Modeling and Analysis of Actors

Bridging Theory and Practice with Rebeca

Ehsan Khamespanah, Marjan Sirjani, Ramtin Khosravi

April 5, 2025

Shalifiers Publisher

Contents

List of Figures

List of Tables

Modeling and Analysis

1 Actors

The Actor model is a well-established paradigm for modeling distributed and asynchronous component-based systems. This model was originally introduced by Hewitt as an agent-based language where goal directed agents did logical reasoning [1]. Subsequently, the actor model developed as a model of concurrent computation for open distributed systems where *actors* are the concurrently executing entities [2]. It is used in distributed computing, concurrent systems, and parallel processing to model and manage the flow of control and communication among different components of a system.

Actors are units of concurrency, with no shared variables, communicating via asynchronous message passing. Actor provides services that may be requested via messages from other actors. A message is buffered until the provider is ready to execute the message. As a result of processing a message, an actor may change its internal state and send messages to other actors, including itself. Thus, the general behavior of each actor is an infinite loop of taking a message from the mailbox and executing the corresponding routine [1]. This way, actors based systems can be assumed as event-driven systems, such that the message passing among actors constitutes events, and consuming an event is realized by serving a received message.

1: The actor waits if there is no message in its mailbox.

An event graph is a powerful tool for modeling interactions of actors. The event graph model represents events (e.g., messages, state changes) and their causal/temporal relationships. In a given event graph, nodes are events and edges denote dependencies (e.g., "event A caused event B"). It often used to model causality, ordering, or data flow in an event based system. As mentioned before, in the Actor Model, messages sent between actors can be treated as events. These events form the nodes of an event graph, with edges reflecting how one message triggers subsequent messages. The message passing among actors creates causal chains (e.g., actor A sends a message to B, which then sends to C). An event graph can explicitly model these chains too. The event graph helps represent the flow of messages and how message can be associated with actors, providing a clear way to track the progression of events and the interactions between different components of a system. Here, we use event graph to model sequence of messages in the system and how they are associated with actors, then model it using the Rebeca family modeling languages.

An event graph is a directed graph where the graph nodes represent events in a system. Edges correspond to causing other events [3]. Jagged incoming edges denote an initial event. Edges can optionally be associated with a boolean condition for causing an event and/or a time delay which means that an event will be caused after the delay. Figure 1.1 shows an example of an event graph where *EventA* causes *EventB* after a delay of `t` time units and if condition *cond* is true. In the same way, *EventB* causes *EventC* unconditionally. In this figure *EventA* is associated with *Actor 1* and two events *EventB* and *EventC* are associated with *Actor 2*.

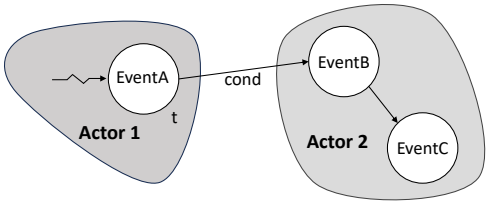


Figure 1.1: Example of an event graph

For the better understanding of how actors can be used for modeling a system, we developed a simple running example. This system is defined as a single railway bridge, which is connected to two ground railways on each side. Each of the railways on the ground is used for trains traveling in one direction, so, only one train is allowed to be on the bridge from each side. As the bridge has only one railway, trains arriving from each direction can not pass simultaneously. The bridge controller system, is a system to signal the trains to stop or pass from the bridge. The controller gets notified when a train arrives from each side, and when the train leaves the bridge.

In the first step we model the bridge control system using the event graph, shown in Figure 1.2. In this event graph, the `reachBridge` message is the initial message and serving it causes asking for arrival to the bridge. Based on the current state of the bridge, it may allow the train to pass, realized by sending `youMayPass`. A train on the bridge sends `leave` message to inform the controller that it leaves the bridge. The controller sends `passed` to the train to confirm that leaving is completed. To model the periodic behavior of the system, trains retry to pass the bridge as soon as passing the bridge.

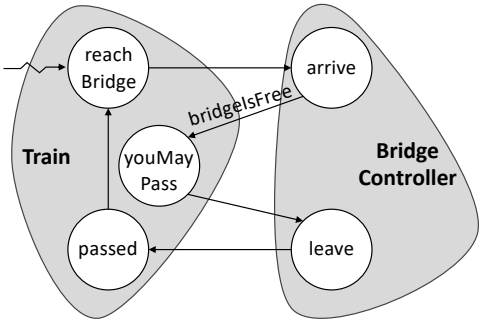


Figure 1.2: The event graph of the bridge controller system

Core Rebeca 2

2.1 What is Rebeca

Rebeca [4, 5] is a class-based, imperative interpretation of the actor model. Rebeca has been designed in an effort to facilitate the verification process for practitioners who are not experts in formal methods. From one point of view Rebeca as a Java-like language, it is easy to use for software engineers, and from another point of view it is a modeling language with formal semantics and formal verification support. A model in Rebeca consists of concurrently executing reactive objects, rebecs.

Computation takes place by asynchronous message passing between actors and execution of the corresponding message servers of messages. Each message is put in the queue of the receiver actor and specifies a unique method to be invoked when the message is serviced. In Rebeca, each actor has a unique thread of control. An actor takes a message from its queue and executes the corresponding message server then takes another message. There is no intra-actor concurrency, meaning that the execution of a message server must be completed 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. This arbitrary ordering of actors is a source of nondeterminism in the behavior of the model in Rebeca.

The core of Rebeca modeling language is intentionally kept simple and the language with simple constructs is called Core Rebeca. For various purposes, several extensions have been proposed, including Timed Rebeca [6] for the domain of real-time systems, Hybrid Rebeca [7] for the domain of cyber-physical systems, pRebeca [8] for modeling and analysis of probabilistic systems, and PTRebeca [9] for probabilistic timed systems. Rebeca is equipped Afra [10] as an Eclipse-based modeling and verification development environment for Core Rebeca, Timed Rebeca, and Probabilistic Timed Rebeca.

2.2 The Structure of Core Rebeca Models

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. The simplified Rebeca model of the running example is presented in Listing 2.1. In this Rebeca model, there are two classes of actors: `Train` and `BridgeController`.

The `main` block in lines 9-12 defines one instance of each class and specifies arguments passed as known rebec identifiers and parameters of constructors. We will talk about constructors and known rebecs in the following section.

Listing 2.1: The structure of the Rebeca model of the bridge control system

```
1  reactiveclass Train(2) {
2    /* Reactive class definition */
3  }

5  reactiveclass BridgeController(2) {
6    /* Reactive class definition */
7  }

9  main {
10   BridgeController controller():();
11   Train train1(controller):(1);
12 }
```

An instance of a reactive class is an actor in the system (which is also called a *rebec*). Each actor has a mailbox, which is a bounded FIFO queue for Core Rebeca. Although in the actor model, the queue length is unbounded, in Rebeca, the maximum queue size has to be defined in the class definition[1]. This size shall be indicated in parenthesis next to the reactive class name, as shown in line 1, i.e. queue size of 2 for actors of type `Train`.

1: The size of the queue is bounded to prevent the creation of infinite transition systems and makes analysis of models possible.

2.2.1 Reactive Class Definition

The class definitions of `Train` and `BridgeController` reactive classes are presented in Listing 2.2. In a class definition, there are two variable definition blocks: `knownrebecs` and `statevars`. Known rebecs block contains references to actors which the actors instantiated from this reactive class are allowed to send messages to [2]. As shown in line 3, the actors of type `Train` are allowed to send messages to an actor of type `BridgeController` using `bc` variable. The binding of known rebecs is performed in the main block. As shown in line 44, `controller` is put in the first parenthesis of the definition of `train1` as its only known rebec. In the case of having more than one known rebecs, a comma-separated list of variables is put in the first parenthesis. The statevars block contains variable definitions which are needed for holding the state of an actor. The state of the actors of type `Train` is presented by an id, mentioned in line 6. The state of `BridgeController` is presented by the number of trains on the bridge, shown in line 25. `BridgeController` does not have a known rebec.

2: Known actors can also be used if there is a need to examine the sender of messages. An example of using known rebecs for this purpose is presented in Listing 2.15.

After these two variable definition sections, method definitions are presented. Modelers are allowed to guarantee the initialization of the actors' state variables by providing constructor methods. If a reactive class has a constructor, it is called automatically when actors are instantiated from this reactive class in the `main` section. In Rebeca, the name of the constructor is the same as the name of the reactive class. The constructor may have input parameters to allow a modeler to specify the initialization values of actors. lines 9 to 11 shows the constructor definition of `Train`. This constructor has one input parameter and the value of this parameter is set in the second parenthesis in line 44. Note that in the case of not having a constructor definition, a *default constructor* is added to the reactive class definition[3].

3: A default constructor does not have any input parameter and statements.

The reactive class also contains message-handling methods. These methods are called message servers of this reactive class and their duty is to serve the incoming messages. In addition to message servers which are defined using the keyword `msgsrv`, local methods can also be defined in the body of a reactive class. Local methods are private methods of the reactive class and can be called by constructors, message servers, and other local methods of the reactive class which contain them. A method call can result in a return value to the caller[4]. The same as constructors, definitions of message servers and local methods can have input parameters. In this document, we are using *method* to refer to any of the message servers, local methods, and constructors. Listing 2.2 shows that `Train` has three messages servers and one constructor. `BridgeController` has two message servers and one local method, i.e. `updateNumberOfTrainsOnTheBridge`.

4: From access control point of view, constructors and message servers of a reactive class can be assumed as public methods and local methods, state variables, and known rebecs as protected.

```
1  reactiveclass Train(2) {
2    knownrebecs {
3      BridgeController bc;
4    }
5    statevars {
6      byte id;
7    }
8
9    Train(byte myId) {
10     /* Constructor definition statements*/
11   }
12   msgsrv reachBridge() {
13     /* Message server definition statements*/
14   }
15   msgsrv youMayPass() {
16     /* Message server definition statements*/
17   }
18   msgsrv passed() {
19     /* Message server definition statements*/
20   }
21 }

23 reactiveclass BridgeController(2) {
24   statevars {
25     byte trainsOnTheBridge;
26   }
27
28   BridgeController() {
29     /* Constructor definition statements*/
30   }
31   msgsrv arrive() {
32     /* Message server definition statements*/
33   }
34   msgsrv leave() {
35     /* Message server definition statements*/
36   }
37   void updateNumberOfTrainsOnTheBridge(byte value) {
38     /* Local method definition statements*/
39   }
40 }

42 main {
43   BridgeController controller():();
44   Train train1(controller):(1);
```

Listing 2.2: The class definition of `Train` and `BridgeController` of the bridge control system

```
45 | }
```

2.2.2 Variable Types

Attention

The state variables of reactive classes and parameters of message servers can not be floating-point variables.

The primitive types in Rebeca are presented in Table 2.1. It includes boolean, numerical, and floating-point types with different ranges. The definition of these types is the same as that of in Java. Names of defined reactive classes are also assumed as type names. These variables are called *reference to actor* variables.

Table 2.1: Primitive types in Rebeca.

Primitive Type	Size	Minimum	Maximum
boolean	1 bit	—	—
byte	8 bits	-128	$+127$
short	16 bits	-2^{15}	$+2^{15}-1$
int	32 bits	-2^{31}	$+2^{31}-1$
float	32 bits	IEEE754	IEEE754
void	—	—	—

Arrays can be defined in a Rebeca model. The length of an array should be specified in its declaration. As an example, x is an array of bytes with length 3 in the following declaration:

```
byte[3] x;
```

An array is indexed from 0. So there are three indexes for the above declaration, namely 0, 1, and 2. The elements of an array can be accessed as in general-purpose programming languages. For example, `x[2]` denotes the last element of the array. Arrays have strict type checking in Rebeca, i.e. if a message server is willing to accept a variable of type `byte[2]`, a variable of type `byte[3]` cannot be passed to it. Rebeca supports multi-dimensional array definition too. As an example, y is a two-dimensional array of int variables in the following declaration:

Attention

Only one-dimensional arrays are allowed to be used as the type of parameters of message servers and local methods.

```
int[4][5] y;
```

2.2.3 Expressions

The logical and arithmetic expressions in Rebeca are similar to Java, and the syntax is not included here. One can refer to Java documents for the syntax. The set of acceptable operators is given in Table 2.2. We should emphasize that casting in Rebeca is the same as in Java.

