Actors Upgraded for Variability, Adaptability, and Determinism

Ramtin Khosravi[1(✉)], Ehsan Khamespanah[1], Fatemeh Ghassemi[1], and Marjan Sirjani[2]

[1] School of ECE, University of Tehran, Tehran, Iran
{r.khosravi,e.khamespanah,fghassemi}@ut.ac.ir
[2] School of IDT, Mälardalen University, Västerås, Sweden
marjan.sirjani@mdu.se

Abstract. The Rebeca modeling language is designed as an imperative actor-based language with the goal of providing an easy-to-use language for modeling concurrent and distributed systems, with formal verification support. Rebeca has been extended to support time and probability. We extend Rebeca further with inheritance, polymorphism, interface declaration, and annotation mechanisms. These features allow us to handle variability within the model, support non-disruptive model evolution, and define method priorities. This enables Rebeca to be used more effectively in different domains, like in Software Product Lines, and holistic analysis of Cyber-Physical Systems. We develop specialized analysis techniques to support these extensions, partly integrated into Afra, the model checking tool of Rebeca.

Keywords: Actor Languages · Variability Modeling · Cyber-Physical Systems · Model Checking

1 Introduction

The Actor model of computation was first proposed by Carl Hewitt in the 1970s [29], and further developed by Gul Agha [3], as a mathematical framework for concurrent and distributed computing systems. The model describes computation as a collection of autonomous entities called actors that encapsulate their states and communicate with each other by sending messages [30]. Actors have been used as a framework for theoretical understanding of concurrent and distributed computation, as the basis for designing many modeling and programming languages, and as a model for many practical implementations of concurrent systems [12,26].

Rebeca (standing for *Re*active O*bjec*ts L*anguage*) is an actor-based modeling language with model checking support designed in 1999–2001 [65,66]. One of the main design decisions in creating Rebeca is to keep the core language as simple as possible. One can still use core Rebeca for modeling using a small set of features for coding. However, Rebeca is extended to work for timed systems [40] and

F. de Boer et al. (Eds.): *Active Object Languages: Current Research Trends*, LNCS 14360, pp. 226–260, 2024.
https://doi.org/10.1007/978-3-031-51060-1_9

address probability [32]. Timed Rebeca is used for the modeling and analysis of several applications [42,60,61,71]. In order to model complex systems the language is evolved in different directions [12]. A brief overview of Timed Rebeca language features is presented in Sect. 2. Rebeca is equipped with an integrated modeling and analysis tool, Afra, which provides LTL model checking for Rebeca as well as schedulability and deadlock-freedom analysis, and assertion check for Timed Rebeca [41].

In the new era of digitalization, smart factories, and systems of cyber-physical systems we are dealing with heterogeneous and dynamic systems. This introduces different types of variability in behavior, including those arising from different contexts in which the model is used (which is common in software product lines [51]), the need to dynamically adapt to the changes in the environment and in the system itself at runtime (like in self-adaptive and reconfigurable systems [70]), and the combination of these two types of variability (as in dynamic software product lines [4,31]). Hence, for a modeling language to address these requirements, it needs proper linguistic constructs to capture the variability in the behavior in a structured way. Current trends in the research community of software-intensive cyber-physical systems also confirm this [22,43,52].

When coping with systems of cyber-physical systems, we have to consider aspects of embedded and real-time systems together with complexities in concurrent and distributed systems[1]. In distributed and concurrent systems we are faced with uncertainties mainly caused by the network. The mainstream approach in the concurrency theory community uses nondeterminism to model concurrency. While the uncertainties in the environment may remain, we can aim for a deterministic design for the behavior of the system itself which is crucial for embedded and real-time systems communities. The recent work of Edward Lee and his group on the coordination language, Lingua Franca, shows one direction focusing on determinism [46,47] and PLC-like semantics [59].

Since its introduction, Rebeca has been used to model adaptive behavior in various domains, such as self-adaptive systems [38,39] and flow management systems [24]. Also, it has been used in [56] to model and analyze dynamic software product lines. However, until recently, Rebeca has not been equipped with special language features to support variable and adaptive behavior in a structured way.

The purpose of this paper is to demonstrate how the recent extensions of Timed Rebeca can be used by a wider community to model and analyze real-world applications, with a focus on how and where the language features can be used. The language extensions presented in this paper are summarized below.

Feature annotations as an explicit variability handling mechanism which are used to bind parts of the model to specific products in a software product line. This language extension is presented in this paper for the first time.

[1] Cyber-physical systems are also hybrid systems, bringing together cyber and physical components which are generally modeled differently. The interface between the cyber and the physical parts is also a source of complexity and an important research area that is not a topic of interest in this paper. This matter is addressed in Hybrid Rebeca introduced in [34].

Inheritance, Interfaces, and Polymorphism as language features that can be used to support variability in model in a structured way. These features have been added to Timed Rebeca in [71] and are used to support alternative communication schemes among actors. In this paper, we present them as a variability handling mechanism.

Priorities for actors and for message handlers to make the behavior of the system more deterministic. This feature enables better modeling and verification of different types of cyber-physical systems. Priority in Timed Rebeca has been introduced in [64] which illustrates through a few examples how Lingua Franca code can be naturally mapped to Timed Rebeca extended with priorities.

We also provide three case studies to demonstrate the applicability of the above-mentioned language extensions in modeling real-world systems in practice. We have also verified the models for schedulability and deadlock-freedom and demonstrated how the Afra toolset is capable of analyzing systems for relatively large state spaces (with more than 37 million states) on a personal computer in a reasonable time (Sect. 3.6).

After a brief overview of Timed Rebeca we review the upgrades to the language (and the analysis tool) to support systematic variability management (Sect. 3) and illustrates their applicability in a case study (Sect. 3.6). We explain how Rebeca is extended to include priorities for actors and for message handlers to address the need for determinism in the model (Sect. 4) and demonstrate its applicability using a case study (Sect. 4.3). Afra provides complete support for this feature in the modeling and analysis of Timed Rebeca models. In the last section (Sect. 5), we explain how we can put together both features supporting variability and priority and hence support the possibility of a holistic analysis for modern cyber-physical systems. We may then formally verify the model to check safety properties as well as schedulability and end-to-end timing properties.

2 Rebeca Overview

Rebeca [62,67] is a class-based, imperative interpretation of the well-known actor model of computation [3]. It describes the behavior of a system as a collection of active objects with isolated states, communicating via asynchronous message passing. Rebeca is a strongly typed modeling language with a Java-like syntax to make it easy to learn and use by practitioners. It is equipped with an LTL model checker integrated into Afra [2], an Eclipse-based development environment. The core Rebeca modeling language is intentionally kept simple, but for various purposes, several extensions have been proposed, including Timed Rebeca [63] for the domain of real-time systems, Hybrid Rebeca [34] for the domain of cyber-physical systems, pRebeca [69] for modeling and analysis of probabilistic systems, and PTRebeca [32] for probabilistic timed systems.

2.1 Running Example

To make our explanation of Rebeca and Timed Rebeca easier to follow, we explain the language features over a simple running example. The example is a highly simplified version of a Wireless Sensor LAN (WSLAN) system [42], in which a sensor periodically gathers and sends data to a computation unit. The computation unit buffers the received data and hands in a packet of data to a network whenever the buffer is full. The network transmits the data according to the TDMA network protocol [14].

2.2 Core Rebeca

A Rebeca model mainly consists of a number of *reactive class* definitions, which define the behavior of the classes of the actors in the model, as well as a `main` block that defines the instances of the actor classes. In the Rebeca model of the running example listed in Fig. 1, there are three classes of actors: `Sensor` (lines 1–16), `CompUnit` (lines 18–30), and `Network` (lines 32–37). The `main` block in lines 39–43 defines one instance of each class and specifies the arguments passed to their constructors. An instance of a reactive class is an actor in the system (which is also called a *rebec*).

The declaration of a reactive class starts with the keyword `reactiveclass`, followed by the reactive class name. The size of the queue is specified in the parentheses right after the reactive class name (e.g., line 1). A reactive class has a number of *state variables*, representing the local state of the actors. They

```
1  reactiveclass Sensor(3) {
2    statevars{
3      CompUnit cu;
4    }

6    Sensor(CompUnit cu1) {
7      cu = cu1;
8      self.gatherData();
9    }

11   msgsrv gatherData() {
12     byte data = ?(1,3);
13     cu.receiveData(data);
14     self.gatherData();
15   }
16 }

18 reactiveclass CompUnit(3) {
19   statevars {
20     Network network;
21   }

23   CompUnit(Network net) {
24     network = net;
25   }

27   msgsrv receiveData(byte data) {
28     network.send(data);
29   }
30 }

32 reactiveclass Network (3) {
33   msgsrv send(byte data) {
34     // Send data according
35     // to the TDMA protocol
36   }
37 }

39 main {
40   Sensor sensor():(cu);
41   CompUnit cu():(network);
42   Network network():();
43 }
```

Fig. 1. The Rebeca model of the running example (a simple sensor network)

may contain variables of basic data types, including booleans, integers, arrays, or references to other actors. The classes in the running example only contain state variables of the reference types. For example, every instance of `Sensor` has a reference to an instance of `CompUnit` (line 3). Each class can have a number of *constructors*, which are used to initialize instances of the class by initializing the state variables and possibly sending messages to other actors or themselves. For example, the constructor of `Sensor` (lines 6–9) initializes a sensor by setting its reference to `CompUnit` as well as sending itself a `gatherData` message. Each reactive class accepts a number of message types which are handled using *message servers*[2]. The message server `gatherData` of `Sensor` (lines 11–16) first chooses a data value in the range 1 to 3 nondeterministically (line 12) and sends a `receiveData` message to the associated `CompUnit` (denoted by the reference variable `cu`), passing the value of `data` as the argument (line 13)[3]. The effect of sending a message is appending the message to the message queue of the receiving actor (sometimes called its mailbox). Sending a `gatherData` to itself (line 14), the sensor exhibits a periodic behavior. In the definition of the message servers, well-known program control structures can be used, including *if-else* conditional statements, *for* and *while* loops, the definition of local variables, and assigning expressions built using usual arithmetic, logic, and comparative operators to local and state variables.

The general behavior of each actor is an infinite loop of taking a message from the mailbox and executing the corresponding message server. The actor waits if there is no message in the mailbox. The mailbox is a bounded FIFO queue. The queue size is bounded to prevent infinite state spaces during model checking. If a message is sent to an actor with a full mailbox, a *queue overflow* error happens and the state space generation is terminated. As we will see shortly in more detail, the model in Fig. 1 suffers from this problem when the sensor repeatedly sends itself `gatherData` messages. To remedy this, the sensor can send the next `gatherData` only after receiving some kind of acknowledgement message from the computation unit. Another solution is to use timing constraints introduced in Timed Rebeca.

It is important to note that in Rebeca there is no intra-actor concurrency, meaning that the execution of a message server must complete before the executing actor takes the next message from its mailbox. To make the behavior of the models more deterministic, we assume that two messages sent from one actor to another are delivered to the receiver's mailbox in order. The order of execution of enabled actors are arbitrary. An actor is enabled if it is not busy handling a message and its message queue is not empty. This arbitrary ordering of actors is a source of nondeterminism in the behavior of the model, requiring the model checker to inspect all possible interleavings of the message processing by different actors.

