Afra: An Eclipse-Based Tool with Extensible Architecture for Modeling and Model Checking of Rebeca Family Models

Ehsan Khamespanah[1,2(✉)], Marjan Sirjani[2,3], and Ramtin Khosravi[1]

[1] School of Electrical and Computer Engineering, University of Tehran, Tehran, Iran
[2] School of Computer Science, Reykjavik University, Reykjavik, Iceland
e.khamespanah@ut.ac.ir
[3] School of Innovation, Design, and Engineering, Mälardalen University, Västerås, Sweden

Abstract. Afra is an *Eclipse*-based tool for the modeling and model checking of *Rebeca* family models. Together with the standard enriched editor, easy to trace counter-example viewer, modular temporal property definition, exporting a model and its transition system to some other formats facilities are features of Afra. Rebeca family provides actor-based modeling languages which are designed to bridge the gap between formal methods and software engineering. Faithfulness to the system being modeled, and the usability of Rebeca family languages help in ease of modeling and analysis of the model, together with the synthesis of the system based on the model. In this paper, architectural decisions and design strategies we made in the development of Afra are presented. This makes Afra an extensible and reusable application for the modeling and analysis of Rebeca family models. Here, we show how different compilers can be developed for the family of languages which are the same in general language constructs but have some minor differences. Then we show how the model checking engine for these different languages is designed. Despite the fact that Afra has a layered object-oriented design and is developed in Java technology, we use C++ codes for developing its model checking for the performance purposes. This decision made the design of the application even harder.

Keywords: Actors · Rebeca · Afra · Model Checking · Eclipse

1 Introduction

The actor model is a well-known model for the development of highly available and high-performance applications. It benefits from the universal primitives of concurrent computation [1], called actors, which are distributed, autonomous objects that interact by asynchronous message passing. Each actor provides a number of services, and other actors send messages to it to run the services.

Messages are put in the mailbox of the receiver, the receiver takes a message from the mailbox and executes its corresponding service. Hewitt introduced the actor model as an agent-based language [2] and is later developed by Agha as a mathematical model of concurrent computation [1].

Rebeca is an operational interpretation of the actor model with formal semantics. Rebeca is designed to bridge the gap between formal methods and software engineering. The formal semantics of Rebeca is a solid basis for its formal verification [3]. Compositional and modular verification, abstraction, symmetry and partial-order reduction have been investigated for verifying Rebeca models [4]. The theory underlying these verification methods is already established and is embodied in verification tools [5,6]. Different extensions have been provided for modeling and analyzing of different aspects of actor systems. Timed Rebeca is an extension on Rebeca with time features for modeling and verification of time-critical systems [7]. Probabilistic Rebeca is another extension of Rebeca which is developed to consider the probabilistic behavior of actor systems [8]. Probabilistic Timed Rebeca (PTRebeca) is an extension of Rebeca which benefits from modeling features of Timed Rebeca and Probabilistic Rebeca, combining the syntax of both languages [9]. More details about these extensions are provided in Sect. 2. RebecaSys is another extension of Rebeca which is developed to support hardware/software co-design (i.e. system-level design) [10]. In Broadcasting Rebeca [11] and Wireless Rebeca [12] the Core Rebeca is extended from a different dimension to provide broadcasting and multi-casting among actors which is crucial for modeling and verification of network protocols.

Afra is a toolset which is developed for the purpose of providing modeling and analysis facilities for the Rebeca family languages. As the same as many other Eclipse plugins, Afra contains a set of Eclipse views and editors together with a set of Java components for implementing models and analyzing them. In addition to the syntax-highlighting editor, Afra provides easy to use counterexample browser which made debugging of models easier. The focus of these futures is in improving the usability of the developed toolset. Beside the essence of providing usability, there is a need for considering extensibility and maintainability of the model checking toolset. This need becomes more important for the case of Afra as it has to support a set of modeling languages which require different compilers and model checking algorithms.

In this paper, we show how Afra is designed to make it extensible and maintainable for different languages of the Rebeca family. Starting from the architectural view (Sect. 3) we make clear how the main functional requirements of Afra are placed in a set of Java components. Then, we describe the techniques which are used for the implementation of compilers of the Rebeca family languages from syntax and semantics points of view (Sect. 4). To this end, we discussed techniques which can be used to develop the hierarchy of compilers using ANTLR [13] toolset. Then, we introduce the class diagram of the semantics-checker which we developed for performing semantical analysis of the Rebeca family models and make it clear that how it can be extended to consider the future extension on Rebeca.