Non-deterministic expressions are the only expressions which are not in Java. The evaluation of the expression $?(e_1, e_2, \cdots, e_n)$ does not result in one value. The result of the evaluation of each of e_i's is the possible value of this expression. In Rebeca, e_i can be an expression which should be determined at compile time and different options of the non-deterministic expression have the same type. Based on this definition, the following expressions are valid non-deterministic expressions.

Operation	operators	Description
Arithmatic	+ − ∗ \ % ++ −−	mod
Assignment	= += −= /= ∗= %=	
Conditional	? :	ternary condition
Logic	&&	logical and
	||	logical or
	!	
	&	bitwise and
	|	bitwise or
Comparative	> < <= >=	
	==	equality
	! =	inequality
Cast		like Java
Non-deterministic	?(e_1, e_2, ⋯)	
Instance Of	`instanceof`	like Java

Table 2.2: Arithmetic and logic operators in Rebeca

```
?(1, 2)
?(true, false)
?(1.1, 2.6, 3.7)
?(1 + 4, 2, 9)
```

and the following non-deterministic expressions are invalid because of having inconsistent value types and can not be determined at the compile time.

```
?(1, 2, false)
?(2 + a, 3)
```

2.2.4 Statements

Methods in Rebeca contain one or more statements. There are different types of statements in Rebeca, including local variable declaration, assignment, conditional statement, for-loop and while-loop statement, method call, sending message, return, continue, and break statements.

Local Variable Declaration

In many cases, a method needs to work with a variable that is not a part of its corresponding actor's state. We call these variables local variables. These variables are declared locally in methods and the value is accessible only in the method context. The possible types of local variables are presented in Table 2.1.

Assignment Statement

Using assignment expression, a modeler is able to assign a value to state variables and local variables. Note that assigning new values to known

rebecs is not allowed in Rebeca.

Conditionals Statement

The if-else and switch conditional statements have been added to Rebeca and its syntax is like Java. The keywords `if`, `else`, `switch`, `case`, `break`, and `default` have been added too. The case expression will be valid if its value is determined at compile time.

Loops

In Rebeca, for-loop and while-loop constructs are introduced with the same syntax as in Java. To facilitate the loop iteration, `break` and `continue` keywords are included too.

Sending Message

The sending message statement is the same as the method calls of objects in Java, i.e. a reference to an actor followed by the name of one of its message servers. Reference to the actor can be one of the following:

- **known rebec**: one of the known rebecs can be used as a reference to the target actor.
- **state variables**: one of the state variables of a reactive class can be used as a reference to the target actor.
- **input parameter**: one of the input parameters of a local method or message server can be used as a reference to the target actor.
- **`self`**: the keyword `self` is used as a reference to the actor that the method has been called for [5]. You can treat the `self` just like any other object reference.
- **sender**: there is a predefined variable named `sender` in Rebeca. The receiver of a message can get a reference to the sender of the message by accessing the value of the variable `sender`. Note that to send a message using `sender` keyword, it has to be cast to one of the reactive class types. This keyword can also be used in the conditional statement using equality and inequality keywords to compare its value with another variable.

5: `self` in Rebeca is the same as `this` in Java.

Local Method Call and Returning a Value

A local method call statement is realized by using the name of the local method followed by a parenthesis containing its required parameters. A local method in Rebeca has to return a value using the `return` keyword and its syntax is the same as Java (except local methods with `void` return type).

2.3 The Model of the Bridge Control System with Statements

Listing 2.3 presents the model of the bridge control system in which its methods contain corresponding statements. Here, the constructor of `Train` initializes a train by setting its id to the given parameter value as well as sending itself a `reachBridge` message. The message server `reachBridge` of `Train` first decides nondeterministically between arriving at the bridge or trying again later (lines 14 and 15. Request for arrival is modeled by sending `arrival` message to the corresponding `BridgeController` (denoted by the reference variable `bc`). The effect of sending a message is appending the message to the message queue of the receiving actor (sometimes called its mailbox).

Upon receiving an `arrival` message, the bridge controller updates the value of the number of trains on the bridge and sends an acknowledgment to the train to allow it to be on the bridge. Note that updating the value of `trainsOnTheBridge` takes place in lines 37 and 41 by calling the local method `updateNumberOfTrainsOnTheBridge`.

```
1  reactiveclass Train(2) {
2    knownrebecs {
3      BridgeController bc;
4    }
5    statevars {
6      byte id;
7    }
8
9    Train(byte myId) {
10     id = myId;
11     self.reachBridge();
12   }
13   msgsrv reachBridge() {
14     boolean isReached = ?(true, false);
15     if(isReached)
16       bc.arrive();
17     else
18       self.reachBridge();
19   }
20   msgsrv youMayPass() {
21     self.passed();
22   }
23   msgsrv passed() {
24     bc.leave();
25   }
26 }

28 reactiveclass BridgeController(2) {
29   statevars {
30     byte trainsOnTheBridge;
31   }
32
33   BridgeController() {
34     trainsOnTheBridge = 0;;
35   }
36   msgsrv arrive() {
37     updateNumberOfTrainsOnTheBridge(+1);
38     ((Train)sender).youMayPass();
```

Listing 2.3: The complete Core Rebeca model of the bridge control system

```
39    }
40    msgsrv leave() {
41      updateNumberOfTrainsOnTheBridge(-1);
42    }
43    void updateNumberOfTrainsOnTheBridge(byte value) {
44      trainsOnTheBridge += value;
45    }
46  }

48  main {
49    BridgeController controller():();
50    Train train1(controller):(1);
51  }
```

2.4 Analysis of Core Rebeca Models

Afra [10] is the graphical user interface for modeling and analysis of Rebeca family models. Afra user interface consists of five main sections which are, the projects browser, model and property editor, model-checking result view, and counterexample and its details views. An overview of Afra is depicted in Figure 2.1. Afra generates transition systems of given models and analyzes them against given correctness conditions. Transition systems are basically directed graphs where nodes represent the states of the system, and edges model how transitions among states take place. This way, a state can be assumed as a snapshot of the system at a certain moment of its behavior [11]. Considering the running example, a state of this model indicates valuations of the state variables of `train1` and `controller`, in addition to their queues content at that moment. Transitions specify the way that the system evolves from one state to another state. In the case of the running example, a transition shows how serving a message of an actor changes the values of its state variables and what the newly sent messages are.

Correctness conditions in Afra are presented as a set of propositional logic expressions (correctness invariants) and a set of temporal logic formulae in the form of LTL [11, 12]. Afra analyzes models against a set of predefined correctness invariants, together with user-defined correctness conditions. The predefined correctness invariants are:

- Deadlock Avoidance: there is no state that all of the queues of actors are empty,
- Queue Overflow Avoidance: there is no condition in which an actor wants to send a message to another actor that its queue is full,
- No Access to Null Reference: access to all of the reference-to-actor variables is safe. Reference-to-actor variables can be set to `null`. In this case, upon access to a reference-to-actor variable which is set to null, the violation of this correctness condition is reported.
- Safe Access to Arrays Elements: There is no out-of-bound access to the elements of arrays.

In the case of violation of the above-mentioned correctness invariants in a state of a Rebeca model, its corresponding counterexample is shown in Afra. A counterexample is a trace of states of the model that starts from the initial state of the model and ends in the state that violates

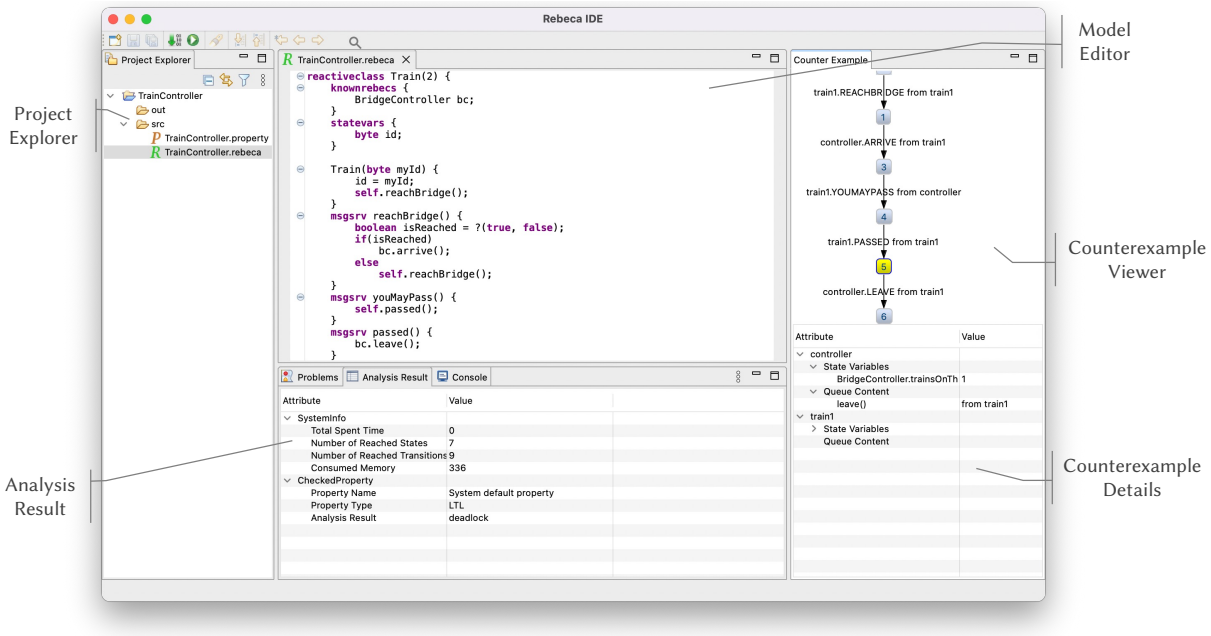


Figure 2.1: Afra: the graphical user interface of modeling and analysis of Rebeca family models

correctness invariants. The same happens for the violation of user-defined correctness invariants.