[2] In this paper we use the words *message server* and *method* interchangeably.

[3] Note that this data value has no effect on the behavior of the actors in this specific model and is only generated to demonstrate the use of nondeterministic choice expression.

2.3 Timed Rebeca

The models in core Rebeca are time abstract in the sense that the passage of time is not modeled explicitly. The nondeterminism in the processing of messages by different actors implicitly models the temporal ordering of events. For example, upon execution of `gatherData`, the sensor sends two messages: a `receiveData` to `cu` and another `gatherData` to itself. Now if the next message processed is `receiveData`, this indicates that the sensor gathers data in a time period relatively larger than the time needed by the computation unit to process the data (including the time needed to receive the message from the sensor). Conversely, if `gatherData` is processed first, it indicates that the sensor gathers data relatively faster. If this case happens routinely, both the sensor's and the computation unit's mailboxes quickly overflow.

To put constraints on the timings of delivering and processing of the messages, we can use an extension of Rebeca, named Timed Rebeca, which provides features for this purpose. Rewriting the two mentioned message servers as below fixes the queue overflow problem.

```
msgsrv gatherData() {
  byte data = ?(1,3);
  cu.receiveData(data);
  self.gatherData() after(2);
}
```

```
msgsrv receiveData(byte data) {
  delay(1);
  network.send(data);
}
```

The clause `after(2)` after sending `gatherData` specifies the message needs two units of time to be delivered to the mailbox of the sensor, hence specifying the time period of two for gathering data. On the other hand, `delay(1)` statement in `receiveData` indicates that the computation unit needs one unit of time to process the message and send the data over the network. Timed Rebeca offers the following features to model the timed behavior of actors.

delay is a statement used to model computation times. Timed Rebeca assumes all statements other than delays are executed instantaneously. So, the computation time must be specified by the modeler using the delay statement. A statement `delay(`t`)` indicates the actor does not perform any action within the next t units of time.

after is a time tag attached to a message and defines the earliest time the message can be served, relative to the time when the message was sent. A clause `after(`t`)` may be added to a message send statement, indicating that the receiver can take the message from its mailbox only after t units of time.

deadline is a time tag attached to a message which determines the expiration time of the messages, relative to the time when the message was sent. A clause `deadline(`t`)` may be added to a message send statement, indicating that the message remains only t units of time in the receiver's mailbox, and purged afterward if its processing has not already started.

The same as Core Rebeca, the order of execution of enabled actors in Timed Rebeca are arbitrary. In Timed Rebeca, an actor is enabled if it is not busy

handling a message and its message bag has a message whose time tag is less than the time tag of all the messages of other actors. This message is also called an enabled message. Timed Rebeca is also supported by Afra toolset for schedulability and deadlock freedom analysis. It makes use of special properties of the Timed Rebeca semantics (isolated actor states and serial execution within a single actor) to generate a data structure called *floating time transition system* which enables a coarse grain discretization of the state space [40].

2.4 Inheritance and Polymorphism

Like most other object-oriented programming and modeling languages, Rebeca provides mechanisms for reusing code through subclassing. A modeler is able to define a new reactive class as a subclass of an existing reactive class, using an inheritance mechanism. This is stated using the `extends` keyword followed by the name of the base reactive class, prior to the queue size declaration. This way, the new reactive class inherits all the state variables and message servers of the base reactive class. Rebeca also supports polymorphism through dynamic binding of the message servers. Since a subclass cannot remove any message server inherited from its superclass, its type is compatible with that of the superclass. Hence, it is possible to assign an instance of a subclass to a reference of the superclass. The actual message server invoked when processing a message is determined by the class of the receiving actor (not the type of the reference). This allows for improving code organization and readability as well as the creation of extensible programs [21]. An example of the usage of inheritance and dynamic binding in Rebeca is demonstrated in the Elevator case study (Sect. 3.6, Fig. 7).

An abstract reactive class is defined when a modeler wants to manipulate a set of classes through their common interface. Rebeca provides this by enabling `abstract` message server definition. An abstract message server has only a declaration and no implementation. A reactive class containing abstract message servers is called an abstract reactive class. Inheriting from an abstract reactive class requires providing definitions for all the abstract message servers in the base reactive class. Otherwise, the derived reactive class is also abstract, and the compiler forces the modeler to qualify that reactive class with the `abstract` keyword.

In some cases, there is a need for defining a completely abstract reactive class, i.e., a reactive class that provides no implementation at all. This is done by defining `interface` instead of reactive classes. It allows the modeler to determine message server names and their argument lists, but no bodies and no state variables. So, it provides only a type, not any implementation. In Rebeca, defining multiple interface implementation is allowed, which can be assumed as a kind of multiple inheritances. More details about the inheritance mechanism and polymorphism in Rebeca are presented in [71]. An example of using interfaces in Rebeca is illustrated in Fig. 3, lines 62–78.

3 Modeling Variability in Rebeca

In this section, we review the language features that can be used to capture variability in Rebeca models. At the finer level of granularity, we have feature annotations that can bind state variables, methods, and statements to feature expressions. On the other hand, polymorphism allows reactive classes to act as different implementations of abstract interfaces, hence providing a coarse-grained variability handling mechanism at the component level. Before going into the details of each language feature, we extend the running example with a few variable features.

3.1 Running Example with Variability

To demonstrate how variability is handled in Rebeca, we extend the running example with a few variable features. The feature diagram of the extended example is illustrated in Fig. 2. The whole system (represented by WSAN) has three sub-features Sensor, Computation Unit, and Network. The filled circle at the top of these features indicates that they are mandatory sub-features of WSAN, meaning that they must be included in every product configuration. There are three variation points in this example. The sensor can gather data with a fixed period, or sporadically. The arc between the edges to Periodic and Sporadic indicates that these sub-features are mutually exclusive. The computation unit can either immediately send the data received from the sensor, or decouple receiving and sending data. In the latter case, it buffers the received data and periodically sends a packet from the buffer (if available). Finally, the system can support the network protocols TDMA, MACB, or both (as indicated by the filled arc between the edges to the sub-features). The Timed Rebeca model of the extended running example is listed in Fig. 3. We will explain the details of the variability handling mechanisms in the following.

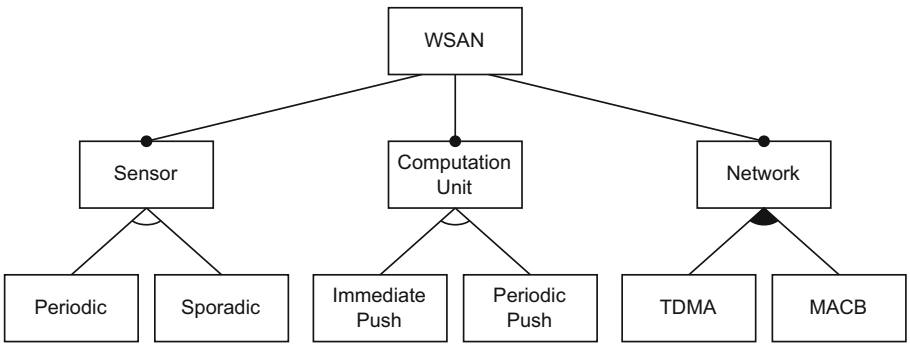


Fig. 2. The feature diagram of the running example of a simple sensor network with three variation points, periodic or sporadic for the sensor, immediate or periodic push for the computation unit, and two different protocols, TDMA and MACB, for the network

```
1  featurevar FT_PERIODIC_SENSOR;
2  featurevar FT_SPORADIC_SENSOR;
3  featurevar FT_IMMEDIATE_PUSH;
4  featurevar FT_PERIODIC_PUSH;
5  featurevar FT_SIMPLE_NETWORK;
6  featurevar FT_TDMA_NETWORK;

8  reactiveclass Sensor(3) {
9    statevars{
10     CompUnit cu;
11   }

13   Sensor(CompUnit cu1) {
14     cu = cu1;
15     self.gatherData();
16   }

18   msgsrv gatherData() {
19     cu.receiveData(0);
20     if (FT_PERIODIC_SENSOR)
21       self.gatherData() after(2);
22     else
23       self.gatherData()
               after(?(1,2,3));
24   }
25 }

27 env int BUFFER_SIZE = 4;

29 reactiveclass CompUnit(3) {
30   statevars {
31     Network network;
32     @feature(FT_PERIODIC_PUSH)
33     int[BUFFER_SIZE] buffer;
34     @feature(FT_PERIODIC_PUSH)
35     int cnt;
36   }

38   CompUnit(Network net) {
39     network = net;
40     @feature(FT_PERIODIC_PUSH)
41     self.process();
42   }

44   @feature(FT_IMMEDIATE_PUSH)
45   msgsrv receiveData(byte data) {
46     network.send(data);
47   }

49   @feature(FT_PERIODIC_PUSH)
50   msgsrv receiveData(byte data) {
51     buffer[cnt++] = data;
52   }

54   @feature(FT_PERIODIC_PUSH)
55   msgsrv process() {
56     for (int i=cnt; cnt>0; cnt--)
57       network.send(buffer[i]);
58     self.process() after(1);
59   }
60 }

62 interface Network {
63   msgsrv send(byte data);
64 }

66 reactiveclass MACBNetwork
       implements Network(3) {
67   msgsrv send(byte data) {
68     // Send data according
69     // to the MACB protocol
70   }
71 }

73 reactiveclass TDMANetwork
       implements Network(3) {
74   msgsrv send(byte data) {
75     // Send data according
76     // to the TDMA protocol
77   }
78 }

80 main {
81   Sensor sensor():(cu);
82   CompUnit cu():(network);
83   @feature(FT_SIMPLE_NETWORK)
84   MACBNetwork network():();
85   @feature(FT_TDMA_NETWORK)
86   TDMANetwork network():();
87 }
```

Fig. 3. The Timed Rebeca model of the running example extended by variability (the sensor network example extended with the variation points of Fig. 2)

3.2 Feature Annotations

In the context of software product line engineering, it is common to capture the variabilities in a separate variability model. Some well-known models for this purpose include the widely-used Feature Models [37], UML-based variability models [9], and Common Variability Language (CVL) [28]. We assume that the variability is captured in a feature model. The features are represented by global boolean *feature variables*. A *True* (resp. *False*) value for a feature variable indicates that the corresponding feature is included in (resp. excluded from) the product under analysis. In the running example (Fig. 3), the variables defined in lines 1 to 6 represent the 'leaf' features in the feature model of Fig. 2. Note that it is not necessary to define variables for the mandatory features included in every configuration.

We assume that the values for all feature variables are defined as parameters of the analysis process. Hence, Afra is currently capable of analyzing one product at a time. As we will see later, this limitation can be relaxed based on the existing theories for model checking several products at a time. We also assume that the values assigned to the feature variables are checked externally to satisfy the validity of the feature model (e.g., not including two alternatives in the configuration).

