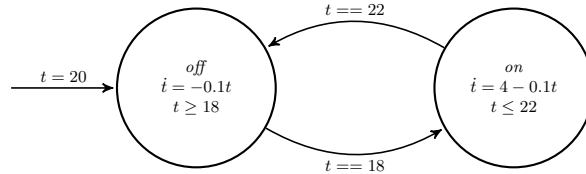
Fig. 1. A hybrid automaton for a heater which consists of two locations (modes) named *on* and *off*. Each location defines a flow and an invariant on the variable t which is the temperature. The mode of the heater changes by means of guarded transitions between the locations. The initial location is *off* and the initial value of t is 20.

Let a valuation $v : V \rightarrow Val$ be a function that assigns a value to each variable of V where *Val* is the set of values, defined by the context. We denote the set of valuations on the set of variables V as $\mathcal{V}(V)$. Formally 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 $\mu \in \mathcal{V}(Var)^2$ is a transition relation on variables. The elements of $\mu = (v, v')$ represents 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 $\mathbb{R}^{\geq 0} \to \mathcal{V}(Var)$. Each flow specifies how the values of variables evolve over time. A flow is usually defined by a doted variable $\dot{v}$ which represents the first derivative.
– *Inv* is a labeling function that assigns an invariant $Inv(l) \subseteq \mathcal{V}(Var)$ to each location $l \in Loc$,
– *Init* is a labeling function that assigns an initial condition $Init(l) \subseteq \mathcal{V}(Var)$ 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, $Lab = \emptyset$ and the labels over the transitions are ϵ which is not shown for brevity. Also the transition relations only consist of guards and the assignments are empty. The flows and the invariant of each location is defined on the location itself. The initial condition for location *off* is $Init(off) = \{t = 20\}$ and for location *on* is $Init(off) = \emptyset$. Note that in our language, we do not use primed variables of the form v' to represent valuation after discrete transitions. We use v' instead of $\dot{v}$ to represent the first derivative of variable v

2.2   Actor Model and Timed Rebeca

Actor model is used for modeling distributed systems. It was originally proposed by Hewitt [11]. 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, node crashes, etc. In this model each actor has a an address and a mailbox which stores the received messages. The behavior of an actor is defined in 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 [23] 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 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 [22]. For example, different extensions have been introduced in various domains such as probabilistic systems [24], real-time systems [1], software product lines [20], and broadcasting environment [26].

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 a Rebec's mailbox is modeled by a message queue. A reactive class consists of *known rebecs* to specify the rebecs it can communicate with, *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 new constructs: *delay*, *after*, and *deadline*. In Timed Rebeca model, each rebec has it's 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.

3 Actor Model for CPSs

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

Rebeca offers a fair and nondeterministic network model. For many application of CPSs this network model is too abstract or completely invalid. For example Control Area Network (CAN) [18] protocol is a dominant networking protocol in automotive industry, which can not be faithfully modeled by Rebeca's network model as by 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 massage delivery sequence by the network actor depends on the execution order of sending actors. So we modeled 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 actions where actions are a set of statements. The actions are 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 easily be 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 a message 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 [6]. To implement this protocol, a unique priority must be assigned to each message and 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
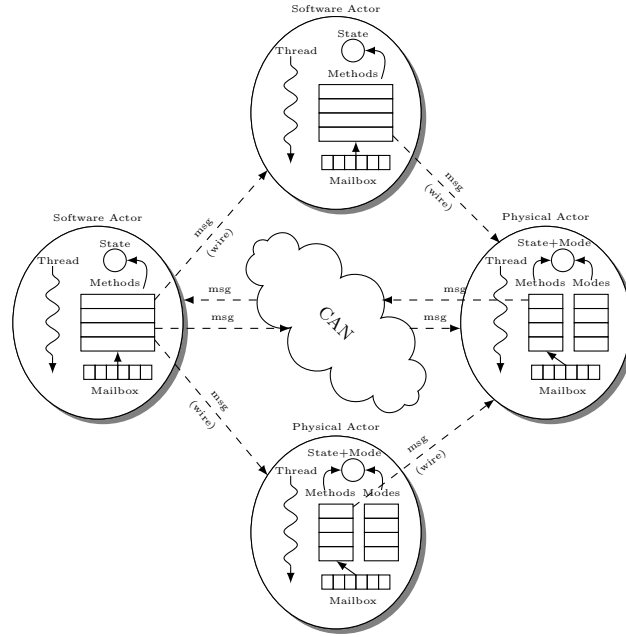
Fig. 2. Hybrid Rebeca model: each actor has its own thread of control, message queue, and ID. In addition to these, physical actors have modes that are defined by a guard, an invariant, and actions. Actors can communicate with each other either by sending messages through CAN or directly by wire.