As shown in Figure 2.2, the model of Listing 2.3 has a trace which results in the violation of deadlock avoidance assertion. It is because of the fact that after one round of execution, `train1` passed the bridge and nothing happens after that. To avoid this assertion violation, we have to modify the model to let it continue its execution. Passing the bridge periodically is one of the solutions to this requirement. This can be realized by sending the `reachBridge` message to `train1` after passing the bridge, as shown in line 5 of Listing 2.4. In this listing, only the parts that are modified in comparison with the code of Listing 2.3 are shown. The "`...`" symbols at the beginning of each part are representatives of not-modified codes.

```
1 reactiveclass Train(2) {
2   ...
3   msgsrv passed() {
4     bc.leave();
5     self.reachBridge();
6   }
7 }
```

Listing 2.4: The modified version of the bridge control system which does not have deadlock

Applying this modification fixes the deadline of the model and a transition system with 7 states is generated by Afra for this model. This transition system is shown in Figure 2.3. In the initial state of the transition system (the state `S0`), the only actor which has a message in its queue is `train1` and it can start serving the `reachBridge` message. But, `S0` has two outgoing transitions with the same label, i.e. `train1.reachBridge`. It is because of the fact that there is a nondeterministic assignment in the `reachBridge` message server and executing this message server results in two different behaviors. In one of them `train1` sends `reachBridge` to

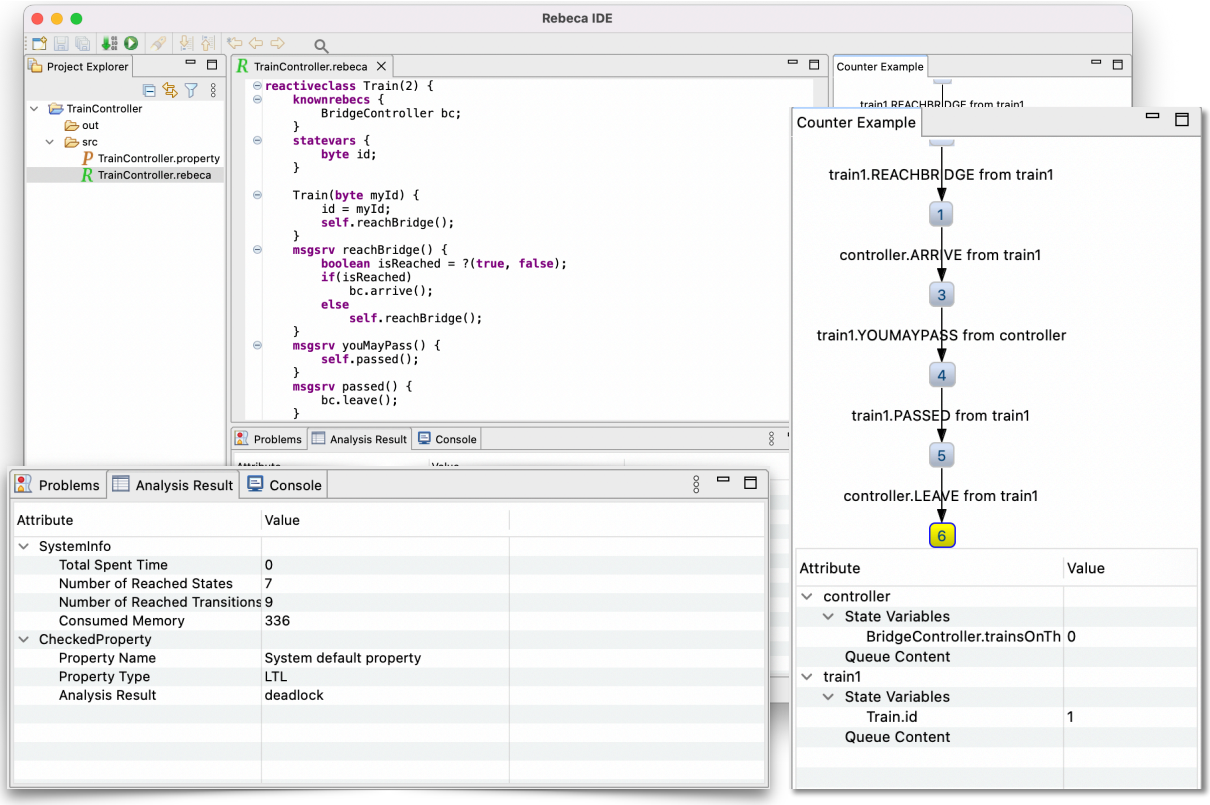


Figure 2.2: The analysis result viewer of Afra which reports having deadlock in the model and presents the counterexample of the violation of this correctness invariant

itself and in the other one it sends `arrive` to `controller`. In the states `S1`, `train1` successfully sent `arrive` to the bridge controller and waits for the response of the bridge controller. In the state `S5`, both the train and bridge controller have one message in their queues. So, either of them can start serving its received message and `S5` has to have two different outgoing transitions. But, `train1` has `reachBridge` in its message queue, which contains a non-deterministic assignment with two alternative values. So, there are three different outgoing transitions from `S5`. In `S6`, the bridge controller has two messages in its queue. As `controller` is the only actor which has a message in its queue, it starts serving the `leave` message [6].

In the second step of developing the model of the bridge control system, we add more trains to the model and try to examine its correctness properties. The only needed modification for this goal is shown in line 4 of Listing 2.5. As depicted in Figure 2.4, this model violates *queue overflow avoidance* correctness invariant. There is an execution trace in this model that results in trying to send messages to `controller` with a full queue. To resolve this issue, the queue size of `BridgeController` has to be increased to four to satisfy all of the predefined correctness invariants for the model of two trains.

6: As mentioned before, the queues of actors in Core Rebeca follow FIFO policy. So, although `controller` in `S6` has two messages in its message queue, the earliest received message has to be served and there is one possible action in `S6`. As a result, there is one outgoing transition in `S6`.

Listing 2.5: The modified version of the bridge control system which have two trains

```
1 main {
2   BridgeController controller():();
3   Train train1(controller):(1);
4   Train train2(controller):(2);
5 }
```

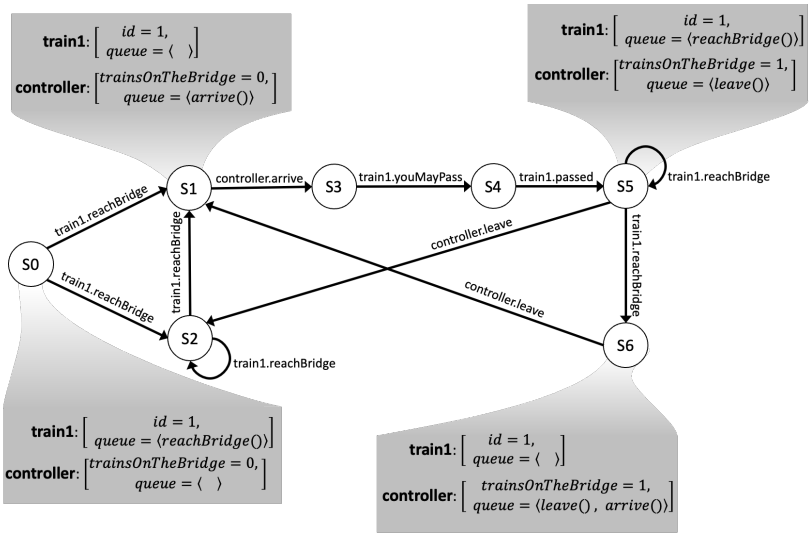


Figure 2.3: The transition system of the model of the bridge control system after resolving the violation of deadlock avoidance property. The content of states S0, S1, S5, and S6 are shown in detail in gray boxes. In each box, the valuation of state variables and queue contents of actors are shown.

Exercise 2.4.1 Set the size of the queue of `BridgeController` to three instead of four. Verify the model in Afra and try to find an interpretation for the newly reported counterexample.

In addition to the predefined correctness invariants, we want to make sure that the bridge controller provides safe passage of trains. By safe passage we mean there is no scenario in the model that more than one train passes the bridge simultaneously. This correctness property can be presented as a correctness invariant in the model using `assertion` method. As shown in line 5 of Listing 2.6, the desired invariant is presented in a boolean-valued expression and given as the input parameter of the `assertion` method. In the case of the evaluation of the given parameter to `false`, Afra reports a counterexample to show the violation of the given correctness invariant, depicted in Figure 2.5.

```
1 reactiveclass BridgeController(4) {
2   ...
3   msgsrv arrive() {
4     updateNumberOfTrainsOnTheBridge(+1);
5     assertion(trainsOnTheBridge <= 1);
6     ((Train)sender).youMayPass();
7   }
8 }
```

Listing 2.6: The modified version of the bridge control system which the safe passage of trains is presented using `assertion` method

More complex correctness properties of a model have to be specified in the property file of the model. To illustrate how correctness properties are defined in property files, we remove line 5 of Listing 2.6 and try to define an invariant for safe passage in the property file. Listing 2.7 shows the structure of specifying correctness properties in the property file.

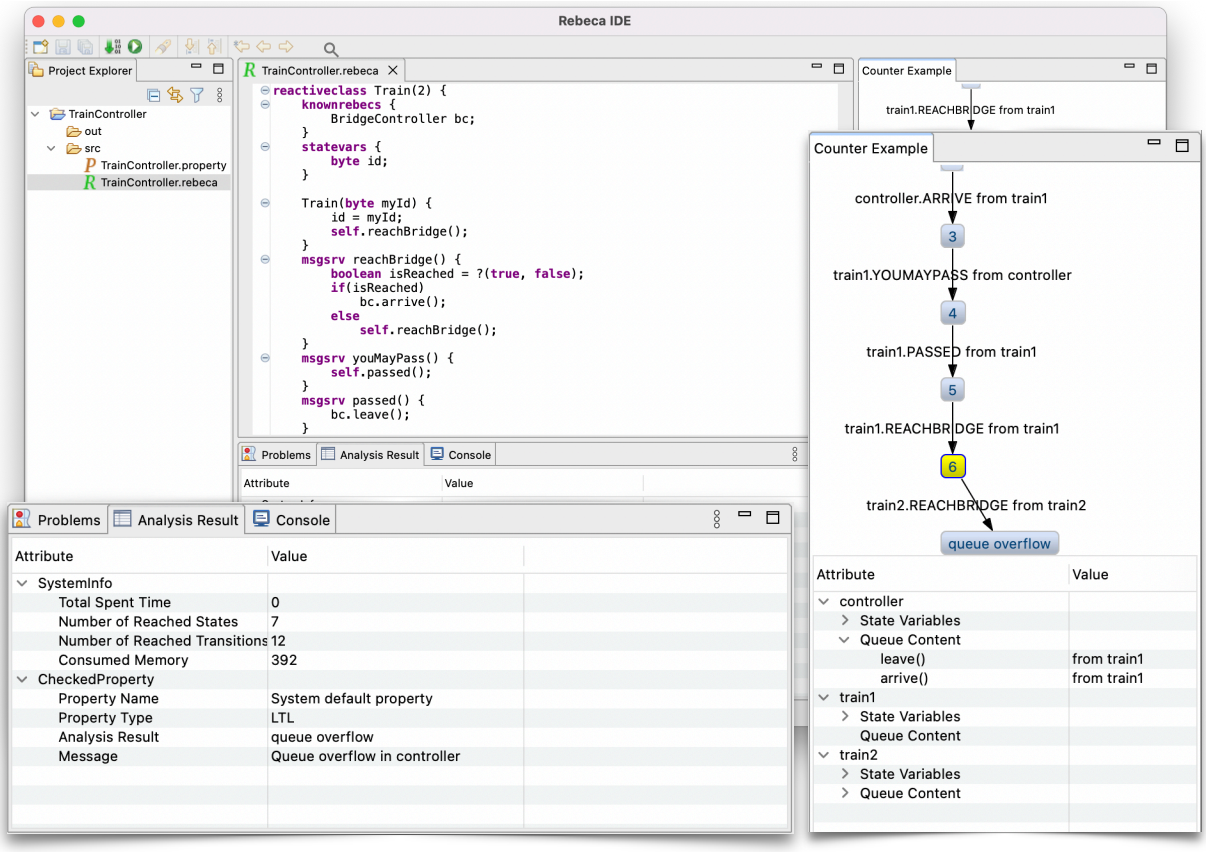


Figure 2.4: The analysis result viewer of Afra which reports having queue overflow in the model after adding a new train and presents the counterexample of the violation of this correctness invariant

Listing 2.7: The structure of defining complex correctness properties in a property file

```
1  property {
2    define {
3      // Defining atomic propositions
4    }
5    Assertion {
6      // Correctness specifications in assertion format
7    }
8    LTL {
9      // Correctness specifications in LTL format
10   }
11 }
```

A property specification has three parts. In the first part, the atomic propositions are specified. Atomic propositions will be used in the definition of correctness properties, defined by its name and a boolean expression as its value. The expression corresponds to an atomic proposition is defined using the specifiers of the state variables of actors. For example, line 3 of Listing 2.8 expresses an atomic proposition which considers the number of trains on the bridge.