The feature variables can be used to define feature-specific behavior in two ways. The first is to use a feature variable as an ordinary global variable. Line 20 of Fig. 3 is an example of this type. It is possible to mix feature variables with state (or local) variables. The second way is to use *feature annotations*. The syntax `@feature(`*feature_expr*`)` may come before various language constructs which causes that construct to be included in the model only if *feature_expr* evaluates to *True*. As an example, the state variable `buffer` is included in the reactive class `CompUnit` only if the feature *Periodic Push* is present in the configuration (represented by the feature expression `FT_PERIODIC_PUSH` in the feature annotation of line 32). Note that the annotation only affects its immediately following declaration. Hence, the variable `cnt` in line 35 must be annotated separately (line 34). Other model elements can be annotated as well, e.g., statements (line 41), message servers (lines 44, 49, and 54), and actor instantiations (lines 83 and 85). As the Timed Rebeca syntax allows grouping of statements into blocks, which itself is a statement, one can annotate a group of statements within a message server:

```
...
@feature(SOME_FEATURE_EXPR) {
  statement 1;
  statement 2;
  ...
  statement n;
}
...
```

As illustrated by the feature annotations in lines 44 and 49, two alternative implementations of the same message server may be provided. However, in case

the feature expressions of the annotations are not mutually exclusive, a duplicate definition error may be raised when compiling an individual product model which includes more than one definition for the same message server. In the case of verifying the whole product line without projecting the model onto an individual product configuration (Sect. 3.5), this check is more involved. Assuming that there are two definitions for the same message server, one annotated with the feature expression e_1 and another with e_2, an error must be raised if $e_1 \wedge e_2$ is satisfiable[4], which can be checked using a SAT solver.

3.3 Reactive Class Polymorphism

As stated in Sect. 2.4, the statically typed, class-based nature of Timed Rebeca allows polymorphic modeling with respect to the interfaces of the reactive classes. As an example, the `Network` interface defined in lines 62 to 64 of Fig. 3 specifies a single message server `send(byte)` without defining its behavior. Any class implementing `Network` must implement the message server, as illustrated by the classes `MACBNetwork` and `TDMANetwork`. To keep the running example as small as possible, the interface is defined in its simplest form and the implementations are omitted. However, the modeler can take advantage of more involved features of interfaces, e.g., by making classes implement multiple interfaces, defining inheritance hierarchies among interfaces, etc.

An interface can be used as the type of state variables (line 31) or parameters (line 38). An instance of any reactive class implementing that interface may be assigned to such a state variable or parameter (line 82). This use of polymorphic modeling provides a coarser-grained variability implementation mechanism (compared to feature annotations), where the variability is resolved by choosing among several components implementing the same interface.

3.4 Handling Reconfiguration

If we allow feature variables to change during execution, it is possible to change the configuration at runtime which allows the modeling of reconfigurable systems. The reconfiguration can take place using both variability mechanisms, feature annotation, and polymorphism. As an example, executing the following code will change the behavior of all sensors from periodic to sporadic[5].

```
if (someCondition) {
  FT_PERIODIC_SENSOR = false;
  FT_SPORADIC_SENSOR = true;
}
```

[4] More precisely, the satisfiability check must incorporate the constraints imposed by the feature model too. To this end, a feature expression F must be derived from the feature model (as explained in [5]), and the satisfiability of $e_1 \wedge e_2 \wedge F$ must be checked.

[5] Of course, since the two features are mutually exclusive, this could have been done using only one feature variable.

Note that this code can be placed at any reactive class, possibly other than `Sensor`. This allows the separation of reconfiguration logic from the actors' behavior. There is a limitation in using this type of reconfiguration where the feature variable is used to annotate some state variables or an entire reactive class (as opposed to a message server or a part of it). Since this changes the structure of the states of the system, it complicates the generation and analysis of the state space and thus is forbidden. If a reconfiguration of this type is required it is recommended to use polymorphism to handle the variability (as illustrated shortly in an example).

Moreover, a number of semantic issues arise when using annotative reconfiguration which are studied in [56]. The most important happens when a reconfiguration eliminates a message server, while there are messages of that type in some actor's mailbox. The solution proposed is to make the receiver actor perform a configuration check whenever it takes a message from its mailbox for execution and drop the message in case it is excluded from the model with respect to the configuration at the time of taking the message. In [56], a variability-aware semantics has been proposed for Rebeca supporting reconfiguration.

When using reactive class polymorphism, the reconfiguration can happen without the need to change the Rebeca semantics. As an example, the following method can be used to change the network protocol at runtime.

```
// in CompUnit:
  statevars {
    Network defaultNet;
    Network alternativeNet;
    Network network;
  }

  CompUnit(Network def, Network alt) {
    defaultNet = def;
    alternativeNet = alt;
    network = def;
  }

  msgsrv switchNetwork() {
    network = alt;
  }

// in reconfiguration logic (anywhere in the model):
  if (someCondition) {
    cu.switchNetwork;
  }

// in the main block:
  MACBNetwork macb():();
  TDMANetwork tdma():();
  CompUnit cu():(macb, tdma);
```

Note that both network classes must be instantiated in the `main` block, as Rebeca does not support the dynamic creation of actors. It is possible that in

the implementation of the system the actors are instantiated just upon reconfiguration. In this case, special care must be taken during the implementation to keep the verification results valid.

We also emphasize that the change of the network protocol happens whenever the `switchNetwork` message is handled. So, the computation unit may work with the default network for a while after the reconfiguration happens. If this makes a problem, in Timed Rebeca, the reconfiguration logic should be given priority over normal behavior using the technique explained in Sect. 4.

3.5 Model Checking in the Presence of Variability

When it comes to verification, one can derive the Rebeca model for each valid configuration, and model check each model separately. However, this way we cannot benefit from the commonalities among the behavior of the products. The problem of model checking the whole product line at once has been the subject of various studies, like [18]. In the context of Rebeca, [56] has addressed model checking reconfigurable families of actor systems, based on a feature-annotated state space generated for the whole product line.

One can statically analyze the product line model to detect the features whose presence does not affect the satisfaction of a given property. For such features, it suffices to verify the products that exclude those features. A similar technique can be used regarding the alternative features (according to the feature model). These improvements (as well as some others regarding evolving product lines) have been studied in [57], using a variability-aware data and control dependency graph generated from the model. The experimental results indicate a significant reduction in the verification cost of the whole product line. Note that the model checking of the whole product line at once has not been yet integrated into Afra and is planned for future releases.

3.6 Case Study: Elevator Scheduling with Variability

To demonstrate how variability handling mechanisms can be used in practice to enable an analysis of a real-time software product line, we studied an elevator scheduling system which is originally defined in [55] and analyzed for schedulability using a Timed Automata Family. The feature model of the case study is depicted in Fig. 4. The elevator system consists of three to five floors, as indicated by the numeric feature Floors. A central controller is responsible for scheduling the movement of the elevator. The time between two consecutive requests on the same floor is assumed to be within a certain discrete range of $[LOW, HIGH]$. The scheduling algorithm must guarantee a maximum waiting time for each request ($TIMEOUT$). The system may support VIP floors (indicated by the optional feature VIP Floor), where the maximum waiting time is less than normal floors ($TIMEOUT\_VIP$). On the other hand, the time between two consecutive requests on a VIP floor may be different from non-VIP floors and is assumed to be within the discrete range of $[VLOW, VHIGH]$. The elevator system may be equipped with a weight sensor (indicated by the optional feature

Weight Sensor) which prevents the elevator from moving if the total weight in the cabin exceeds a limit. This increases the time the elevator waits at a floor in the worst case by *LVL_DELAY*.

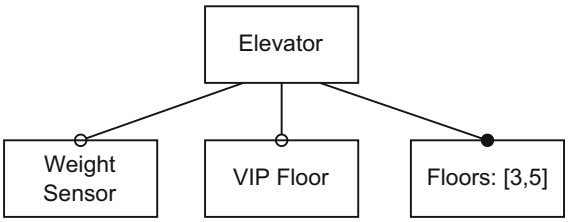


Fig. 4. The elevator case study feature model [55]

The Timed Rebeca model for the case study with four floors is listed in the Figs. 5, 6 and 7. To save space, we have omitted a few less important parts. The current implementation of Afra does not support the dynamic creation of actors, so the variability in the number of floors must be handled manually, by instantiating the desired number of actors in the main block (as in Fig. 7). The other two features are modeled by `FT_VIP` and `FT_WEIGHT_SENSOR`.

```
1  featurevar FT_VIP;
2  featurevar FT_WEIGHT_SENSOR;
3
4  reactiveclass Floor(5) {
5    knownrebecs {
6      Controller ctrl;
7    }
8    statevars {
9      int level;
10     boolean isWaiting;
11     @feature(FT_VIP)
12     boolean isVIP;
13   }
14   // constructor omitted
15   msgsrv makeReq() {
16     ctrl.requestFor(level);
17     isWaiting = true;
18     int timeout = TIMEOUT;
19     @feature(FT_VIP)
20     if (isVIP)
21       timeout = TIMEOUT_VIP;
22     self.timeOut()
           after(timeout);
23   }
24   msgsrv timeOut() {
25     assertion(!isWaiting);
26   }
27   msgsrv served() {
28     isWaiting = false;
29     int rqDly = ?(LOW, HIGH);
30     @feature(FT_VIP)
31     if (isVIP)
32       rqDly = ?(VLOW, VHIGH);
33     self.makeReq() after(rqDly);
34   }
35 }
```

Fig. 5. The elevator scheduling case study - Timed Rebeca model of the floors

Each floor actor, an instance of `Floor` reactive class (Fig. 5), knows its level, whether it is waiting for its request to be served, and if it is a VIP floor (only if VIP feature is on), modeled by the corresponding state variables (lines 9–12). Upon construction, a floor makes a request for the elevator. The body of the constructors are omitted to save space. When receiving a `makeReq` message (lines

15–23), the floor sends the controller a `requestFor` message along with its level number and sets itself in the waiting mode. To check the schedulability of the model, the floor schedules a `timeOut` message for either `TIMEOUT` or `TIMEOUT_VIP` to be sent to itself. Upon the timeout (lines 24–26), an assertion fails if the floor is still waiting. If the elevator arrives on a waiting floor (lines 27–35), the floor exits the waiting state and schedules the subsequent request for some time in the range [`LOW`, `HIGH`] (or [`VLOW`, `VHIGH`] for a VIP floor). To avoid the complexity of handling recurrent requests at a floor (i.e., a second request is made before the first one is served), we assume that *TIMEOUT* is reasonably smaller than *LOW*.

```
1  reactiveclass Controller(20) {
2    statevars {
3      Floor[LVL_CNT] floor;
4      boolean[LVL_CNT] requested;
5      int dir;
6      int atLevel;
7      boolean stopped;
8    }
9    // constructor omitted
10   msgsrv requestFor(int dest) {
11     requested[dest] = true;
12     if (dir == NOT_MOVING)
13       if (atLevel < dest)
14         move(UP);
15       else if (atLevel > dest)
16         move(DOWN);
17       else
18         serve(dest);
19   }
20   msgsrv arrive(int level) {
21     handleArrival(level);
22     reschedule(level);
23   }
24   void move(int direction) {
25     dir = direction;
26     int next_arrival =
           TIME_FOR_ONE_LEVEL;
27     @feature(FT_WEIGHT_SENSOR)
28     if (stopped)
29       next_arrival += LVL_DELAY;
30     if (direction == UP)
31       self.arrive(atLevel + 1)
             after(next_arrival);
32     else if (direction == DOWN)
33       self.arrive(atLevel - 1)
             after(next_arrival);
34   }
35   void stop() {
36     dir = NOT_MOVING;
37   }
38   void handleArrival(int level) {
39     atLevel = level;
40     stopped = false;
41     if (requested[level]) {
42       serve(level);
43       stopped = true;
44     }
45   }
46   void serve(int level) {
47     requested[level] = false;
48     floor[level].served();
49   }
50   boolean higherLevelsRq() ...
51   boolean lowerLevelsRq() ...
52   void reschedule(int level) {
53     if (dir == UP)
54       if (higherLevelsRq())
55         move(UP);
56       else
57         if (lowerLevelsRq())
58           move(DOWN);
59         else
60           stop();
61     else if (dir == DOWN)
62       if (lowerLevelsRq())
63         move(DOWN);
64       else
65         if (higherLevelsRq())
66           move(UP);
67         else
68           stop();
69   }
70 }
```

Fig. 6. The elevator scheduling case study - the model of the controller in a non-VIP setting.