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) got 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 has been illustrated in Fig. 2.

4   Hybrid Rebeca

Hybrid Rebeca is an extension of Timed Rebeca to support physical behaviors. Timed Rebeca supports deadline specification for messages, but because of technical issues, Hybrid Rebeca does not support this feature. Other timing behaviors like network delay and computation time are supported.

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 services 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 communicated through wire, the communication delay is considered to be zero. After instantiating rebecs, the CAN specification must be defined.

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. 3. 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, expect 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 set of actions. The comparison expression of an invariant and a guard expression are specified by the reserved words inv and guard, respectively. The actions following the guard expression, which are expressed as a statement block, define 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 rebecs by sending the message *setMode*.

Three primitive data types are available in Hybrid Rebeca: *int*, *real*, and *float*. Variables of types *int* and *real* are only allowed in software and physical classes, respectively. Message parameters and state variables can be only defined of primitive type with some restrictions. 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 evolve with 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 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 binded 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 (@Wire b, @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.

$$
\begin{aligned}
\text{Model} &::= (\text{SClass} \mid \text{PClass})^+ \text{ Main} \\
\text{Main} &::= \textsf{main } \{\text{InstanceDcl}^* \text{ CANSpec}\} \\
\text{InstanceDcl} &::= \textsf{C r } (\langle \text{ @CAN} \mid \text{@Wire } \text{ r}\rangle^*) : (\langle c \rangle^*) \\
\text{CANSpec} &::= \textsf{CAN } \{\text{Priorities Delays}\} \\
\text{Priorities} &::= \textsf{priorities } \{\langle \textsf{r r.m c};\rangle^*\} \\
\text{Delays} &::= \textsf{delays } \{\langle \textsf{r r.m c};\rangle^*\} \\
\text{SClass} &::= \textsf{softwareclass } \textsf{C } \{\textsf{KnownRebecs Vars MsgSrv}^*\} \\
\text{PClass} &::= \textsf{physicalclass } \textsf{C } \{\textsf{KnownRebecs Vars MsgSrv}^* \textsf{ Mode}^*\} \\
\text{KnowRebecs} &::= \textsf{knownrebecs } \{\text{VarDcl}^*\} \\
\text{Vars} &::= \textsf{statevars } \{\text{VarDcl}^*\} \\
\text{VarDcl} &::= \textsf{T } \langle v \rangle^+; \\
\text{MesgSrv} &::= \textsf{msgsrv } \text{m } (\langle \text{T v}\rangle^*) \{\text{Stmt}^*\} \\
\text{Mode} &::= \textsf{mode } \textsf{m } \{\textsf{inv}(e) \; (v' = e)^+ \textsf{ guard}(e) \text{ MSt}\} \\
\text{Stmt} &::= v = e; \mid \text{Call}; \mid \textsf{if}(e) \text{ MSt } [\textsf{else MSt}] \mid \textsf{delay}(t); \mid \textsf{setmode}(m); \\
\text{Call} &::= \textsf{r.m}(\langle e \rangle^*) \mid \textsf{r}.setMode(\textsf{m}) \\
\text{MSt} &::= \{\text{Stmt}^*\} \mid \text{Stmt}
\end{aligned}
$$

Fig. 3. Abstract syntax of Hybrid Rebeca. The main differences in syntax compared to Timed Rebeca, are highlighted with color green. Angle brackets $\langle \rangle$ denotes meta parenthesis, superscripts $+$ and $*$ respectively are used for repetition of one or more and repetition of zero or more times. Combination of $\langle \rangle$ with repetition is used for comma separated list. Brackets [ ] are used for optional syntax. Identifiers C, T, m, $\mathfrak{m}$, v, c, r and e respectively denote class, primitive type, method name, mode name, variable, constant, and rebec name, respectively; and e denotes an expression.