Listing 2.8: Defining safe passage of trains in a property file

```
1  property {
2    define {
3      AtMostOneTrain = controller.trainsOnTheBridge < 2;
```

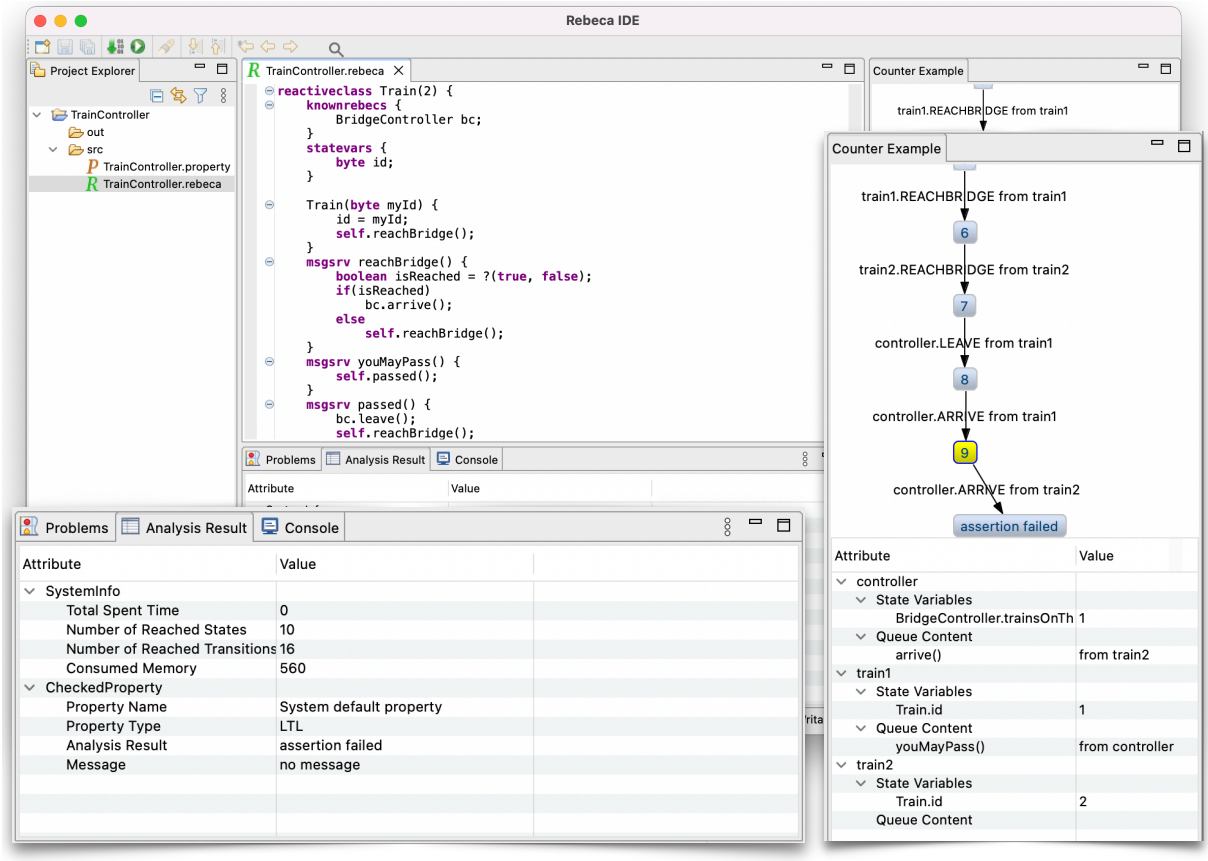


Figure 2.5: The analysis result viewer of Afra which reports the violation of the user-defined correctness invariant in the model

```
4   }
5   Assertion {
6     SafePass : AtMostOneTrain;
7   }
8 }
```

The second part of a property file contains user-defined correctness invariants, defined by its name and a boolean expression as its value. Clauses of the boolean expressions are user-defined atomic propositions (defined in the `define` section) which can be combined by standard boolean operators to produce more complex expressions. Core Rebeca supports &&, ||, and ! as symbols of *and*, *or*, and *negation* logical operators, respectively. Back to the running example, the safe passage property can be simply defined by `AtMostOneTrain` clause without using additional boolean operator, as depicted in line 6 of Listing 2.8. Model checking against this property results in reporting a counterexample the same as the counterexample of Figure 2.5. To provide safe passage for trains, the model has to be modified as shown in Listing 2.9. In this implementation of the system, the bridge controller allows passage of trains only if the number of trains on the bridge is zero. Afra generates a transition system with 40 states and 113 transitions for this model and reports the correctness of the model considering both predefined and user-defined correctness invariants.

Listing 2.9: The modified version of the bridge control system which provides safe passage of trains

```
1  reactiveclass BridgeController(4) {
2    ...
3    msgsrv arrive() {
4      if(trainsOnTheBridge == 0) {
5        updateNumberOfTrainsOnTheBridge(+1);
6        ((Train)sender).youMayPass();
7      } else {
8        ((Train)sender).reachBridge();
9      }
10   }
11 }
```

There is a set of complex correctness properties which can not be specified by invariants. To support this set of correctness properties, Afra provides verification of models against linear temporal logic (LTL) properties[7]. The third part of the property file contains LTL correctness formulae. An LTL formula over a set of atomic propositions is defined by its name and the combination of logical expressions and LTL modalities as the following:

7: More details about LTL are presented in Appendix **??**

- p: is the label of an atomic proposition which is defined in the `define` part,
- `G`(ϕ): represents $\Box\,\phi$ LTL formula (*always*) which ϕ is another LTL formula,
- `f`(ϕ): represents $\Diamond\,\phi$ LTL formula (*eventually*) which ϕ is another LTL formula,
- `U`(ϕ, ψ): represents $\phi\,\mathbf{U}\,\psi$ LTL formula (*until*) which ϕ an ψ are another LTL formulae,
- $\phi||\psi$: represents $\phi \vee \psi$ LTL formula which ϕ an ψ are another LTL formulae,
- $\phi\&\&\psi$: represents $\phi \wedge \psi$ LTL formula which ϕ an ψ are another LTL formulae,
- $!\phi$: represents $\neg\phi$ LTL formula which ϕ is another LTL formula.

Back to the running example, one of the desired correctness properties of the bridge control system is avoiding starvation in allowing trains to pass the bridge. To define an LTL specification to examine this property, some minor modifications to the model have to be applied. The message server `arrive` is modified by adding an input parameter to it whose value is the identifier of the train that wants to pass the bridge. If the bridge decides to allow a train to pass, store the identifier of the train in the `lastTrainOnTheBridge` state variable as shown in line 17 of Listing 2.10.

Listing 2.10: The modified version of the bridge control system which enabled defining starvation avoidance property

```
1  reactiveclass Train(2) {
2    msgsrv reachBridge() {
3      bc.arrive(id);
4    }
5    ...
6  }

8  reactiveclass BridgeController(4) {
9    statevars {
10     byte trainsOnTheBridge;
```

```
    byte lastTrainOnTheBridge;
  }

  msgsrv arrive(byte trainId) {
    if(trainsOnTheBridge == 0) {
      updateNumberOfTrainsOnTheBridge(+1);
      lastTrainOnTheBridge = trainId;
      ((Train)sender).youMayPass();
    } else {
      ((Train)sender).reachBridge();
    }
  }
  ...
}
```

Using this variable, the starvation avoidance property can be defined in the property file. As shown in Listing 2.11, two atomic propositions are defined to capture passing the bridge by `train1` and `train2`. Using these atomic propositions, the LTL formula `NoStarvation` is defined to ensure that in all states, both trains eventually pass the bridge. Model checking of the model against `NoStarvation` reports the counterexample of Figure 2.6. This counterexample shows a cycle in which infinite execution of the model in this cycle results in violation of `NoStarvation`[8]. The reported counterexample illustrates that there is an execution trace (the cycle of states 18, 19, 20, 21, and 81) that `train1` sends `arrive` message to `controller` as passes the bridge, but non-deterministic behavior of `train2` in `reachBridge` results in assigning `false` to `isReached` and no try to pass the bridge.

8: As described in Appendix **??**, a counterexample of a LTL correctness property is a trace of execution which ends in a loop.

```
property {
  define {
    AtMostOneTrain = controller.trainsOnTheBridge < 2;
    FirstTrainOnTheBridge = controller.lastTrainOnTheBridge == 1;
    SecondTrainOnTheBridge = controller.lastTrainOnTheBridge == 2;
  }
  Assertion {
    SafePass : AtMostOneTrain;
  }
  LTL {
    NoStarvation : G(F(FirstTrainOnTheBridge) &&
     F(SecondTrainOnTheBridge));
  }
}
```

Listing 2.11: Defining safe passage of trains and starvation avoidance in the property file

To resolve this problem the described LTL property has to be modified. The modified version which is shown in Listing 2.13 illustrates that if a train wants to pass the bridge, the bridge controller has to allow it. So, remaining in the loop of not trying to pass the bridge in `reachBridge` does not violate the `NoStarvation` property. Note that as the Rebeca expression engine does not support *implies* operator, an expression in the form of $\phi \rightarrow \psi$ has to be expressed in the form of $\neg\phi \vee \psi$ expression. So, in the definition of the `NoStarvation` instead of using the sub-formula `FirstTrainWantsToPass` $\rightarrow$ `F(FirstTrainOnTheBridge`, we used `(!FirstTrainWantsToPass) || F(FirstTrainOnTheBridge)` expression.

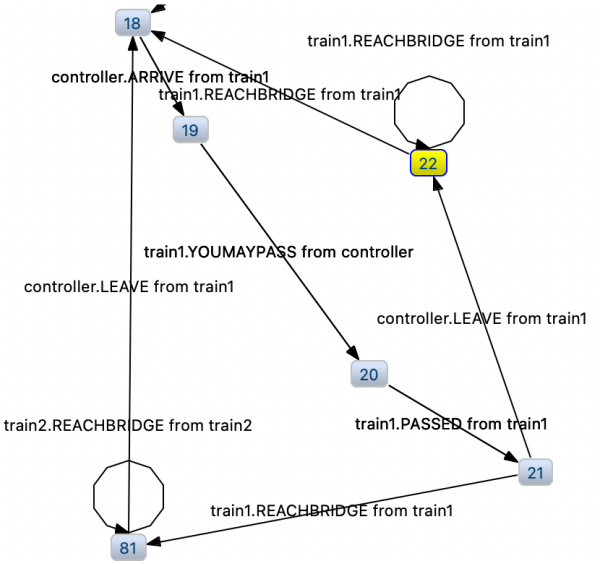


Figure 2.6: A part of the counterexample which reports the violation of the LTL correctness property of `NoStarvation` in the model

Applying this modification in the property file requires adding the `wantsToPass` variable to the model as shown in Listing 2.13.