To increase the performance of model checking, Afra transforms models into a set of C++ source codes. Running these codes results in generating the transition system of the model and performs property checking. This approach is very similar to the development approach of SPIN [14]. Decisions which are made in the design of C++ classes and how third-party template generators help in code reuse are issues which we address in Sects. 5 and 5.2.

2 Rebeca Family Modeling Languages

A Rebeca model is similar to the actor model as reactive objects without shared variables are its only computation units. Objects in Rebeca are reactive, self-contained, and each of them is called a *rebec* (reactive object). Note that in this paper we use rebec and actor interchangeably. Each actor has an unbounded buffer, called message *queue*, for its arriving messages. Communication among actors takes place by asynchronous message passing with no blocking send and no explicit receive. Computation is event-driven, meaning that each actor takes a message that can be considered as an event from the top of its message queue and executes the corresponding message server (also called a method). In Rebeca, the execution of a message server is atomic, i.e. there is no way to preempt the execution of a message server of an actor and start executing another message server of that actor. Note that we call the basic extension of Rebeca as Core Rebeca to avoid misunderstanding.

2.1 Core Rebeca

A Core Rebeca model consists of a set of reactive classes definitions and the main block. In the main block, actors which are instances of the reactive classes are declared. The body of the reactive class includes the declaration of its known actors, state variables, and message servers. Message servers consist of the declaration of local variables and the body of the message server. The statements in the body can be assignments, conditional statements, enumerated loops, non-deterministic assignment, and method calls. Method calls are sending asynchronous messages to other actors (or to itself). A reactive class has an argument of type integer denoting the maximum size of its message queue. Although message queues are unbounded in the semantics of Rebeca, to ensure that the state space is finite, we need a user-specified upper bound for the queue size. The operational semantics of Rebeca has been introduced in [15] in more detail. In comparison with the standard actor model, dynamic creation and dynamic topology are not supported by Core Rebeca. Also, actors in Core Rebeca are single-threaded.

We illustrate the Core Rebeca language with an example. Listing 1.1 shows the Core Rebeca model of the ticket service system. The model consists of three reactive classes: `TicketService`, `Agent`, and `Customer`. In this model, `Customer` sends the `requestTicket` message to `Agent` (line 32) and `Agent` forwards the message to `TicketService` (line 18). `TicketService` replies to `Agent` by sending a `ticketIssued` message (line 8) and `Agent` responds to `Customer` by sending the issued ticket (21). Upon receiving a ticket, `Customer` tries for another ticket (line 37).

Listing 1.1. The Rebeca model of Ticket Service System

```
 1 reactiveclass TicketService (3) {
 2     knownrebecs {Agent a;}
 3     statevars {int nextId;}
 4     TicketService() {
 5         nextId = 0;
 6     }
 7     msgsrv requestTicket() {
 8         a.ticketIssued(nextId);
 9         nextId = nextId + 1;
10     }
11 }
12 reactiveclass Agent (3) {
13     knownrebecs {
14         TicketService ts;
15         Customer c;
16     }
17     msgsrv requestTicket() {
18         ts.requestTicket();
19     }
20     msgsrv ticketIssued(byte id) {
21         c.ticketIssued(id);
22     }
23 }
24 reactiveclass Customer (2) {
25     knownrebecs {Agent a;}
26     statevars {boolean sent;}
27     Customer() {
28         self.try();
29         sent = false;
30     }
31     msgsrv try() {
32         a.requestTicket();
33         sent = true;
34     }
35     msgsrv ticketIssued(byte id) {
36         sent = false;
37         self.try();
38     }
39 }
40 main {
41     Agent a(ts, c):();
42     TicketService ts(a):(3);
43     Customer c(a):();
44 }
```

For a given Core Rebeca model, a modeler can specify the correctness properties of the model as a set of assertions or LTL formula. As shown in Listing 1.2, a property specification has three parts. In the first part the atomic propositions of the properties are defined. An atomic proposition is defined by its name and a boolean expression as its value. `cIsSent` and `idCounter` are two atomic propositions in Listing 1.2.