Example Here we use a part of our case study to show the basics of our language. The model of the case study is presented in Fig. 5. Our case study is a Brake-By-Wire (BBW) with an Antilock Braking System (ABS). Here we only describe two classes *Wheel* and *WCtlr* which define the behavior of wheels and wheel controllers, respectively. The physical class *Wheel* has one known

rebec *ctlr*, which is the wheel controller of the wheel. It also has three variables *trq*, *spd* and *t* which are respectively the amount of torque that must be applied during brake, the speed of the wheel and an auxiliary timer for periodic behavior of the wheel's sensor. In its *initial* message server, the *spd* is initialized to the given value and the active mode of the rebec is set to *Rolling* mode by using the *setmode* statement. *Wheel* defines a setter message server *setTrq* for updating the value of *trq*. This class has one mode named *Rolling*. This mode has a periodic behavior that every 50 milliseconds, sends the value of *spd* to its wheel controller defined by *ctlr*. For the periodic behavior the invariant $t <= 0.05$, the flow $t' = 1$ and the guard $t == 0.05$ are defined. The flow equation for *spd* variable is defined simply as $spd' = -0.1 - trq$. The constant -0.1 models the friction of the wheel with the road. In the actions of this mode, after reseting the timer value and sending the wheel's speed to its controller, the rebec's mode is again set to the *Rolling* mode if the wheel's speed is greater than zero.

The ABS behavior is defined by the software class *WCtlr*. It has two known rebecs *w* and *bctrl*, which are the controlled wheel of the controller and the global brake controller of the system, respectively. It also has three state variables *id*, *wspd* and *slprt*. The variable *id* is the identifier of the wheel controller and is used to differentiate between multiple wheel controllers in the model, *wspd* is the speed of the controlled wheel, and *slprt* is an auxiliary variable for calculating the slip rate. The variable *slprt* is used to determine whether the brake must be applied or not (this will be explained more in section 5). The *wspd* variable gets updated by *setWspd* message server. This message server also sends the wheel's speed along side the controller's identifier, to the *bctrl*. To calculate the brake torque for wheels, the global brake controller sends an *applyTrq* message to each *WCtlr* in the system. The corresponding message server has two parameters named *reqTrq* and *vspd* which are the requested braking torque and the estimated vehicle speed, respectively. In this message server first the slip rate, denoted by the variable *slprt*, is calculated. Here the constant *WRAD* denotes the radius of the wheel. After calculating the slip rate, if the value of *slprt* is greater than 0.2, it sets its wheel's brake torque to zero by using the *setTrq* message. Otherwise it sets the wheels brake torque to the requested torque.

In the main block, two rebecs of type *Wheel* and two rebecs of type *WCtlr* are instantiated. Here we explain one instantiation of each class. The term *Wheel wR (@Wire wctlrR) : (1);* is used to instantiate a rebec named *wR* of type *Wheel*. The known rebec of *wR* is assigned to *wctlrR*. The tag *@Wire* is used to indicate that the communications of this rebec to *wctlrR* are via wire. The parameter (1) is the parameter of the *initial* message server of the rebec, which is the initial speed of the wheel. The term *WCtlr wctlrR (@Wire wR, @CAN bctlr) : (0);* is used to instantiate a rebec named *wctlrR* of type *WCtlr*. The known rebecs of *wctlrR* are assigned to *wR* and *bctlr*. For the second known rebec, the tag *@CAN* is used to indicate that the communications of this rebec to *bctlr* are via CAN. The parameter (0) is the identifier of the wheel controller. In the priorities of CAN specification, the term *wctlrR bctlr.setWspd 3*, specifies that the priority of a message sent from *wctlrR* to *bctlr* containing the message server name *setWspd* is 3. The term *wctlrR bctlr.setWspd -> 0.01* in the delays of CAN specification, specifies that the delay of the same message is 0.01.

4.2 Operational Semantics

Rebecs are executed concurrently in response to a physical mode being finished or by taking a message from their message queues. The actions of a physical mode are executed when its behavior is finished, and each message is processed 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. The message selection mechanism of CAN protocol is time consuming. We assume that this time is negligible. We abstract away from this time by considering the effect of the network entity when no actor can progress instantaneously. For communication via wire, the message is directly inserted into the receiver's message queue instantaneously.