Listing 2.12: Defining safe passage of trains and starvation avoidance in the property file

```
property {
  define {
    AtMostOneTrain = controller.trainsOnTheBridge < 2;
    FirstTrainOnTheBridge = controller.lastTrainOnTheBridge == 1;
    SecondTrainOnTheBridge = controller.lastTrainOnTheBridge == 2;
    FirstTrainWantsToPass = train1.wantsToPass;
    SecondTrainWantsToPass = train2.wantsToPass;
  }
  Assertion {
    SafePass : AtMostOneTrain;
  }
  LTL {
    NoStarvation : G(
      ((!FirstTrainWantsToPass) || F(FirstTrainOnTheBridge)) &&
      ((!SecondTrainWantsToPass) || F(SecondTrainOnTheBridge))
    );
  }
}
```

Listing 2.13: The modified version of the bridge control system which enabled defining starvation avoidance property in way that if a train wants to pass the bridge, it will be allowed

```
reactiveclass Train(2) {
  knownrebecs {
    BridgeController bc;
  }
  statevars {
    byte id;
    boolean wantsToPass;
  }

  msgsrv reachBridge() {
    boolean isReached = ?(true, false);
    if(isReached) {
      bc.arrive(id);
      wantsToPass = true;
    } else
```

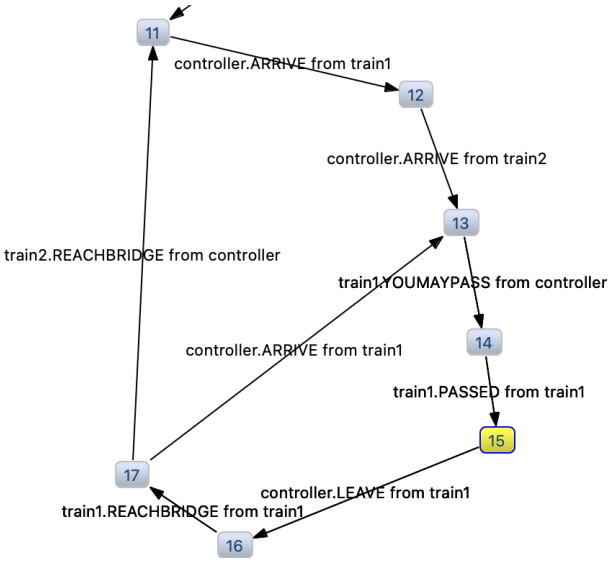


Figure 2.7: A part of the counterexample which reports the violation of the LTL correctness property of `NoStarvation` in the model because of the unfair algorithm of associating resources in the bridge controller

```
16        self.reachBridge();
17    }
18    msgsrv passed() {
19      wantsToPass = false;
20      bc.leave();
21      self.reachBridge();
22    }
23    ...
24  }
```

Model checking the modified version of the model after applying the mentioned modifications, reports another counterexample which is shown in Figure 2.7. The loop of states 11, 12, 13, 14, 15, 16, 17 shows an execution loop in which `train1` passes the bridge infinitely, but all of the requests of `train2` are rejected by the bridge controller as `train2` always sends its requests after `train1`. Having a fair algorithm for the association of resources in the bridge controller requires allowing `train2` to pass the bridge after `train1`, as it already sent its request.

Resolving this problem requires storing the identifier of the train that its request is rejected by the bridge controller, then allowing it to pass the bridge as soon as the bridge is free. Two variables `rejectedTrainId` and `rejectedTrain` are defined in Listing 2.14 to store the mentioned information. Consequently, two message servers `arrive` and `leave` have to be modified to update and use the values of the mentioned two variables in the case of rejecting the request of a train. Verification of the model of Listing 2.14 reports satisfaction `NoStarvation` LTL property in a transition system with 287 states and 1596 transitions.

```
1  reactiveclass BridgeController(4) {
2    statevars {
3      byte trainsOnTheBridge;
4      byte lastTrainOnTheBridge;
5      byte rejectedTrainId;
6      Train rejectedTrain;
7    }
```

Listing 2.14: The modified parts of the bridge control system of Listing 2.10 which resolves starvation in the model

```
8    ...
9    msgsrv arrive(byte trainId) {
10     if(trainsOnTheBridge == 0) {
11       updateNumberOfTrainsOnTheBridge(+1);
12       lastTrainOnTheBridge = trainId;
13       ((Train)sender).youMayPass();
14     } else {
15       rejectedTrain = (Train)sender;
16       rejectedTrainId = trainId;
17     }
18   }
19   msgsrv leave() {
20     updateNumberOfTrainsOnTheBridge(-1);
21     if(rejectedTrain != null) {
22       ((Train)rejectedTrain).youMayPass();
23       updateNumberOfTrainsOnTheBridge(+1);
24       lastTrainOnTheBridge = rejectedTrainId;
25       rejectedTrain = null;
26     }
27   }
28 }
```

Exercise 2.4.2 The proposed modification of Listing 2.14 only supports having two trains. Use Afra to illustrate why it does not support having three or more trains. Try to modify the model to support three or more trains.

2.5 Case Studies

In the following, different case studies are presented to get more familiar with how modeling and analysis of actor based systems is performed using Core Rebeca.

2.5.1 Dining Philosophers

There is a group of philosophers sitting at a round table. Between each adjacent pair of philosophers is a chopstick. In other words, the chopsticks are equal to philosophers number. Each philosopher does two activities: think and eat. The philosopher thinks for a while, and then stops thinking and becomes hungry. When the philosopher becomes hungry, he cannot eat until he owns the chopsticks to his left and right sides. When the philosopher is done the eating, he puts down the chopsticks and begins thinking again. The model of dining philosophers problem that philosophers are communicating asynchronously is presented in Listing 2.15.

Listing 2.15: The Rebeca model of the Dining Philosophers problem

```
1  reactiveclass Philosopher(3) {
2    knownrebecs {
3      Chopstick chpL, chpR;
4    }
5    statevars {
6      boolean eating;
```

```
  }
  Philosopher() {
    eating = false;
    self.think();
  }
  msgsrv think() {
    chpL.request();
  }
  msgsrv permit() {
    if (sender == chpL) {
      chpR.request();
    } else {
      self.eat();
    }
  }
  msgsrv eat() {
    eating = true;
    self.leave();
  }
  msgsrv leave() {
    eating = false;
    chpL.release();
    chpR.release();
    self.think();
  }
}

reactiveclass Chopstick(3) {
  knownrebecs {
    Philosopher philL, philR;
  }
  statevars {
    boolean lAssign, rAssign, leftReq, rightReq;
  }
  Chopstick() {
    lAssign = false;
    rAssign = false;
    leftReq = false;
    rightReq = false;
  }

  msgsrv request() {
    if (sender == philL) {
      leftReq = true;
      if (!rAssign) {
        lAssign = true;
        philL.permit();
      }
    } else {
      rightReq = true;
      if (!lAssign) {
        rAssign = true;
        philR.permit();
      }
    }
  }
  msgsrv release() {
    if (sender == philL){
      leftReq = false;
      lAssign = false;
      if (rightReq) {
        rAssign=true;
        philR.permit();
```

```
70        }
71      }
72      if (sender == philR){
73        rAssign = false;
74        rightReq = false;
75        if (leftReq) {
76          lAssign=true;
77          philL.permit();
78        }
79      }
80    }
81  }

83  main {
84    Philosopher phil0(chp0, chp2):();
85    Philosopher phil1(chp0, chp1):();
86    Philosopher phil2(chp1, chp2):();

88    Chopstick chp0(phil0, phil1):();
89    Chopstick chp1(phil1, phil2):();
90    Chopstick chp2(phil2, phil0):();
91  }
```

The correctness properties of the dining philosopher model are presented in Listing 2.16. Using the `Safety` assertion, mutual exclusion is examined. Model checking against this correctness formula, we make sure that it is impossible that two philosophers access a same chopstick at a same time. By `NoStarvation` formula, we make sure that all of the philosophers can eat infinitely often. Finally, `NoDeadlock` makes sure that there is no system-level deadlock in the model. So, in all conditions, at least one of the philosophers is eating infinitely often.

Listing 2.16: The correctness property specification of the Rebeca model of the Dining Philosophers problem

```
1  property {
2      define {
3          p0eat = phil0.eating;
4          p1eat = phil1.eating;
5          p2eat = phil2.eating;
6          c0s = !(chp0.lAssign && chp0.rAssign);
7          c1s = !(chp1.lAssign && chp1.rAssign);
8          c2s = !(chp2.lAssign && chp2.rAssign);
9      }

11     Assertion {
12       Safety: c0s && c1s && c2s;
13     }

15     LTL {
16       NoStarvation: G(F(p0eat) && F(p1eat) && F(p2eat));
17       NoDeadlock: G(F(p0eat || p1eat || p2eat));
18     }
19 }
```

Exercise 2.5.1 Define more instances of philosophers in the model. Try to answer the following questions:

- Is there a possibility of queue overflow when more instances are

defined?
- What is the growth rate of the size of the state space in relation with the number of philosophers?

2.5.2 Leader Election Problem

Leader election problem is selecting a node as a leader in a ring of n nodes. In this ring, each node has a unique identifier which is supposed to be an integer number. The leader shall be the node with the least id among all of the ids. Each node knows its own id and can send messages to one of the nodes next to it or both of them. The leader is selected by sending messages to other nodes. At the beginning, each node introduces itself as the leader to its neighbor(s). Each node compares the id in the received message to its own leader id, and substitutes its leader id with the new id received in the case that the received id is less than the current leader id. If a change is made to a node's leader id, it will declare this change to its neighbors by sending messages to them, containing its new leader id.

The HS algorithm is one of the solutions for the leader election problem. Using HS algorithm, each node acts in a set of phases. Node i that is in phase 1, sends a message containing its ID in two directions. These messages pass through a 2^0 length way (forward-trip) and then return to the sender (backward-trip). If both of the send messages are returned to the sender, node i will continue acting in the next phase. In the next phase, the message will pass through a 2^1 length way, then 2^2 and so on. The sent messages might not get back to the node. When the message sent by node i moves outwards this node, every node located in its way compares its own leader ID to the ID in the message. If their own leader ID is less than the ID in the message, it will be substituted. If their own leader ID is greater than it, the message will be ignored. In the case they are equal, this means that the node has received its own ID, so the node selects itself as the leader. In the returning way, nothing is done to the message and it just passes through the nodes. The algorithm terminates when a node receives its sent messages from both sides with its own ID, and each message has passed through half of the ring.

The Rebeca code of the HS algorithm is presented in Listing 2.17. Forward-trip in this code is presented in lines 22 to 38. At the beginning of this part, if the node receives a message which contains it id, assumes it self as the leader and informs the other. If it is not this case, if it is the last node in forward-trip, starts backward-trip by sending back the message to its sender. Otherwise, the message is send to the next node if its id is less than the id of the node. Backward-trip is presented in lines 40 to 55. In this part, if the node receives its id for the first time, it sets the value of `oneResponseIsAlreadyReceived` to `true`. The second receive of its own id, the node is informed that it survived in this phase, so goes for the next phase as shown in lines 42 to 45. If the received node is not its own, the node send it to the next node.

```
1 reactiveclass Node(5) {
2   knownrebecs {
3     Node nodeL;
```

Listing 2.17: The Rebeca model of the Leader Election problem

```
4     Node nodeR;
5   }
6   statevars {
7     int id;
8     int phase;
9     boolean oneResponseIsAlreadyReceived;
10    int leaderId;
11  }
12  Node (int myId) {
13    id = myId;
14    phase = 0;
15    oneResponseIsAlreadyReceived = true;
16    leaderId = -1;
17    self.broadcast(id, false, 0);
18  }

20  msgsrv broadcast(int msgId, boolean out, int hopCount) {
21    if(out) {
22      if(msgId == id) {
23        leaderId = id;
24        nodeR.leaderIsElected(id);
25      } else {
26        if(hopCount == 1) {
27          if(msgId < id)
28            ((Node)sender).broadcast(msgId, false, 0);
29        } else {
30          if(msgId < id) {
31            if(sender == nodeL) {
32              nodeR.broadcast(msgId, out, hopCount - 1);
33            } else {
34              nodeL.broadcast(msgId, out, hopCount - 1);
35            }
36          }
37        }
38      }
39    } else {
40      if(msgId == id) {
41        if(oneResponseIsAlreadyReceived) {
42          nodeL.broadcast(id, true, (int)pow(2, phase));
43          nodeR.broadcast(id, true, (int)pow(2, phase));
44          phase++;
45          oneResponseIsAlreadyReceived = false;
46        } else {
47          oneResponseIsAlreadyReceived = true;
48        }
49      } else {
50        if(sender == nodeL) {
51          nodeR.broadcast(msgId, false, 0);
52        } else {
53          nodeL.broadcast(msgId, false, 0);
54        }
55      }
56    }
57  }
58  msgsrv leaderIsElected(int selectedLeaderId) {
59    leaderId = selectedLeaderId;
60    if(leaderId != id)
61      nodeR.leaderIsElected(selectedLeaderId);
62    else
63      self.leaderIsElected(id);
64  }
65 }
```

```
main {
  Node node1(node4,node2):(1);
  Node node2(node1,node3):(2);
  Node node3(node2,node4):(3);
  Node node4(node3,node1):(4);
}
```