As its state variables, the (non-VIP) `Controller` (Fig. 6) knows the floors, whether there is a request for each floor, its direction (`NOT_MOVING`, `UP`, or `DOWN`), the level at which it just arrived, and whether it has stopped at that level (or just passed by). Upon receiving a request for a destination level (lines 10–19), the controller marks the floor as requested and starts to move the elevator toward the destination if it is not moving already. When the elevator arrives at a level (either as a destination or just passing by), it notifies the controller via `arrive` message (lines 20–23). The controller first handles the arrival, and then reschedules the elevator's movement if necessary. The movement (for one level) is handled in `move` method, whose function is to schedule an `arrive` message at the next visited floor (determined according to the current level and the direction). The time the elevator arrives on the next floor is `TIME_FOR_ONE_LEVEL`, plus the extra time needed to wait a the level if the weight sensor feature is included. This extra time is needed only if the elevator has been stopped to serve a request (hence the conditional statement in line 64). The functions of `handleArrival` and `server` are straightforward. After arrival, a rescheduling must happen if necessary (lines 52–69). If the elevator has been going up, and there are requests for upper levels, it continues in that direction. Otherwise, if there are requests for the lower directions, it changes direction downwards. If there are no other requests, it stops. A similar logic is followed if the elevator has been going down. The bodies of the two boolean methods `higherLevelsRq` and `lowerLevelsRq` are omitted to save space.

The weight sensor variability can be resolved in just a few annotations. To support VIP floors, special care must be taken when rescheduling to be able to meet the shorter waiting deadline of such floors. Hence, the basic controller is extended by `VIPController` to support VIP scheduling (Fig. 7). It inherits all members of the basic `Controller` and additionally knows which floors are of VIP type (line 4). The message server `arrive` is overridden in the way that it first determines its next direction considering only the requests for VIP floors. If no such request exists, the ordinary rescheduling algorithm is used by calling the (inherited) `reschedule` method. Again, the bodies of the two boolean methods `higherVIPLvlRq` and `lowerVIPLvlRq` are omitted to save space.

Each configuration of the model can be analyzed for schedulability using Afra. The results of the verification of a few products is reported in Table 1. To keep the size of the table small, we have only reported the configurations with four floors, and two configuration with five floors. For the configurations including VIP floors, only the topmost floor is considered as VIP.

The complexity of the analysis is greatly affected by the size of the intervals specifying the minimum and maximum amount of times between two consecutive requests for each floor (shown in the Rq.Int. column), as Afra checks for each value within the interval systematically. For the first four configurations, we set this parameter to three ($LOW = 20$, $HIGH = 22$, $VLOW = 22$, $VHIGH = 24$). The last two configurations have five floors, one with an interval of size two and the other with size three. The models are analyzed for schedulability and deadlock-freedom on a single core from a 3.6 GHz Core-i7 machine with 16 GB of RAM.

```
1  @feature(FT_VIP)
2  reactiveclass VIPController
       extends Controller(20) {
3    statevars {
4      boolean[LVL_CNT] isVIP;
5    }
6    // constructor omitted
7    msgsrv arrive(int level) {
8      handleArrival(level);
9      if (dir == UP)
10       if (higherVIPLvlRq())
11         move(UP);
12       else if (lowerVIPLvlRq())
13         move(DOWN);
14       else
15         reschedule(level);
16     else if (dir == DOWN)
17       if (lowerVIPLvlRq())
18         move(DOWN);
19       else if (higherVIPLvlRq())
20         move(UP);
21       else
22         reschedule(level);
23   }
25   boolean higherVIPLvlRq() ...
26   boolean lowerVIPLvlRq() ...
27 }
29 main {
30   @feature(!FT_VIP) {
31     Controller ctrl():(f0, f1,
           f2, f3);
32     Floor f0(ctrl):(0);
33     Floor f1(ctrl):(1);
34     Floor f2(ctrl):(2);
35     Floor f3(ctrl):(3);
36   }
37   @feature(FT_VIP) {
38     VIPController ctrl():(f0,
           f1, f2, f3, false,
           false, false, false);
39     Floor f0(ctrl):(0, false);
40     Floor f1(ctrl):(1, false);
41     Floor f2(ctrl):(2, false);
42     Floor f3(ctrl):(3, false);
43   }
44 }
```

Fig. 7. The elevator scheduling case study - the model of the VIP controller and the instantiation of the actors.

Table 1. The number of states and transitions, and the time required to model check a few configurations of the elevator product line. Each row specifies a configuration by assigning values to the features Weight Sensor (WS), VIP Floor (VIP), and the number of floors (Floors). The parameter Rq.Int. specifies the size of the time interval between two consecutive requests ($[LOW, HIGH]$).

Config.	WS	VIP	Floors	Rq.Int	States	Transitions	Time (sec.)
1	✓		4	3	106,234	165,326	1
2	✓	✓	4	3	185,145	196,939	2
3			4	3	380,794	491,662	3
4		✓	4	3	1,221,333	1,543,755	10
5		✓	5	2	1,435,246	1,818,949	14
6		✓	5	3	37,178,658	48,576,931	384

Assuming the elevator waits for one time unit at each floor, and adds another time unit if it has a weight sensor, having the mentioned intervals between two consecutive requests yields in the smallest values for time outs as shown in Table 2. In case the time out values are infeasible to satisfy, Afra reports a schedulability violation and provides a counterexample trace as illustrated in Fig. 8.

Table 2. The smallest possible time out values for different configurations

Config.	WS	VIP	Floors	Rq.Int	*TIMEOUT*	*TIMEOUT_VIP*
1	✓		4	3	16	N/A
2	✓	✓	4	3	16	10
3			4	3	11	N/A
4		✓	4	3	11	8
5		✓	5	2	13	10
6		✓	5	3	13	10

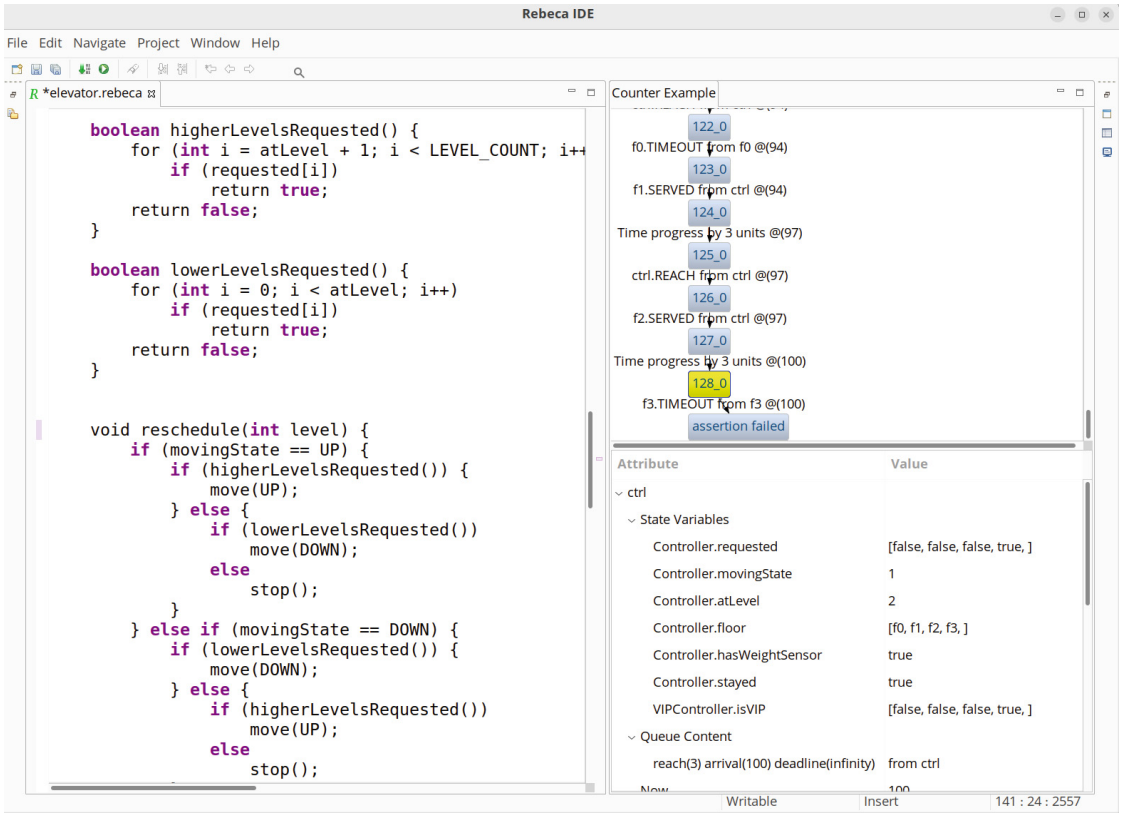


Fig. 8. The counterexample provided by Afra when a time out happens

4 More Deterministic Models Using Priorities

In concurrency theory, nondeterminism is used to model concurrency. Hewitt actors are designed for building distributed and network systems. There is a trend to add more determinism to the language models inspired from synchronous languages. Edward Lee and his team are proposing deterministic concurrency in [49]. Apart from that, in many applications, there is a predefined priority used for ordering the tasks in hand. Here we explain how priorities are added as annotations to Timed Rebeca to better support such applications.

In Rebeca, the semantics of the language is defined to order the execution of enabled actors nondeterministically. An actor is enabled if the actor is not busy handling a message and its message queue is not empty. Each actor has a message queue and the messages sent to an actor from another actor are put in the receiver's message queue with the same order that the messages are sent. So, in Rebeca, we have a point-to-point in-order message delivery, but we cannot have any assumptions about messages sent by different actors. For Timed Rebeca, the order of handling messages of an actor depends on the time tags of the messages. If there is more than one message with the same time tag then these messages are handled in a nondeterministic order (see [53] for a formal definition of the semantics). To make the behavior of actors in Rebeca models more deterministic, which is required for real-time and embedded systems, Rebeca allows associating priority to message servers and actors. The messages with the same time tag are handled in the order which is defined by the priorities of their corresponding message servers. Priorities for the actors are defined in the main part of the code where we instantiate actors from the reactive classes. This way, the execution of enabled actors takes place considering the associated priorities.