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 either be CAN or wire, the message priorities and message delivery delays.

Definition 1 (Hybrid Rebeca model). *A Hybrid Rebeca model is defined as a 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 $r_{s_i} \in R_s$ and physical rebec $r_{p_i} \in R_p$ with a unique identifier i, are defined by tuples $(i, V_i, msgsrvs_i, K_i)$ and $(i, V_i, msgsrvs_i, modes_i, K_i)$, respectively, where V_i is the set of its variables, $msgsrvs_i$ is the set of its message servers, K_i is the set of its known rebecs, and $modes_i$ is the set of modes.

A network specification is defined as a tuple $N = (conn, netPriority, netDelay)$ where conn is a partial function $R \times R \to \{Wire, CAN\}$ which defines the one way connection type from a rebec to another rebec, $netPriority : Msg \to \mathbb{N}$ and $netDelay : Msg \to \mathbb{R}$ define the priority and the network delivery delay for a message, respectively. Msg denotes the set of all messages in the model.

Definition 2 (Message). *A message is defined as a tuple $(sender, m, receiver) \in R \times Name \times R$ where sender is the sending rebec, m is the name of the message server in the receiver, and receiver is the receiving rebec.*

The operational semantics of a Hybrid Rebeca model is defined as a monolithic hybrid automaton. The semantics could be defined in a compositional way by providing a translation for constitutive elements of a hybrid rebeca model. By composing the translations with the operational semantics of hybrid automata, the final model can be derived. However, this approach will lead to many real-valued variables which reduces analyzability of the resulting model.

Definition 3 (Hybrid automaton for 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 has four entities: the states of software rebecs, physical rebecs, network, and pending event list. The state of a software rebec consists of the valuation of its discrete variables, the state of its message queue and a program counter. The program counter points to a statement that the rebec must execute. The state of a physical rebec consists of its active mode, the state of its message queue and a program counter. 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. 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. Delay statements are not allowed in physical rebecs.

Definition 4 (State of a rebec). *The state of a software rebec is denoted by the tuple (v, q, c) where v is the valuation of its variables, q is the message queue of the rebec, and c denotes the program counter. The state of a physical rebec is a tuple of the form $(\mathfrak{M}, q, c)$ where $\mathfrak{M}$ is the active mode and q and c are defined as in the software rebec's state.*

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 sending a message or is ready to send one.

Definition 5 (State of network). *The network state is defined by the pair (B, r), where B is the network buffer and the boolean flag r indicates the status of the network, which can be ready or busy.*

The forth entity, pending event list, represents the sequence of pending events. Events are used for time consuming actions. For a time consuming action an event is stored, to be triggered at the time that the action is over. Two types of events are defined in the semantics of Hybrid Rebca: *Resume* and *Transfer*. A pending event with event *Resume* is generated and inserted into the pending event list when a delay statement is seen in Rebeca model, and the corresponding rebec is suspended. To model the passage of time for the delay statement, a timer variable is used by 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. A pending event with event *Transfer* is generated when a message from the network buffer is chosen to be sent 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.

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 variable that is assigned to this event. The event e can either be a Resume or Transfer event. The timer variable is 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.*

Transitions We define two general types of transitions: urgent and nonurgent transitions. The urgent transitions are further divided to message, statement and network transitions which are respectively shown as $\Rightarrow_m$, $\Rightarrow_s$ and $\Rightarrow_n$. The nonurgent transitions are shown as $\Rightarrow_N$. We use these 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. This transition is only about choosing the message and not the act of sending. Non-urgent transitions are used to model the passage of time. These transitions include the behaviors of physical actors' active modes and pending time of events since they are time consuming.

An ordering is defined among these transitions. The ordering is $\Rightarrow_m = \Rightarrow_s > \Rightarrow_n > \Rightarrow_N$. Whenever a higher order transition is enabled in a location, no lower order transition can be taken in that location. The semantics of the actions in Hybrid Rebeca are defined using these transitions on the locations. In the following we define these transitions.