The correctness properties of the leader election model are presented in Listing 2.18. Using the `Safety` assertion, We make sure that in all of the states, there is no possibility for selection a wrong node as a leader, even for a very short period of time. By `CorrectLeader` formula, we examine that if `node1` decides to announce itself as a leader, eventually all of the nodes accept it as the leader.

```
property {
  define {
    newLeaderIsElected = (node1.leaderId == 1);
    TheSameLeaderIds = (node1.leaderId == node2.leaderId) &&
                (node2.leaderId == node3.leaderId) &&
                (node3.leaderId == node4.leaderId);
    validIdsInNode1 = node1.leaderId == -1 || node1.leaderId == 1;
    validIdsInNode2 = node2.leaderId == -1 || node2.leaderId == 1;
    validIdsInNode3 = node3.leaderId == -1 || node3.leaderId == 1;
    validIdsInNode4 = node4.leaderId == -1 || node4.leaderId == 1;
  }
  Assertion {
    Safety: validIdsInNode1 && validIdsInNode2 && validIdsInNode3
     && validIdsInNode4;
  }
  LTL {
    CorrectLeader : G(!newLeaderIsElected || F(TheSameLeaderIds));
  }
}
```

Listing 2.18: The correctness property specification of the Rebeca model of the Leader Election problem

3 Timed Rebeca

Timed Rebeca [13] is an extension of Rebeca with time features for modeling and verification of time-critical systems. In a Timed Rebeca model, each actor has its own local clock and the local clocks evolve uniformly. Methods are still executed atomically, however passing time while executing a method can be modeled. In addition, instead of a queue for messages, there is a bag of messages for each actor. The size of message bags is specified the same as setting queue sizes in Core Rebeca. Three primitives are added to Rebeca to address *computation time*, *message delivery time*, *message expiration*, and *period of occurrence of events* features. These timing primitives are `delay`, `deadline` and `after`.

The `delay` statement models the passing of time for an actor during the execution of a message server. The input parameter of `delay` illustrates the delay time which can be any kind of integer-valued expression, as shown in the below examples.

```
delay(10);
delay(a + 6);
delay(localMethod());
delay(?(1, 4, 5));
```

The keywords `after` and `deadline` can only be used in conjunction with a message sending statement. The value of the argument of `after` shows how long it takes for the message to be delivered to its receiver actor[1]. The `deadline` shows the timeout for the message, i.e., how long it will stay valid. The given parameters of `after` and `deadline` can be any kind of integer-valued expressions, as shown in the following examples.

1: `after` can also be used to model periodic behavior in realtime systems. For example, if a message server `task1` has to be executed periodically in every 3 units of time, its body has to contains the statement `self.task1() after(3);`

```
actor1.msgsrv1() after(10);
actor1.msgsrv1() deadline(a + 6);
actor1.msgsrv1() after(4) deadline(3);
actor1.msgsrv1() after(?(2, 3)) deadline(localMethod());
```

We illustrate the application of these keywords by modifying the first implementation of the running example. Listing 3.1 shows the Timed Rebeca model of the bridge controller system. The clause `after(1)` associated with `arrive` in line 16 specifies the network delay in communication between trains and the controller. The same happened in lines 24 and 24. The clause `deadline(5)` associated with `arrive` in line 16 illustrates that this message has to be server prior to passing 5 units of time which is its execution period. By sending `reachBridge` in line 17 which will be delivered to the train after 5 units of time, the recurrent behavior of the model is specified. In this model, `delay(2)` in 20 illustrates that passing the bridge by a train takes 2 units of time. Note that all statements other than delays are assumed to be executed instantaneously in Timed Rebeca.

```
1 reactiveclass Train(3) {
```

Listing 3.1: The Timed Rebeca model of the bridge control system

```
  knownrebecs {
    BridgeController bc;
  }
  statevars {
    byte id;
  }

  Train(byte myId) {
    id = myId;
    self.reachBridge();
  }
  msgsrv reachBridge() {
    boolean isReached = ?(true, false);
    if(isReached)
      bc.arrive(id) after(1) deadline(5);
    self.reachBridge() after(5);
  }
  msgsrv youMayPass() {
    delay(2);
    self.passed();
  }
  msgsrv passed() {
    bc.leave() after(1);
  }
}

reactiveclass BridgeController(10) {
  statevars {
    byte trainsOnTheBridge;
    byte lastTrainOnTheBridge;
  }

  BridgeController() {
    trainsOnTheBridge = 0;;
  }
  msgsrv arrive(byte trainId) {
    delay(1);
    if(trainsOnTheBridge == 0) {
      updateNumberOfTrainsOnTheBridge(+1);
      ((Train)sender).youMayPass() after(1);
    }
  }
  msgsrv leave() {
    updateNumberOfTrainsOnTheBridge(-1);
  }
  void updateNumberOfTrainsOnTheBridge(byte value) {
    trainsOnTheBridge += value;
  }
}

main {
  BridgeController controller():();
  Train train1(controller):(1);
  Train train2(controller):(2);
  Train train3(controller):(3);
  Train train4(controller):(4);
  Train train5(controller):(5);
}
```

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 with handling a message and its message bag has a message

which its time tag is less than the local time tag of that actor. In the case of having more than one message which satisfies this condition, the message with the smallest time tag will be executed. Finally, if there are more that one message with the smallest time tags, one of them is nondeterministically selected. When there is no more enabled actor in the model, the progress of time happens. The amount of time progress is the smallest value that makes at least one actor enabled.

3.1 Analysis of Timed Rebeca Models

The current version of Afra provides limited facilities for the analysis of Timed Rebeca models, which is verification against satisfaction of propositional logic expressions (correctness assertions). Afra analyzes models against a set of predefined assertions which are the predefined assertions of Core Rebeca models, in addition to deadline-missed avoidance. Besides the mentioned predefined correctness conditions, Afra supports verification against user-defined correctness assertions for Timed Rebeca models. User-defined correctness assertions in Timed Rebeca are defined the same as user-defined correctness assertions in Core Rebeca. Note that there is no way to access to the local time of actors in the used-defined correctness properties.

Model checking of the model of Listing 3.1 shows that there is no assertion violation in the model by analyzing an state space with more than four million states. However, having six instances of trains in the model results in missing the deadline of the execution of `arrival` messages at time 6.

Exercise 3.1.1 Try to find an interpretation for the missed deadline of this model using the reported counter example of Afra. Try to fix this error by changing the timing of the model.

3.2 Case Studies

In the following, different case studies are presented to get more familiar with how modeling and analysis of actor based systems is performed using Timed Rebeca.

3.2.1 A Ticket Service System

Listing 3.2 shows the Timed Rebeca model of a ticket service system. In this system, a client asks a ticket from an agent and the agent tries to issue a ticket by interacting with a ticket service server. As shown in the model, receiving a new ticket by a customer result in asking for the next ticket after 30 units of time. So, the customer shows a periodic behavior. This model also illustrates that issuing a ticket may takes 2 or 3 units of time as mentioned in the message server `requestTicket` of `TicketService`.

Listing 3.2: The Timed Rebeca model of the Ticket Service problem

```
reactiveclass Customer(3) {
  knownrebecs { Agent agent; }
  statevars { byte id; }
  Customer(byte myId) {
    id = myId;
    self.try();
  }
  msgsrv try() {
    agent.requestTicket();
  }
  msgsrv ticketIssued() {
    self.try() after(30);
  }
}
reactiveclass Agent(10) {
  knownrebecs { TicketService ticketService; }
  msgsrv requestTicket() {
    ticketService.requestTicket((Customer)sender) deadline(24);
  }
  msgsrv ticketIssued(Customer customer) {
    customer.ticketIssued();
  }
}
reactiveclass TicketService(10) {
  msgsrv requestTicket(Customer customer) {
    delay(?(2, 3));
    ((Agent)sender).ticketIssued(customer);
  }
}
main {
  Agent a(ts):();
  TicketService ts():();
  Customer c1(a):(1);
  Customer c2(a):(2);
}
```

Exercise 3.2.1 Try to find the maximum number of customer that the ticket service system can serve without missing deadlines. Try to change timings to support more clients.

3.2.2 A Toxic Gas Sensing System

The second case study of this section is the model of a lab environment in which the level of a toxic gas changes over time. If this level rises above a certain threshold, the scientist's life of the lab is in danger. Sensors in the lab constantly measure the amount of toxicity in the air and send the measurements to a central controller which periodically checks whether the scientist is in danger. If so, it notifies the scientist about the danger. The scientist should acknowledge the alarm; if he fails to do so in a timely manner, the controller notifies a rescue team. When the team reaches the lab it notifies the controller that the scientist has been rescued. If the controller does not receive this notification, it is concluded that the scientist has lost his life.

The Timed Rebeca model of this system is shown in Listing 3.3 and contains four reactive classes: `Sensor`, `Controller`, `Scientist`, and `Rescue`.

The sensors periodically measure the level of toxic gas in the environment (which is modeled by the nondeterministic assignment of line 10). After sensing, they report the measured data to the only `Controller` instance. Upon receiving the measured data from each `Sensor` (in the `report` message server), `Controller` stores the value in its corresponding location of `sensorValues` state variable. Periodically, `checkSensors` examines if the received values are above the normal. If high toxicity is detected, `Controller` alarms `Scientist`, and schedules a task to check for receiving the scientist's acknowledgment (line 38). If the controller does not receive an `ack` message from `Scientist`, the rescue team is informed. The controller sets a time-out for receiving the rescue notification by sending itself a `checkRescue` message (line 52). Note that all of the communications among actors in this model takes 1 unit of time, shown by `after(1)`.

```
1  reactiveclass Sensor(2) {
2    knownrebecs { Controller ctrl; }
3    statevars { byte period, id; }
4    Sensor(byte myId, byte myPeriod) {
5      period = myPeriod;
6      id = myId;
7      self.doReport();
8    }
9    msgsrv doReport() {
10     ctrl.report(id, ?(0,1)) after(1);
11     self.doReport() after(period);
12   }
13 }
14 reactiveclass Controller(4) {
15   knownrebecs {
16     Scientist scientist;
17     Rescue rescue;
18   }
19   statevars {
20     byte[2] sensorValues;
21     boolean sciAck, sciRescued, sciDead;
22     int ctrlCheckPeriod, scientistDeadline, rescueDeadline;
23   }
24   Controller() {
25     scientistDeadline = 5;
26     rescueDeadline = 4;
27     ctrlCheckPeriod = 15;
28     self.checkSensors();
29   }
30   msgsrv report(byte id, byte value) {
31     sensorValues[id] = value;
32   }
33   msgsrv checkSensors() {
34     boolean dangerousConditionIsDetected = false;
35     for(int i = 0; i < 2; i++) {
36       if(sensorValues[i] != 0) {
37         scientist.abortPlan() after(1);
38         self.checkSciAck() after(scientistDeadline);
39         dangerousConditionIsDetected = true;
40         break;
41       }
42     }
43     if(!dangerousConditionIsDetected)
44       self.checkSensors() after(ctrlCheckPeriod);
```

Listing 3.3: The Rebeca model of the Scientific Lab which may have toxic gas

```
45    }
46    msgsrv ack() {
47      sciAck = true;
48    }
49    msgsrv checkSciAck() {
50      if (!sciAck) {
51        rescue.go() after(1);
52        self.checkRescue() after(rescueDeadline);
53      }
54      sciAck = false;
55    }
56    msgsrv rescueReached() {
57      sciRescued = true;
58    }
59    msgsrv checkRescue() {
60      if (!sciRescued)
61        sciDead = true;
62    }
63  }
64  reactiveclass Scientist(3) {
65    msgsrv abortPlan() {
66      if(?(true, false))
67        ((Controller)sender).ack() after(1);
68    }
69  }
70  reactiveclass Rescue(3) {
71    msgsrv go() {
72      delay(1);
73      ((Controller)sender).rescueReached() after(1);
74    }
75  }
76  main {
77    Sensor sensor0(ctrl):(0, 10);
78    Sensor sensor1(ctrl):(1, 9);
79    Scientist scientist():();
80    Rescue rescue():();
81    Controller ctrl(scientist, rescue):();
82  }
```

Exercise 3.2.2 What is the maximum delay time of the rescue team (mentioned in line 72) which does not result in the death of the scientist?