4.1 Incorporating Priorities into the Running Example

In the extended version of the running example in Fig. 10, we want to make sure that in each round of execution, all of the gathered data by Sensor is processed by CompUnit. So, there is a need for Sensor to have a higher priority in the execution in comparison with CompUnit. Figure 9 shows a diagram representing the program model of the running example, inspired from Lingua Franca [27]. The program is assembled from three actors, Sensor, CompUnit, and Network, shown as light gray boxes. The numbers in the top-left side of the boxes show the priorities of actors. Black triangles in the diagram show communication ports. In this model, both Sensor and CompUnit have output ports that are connected to corresponding input ports. In the diagram, methods are represented by dark gray chevrons. The order of defining methods in the figure shows the execution priority of methods, e.g., `receiveData` has a higher execution priority compared to `process` in the `CompUnit` actor. In Fig. 9, Sensor and CompUnit define methods that are triggered periodically.

As depicted in lines 50 to 55, of Fig. 10, three different priority levels are associated with instances of reactive classes using *priority annotations*. Having

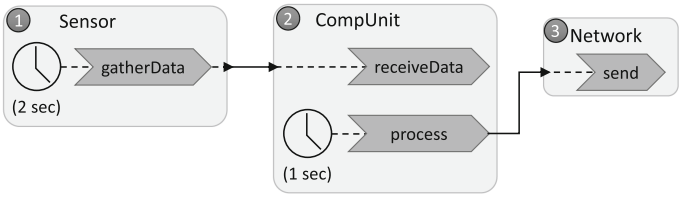


Fig. 9. A diagrammatic representation of the program model of the running example of sensor network with priorities, inspired with the Lingua Franca diagram notation

a smaller value for *priority annotations* means that the actor has a higher execution priority. Note that associating the same priority level with actors results in the nondeterministic choice among the actors when more than one of them are enabled.

In addition to the cases mentioned above, each reactive class is allowed to prioritize the execution of its message servers. It means that in the case of receiving two messages with the same time tag, the message server which is annotated with a higher priority will be executed first. In Fig. 10, we make sure that the method for receiving data from Sensor has a higher priority than the method for processing data in CompUnit. This decision is because CompUnit has to receive the data prior to processing it. This way, the priority among reactions 1 and 2 in Fig. 9 is addressed.

```
1  reactiveclass Sensor(3) {
2    statevars{
3      CompUnit cu;
4    }

6    Sensor(CompUnit cu1) {
7      cu = cu1;
8      self.gatherData(1);
9    }

11   msgsrv gatherData(byte data) {
12     cu.receiveData(1);
13     self.gatherData(1) after(2);
14   }
15 }

17 reactiveclass CompUnit(3) {
18   statevars {
19     Network network;
20     byte[4] buffer;
21     int cnt;
22   }

24   CompUnit(Network net) {
25     network = net;
26     self.process();
27   }

29   @priority(1)
30   msgsrv receiveData(byte data) {
31     buffer[cnt++] = data;
32   }

34   @priority(2)
35   msgsrv process() {
36     for (int i=cnt; cnt>0; cnt--)
37       network.send(buffer[i]);
38     self.process() after(1);
39   }
40 }

42 reactiveclass Network (3) {
43   msgsrv send(byte data) {
44     // Send data according
45     // to a protocol
46   }
47 }

49 main {
50   @priority(1)
51   Sensor sensor():(cu);
52   @priority(2)
53   CompUnit cu():(network);
54   @priority(3)
55   Network network():();
56 }
```

Fig. 10. The Timed Rebeca model of the running example with priorities (the sensor network example with priorities for the actors and for the message servers)

In some cases, associating priorities to actors and methods within classes does not give us the order of execution of methods we are looking for. Hence, we also added another feature to Timed Rebeca, by which we can associate priorities with each method. This is a flat type of priority throughout the whole model

which we call Global Priority (and is not shown in the examples). Note that using both `GlobalPriority` and `Priority` in one model is not allowed.

4.2 Analysis of Rebeca Models with Priorities

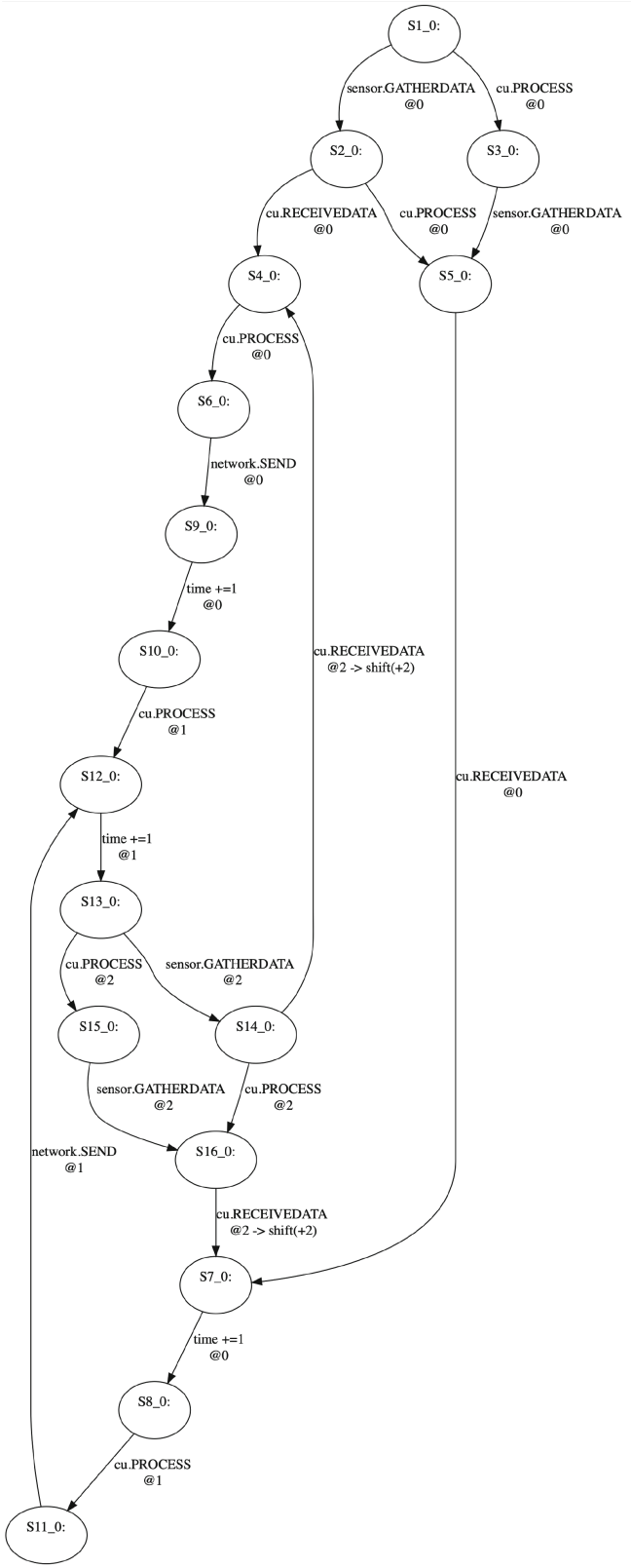
The model checking engine of Afra assumes that in the given model all of the actors and methods have priorities, if there is no priority associated to an actor or a method, then Afra assumes the lowest priority for it. At each step of the state space generation, Afra selects the highest priority enabled message from the enabled actor with the highest priority. In the case of having methods or actors with the same priority level, one of them is selected nondeterministically. During model checking, Afra generates the state space for all possible combinations.

Figure 11 compares the transition systems of the model of Fig. 10. As mentioned before, including priorities eliminates some nondeterministic choices which results in smaller transition systems. Two outgoing transitions of $S1\_0$ of Fig. 11(a) illustrates nondeterministic choice between executing the messages of `sensor` and `cu`. This nondeterminism is resolved by associating priorities to actor instances in $S1\_0$ of Fig. 11(b). Another kind of nondeterminism is depicted in $S2\_0$ for executing `receiveData` or `process` of the actor `cu`. In its corresponding state in Fig. 11(b), `receiveData` has a higher priority and there is no nondeterministic choice.

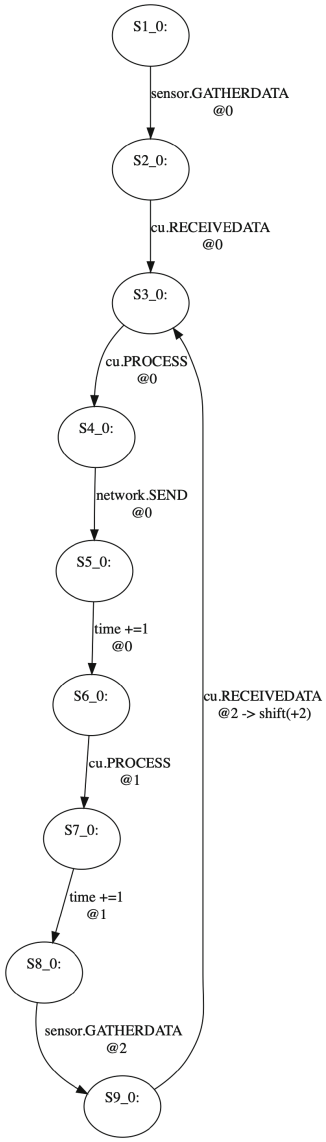
4.3 Case Study: Anti-lock Braking System, with Priority

We demonstrate the applicability of the priority feature of Rebeca on a simplified Brake-by-Wire (BBW) system with Anti-lock Braking System (ABS) [23,36,48]. To prevent uncontrolled skidding, ABS releases the brakes based on the slip rate, computed in terms of the torque and speed of wheels read by the wheel sensors. We previously specified and analyzed this case study within Hybrid Rebeca [33], an extension of Rebeca with continuous real variables that change over time, specified by ordinary differential equations (ODEs). Due to the absence of the priority feature, we handled the required priorities among the actors in the semantic model (this priority was hard-coded in the semantics). We revisit this example by replacing ODEs with simple expressions updating real-valued variables at discrete time intervals.

In this system, there is a wheel controller (WheelCtrl) for each wheel and a global brake controller (BrakeCtrl). Each wheel and the brake pedal are equipped with a sensor. The brake pedal sensor calculates the brake percentage based on the brake pedal's position and sends this value to BrakeCtrl. Each wheel sensor sends the speed of its wheel to its corresponding WheelCtrl which sends this value to BrakeCtrl. Then, BrakeCtrl computes the desired brake torque and the speed of all wheels and sends these values to each WheelCtrl to apply them. Depending on the slip rate, computed based on the current and desired speed, WheelCtrl releases the brake if the slip rate is greater than a specified value to prevent skidding.



(a) The transition system without priorities



(b) The transition system with priorities

Fig. 11. Comparing transition systems of the model of Fig. 10 without priorities (a) and with priorities (b).

Each pair of a sensor and its corresponding controller are connected directly by a pair-to-pair link. All other communications are managed through a shared Controller Area Network (CAN) [50] which is a dominant networking protocol in the automotive industry. CAN is a serial bus network where nodes can send messages anytime. Upon multiple simultaneous send requests, 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. A CAN bus can be conceived as a single global priority-based queue [20] that deterministically dispatches messages based on their arrival times and for those messages arrived at the same time based on their priorities. Thus, we model the CAN network as a Rebeca class, called `CANBusNetwork`, with a message server for each message priority. We define an abstract class called `Ent` as the supertype for connected entities, e.g., ECUs in this example, over the CAN bus. Connected entities send their messages to `CANBusNetwork` by calling the appropriate message server corresponding to the message priority. Then, `CANBusNetwork` will transfer the message to the target entity by sending a `rcv` message. We assume that entities communicate by sending a pair of type and value, modeled as the parameters of `rcv` messages. We have considered three message priorities by defining three message servers `sndH`, `sndM`, and `sndL` as given in Fig. 12. For simplicity, we have considered two wheels in the model. The model consists of four other classes shown in Fig. 12: `WheelSensor`, `WheelCtrl`, `BrakeSensor`, and `BrakeCtrl`.