Message Transitions: Message transitions define the act of taking a message. A message transition can take place whenever a rebec is not suspended. A rebec is suspended when it executes a delay statement. Let the tuples (v, q, c) and $(\mathfrak{M}, q, c)$ denote the state of a software rebec and a physical rebec respectively. The message transition is defined as follows:

- *Taking a Message:* A rebec takes a message from the head of its message queue q, whenever the rebec has no statement to execute. When a message is taken, the program counter c is updated to point to the beginning of the corresponding message server, and the message is removed from the message queue q.

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

- *Assignment Statement:* This statement has two cases. When assigning to a discrete variable, the value of the variable is update in the valuation v 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 transfered over to the transition to be handled by the resulting hybrid automaton.
- *Conditional Statement:* This statement has two cases. If the value of the condition is determined, the program counter c is updated to point to the appropriate statement block. 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.
- *Send Statement:* This statement, depending on the communication type, has two behaviors. 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.
- *Delay Statement:* This statement suspends the software rebec by updating the corresponding variable (suspension status) in the valuation v and creates a pending event $(d, Resume, t)$ for resuming the rebec after d units of time. d is the delay specified in the delay statement and t is a timer variable.
- *Set Mode Statement:* This statement changes the active mode $\mathfrak{M}$ of the physical rebec to the specified mode.

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. The choosing transition is as follows:

- *Choosing a Message:* For this behavior, the message with the highest priority is removed from the network buffer B, a pending event $(d, Transfer, t)$ 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 of the message.

Nonurgent Transitions: Nonurgent 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 $(\mathfrak{M}, q, c)$ if $\mathfrak{M}$ is not none, the guard of the active mode $\mathfrak{M}$ is transfered to the transition, and the program counter c of the rebec is updated to point to the actions of this active mode, and the active mode is set to none.
- *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.

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. The functions $flows(r_p, \mathfrak{M})$ and $invariant(r_p, \mathfrak{M})$ return the flow and invariant of a mode $\mathfrak{M}$ in a physical rebec r_p respectively. Instantaneous behaviors 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.

Definition 7 (Urgent location flow and invariant). *A possible implementation for an urgent location is urg$' = 1$ as flow and urg $<= 0$ as 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 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, the flows for timers of pending events, and a flow of zero for each float variable to freeze its value. The flow of a pending event is simply defined as $t' = 1$ where t is the timer variable of the pending event.

Similarly, if a location is urgent, its invariant is set to urgency invariant, otherwise it inherits the invariants of all physical rebecs' active modes, and the invariants for corresponding pending events' timers. 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.

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 variable to zero in the initial condition of the initial location.

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. But in practice a specific size must be specified for message queues of rebecs.

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 during the translation to hybrid automata, so it's 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.

5   Case Study

We demonstrate the applicability of our language on a simplified Brake-by-Wire (BBW) system with Anti-lock Braking System (ABS) [12,16,8]. 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 controllers 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. Furthermore, 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. A representation of the system is shown in Figure 4.
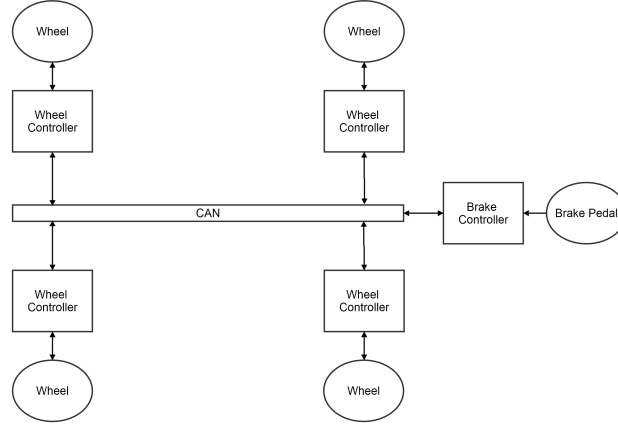
Fig. 4. The schematic of the BBW system. The physical components are shown as ellipses and computational components are shown as rectangles.

5.1   Model Definition

The model is defined in Fig. 5. Note that For simplicity, we have considered two wheels in our model. The model consists of 5 classes. *Wheel* and *WCtlr* classes were defined previously in 4.1. *Wheel* class models the sensors and actuators of the wheel. It periodically sends the speed of the wheel to the controller and defines the effect of braking on the wheel speed. *WCtlr* 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.