4 Probabilistic Timed Rebeca

Probabilistic Timed Rebeca supports modeling and verification of real-time systems with probabilistic behaviors [14]. In Probabilistic Timed Rebeca, the time model is discrete (the same as Timed Rebeca), and discrete probability distributions are used. For Probabilistic Timed Rebeca, the syntax of Timed Rebeca is extended, considering possible probabilistic aspects that could exist in an actor based system. An actor can exhibit different alternative behaviors with some probability. Such as a node in the network which can behave differently in each of the safe or failure states. Also, messages can be lost in the case of unreliable communication media [15]. To address the mentioned probabilistic features, *probabilistic expression* and *pAlt* statements are added to the language.

In a probabilistic expression, the result value is one of its alternative with the specified probability. The value of the probabilistic expression $?(p_1{:}e_1\,, p_2{:}e_2\,, \cdots\,, p_n{:}e_n)$ is e_1 with the probability of p_1, it is e_2 with the probability of p_2, etc. Here, $p_1 \ldots p_n$ are real value expressions between 0 and 1, and sum up to 1. Note that $p_1 \ldots p_n$ have to be evaluatable in compile time to make sure that their summation in 1. Using probabilistic assignments, the value of the timing constructs (`delay`, `after`, and `deadline`) can also become probabilistic. Considering the following example, the value of variable `delayTime` is `1` with the probability of `0.7` and `2` with the probability of `0.3`.

```
delayTime = ?(0.7:1, 0.3:2);
```

The `pAlt` statement denotes probabilistic choice between alternative behaviors. In the `pAlt` structure, each block of statements may be executed by the mentioned probabilities. The same as probabilistic expression, the given probabilities have to be evaluatable in compile time and their summation has to be 1. Considering the following example, the value of `x` may be incremented by one with probability 0.3 or it would be decremented by one with probability 0.7.

```
1  msgsrv performingProbabilisticBehavior() {
2    /* Some Statements*/
3    pAlt {
4      prob(0.3): {
5        x=x+1;
6      }
7      prob(0.7): {
8        x=x-1;
9      }
10   }
11   /* Some Statements*/
12 }
```

Listing 4.1: The way of using the `pAlt` statement in Probabilistic Timed Rebeca

We illustrate the application of Probabilistic Timed Rebeca in the extended version of the running example in Listing 4.2. The `pAlt` statement in lines **??** to **??** shows that there is a probability of 0.1 for not sending the

gathered data to `CompUnit` (it models the behavior of loosing a massage in an unreliable communication medium). In this case, the value of `after` clause is set to 1. Another probabilistic behavior of this model is in the computation delay time in line **??**.

Listing 4.2: The Probabilistic Timed Rebeca model of the WSAN application

```
1  reactiveclass Sensor(3) {
2    knownrebecs{ CompUnit cu; }
3    statevars { int id; }
4    Sensor(int myId) { ... }
5    msgsrv gatherData() {
6      byte data = ?(1,2,3);
7      pAlt
       {|\label{listing::probabilistic-timed-rebeca::pAlt::begin}|
8       prob(0.9): {
9         cu.receiveData(data) deadline(3);
10        self.gatherData() after(2);
11      }
12      prob(0.1): {
13        self.gatherData() after(1);
14      }
15     }
16     cu.receiveData(data) deadline(3);
17     self.gatherData() after(2);
18   }|\label{listing::probabilistic-timed-rebeca::pAlt::end}|
19 }

21 reactiveclass CompUnit(3) {
22   knownrebecs { RCD rcd; }
23   statevars { byte receivedDataCounter; }
24   CompUnit() { ... }
25   msgsrv receiveData(byte data) {
26     delay(?(0.6:1, 0.3:2,
        0.1:4));|\label{listing::probabilistic-timed-rebeca::delay}|
27     if(isValid(data)
28       rcd.send(data);
29   }
30   bool isValid(byte data) { ... }
31 }

33 reactiveclass Network (3) {
34   msgsrv send(byte data) { ... }
35 }

37 main {
38   Sensor sensor(cu):(1);
39   CompUnit cu(network):();
40   Network network():();
41 }
```

4.1 Analysis of Probabilistic Timed Rebeca Models

5 Advanced Modeling Features

5.1 Environment Variables

In Rebeca, modelers are allowed to define constant variables using `env` keyword. In Listing 5.1 the network delay between trains and the bridge controller is set using the `NET_DELAY` environment variable. The type of environment variables can be any of primitive types. For the case of defining arrays, the size of arrays (as a part of state variables) should be compile-time evaluatable integer expressions.

```
1  env int NET_DELAY = 1;
2  reactiveclass Train(3) {
3    ...
4    msgsrv reachBridge() {
5      boolean isReached = ?(true, false);
6      if(isReached)
7        bc.arrive(id) after(NET_DELAY) deadline(5);
8      self.reachBridge() after(5);
9    }
10   msgsrv passed() {
11     bc.leave() after(NET_DELAY);
12   }
13 }

15 reactiveclass BridgeController(10) {
16   ...
17   msgsrv arrive(byte trainId) {
18     delay(1);
19     if(trainsOnTheBridge == 0) {
20       updateNumberOfTrainsOnTheBridge(+1);
21       ((Train)sender).youMayPass() after(NET_DELAY);
22     }
23   }
24 }
```

Listing 5.1: The modified parts of the Timed Rebeca model of Listing 3.1 which illustrates how network delays are set using an environment variable.

5.2 Inheritance and Polymorphism

Like most other object-oriented programming and modeling languages, Rebeca provides mechanisms for reusing codes through subclassing. A modeler is able to define a new reactive class as a subclass of an existing reactive class, using inheritance mechanism. This is stated using `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 known rebecs, state variables, message servers, and local methods 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 [16] .

Attention

The current version of Afra only supports declaration of default constructor for super classes.

Listing 5.2 illustrates how the behavior of trains that contain hazardous cargo are implemented using inheritance. For the case of trains with none-hazardous cargo, upon leaving the bridge, other trains are allowed to pass the bridge. For the case of trains with hazardous cargo, the bridge has to remain reserved for some amount of time, then other trains are allowed to pass it, as depicted in the `considerSafetyDistance` message server.

Listing 5.2: The Timed Rebeca model of the bridge control system which contains hazardous-cargo trains

```
1  env int NET_DELAY = 1;
2  reactiveclass Train(3) {
3    knownrebecs { BridgeController bc; }
4    statevars { byte id; }
5
6    Train(byte myId) { ... }
7    msgsrv reachBridge() { ... }
8    msgsrv youMayPass() {
9      delay(2);
10     self.passed();
11   }
12   msgsrv passed() {
13     bc.leave() after(NET_DELAY);
14   }
15 }

17 reactiveclass HazardousCargoTrain extends Train (3){
18   HazardousCargoTrain(byte myId) {
19     id = myId;
20     self.reachBridge();
21   }
22   msgsrv passed() {
23     self.considerSafetyDistance();
24   }
25   msgsrv considerSafetyDistance() {
26     delay(5);
27     bc.leave() after(NET_DELAY);
28   }
29 }

31 reactiveclass BridgeController(10) {
32   ...
33 }

35 main {
36   BridgeController controller():();
37   Train train1(controller):(1);
38   Train train2(controller):(2);
39   Train train3(controller):(3);
40   HazardousCargoTrain train4(controller):(4);
41   HazardousCargoTrain train5(controller):(5);
42 }
```

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 Listing 5.3, the new implementation of the bridge control system with the abstract `Train` reactive class is presented. Both `NormalTrain` and `HazardousCargoTrain` have to inherit from `Train` and provide implementation for the `passed` message server.

```
1  env int NET_DELAY = 1;
2  abstract reactiveclass Train(3) {
3    knownrebecs {
4      BridgeController bc;
5    }
6    statevars {
7      byte id;
8    }
9    msgsrv reachBridge() {
10     boolean isReached = ?(true, false);
11     if(isReached)
12       bc.arrive(id) after(NET_DELAY) deadline(5);
13     self.reachBridge() after(5);
14   }
15   msgsrv youMayPass() {
16     delay(2);
17     self.passed();
18   }
19   abstract msgsrv passed();
20 }
21 reactiveclass NormalTrain extends Train(3) {
22   NormalTrain(byte myId) {
23     id = myId;
24     self.reachBridge();
25   }
26   msgsrv passed() {
27     bc.leave() after(NET_DELAY);
28   }
29 }

31 reactiveclass HazardousCargoTrain extends Train (3){
32   HazardousCargoTrain(byte myId) {
33     id = myId;
34     self.reachBridge();
35   }
36   msgsrv passed() {
37     delay(5);
38     self.considerSafetyDistance();
39   }
40   msgsrv considerSafetyDistance() {
41     bc.leave() after(NET_DELAY);
42   }
43 }

45 reactiveclass BridgeController(10) {
46   ...
47 }
```

Listing 5.3: The Timed Rebeca model of the bridge control system with the `Train` abstract reactive class

```
49 main {
50   BridgeController controller():();
51   Train train1(controller):(1);
52   Train train2(controller):(2);
53   Train train3(controller):(3);
54   HazardousCargoTrain train4(controller):(4);
55   HazardousCargoTrain train5(controller):(5);
56 }
```

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 requirement can be realized by defining an `interface` instead of a reactive class. Using interfaces allows a modeler to determine message server names and their argument lists, but no bodies, no known rebecs, and no state variables. So, it provides only a type, not any implementation. Interfaces and abstract classes provide more structured way to separate interface from implementation.

A reactive class is allowed to provide implementation for message servers of an `interface` using `implements` keyword. The reactive class must provide method definitions for all the message servers of the interface. If it doesn't, the reactive class is abstract and it must be explicitly declared using `abstract` keyword. An example of using interfaces in Rebeca is illustrated in Listing 5.4. In this example, an interface is defined for the bridge controller. Using this interface, in addition to the normal bridge controller, a new implementation for the bridge controller which considers the safety requirements of passage of trains with hazardous cargo can be provided.

Attention

In Rebeca, defining multiple interface implementation is allowed, which can be assumed as a kind of multiple inheritances.

Listing 5.4: The Timed Rebeca model of the bridge control system which contains the interface declaration `BridgeController`

```
1  abstract reactiveclass Train(3) {
2    knownrebecs {
3      BridgeController bc;
4    }
5      ...
6  }

8  reactiveclass NormalTrain extends Train(3) { ... }

10 reactiveclass HazardousCargoTrain extends Train(3) { ... }

12 interface BridgeController {
13   msgsrv arrive();
14   msgsrv leave();
15 }

17 reactiveclass ConsidersSaferyBridgeController implements
       BridgeController (10) {
18   msgsrv arrive() { ... }
19   msgsrv leave() { ... }
20 }

22 main {
23   ConsidersSaferyBridgeController controller():();
24   NormalTrain train1(controller):(1);
25   NormalTrain train2(controller):(2);
26   HazardousCargoTrain train3(controller):(3);
```

```
27   HazardousCargoTrain train4(controller):(4);
28 }
```

5.3 More Deterministic Models

In concurrency theory, nondeterminism serves as a foundational concept for modeling concurrent systems. Hewitt actors are specifically designed to facilitate the development of distributed and networked applications. Recently, there has been a growing trend to incorporate greater determinism into language models, drawing inspiration from synchronous languages (e.g. Edward Lee et.al. in [17]. Furthermore, many applications rely on predefined priorities to effectively manage task scheduling. In this context, we describe how priorities can be integrated as annotations in Timed Rebeca, enhancing its support for applications that require structured task ordering.