Listing 1.2. Correctness property specification of Ticket Service System

```
 1 property {
 2   define {
 3     cIsSent = c.sent;
 4     idCounter = ts.nextId;
 5   }
 6   Assertion {
 7     MaxNumberOfTickets: idCounter < 10;
 8   }
 9   LTL {
10     NoStarvation: G(cIsSent -> F(!cIsSent));
11   }
12 }
```

The assertions of models are defined in the second part of property specifications. An assertion is defined by its name and a boolean expression as its value, which its terms are the labels of atomic propositions. `MaxNumberOfTickets` is the only assertion of this model which makes sure that the number of issued tickets in this model is less than 10. Note that this model does not satisfy `MaxNumberOfTickets` as there is no limitation on the number of issued tickets. The last part of the property specification contains LTL formula. An LTL formula is defined by its name and combination of logical expressions and LTL modalities as its value, which its terms are the labels of atomic propositions. $G(\phi)$, $f(\phi)$, and $U(\phi, \psi)$ are used to specify $\Box\,\phi$ (*always*), $\Diamond\,\phi$ (*eventually*), and $\phi\,\mathbf{U}\,\psi$ (*until*) respectively. `NoStarvation` is the only LTL property of this model which makes sure that each request for ticket will be served in the future.

2.2 Timed Rebeca

Timed Rebeca is an extension on Rebeca with time features for modeling and verification of time-critical systems [7]. To this end, three primitives are added to Rebeca to address *computation time*, *message delivery time*, *message expiration*, and *period of occurrence of events*. 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.

In comparison to the syntax of Rebeca, three timing primitives are defined in Timed Rebeca which are `delay`, `deadline` and `after`. The `delay` statement models the passing of time for an actor during the execution of a message server. The keywords `after` and `deadline` can only be used in conjunction with a method call. The value of the argument of *after* shows how long it takes for the message to be delivered to its receiver. The `deadline` shows the timeout for the message, i.e., how long it will stay valid. We illustrate the application of these keywords using the Timed Rebeca version of the ticket service system in Listing 1.3. Note that this source code only contains the parts of the model which are different in the Rebeca and Timed Rebeca models. As shown in line 3 of the model, issuing a ticket takes two or three time units (modeled by a non-deterministic expression). At line 10 the actor instantiated from `Agent` sends a message `requestTicket` to actor `ts` instantiated from `TicketService`, and gives a deadline of five to the receiver to take this message and start serving it. The periodic task of retrying for a new ticket is modeled in line 15 by the customer sending a `try` message to itself and letting the receiver to take it from its bag only after 30 units of time (by stating `after(30)`).

Listing 1.3. The Timed Rebeca model of ticket service system

```
 1 reactiveclass TicketService {
 2     msgsrv requestTicket() {
 3         delay(?(2, 3));
 4         a.ticketIssued(nextId);
 5         nextId = nextId + 1;
 6     }
 7 }
 8 reactiveclass Agent {
 9     msgsrv requestTicket() {
10         ts.requestTicket()
               deadline(5);
11     }
}
reactiveclass Customer {
    msgsrv ticketIssued(byte id) {
        self.try() after(30);
    }
}
```

For a given Timed Rebeca model, a modeler can specify the correctness properties of the model as a set of assertions or TCTL formula. The structure of property specifications for Timed Rebeca models is the same as the property specification of Core Rebeca models except that there is TCTL part instead of LTL part. In TCTL specifications $AU(time <= c, \phi, \psi)$, $EU(time <= c, \phi, \psi)$, $AF(time <= c, \phi)$, $AG(time <= c, \phi)$ are used to specify $\forall \phi \, \mathbf{U}^{\leq c} \, \psi$, $\exists \phi \, \mathbf{U}^{\leq c} \, \psi$, $\forall \Diamond^{\leq c} \, \phi$, $\forall \Box^{\leq c} \, \phi$ respectively. The same formula can be used to express modalities with $\geq c$ time constraint.

2.3 Probabilistic and Probabilistic Timed Rebeca

Probabilistic Rebeca is an extension of Rebeca for modeling actor-based systems with probabilistic and nondeterministic behavior [8]. In order to provide a concise syntax for Probabilistic Rebeca, different possibilities of probabilistic aspects that could exist in an actor based system are investigated and two keywords together with one expression definition are added to Rebeca. The first keyword is `pAlt` which models probabilistic alternative behavior in the `switch-case` style. In a pAlt structure, each block of statements may be executed by its associated probabilities. The second keyword is `probloss` which can only be used in conjunction with a method call. The value of the argument of `probloss` shows the probability of losing this message in the communication among actors. They only new expression definition of Probabilistic Rebeca is the *probabilistic expression* which its definition is like nondeterministic expressions such that a real number is associated with each choice of it. We illustrate the application of these features using the Probabilistic Rebeca version of the ticket service system in Listing 1.4. As shown in line 3, there is a probability of 0.4 percent for the computation delay of 2 and 0.6 for 3. Finally, a customer may decide to not to ask for a new ticket with the probability of 0.5 as shown in line 10.