The `WheelSensor` class models the sensors and actuators of the wheel. The class has one known rebec of `WheelCtrl`. This class periodically updates the speed of the wheel and then sends the new value to the wheel controller (lines 33–35), specified by the message server `sndSpeed`. As each wheel sensor is connected via a pair-to-pair link to its wheel controller, we model this communication by directly sending a message `setWspd` to the wheel controller. Upon handling a message `setTrq`, it applies the effect of braking on the wheel speed (line 31).

The `WheelCtrl` class defines the behavior of the wheel controller which communicates via CAN bus by the global brake controller and via pair-to-pair link with its wheel sensors. So, this class has three known rebecs of `WheelSensor`, `BrakeCtrl`, and `CANBusNetwork`. Upon receiving the speed of the wheel through `setWspd` messages from the wheel sensor, it will send the speed to the brake controller via the CAN network (line 56). It receives the desired speed and torque from the brake controller via CAN bus through `rcv` messages (lines 46–52). We assume that the brake controller first sends the desired speed and then the torque. After receiving the torque, it computes the slip rate of the wheel and then decides to apply the brake by sending the appropriate torque to the wheel (lines 49–51).

The `BrakeSensor` class defines the behavior of the brake pedal sensor. The class has one known rebec `BrakeCtrl` which is the global brake controller. It has the state variable `bpcnt` which is the brake percentage and increased up to the value defined by the state variable `max`. This class sends the value of `bpcnt` periodically to `BrakeCtrl` via `sndBrake` message (lines 65–67). In the

```
env int TORQUE = 0;
env int SPEED = 1;
env int PERIOD = 1;

interface Ent{
   msgsrv rcv(Ent entity,int
        type, float value);
}
reactiveclass CANBusNetwork(4){
 CANBusNetwork(){}
 @priority(1) //high
 msgsrv sndH(Ent n,int t,float v)
   {n.rcv((Ent)sender,t,v);}
 @priority(2) //medium
 msgsrv sndM(Ent n,int t,float v)
  {n.rcv((Ent)sender,t,v) ;}
  @priority(3) //low
 msgsrv sndL(Ent n,int t,float v)
   {n.rcv((Ent)sender,t,v) ;}
}
reactiveclass WheelSensor(1){
  knownrebecs {
   WheelCtrl wCtrl;}
 statevars {
   float spd; float trq;}
  WheelSensor(float _s){
   spd = _s;
   self.sndSpeed();}
 msgsrv setTrq(float _trq){
   trq = _trq;}
 msgsrv sndSpeed(){
  spd = (float)spd-trq-0.1 ;
  wCtrl.setWspd(spd);
  if (spd>0)
   self.sndSpeed() after(PERIOD);
}}
reactiveclass WheelCtrl
     implements Ent(2){
 knownrebecs {
   WheelSensor w;
   BrakeCtrl bCtrl;
   CANBusNetwork CAN;}
 statevars {
   float vspd;float wspd;}
 WheelCtrl(){}
 @priority(2)
 msgsrv rcv(Ent n,int t,float v){
  if (t==SPEED) vspd = v;
  else{
   if(((vspd-wspd*0.74)/vspd)>0.2)
     w.setTrq(0);
    else w.setTrq(v);
 }}
 @priority(1)
 msgsrv setWspd(float spd){
   wspd = spd;
   CAN.sndM(bCtrl,SPEED,spd);}
}
reactiveclass BrakeSensor(1){
 knownrebecs {
   BrakeCtrl bCtrl;}
 statevars {
   float bpcnt;float max;}
 BrakeSensor(float b,float m){
  bpcnt = b; max = m;
  self.Braking();}
 msgsrv Braking(){
  bpcnt = bpcnt + 1 ;
  bCtrl.setBpcnt(bpcnt);
  if (bpcnt<max)
   self.Braking() after(PERIOD);
}}
reactiveclass BrakeCtrl
     implements Ent(4){
 knownrebecs {
   WheelCtrl wCtrlL;
   WheelCtrl wCtrlR;
   CANBusNetwork CAN;}
 statevars {
   float spdR;float rtrq;
   float spdL;float rspd;
   float bpcnt;}
 BrakeCtrl(){
   self.control();}
 @priority(1)
 msgsrv rcv(Ent n,int t,float v){
  if (((WheelCtrl)n)==wCtrlR)
    spdR = v;
   else spdL = v;}
 @priority(2)
 msgsrv control(){
  rtrq = bpcnt;
  rspd = (spdR + spdL) / 2;
  CAN.sndH(wCtrlR,SPEED,rspd);
  CAN.sndH(wCtrlR,TORQUE,rtrq);
  CAN.sndH(wCtrlL,SPEED,rspd);
  CAN.sndH(wCtrlL,TORQUE,rtrq);
  self.control() after(PERIOD;}
  @priority(1)
  msgsrv setBpcnt(float b){
    bpcnt = b ;}
}
// main block in the next figure
```

Fig. 12. The specification of Brake-by-Wire system with Anti-lock Braking System

constructor, the actor sends a `sndBrake` message to itself to start the periodic communication.

The `BrakeCtrl` class is responsible for delegating the brake torque to wheel controllers. It defines three known rebecs, two for each wheel controller named `wCtrlL`, `wCtrlR`, and one for the network called `CAN`. This class has three state variables for the right and left wheels' speed and the brake pedal's brake percentage (`bpcnt`). It also has two auxiliary state variables for computing the desired speed and torque. The message server `control` is executed periodically to calculate the desired brake torque, calculated based on the brake percentage (lines 89–96). It also estimates the speed based on the speed of the wheels. Then, the estimated speed and global torque are sent to each wheel controller via the CAN network. The message server `setBpcnt` updates `bpcnt` based on the received value. The constructor sends a `control` message to itself to start the periodic execution.

The main block of the model is listed in Fig. 13. Figure 14 shows the LF's diagrammatic representation of the program model of the Brake-by-Wire system. As we considered two wheels in the system, the program is assembled from two instances of `WheelSensor` and `WheelCtrl`, one instance of `BrakeSensor`, `BrakeCtrl`, and `CANBusNetwork`. As depicted in Fig. 14, the values are received from `WheelCtrl` and `BrakeSensor` by `BrakeCtrl` to compute the desired brake torque and speed. `WheelCtrl` also receives its value from `WheelSensor`. To correctly compute the desired values in each period, we must guarantee that `BrakeCtrl` has received the most recent sensed values from the sensors. So, we assign the highest priority to the instances of `BrakeSensor` and `WheelSensors`. We assign the next priorities to the components over the path from `WheelSensor` to `BrakeCtrl`, i.e., instances of `WheelCtrls` and then `CANBusNetwork`. We also assign a lower priority to the message server `control` than `rcv` to be sure that it updates the values sensed for this period before its computation. If none of the priorities are considered, `WheelCtrl` may make the computation using stale values. This is in line with the policy of the order of execution of components "from upstream to downstream" in the design of CPS and used in Lingua Franca. We will explain this through a scenario in the following.

Consider the property that states "whenever the slip rate of a wheel exceeds 0.2, the brake actuator of that wheel must be immediately released". We imply from this property that at the end of each period if $(\texttt{rspd} - \texttt{WSL.spd} \times 0.75)/\texttt{rspd} > 0.2$ then `WSL.trq` must immediately become 0, where $\texttt{rspd} = (\texttt{WSL.spd} + \texttt{WSR.spd})/2$. Suppose that initially, the speed of the left and right wheels are 15 and 13, respectively and the initial brake percentage is 60. As only `WSL`, `WSR`, and brake sensor `BS` have messages in their queue, they first send the speed of wheels (i.e., 15 and 13) and brake percentage (i.e., 60) to their corresponding controllers upon handling their messages. Then, the wheel controllers and the brake controller handle `setWspd` and `setBpcnt` messages, respectively, to update their values. The wheel controllers send their speed values to the brake controller via `CAN` by sending a `sndM` message. Then, `CAN` handles its two `sndM` messages from the wheel controllers by sending `rcv` messages to the brake con-

```
93    main {
94       @priority(1)
95       WheelSensor WSL(WCL):(10,12);
96       @priority(2)
97       WheelCtrl WCL(WSL,BC,CAN):();
98       @priority(1)
99       WheelSensor WSR(WCR):(11,12);
100      @priority(2)
101      WheelCtrl WCR(WSR,BC,CAN):();
102      @priority(1)
103      BrakeSensor BS(BC):();
104      @priority(4)
105      BrakeCtrl BC(WCL,WCR,CAN):();
106      @priority(3)
107      CANBusNetwork CAN():();
108   }
```

Fig. 13. Actor instantiations for the Brake-by-Wire system with Anti-lock Braking System

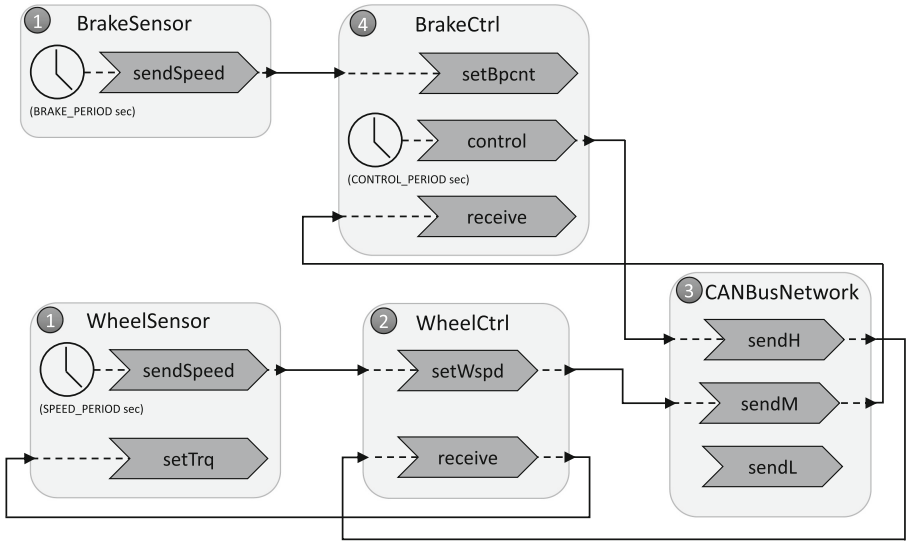


Fig. 14. A diagrammatic representation of the program model of the Brake-by-Wire system presented in Fig. 12, inspired from the Lingua Franca diagram notation

troller `BC`. Please note that `CAN` has a higher priority than the brake controller, so the brake controller `BC` first gets two `rcv` messages before handing its messages. `BC` has the next priority to be executed. It has three messages in its queue: two `rcv` messages and one `control` message. As the priority for handling `rcv` messages is higher than the `control` message, it first handles the `rcv` messages and updates the value speed of wheels, and then by handling the `control` message computes the desired torque and speed as 60 and 14 and sends them via two sequential `rcv` messages through `CAN` to each wheel controllers. The wheel controllers handle their `rcv` messages and compute the slip rate as 0.207 and 0.312 for the left and right wheels which indicates that the brake should be released

by sending `setTrq(0)` to the wheels. This scenario satisfies the given property. Assume that no priority is defined for the message servers of the brake controller or the priority of the brake controller is not less than the others. So, the brake controller may handle `control` first while it has not received any values for the speeds (which are initially 0). Thus, the given property is wrongly violated.