In Rebeca, the semantics of the language dictate that the execution of enabled actors occurs in a nondeterministic order. An actor is considered enabled when it is not busy processing a message and its message queue is not empty. Each actor has a message queue; where messages sent from one actor to another are delivered in the same order as they were sent, ensuring point-to-point, in-order message delivery within an actor. However, no assumptions can be made regarding the order of messages sent from different actors. For Timed Rebeca, the order of handling messages of an actor depends on the time tags of the messages. When multiple messages have the same time tag, they are handled in a nondeterministic manner (see [13] for a formal definition of the semantics).

To enhance the determinism of actor behavior –particularly essential for real-time and embedded systems– Rebeca allows associating priorities with actors and message servers. Priorities for actors are specified in the main code section where actor instances are created from reactive classes. This way, the execution of enabled actors takes place considering the associated priorities. 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. Note that associating the same priority level with message servers of an actor results in the nondeterministic choice among the messages when they have the same time tag.

Finally, 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. This is a flat type of priority throughout the whole model which we call *Global Priority*.

Attention

Note that using both `priority` and `globalPriority` in a model is not allowed.

5.3.1 Incorporating Priorities into the Model

In more deterministic version of the bridge controller system of Listing 5.4, we aim to ensure that trains carrying hazardous cargo are given higher execution priorities. As depicted in Listing 5.5, two different priority levels are associated with the actor instances using *priority annotations*(lines 6 and 8). Having a smaller value for *priority annotations* means that the actor has a higher execution priority. In Listing 5.5, the highest priority is associated with `train4` and `train5` has the next priority. As no priority is associated with `train1`, `train2`, and `train3`, they have the same priority which is the lowest priority.

Listing 5.5: The Timed Rebeca model of the bridge control system which contains the interface declaration `BridgeController`

```
1  main {
2    BridgeController controller():();
3    NormalTrain train1(controller):(1);
4    NormalTrain train2(controller):(2);
5    NormalTrain train3(controller):(3);
6    @priority(1)
7    HazardousCargoTrain train4(controller):(4);
8    @priority(2)
9    HazardousCargoTrain train5(controller):(5);
10 }
```

In Listing 5.6, we prioritize the handling of a `leave` message over an `arrive` in `ConsidersSaferyBridgeController`. This prioritization is essential because the controller must ensure that the bridge is cleared before allowing additional trains to enter.

Listing 5.6: The Timed Rebeca model of the bridge control system which contains association of priority with the message servers of `ConsidersSaferyBridgeController`

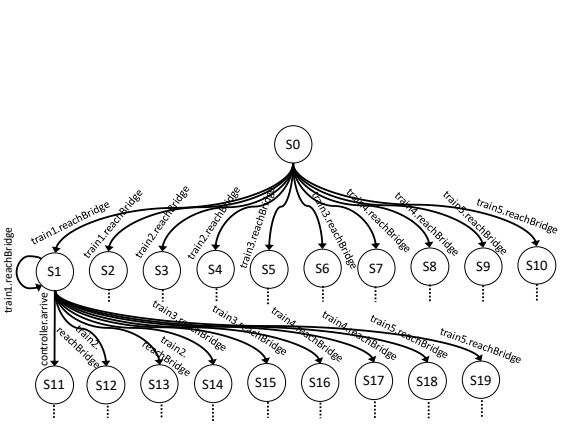
```
1  interface BridgeController {
2    msgsrv arrive();
3    msgsrv leave();
4  }

6  reactiveclass ConsidersSaferyBridgeController implements
       BridgeController (10) {
7    @priority(2)
8    msgsrv arrive() { ... }
9    @priority(1)
10   msgsrv leave() { ... }
11 }
```

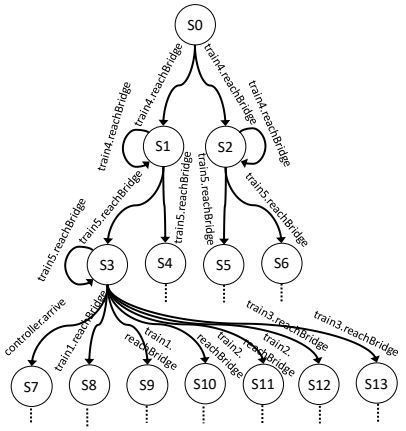
Similar to the prioritization of actor instances and message servers, the handling of global priority is achieved through the annotation mechanism. To associate global priority to a message server, `@globalPriority` annotation is utilized.

5.3.2 Analysis of Rebeca Models with Priorities

The model checking engine of Afra operates under the assumption that all actors and methods in the given model have assigned priorities. If an actor or message server lacks an associated priority, Afra defaults to the lowest priority for it. During each step of generating the transition system



(a) The transition system of the bridge controller system that no priority is associated with actor instances



(b) The transition system of the bridge controller system that priorities are associated with actor instances

Figure 5.1: Comparing transition systems of two implementations of the model the bridge control system, which are no priority is associated with actors (a) and associating priorities with actors (b).

for a model, Afra selects the highest priority enabled message from the actor with the highest priority. In instances where multiple methods or actors share the same priority level, one is chosen nondeterministically.

Figure 5.1 compares the initial segments of the transition systems for the bridge control system, both with and without associated priorities for actor instances. As mentioned before, including priorities reduces nondeterministic choices, resulting in smaller transition systems. As shown in 5.1(a), there are 10 outgoing transitions from the state `S0`, with two transitions for each actor. This is due to the nondeterministic assignment in the `reachBridge` message server for both types of trains. This high branching factor leads to a significantly large transition system, comprising 5,334,795 states and 13,459,394 transitions.

In contrast, when priorities are assigned to the actors, as shown in Listing 5.5, the transition system appears as depicted in 5.1(b). In this case, `train4` has the highest priority, so the outgoing transitions of `S0` pertain to this actor. At the next step, outgoing transitions of `S1` and `S2` correspond to the execution of the message server of `train5`, as it is the enabled actor with the highest priority among the others. Finally, `S3` has seven outgoing transitions due to the presence of four other actors having the same lowest priority level. This transition system consists of 3,656,612 states and 6,819,689 transitions, which is significantly smaller than the transition system presented in 5.1(a).

Bibliography

Here are the references in citation order.

[1] Carl Hewitt, Peter Bishop, and Richard Steiger. 'A Universal Modular ACTOR Formalism for Artificial Intelligence'. In: *IJCAI*. Ed. by Nils J. Nilsson. William Kaufmann, 1973, pp. 235–245 (cited on page 3).

[2] Gul A. Agha. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence. MIT Press, 1990, pp. I–IX, 1–144 (cited on page 3).

[3] Arnold H. Buss. 'Modeling with Event Graphs'. In: *Proceedings of the 28th conference on Winter simulation, WSC 1996, Coronado, CA, USA, December 8-11, 1996*. Ed. by John M. Charnes et al. IEEE Computer Society, 1996, pp. 153–160. DOI: 10.1109/WSC.1996.873273 (cited on page 3).

[4] Marjan Sirjani et al. 'Modeling and Verification of Reactive Systems using Rebeca'. In: *Fundam. Informaticae* 63.4 (2004), pp. 385–410 (cited on page 5).

[5] Marjan Sirjani and Mohammad Mahdi Jaghoori. 'Ten Years of Analyzing Actors: Rebeca Experience'. In: *Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*. Ed. by Gul Agha, Olivier Danvy, and José Meseguer. Vol. 7000. Lecture Notes in Computer Science. Springer, 2011, pp. 20–56. DOI: 10.1007/978-3-642-24933-4_3 (cited on page 5).

[6] Marjan Sirjani and Ehsan Khamespanah. 'On time actors'. In: *Theory and Practice of Formal Methods: Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday* (2016), pp. 373–392 (cited on page 5).

[7] Iman Jahandideh, Fatemeh Ghassemi, and Marjan Sirjani. 'Hybrid rebeca: Modeling and analyzing of cyber-physical systems'. In: *Cyber Physical Systems. Model-Based Design: 8th International Workshop, CyPhy 2018, and 14th International Workshop, WESE 2018, Turin, Italy, October 4–5, 2018, Revised Selected Papers 8*. Springer. 2019, pp. 3–27 (cited on page 5).

[8] Mahsa Varshosaz and Ramtin Khosravi. 'Modeling and Verification of Probabilistic Actor Systems Using pRebeca.' In: *ICFEM*. Springer. 2012, pp. 135–150 (cited on page 5).

[9] Ali Jafari et al. 'PTRebeca: Modeling and analysis of distributed and asynchronous systems'. In: *Science of Computer Programming* 128 (2016), pp. 22–50 (cited on page 5).

[10] Ehsan Khamespanah, Marjan Sirjani, and Ramtin Khosravi. 'Afra: An Eclipse-Based Tool with Extensible Architecture for Modeling and Model Checking of Rebeca Family Models'. In: *Fundamentals of Software Engineering - 10th International Conference, FSEN 2023, Tehran, Iran, May 4-5, 2023, Revised Selected Papers*. Ed. by Hossein Hojjat and Erika Ábrahám. Vol. 14155. Lecture Notes in Computer Science. Springer, 2023, pp. 72–87. DOI: 10.1007/978-3-031-42441-0_6 (cited on pages 5, 12).

[11] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008 (cited on page 12).

[12] Amir Pnueli. 'The Temporal Logic of Programs'. In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32 (cited on page 12).

[13] Arni Hermann Reynisson et al. 'Modelling and simulation of asynchronous real-time systems using Timed Rebeca'. In: *Sci. Comput. Program.* 89 (2014), pp. 41–68. DOI: 10.1016/j.scico.2014.01.008 (cited on pages 29, 41).

[14] Ali Jafari et al. 'PTRebeca: Modeling and analysis of distributed and asynchronous systems'. In: *Sci. Comput. Program.* 128 (2016), pp. 22–50. DOI: 10.1016/j.scico.2016.03.004 (cited on page 35).

[15] Mahsa Varshosaz and Ramtin Khosravi. 'Modeling and Verification of Probabilistic Actor Systems Using pRebeca'. In: *Formal Methods and Software Engineering - 14th International Conference on Formal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12-16, 2012. Proceedings*. Ed. by Toshiaki Aoki and Kenji Taguchi. Vol. 7635. Lecture Notes in Computer Science. Springer, 2012, pp. 135–150. DOI: 10.1007/978-3-642-34281-3_12 (cited on page 35).

[16] Bruce Eckel. *Thinking in Java (4th Edition)*. Prentice Hall, 2006 (cited on page 38).

[17] Christian Menard et al. 'High-Performance Deterministic Concurrency using Lingua Franca'. In: *CoRR* abs/2301.02444 (2023) (cited on page 41).

[18] Fatemeh Ghassemi et al. 'Transparent Actor Model'. In: *11th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE 2023, Melbourne, Australia, May 14-15, 2023*. IEEE, 2023, pp. 97–107. DOI: 10.1109/FORMALISE58978.2023.00018.

[19] Ehsan Khamespanah. 'Modeling, Verification, and Analysis of Timed Actor-Based Models'. PhD thesis. Reykjavik, Iceland: Reykjavik University, June 2018.

[20] Luca Aceto et al. 'Modelling and Simulation of Asynchronous Real-Time Systems using Timed Rebeca'. In: *Proceedings 10th International Workshop on the Foundations of Coordination Languages and Software Architectures, FOCLASA 2011, Aachen, Germany, 10th September, 2011*. Ed. by Mohammad Reza Mousavi and António Ravara. Vol. 58. EPTCS. 2011, pp. 1–19. DOI: 10.4204/EPTCS.58.1.