   Brake class defines the behavior for the brake pedal. Here we assume a simple behavior where the brake percentage is increased by a constant rate until it reaches a predefined max percentage. The class have 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. 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*. *Braking* 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.

   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*. *BrakeCtrl* 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 estimated speed of the vehicle is computed 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. The *initial*, *setWspd* and *setBprcnt* message servers are omitted for brevity. The *setWspd* message server

updates the correct 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 *wR* and *wL* are wired to their respecting wheel controllers by using the tag @Wire. Both wheels are initialized with the speed of 1^4. The wheel controllers *wctlrR* and *wctlrL* 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 *wctlrR* and *wctlrL* 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 *wctlrR* and *wctlrL*. The wheels *wR* and *wL* send *setWspd* message to *bctlr* 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 milliseconds.

5.2   Analysis and Verification

For the analysis of this model the queue size of *bctlr* is set to 4, the queue sizes of both *wctlrR* and *wctlrL* 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 variable pool is set to 11. The hybrid automaton derived from the model consists of 10097 locations and 25476 transitions. We use SpaceEx [9] tool to verify our model. Note that the slip rate equation used in the model is not supported by SpaceEx, since it's a nonlinear equation. We simplified this equation for analysis. By specifying a set of forbidden states, safety properties can be verified by SpaceEx. We developed an initial tool[5] for our language that automatically translates a Hybrid Rebeca model to a SpaceEx model based on the semantics of our language. We verified three properties for our case study. The first property is "design-fault freedom". Here by design-fault we mean a fault caused by following situations: exceeding the capacity of a message queue, running out of pooled variables, and having messages with same priority in CAN buffer. We assume the message with the highest priority must always be unique in CAN buffer. Note that in practice, this property must be implicitly checked for all models in Hybrid Rebeca, but for now it must be manually verified by SpaceEx. 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 seconds. The third property states that whenever the slip rate of a wheel exceeds 0.2, the brake actuator of that wheel must immediately be released.

---

[4] 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.
[5] The implementation can be found in https://github.com/jahandideh-iman/HybridRebeca

```
 1  physicalclass  Wheel{
 2   knownrebecs {WCtlr ctlr;}
 3   statevars {float  trq;  real
           spd;  real  t;}
 4   msgsrv  initial ( float  spd_){
 5    spd = spd_;
 6    setmode(Rolling);
 7   }
 8   msgsrv setTrq(float  trq_){
 9    trq = trq_;
10   }
11   mode Rolling{
12    inv(t <= 0.05){
13     t' = 1;
14     spd' = −0.1−trq;
15    }
16    guard(t == 0.05){
17     t = 0;
18     ctlr .setWspd(spd);
19     if (spd > 0)
20      setmode(Rolling);
21  }}}

23  softwareclass  WCtlr{
24   knownrebecs {Wheel w;
         BrakeCtlr bctlr;}
25   statevars  {int  id;  float
         wspd; float  slprt ;}
26   msgsrv  initial ( int  id_){
27    id = id_;
28   }
29   msgsrv setWspd(float
         wspd_){
30    wspd = wspd_;
31    bctlr .setWspd(id,wspd);
32   }
33   msgsrv applyTrq(float
         reqTrq, float  vspd){
34    if (vspd == 0)
35     slprt  = 0;
36    else
37     slprt  = (vspd − wspd ∗
          WRAD)/vspd;
38    if ( slprt  > 0.2)
39     wheel.setTrq(0);
40    else
41     wheel.setTrq(reqTrq);
42  }}
```

```
44  physicalclass  Brake{
45   knownrebecs {BrakeCtlr
         bctlr;}
46   statevars  {real  bprcnt;  real
         t;  float  mxprcnt;
         float  r}
47   msgsrv  initial ( float
         bprcnt_, float
         mxprcnt_){
48    bprcnt = bprcnt_;
49    mxprcnt = mxprcnt_;
50    r = 1;
51    setmode(Braking);
52   }
53   mode Braking{
54    inv(t <= 0.05){
55     t' = 1;
56     bprcnt' = r;
57    }
58    guard(t == 0.05){
59     t = 0;
60     bctrl .setBprcnt(bprcnt);
61     if (bprcnt>=mxprcnt)
62      r = 0;
63     setmode(Braking);
64  }}}

66  softwareclass  BrakeCtlr{
67   knownrebecs{
68    WCtlr wctlrR;WCtlr
         wctlrL;}
69   statevars  {float
         wspdR;float
         wspdL;float
         bprcnt;float  gtrq; float
         espd;}
70   msgsrv control(){
71    espd = (wspdR +
         wspdL)/2;
72    gtrq = bprcnt;
73    wctlrR.applyTrq(gtrq, espd);
74    wctlrL.applyTrq(gtrq, espd);
75   }
76   // Setters for wspdR,
         wspdL  and  bprcnt
77    ...
78  }

80  physicalclass  Clock{
```

```
 81  knownrebecs {BrakeCtlr
         bctlr;}
 82  statevars  {real  t;}
 83  msgsrv  initial (){
 84   setmode(Running)
 85  }
 86  mode Running(){
 87   inv(t <= 0.05){
 88    t' = 1;
 89   }
 90   guard(t == 0.05){
 91    t = 0;
 92    bctlr .control ();
 93    setmode(Running);
 94  }}}

 96  main {
 97   Wheel wR (@Wire
         wctlrR):(1);
 98   Wheel wL (@Wire
         wctlrL):(1);
 99   WCtlr wctlrR (@Wire wR,
         @CAN bctlr):(0);
100   WCtlr wctlrL (@Wire wL,
         @CAN bctlr):(1);
101   BrakeCtlr bctlr  (@CAN
         wctlrR, @CAN
         wctlrL):();
102   Brake brake(@Wire
         bctlr):(60,65);
103   Clock clock(@Wire bctlr):();

105   CAN{
106    priorities {
107     bctlr  wctlrR.applyTrq  1;
108     bctlr  wctlrL.applyTrq  2;
109     wctlrR bctlr .setWspd  3;
110     wctlrL bctlr .setWspd  4;
111    }
112    delays{
113     bctlr  wctlrR.applyTrq
          0.01;
114     bctlr  wctlrL.applyTrq
          0.01;
115     wctlrR bctlr .setWspd
          0.01;
116     wctlrL bctlr .setWspd
          0.01;
117  }}}
```