The size of the state space generated by Afra has 10,088 states and 12,732 transitions. If we remove the priorities defined for the instances of actors, the resulting size is increased to 1,659,463 states and 6,326,764 transitions. If we also remove the priorities for the message servers within the `BrackCtrl` and `CANBusNetwork` classes, the resulting state space will have 2,523,309 states and 10,313,561 transitions. The priorities among the actors implicitly model a scheduling policy for executing actors to resolve nondeterminism due to their concurrent execution while the priorities among the message servers model a scheduling policy to resolve the nondeterminism caused by messages arriving at the same time.

5 Holistic Analysis of Cyber-Physical Systems

The two orthogonal features of variability handling mechanisms and priority can be used together. This combination of usage makes it possible to specify variability in the domain of embedded and cyber-physical systems. Using the features of upgraded Timed Rebeca, we may define different communication mechanisms, like broadcast or specific protocols like in a CAN bus in a more structured way and hence more usable and understandable for the engineers. We can model periodic and sporadic events and order their handling where necessary. This allows us to model different configurations for cyber-physical systems and perform a holistic analysis of safety and timing features. We revise our running example in Fig. 3 to extend its domain application, inspired from [11] in the automotive domain.

The extension to the Timed Rebeca model in Fig. 3 is brought by three modifications: 1) replacing the feature annotation by polymorphism and making `Sensor` and `ComptUnit` abstract classes, 2) adding another network type `CANBusNetwork`, similar to our case study in Sect. 4.3, and 3) adding variability to the abstract class `CompUnit` to communicate with entities over a CAN bus. As we need all the instances of `CompUnit` variations to either communicate over CAN or not, we add this variability by using a feature annotation (instead of defining two subclasses for each variant). The resulting model is shown in Fig. 15, with its main block listed in Fig. 16. We explain each modification in detail.

We remove the variables `FT_PERIODIC_SENSOR` and `FT_SPORADIC_SENSOR` and instead define `PeriodicSensor` and `SporadicSensor` as the subclasses of the abstract class `Sensor`. By making the superclass `Sensor` an abstract class, we can specify the common behavior between the two variant subclasses in the superclass as much as possible like the constructor. Substituting polymorphism for feature annotation allows having two variants of `Sensor` class within a model simultaneously. With the same discussion, we also remove the variables `FT_PERIODIC_PUSH` and `FT_IMMEDIATE_PUSH` and define `CompUnitPeriodic`

```
featurevar FT_SIMPLE_NETWORK;
featurevar FT_CAN_NETWORK;
featurevar FT_TDMA_NETWORK;

abstract Sensor{
   statevars{CompUnit cu;}
   Sensor(CompUnit cu1) {
      cu = cu1;
      self.gatherData();}
   abstract msgsrv gatherData();
}
reactiveclass PeriodicSensor
     extends Sensor(3){
  // constructor omitted
   msgsrv gatherData() {
      cu.receiveData(0);
      self.gatherData()
           after(2);}
}
reactiveclass SporadicSensor
     extends Sensor(3){
  // constructor omitted
   msgsrv gatherData() {
     cu.receiveData(0);
     self.gatherData()
          after(?(1,2,3));}
}
abstract CompUnit {
   statevars {
      Network network;
      @feature(FT_CAN_NETWORK)
      int priority}
   CompUnit(Network net, int pr){
      network = net;
      @feature(FT_CAN_NETWORK)
      priority = pr;
      init();}
   abstract msgsrv
        receiveData(byte d);
   abstract void init();
   void transfer(byte data){
     @feature(FT_CAN_NETWORK)
     if (priority==1)
        network.sendHigh(data);
     else if (priority==2)
        network.sendMedium(data);
     network.send(data);}
}
env int BUFFER_SIZE = 4;
reactiveclass CompUnitPeriodic
     extends CompUnit(3){
   statevars {
      int[BUFFER_SIZE] buffer;
      int cnt;}
  // constructor omitted
   void init(){self.process();}
   msgsrv receiveData(byte d) {
      buffer[cnt++] = d;}
   msgsrv process() {
    for(int i=cnt;cnt>0;cnt--)
        transfer(buffer[i]);
      self.process() after(1);}
}
reactiveclass CompUnitImmediate
     extends CompUnit(2){
  // constructor omitted
   void init(){}
   msgsrv receiveData(byte d){
     transfer(data);}
}
interface Network {
   @feature(FT_CAN_NETWORK)
   msgsrv sendHigh(byte data);
   @feature(FT_CAN_NETWORK)
   msgsrv sendMedium(data);
   msgsrv send(byte data);
}
reactiveclass MACBNetwork
     implements Network(3) {
  // constructor omitted
   msgsrv send(byte data) { }
}
reactiveclass TDMANetwork
     implements Network(3) {
  // constructor omitted
   msgsrv send(byte data) { }
}
reactiveclass CANBusNetwork
     implements Network(3) {
  // constructor omitted
   @priority(1)
   msgsrv sendHigh(byte data){}
   @priority(2)
   msgsrv sendMedium(data){}
   @priority(3) //low
   msgsrv send(byte data){}
}
```

Fig. 15. Extending the domain application of the sensor network running example shown in Fig. 3 by adding a CAN Bus and using the features of upgraded Timed Rebeca

```
82 main {
83    @priority(1);
84    PeriodicSensor sr1():(cu1);
85    @priority(2);
86    CompUnitPeriodic cu1():(network, FT_CAN_NETWORK?2:0);
87    @priority(1);
88    SporadicSensor sr2():(cu2);
89    @priority(2);
90    CompUnitImmediate cu2():(network, FT_CAN_NETWORK?1:0);
91    @priority(3);
92    @feature(FT_SIMPLE_NETWORK)
93    MACBNetwork network():();
94    @feature(FT_TDMA_NETWORK)
95    TDMANetwork network():();
96    @feature(FT_CAN_NETWORK)
97    CANBusNetwork network():();
98 }
```

Fig. 16. Actor instantiations for the upgraded sensor network running example shown in Fig. 15

and `CompUnitImmediate` as the subclasses of the abstract class `CompUnit`. The superclass `CompUnit` has one abstract function `init` which is called in the constructor. This abstract method contains the specific initialization needed for each variant subclass. In `CompUnitPeriodic`, this method must send a message to itself to start periodic execution while no initialization is required in `CompUnitImmediate`.

Thanks to the priority feature, we add another network type `CANBusNetwork`, suitable for modeling the network in the automotive domain. By using the feature expression `@feature(FT_CAN_NETWORK)`, we add variability to the abstract class `CompUnit`. This feature adds a `priority` state variable to the class. As messages are transmitted over a CAN bus based on their priorities, we define a message server for each message priority in `CANBusNetwork` and assign a priority to each message server using the priority feature. The `priority` variable of `CompUnit` indicates the priority of messages (received from the sensors). The method `transfer` of `CompUnit` sends messages to the network. If the network is a CAN bus, when the variable `FT_CAN_NETWORK` is set, then it calls the corresponding message server of `CANBusNetwork` based on the value of `priority`.

A configuration of the model where only the feature `FT_CAN_NETWORK` is present gives the model of two connected ECUs communicating over a CAN bus in a car. The first sensor instance `sr1` models the wheel sensor which periodically sends the speed of the wheel to its wheel controller, represented by `cu1`. The second sensor instance `sr2` is the gear sensor which sends the level of gear upon any change to its controller, represented by `cu2`.

6 Related Work

Apart from variability-aware extensions of modeling notations based on transition systems and process algebras (comprehensively surveyed in [10]), several formal modeling languages have been extended to support variability, including fPromela [16], fSMV [17], and an extension of Event-B [68]. Having our focus on formal modeling of asynchronously communicating distributed systems, the most notable language is Abstract Behavioral Specification (ABS) [1,35], which follows the concurrent object-oriented style of the actor model, and enables variability modeling using a delta-oriented approach [15,19,58]. Unlike ABS, our way to handle variability in Rebeca family of languages is through feature annotation and polymorphism which models the entire family behavior in one place. Verification of software product lines has a relatively long history. This includes the works based on modal I/O automata [45], PL-CCS [25], and early results based on Featured Transition Systems [18]. More recent advances on the verification of SPLs include a wide range of techniques such as static analysis [6,8], parity games [7], proof plans [44], and correct-by-construction approach [13].

Lee et.al. proposed Lingua Franca as a language for developing deterministic actors [46]. Lingua Franca resolve nondeterminisitc execution among actors using predefined order of executions for actors. The idea of associating priority to actors in Rebeca to make the model more deterministic is inspired from Lingua Franca. In comparison with [46], although actors can be deterministic in Rebeca, they are allowed to have nondeterministic behavior. This means that modelers are allowed to express the required level of nondeterminism in models.

7 Conclusion

In this paper, we presented an overview of the language features of Timed Rebeca to support variability management and modeling determinism. The approach to variability management is feature-oriented and is done using feature variables. By annotating parts of the model source code with feature expressions, we can bind model parts to a number of product configurations. Moreover, class polymorphism can be used to manage variability by providing alternative implementations of model components. This way, the whole product line can be modeled in a single artifact which explicitly models the variability in structure and behavior of the model. This enables the opportunity to analyze the whole product line model at once as opposed to analyzing every product individually. The theory behind verification of the whole product line has been developed and partially implemented and is a future step in the development of Afra, the Timed Rebeca model checker. Currently, Afra supports feature variables and verification of the individual products specified through a valuation of the feature variables. As a future work, we plan to study the feasibility of applying variability encoding [54] to defer the variability resolution time from compile-time to state-space generation time, which may increase the efficiency of whole-family verification.

We also showed how Timed Rebeca models can be made more deterministic by assigning priorities to message servers and actors. This enables the modelers

to bring in assumptions about the execution environment or outside entities. In addition to making the model closer to a set of real world applications, this may result in (possibly significant) reduction in the size of the state space and make it more practical to analyze more complex systems. Both of these aspects enhance the practicality of the Timed Rebeca toolset to be used in industrial settings.

References

1. The ABS language. https://abs-models.org/
2. Afra toolset homepage. https://rebeca-lang.org/alltools/Afra
3. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
4. Ayala, I., Papadopoulos, A.V., Amor, M., Fuentes, L.: ProDSPL: proactive self-adaptation based on dynamic software product lines. J. Syst. Softw. **175**, 110909 (2021)
5. Batory, D.: Feature models, grammars, and propositional formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005). https://doi.org/10.1007/11554844_3
6. ter Beek, M.H., Damiani, F., Lienhardt, M., Mazzanti, F., Paolini, L.: Efficient static analysis and verification of featured transition systems. Empir. Softw. Eng. **27**(1), 10 (2022)
7. ter Beek, M.H., van Loo, S., de Vink, E.P., Willemse, T.A.: Family-based SPL model checking using parity games with variability. In: FASE, vol. 20, pp. 245–265 (2020)
8. Beek, M.H.T., Damiani, F., Lienhardt, M., Mazzanti, F., Paolini, L.: Static analysis of featured transition systems. In: Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A, pp. 39–51 (2019)
9. Behjati, R., Yue, T., Briand, L., Selic, B.: SimPL: a product-line modeling methodology for families of integrated control systems. Inf. Softw. Technol. **55**(3), 607–629 (2013)
10. Benduhn, F., Thüm, T., Lochau, M., Leich, T., Saake, G.: A survey on modeling techniques for formal behavioral verification of software product lines. In: Proceedings of the Ninth International Workshop on Variability Modelling of Software-Intensive Systems, pp. 80–87 (2015)
11. Bengtsson, H.H., Hiller, M., Mattsson, F., Bengtsson, J.: Holistic analysis of task scheduling and message scheduling in automotive centralised E/E architecture. In: IEEE/SA Ethernet/IP@Automotive Techonology Day (2020)
12. Boer, F.D., et al.: A survey of active object languages. ACM Comput. Surv. (CSUR) **50**(5), 1–39 (2017)
13. Bordis, T., Runge, T., Schaefer, I.: Correctness-by-construction for feature-oriented software product lines. In: Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, pp. 22–34 (2020)
14. Cionca, V., Newe, T., Dadârlat, V.: TDMA protocol requirements for wireless sensor networks. In: 2008 Second International Conference on Sensor Technologies and Applications (sensorcomm 2008), pp. 30–35. IEEE (2008)
15. Clarke, D., Muschevici, R., Proença, J., Schaefer, I., Schlatte, R.: Variability modelling in the ABS language. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 204–224. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_11

16. Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.Y.: Model checking software product lines with SNIP. Int. J. Softw. Tools Technol. Transf. **14**, 589–612 (2012)
17. Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.Y.: Formal semantics, modular specification, and symbolic verification of product-line behaviour. Sci. Comput. Program. **80**, 416–439 (2014)
18. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A., Raskin, J.F.: Model checking lots of systems: efficient verification of temporal properties in software product lines. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, vol. 1, pp. 335–344 (2010)
19. Damiani, F., Hähnle, R., Kamburjan, E., Lienhardt, M., Paolini, L.: Variability modules. J. Syst. Softw. **195**, 111510 (2023)
20. 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)
21. Eckel, B.: Thinking in Java, 4th edn. Prentice Hall (2006)
22. Fadhlillah, H.S., Feichtinger, K., Meixner, K., Sonnleithner, L., Rabiser, R., Zoitl, A.: Towards multidisciplinary delta-oriented variability management in cyber-physical production systems. In: Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems, pp. 1–10 (2022)
23. Filipovikj, P., Mahmud, N., Marinescu, R., Seceleanu, C., Ljungkrantz, O., Lönn, H.: Simulink to UPPAAL statistical model checker: analyzing automotive industrial systems. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 748–756. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_46
24. Forcina, G., et al.: Safe design of flow management systems using Rebeca. J. Inf. Process. **28**, 588–598 (2020)
25. Gruler, A., Leucker, M., Scheidemann, K.: Modeling and model checking software product lines. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 113–131. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68863-1_8
26. Haller, P.: On the integration of the actor model in mainstream technologies: the scala perspective. In: Proceedings of the 2nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions, pp. 1–6 (2012)
27. von Hanxleden, R., et al.: Pragmatics twelve years later: a report on lingua franca. In: Margaria, T., Steffen, B. (eds.) ISoLA 2022, Part II. LNCS, vol. 13702, pp. 60–89. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-19756-7_5
28. Haugen, Ø., Wasowski, A., Czarnecki, K.: CVL: common variability language. In: Kishi, T., Jarzabek, S., Gnesi, S. (eds.) 17th International Software Product Line Conference, SPLC 2013, Tokyo, Japan, 26–30 August 2013, p. 277. ACM (2013)
29. Hewitt, C.: Viewing control structures as patterns of passing messages. Artif. Intell. **8**(3), 323–364 (1977)
30. Hewitt, C.: Actor model of computation: scalable robust information systems. arXiv preprint arXiv:1008.1459 (2010)
31. Hinchey, M., Park, S., Schmid, K.: Building dynamic software product lines. Computer **45**(10), 22–26 (2012)
32. Jafari, A., Khamespanah, E., Sirjani, M., Hermanns, H., Cimini, M.: PTRebeca: modeling and analysis of distributed and asynchronous systems. Sci. Comput. Program. **128**, 22–50 (2016)

33. Jahandideh, I., Ghassemi, F., Sirjani, M.: An actor-based framework for asynchronous event-based cyber-physical systems. Softw. Syst. Model. **20**(3), 641–665 (2021)
34. Jahandideh, I., Ghassemi, F., Sirjani, M.: Hybrid Rebeca: modeling and analyzing of cyber-physical systems. In: Chamberlain, R., Taha, W., Törngren, M. (eds.) CyPhy/WESE -2018. LNCS, vol. 11615, pp. 3–27. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-23703-5_1
35. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_8
36. 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)
37. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon University, Pittsburgh, PA, Software Engineering Institute (1990)
38. Khakpour, N., Jalili, S., Talcott, C., Sirjani, M., Mousavi, M.: Formal modeling of evolving self-adaptive systems. Sci. Comput. Program. **78**(1), 3–26 (2012)
39. Khakpour, N., Khosravi, R., Sirjani, M., Jalili, S.: Formal analysis of policy-based self-adaptive systems. In: Proceedings of the 2010 ACM Symposium on Applied Computing, pp. 2536–2543 (2010)
40. Khamespanah, E., Sirjani, M., Kaviani, Z.S., Khosravi, R., Izadi, M.J.: Timed Rebeca schedulability and deadlock freedom analysis using bounded floating time transition system. Sci. Comput. Program. **98**, 184–204 (2015)
41. Khamespanah, E., Sirjani, M., Khosravi, R.: Afra: an eclipse-based tool with extensible architecture for modeling and model checking of Rebeca family models. In: Hojjat, H., Ábrahám, E. (eds.) FSEN 2023. LNCS, vol. 14155, pp. 72–87. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-42441-0_6
42. Khamespanah, E., Sirjani, M., Mechitov, K., Agha, G.: Modeling and analyzing real-time wireless sensor and actuator networks using actors and model checking. Int. J. Softw. Tools Technol. Transf. **20**, 547–561 (2018)
43. Krüger, J., et al.: Beyond software product lines: variability modeling in cyber-physical systems. In: Proceedings of the 21st International Systems and Software Product Line Conference, vol. A, pp. 237–241 (2017)
44. Kuiter, E., Knüppel, A., Bordis, T., Runge, T., Schaefer, I.: Verification strategies for feature-oriented software product lines. In: Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems, pp. 1–9 (2022)
45. Larsen, K.G., Nyman, U., Wąsowski, A.: Modal I/O automata for interface and product line theories. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 64–79. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71316-6_6
46. Lohstroh, M., Menard, C., Bateni, S., Lee, E.A.: Toward a lingua franca for deterministic concurrent systems. ACM Trans. Embed. Comput. Syst. **20**(4), 36:1–36:27 (2021). https://doi.org/10.1145/3448128
47. Lohstroh, M., et al.: Actors revisited for time-critical systems. In: Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, 02–06 June 2019, p. 152. ACM (2019)
48. 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)
49. Menard, C., et al.: High-performance deterministic concurrency using lingua franca. CoRR abs/2301.02444 (2023)
50. Pfeiffer, O., Ayre, A., Keydel, C.: Embedded Networking with CAN and CANopen, 1st edn. Copperhill Media Corporation (2008)
51. Pohl, K., Böckle, G., Van Der Linden, F.: Software Product Line Engineering, vol. 10. Springer, Heidelberg (2005). https://doi.org/10.1007/3-540-28901-1
52. Rabiser, R., Zoitl, A.: Towards mastering variability in software-intensive cyber-physical production systems. Procedia Comput. Sci. **180**, 50–59 (2021)
53. Reynisson, A.H., et al.: Modelling and simulation of asynchronous real-time systems using timed Rebeca. Sci. Comput. Program. **89**, 41–68 (2014)
54. von Rhein, A., Thüm, T., Schaefer, I., Liebig, J., Apel, S.: Variability encoding: From compile-time to load-time variability. J. Log. Algebraic Methods Program. **85**(1), 125–145 (2016)
55. Sabouri, H., Jaghoori, M.M., de Boer, F., Khosravi, R.: Scheduling and analysis of real-time software families. In: 2012 IEEE 36th Annual Computer Software and Applications Conference, pp. 680–689. IEEE (2012)
56. Sabouri, H., Khosravi, R.: Modeling and verification of reconfigurable actor families. J. Univers. Comput. Sci. **19**(2), 207–232 (2013)
57. Sabouri, H., Khosravi, R.: Reducing the verification cost of evolving product families using static analysis techniques. Sci. Comput. Program. **83**, 35–55 (2014)
58. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 77–91. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15579-6_6
59. Sehr, M.A., et al.: Programmable logic controllers in the context of industry 4.0. IEEE Trans. Industr. Inform. **17**(5), 3523–3533 (2021)
60. Sharifi, Z., Khosravi, R., Sirjani, M., Khamespanah, E.: Towards formal analysis of vehicle platoons using actor model. In: 2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), vol. 1, pp. 1820–1827. IEEE (2020)
61. Sharifi, Z., Mosaffa, M., Mohammadi, S., Sirjani, M.: Functional and performance analysis of network-on-chips using actor-based modeling and formal verification. Electron. Commun. Eur. Assoc. Softw. Sci. Technol. **66**, 1–16 (2013)
62. Sirjani, M., Jaghoori, M.M.: Ten years of analyzing actors: Rebeca experience. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 20–56. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24933-4_3
63. Sirjani, M., Khamespanah, E.: On time actors. In: Ábrahám, E., Bonsangue, M., Johnsen, E.B. (eds.) Theory and Practice of Formal Methods. LNCS, vol. 9660, pp. 373–392. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30734-3_25
64. Sirjani, M., Lee, E.A., Khamespanah, E.: Verification of cyberphysical systems. Mathematics **8**(7), 1068 (2020)
65. Sirjani, M., Movaghar, A.: An actor-based model for formal modelling of reactive systems: Rebeca. Technical report CS-TR-80-01, Tehran, Iran (2001)
66. Sirjani, M., Movaghar, A., Mousavi, M.: Compositional verification of an object-based reactive system. In: Workshop on Automated Verification of Critical Systems (AVoCS 2001) (2001)
67. Sirjani, M., Movaghar, A., Shali, A., De Boer, F.S.: Modeling and verification of reactive systems using Rebeca. Fund. Inform. **63**(4), 385–410 (2004)

68. Sorge, J., Poppleton, M., Butler, M.: A basis for feature-oriented modelling in event-B. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, p. 409. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11811-1_42
69. Varshosaz, M., Khosravi, R.: Modeling and verification of probabilistic actor systems using pRebeca. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 135–150. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34281-3_12
70. Weyns, D.: Software engineering of self-adaptive systems. In: Handbook of Software Engineering, pp. 399–443. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-00262-6_11
71. Yousefi, F., Khamespanah, E., Gharib, M., Sirjani, M., Movaghar, A.: VeriVANca framework: verification of VANETs by property-based message passing of actors in Rebeca with inheritance. Int. J. Softw. Tools Technol. Transf. **22**(5), 617–633 (2020)