Fig. 5. The model definition of the BBW model.

For the first property, a specific location, called *Fault*, is created for the mentioned situations, and during the semantic derivation, occurrences of such design-faults are handled by generating a transition to the specified location *Fault*. The verified forbidden condition in SpaceEx for this property is *loc*() == *Fault*, where the term *loc*() specifies the current location in SpaceEx.

The second property can not directly be specified with a set of forbidden states, since there is no direct concept of time in hybrid automata. To this aim, a monitor class is added to the model to measure the time between two events. Here the events are sending the brake percentage from the brake pedal to the global brake controller and processing the received brake torque in the wheels. The monitor class is a simple physical class with one physical mode and two stop and start message

servers. The physical mode tracks the time and the message servers are used to stop and start the tracking. Note that in the start message server the tracked time is reset.

For the second property, one monitor rebec is instantiated and is wired to the brake pedal and one of the wheels. A start message is sent in the actions of the *Braking* mode of the brake pedal after the brake percentage is sent to the brake controller and a stop message is sent by *setTrq* message server of the wired wheel. In SpaceEx the forbidden condition *monitor_time* > 0.2 is verified where *monitor_time* is the name of the monitor's timer in the resulted hybrid automaton.

The third property may seem to be straightforward, but because the semantics of our language is fine-grained, the request of releasing the brake and the actual act, take place in different locations, even though the time does not advance between these locations because they are urgent locations. For this property, the monitor class is used again. This time, a monitor rebec is wired to one of the wheels and its wheel controller. A start message is sent from the wheel controller when the slip rate is greater than 0.2 and a stop message is sent like the second property. By using the monitor rebec, the states which the brake is not released and time has not progressed while the slip rate is greater than 0.2 can be considered safe. The verified forbidden condition in SpaceEx is

$$wctlr\_slprate > 0.2 \wedge wheel\_torque > 0 \wedge monitor\_time > 0$$

where *wctlr_slprate*, *wheel_torque* and *monitor_time* are the names of the slip rate of the wheel, the brake torque of the wheel and the timer of the monitor, respectively, in the resulted hybrid automaton.

The resulted hybrid automaton for the first property has 10097 locations and 25476 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 properties mentioned here. After aggregating these urgent locations, the size of the resulting hybrid automaton is reduced to 21 locations and 1148 transitions. The aggregation process is implemented in our tool. The three properties are verified on their respective reduced hybrid automaton. The verification result of these properties are provided in the Table 1.

Property	Derived HA		Gen Time (s)	Red HA		Verif Result	Verif Duration (s)
Design-fault Freedom	10097	25476	12	21	1148	Passed	3705
Reaction Time	16317	42976	20	21	1168	Passed	7521
Brake Release	54097	175036	64	21	1168	Passed	3541

Table 1. The verification result of the case study. Legends: **Property**: verified property, **Derived HA**: derived hybrid automaton size where the first and second columns are the number of locations and transitions, respectively, **Gen Time**: duration of hybrid automaton generation in seconds, **Red HA**: reduced hybrid automaton size, **Verif Result**: result of verified property, **Verif Duration**: duration of verification in seconds.

6   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.

Ptolemy II [19] relies on simulation for analysis and 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. In [4] an agent-based and control centric methodology 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. In [17] a modular approach for specifying and validating CPSs using rewriting logic-based technique is purposed. 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 [5] can also be used to model hybrid systems and CPSs. For analysis of these hybrid Petri nets in [5] a translation to hybrid automata is presented. However, Petri-net based approaches prohibit modular specification of systems. The framework of [13] provides a hybrid process calculus tailored for modeling CPSs and analyzing their security properties [14,15]. In this approach network governing the interactions between physical and cyber entities is not addressed.

7   Discussions

In Section 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 software 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. 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 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.

8   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, to allow for formal verification of the 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.

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's also possible to allow for user-defined network models. To this aim, a set of basic functionalities must be defined to enable defining most network models. Also, defining multiple instances of a network model (e.g. multiple CAN networks) may be needed in some systems. Like network models, it's possible to allow for defining new internal message schedulers for the rebecs, since the FIFO scheduler can be inadequate for some systems.

Acknowledgments

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 (1990)
3. 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. Theoretical Computer Science **138**(1), 3–34 (1995)
4. Cicirelli, F., Nigro, L., Sciammarella, P.F.: Model continuity in cyber-physical systems: A control-centered methodology based on agents. Simulation Modelling Practice and Theory **83**, 93–107 (2018)
5. David, R., Alla, H.: On hybrid petri nets. Discrete Event Dynamic Systems **11**(1-2), 9–40 (2001)
6. Davis, R.I., Burns, A., Bril, R.J., Lukkien, J.J.: Controller area network (CAN) schedulability analysis: Refuted, revisited and revised. Real-Time Systems **35**(3), 239–272 (2007)
7. Derler, P., Lee, E.A., Sangiovanni-Vincentelli, A.L.: Modeling cyber-physical systems. Proceedings of the IEEE **100**(1), 13–28 (2012)

8. 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)
9. 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)
10. 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)
11. Hewitt, C.: Description and theoretical analysis (using schemata) of planner: A language for proving theorems and manipulating models in a robot. Tech. rep., Massachusetts Institute of Technology, Artificial Intelligence Laboratory (1972)
12. 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. Reliability Engineering & System Safety **120**, 127–138 (2013)
13. 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)
14. 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)
15. 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. to appear (2018)
16. 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)
17. 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)
18. Pfeiffer, O., Ayre, A., Keydel, C.: Embedded Networking with CAN and CANopen. Copperhill Media Corporation, 1st edn. (2008)
19. Ptolemaeus, C. (ed.): System Design, Modeling, and Simulation using Ptolemy II. Ptolemy.org (2014)
20. 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)
21. Sirjani, M.: Power is overrated, go for friendliness! expressivness versus faithfulness and usability in modeling-actor experience. In: Edward A. Lee Festschrift. pp. 1–21. LNCS, Springer (2018)
22. 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)
23. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using rebeca. Fundamenta Informaticae **63**(4), 385–410 (2004)
24. 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)
25. Wolf, W., Madsen, J.: Embedded systems education for the future. Proceedings of the IEEE **88**(1), 23–30 (2000)
26. 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)