Listing 1.4. The Probabilistic Rebeca model of Ticket Service System

```
 1 reactiveclass TicketService {
 2     msgsrv requestTicket() {
 3         delay(?(0.4:2, 0.6:3));
 4         a.ticketIssued(nextId);
 5         nextId = nextId + 1;
 6     }
 7 }
 8 reactiveclass Customer (2) {
 9     msgsrv try() {
10         pAlt{
11             0.5: a.requestTicket();
12             0.5: self.try();
13         }
14     }
15 }
```

Probabilistic Timed Rebeca (PTRebeca) is an extension of Rebeca which benefits from modeling features of Timed Rebeca and Probabilistic Rebeca, combining the syntax of both languages [9]. This aims at enhancing modeling abilities in order to cover performance evaluation of probabilistic real-time actors.

Although there is no new feature in the syntax of PTRebeca a new semantics is defined for it to support timing, probabilistic, and nondeterministic features [16]. PTRebeca is the first actor-based language which supports time, probability, and nondeterminism in modeling distributed systems with asynchronous message passing.

3 Afra Architecture

Afra is the modeling and analysis IDE of the Rebeca family models[1]. It is developed as an Eclipse plugin and released as a standalone Eclipse product. It contains a set of Eclipse views and editors together with three Java components for implementing models and analyzing them. As shown in Fig. 1, the Afra plugin contains compiler component for compiling its given models, RMC component for generating model checking codes for models, and model transformer component to transform the Rebeca family models to some other well-known models and programs.

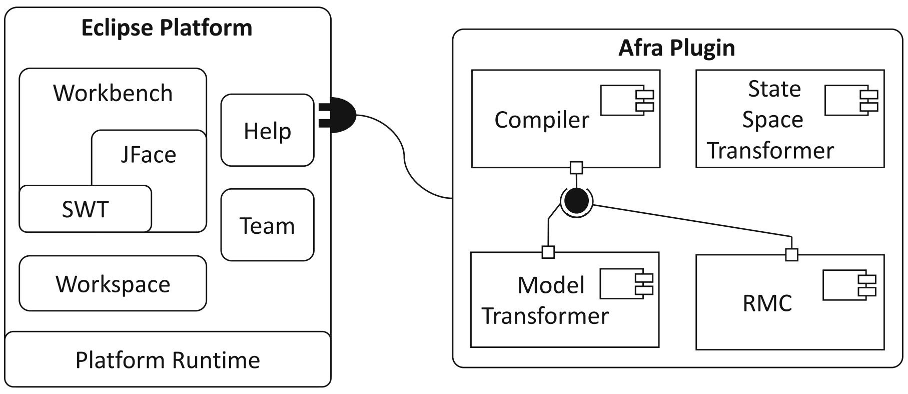


Fig. 1. Components and connectors view of Afra

Using Afra, the compiler component makes sure that a given model is syntactically and semantically correct. At the second step, the transition system of the given model has to be generated and it has to be analyzed against given correctness properties. To this aim, the given model is transformed to a set of C++ source. Running the generated C++ codes provides the model checking result by generating the transition system of the model. The summary of the user activities to this end is shown in the Activity Diagram of Fig. 2.

The first release of the Afra benefits from the model checking engine which was developed in 2006 for Core Rebeca models [17], called Modere. Modere has an object-oriented design and the next model checking engines for the other members of the Rebeca family are developed by extending Modere classes. The overview of the design of model checking classes of Afra is presented in Fig. 3. We

[1] Afra can be downloaded from http://rebeca-lang.org/alltools/Afra.

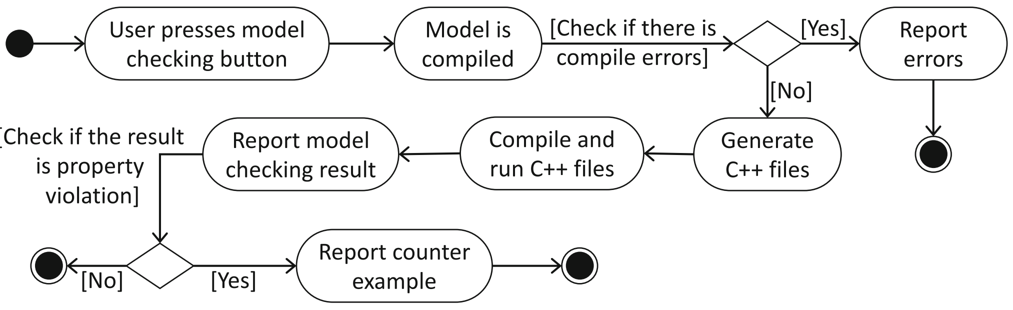

User presses model checking button
Model is compiled
[Check if there is compile errors]
[Yes]
Report errors
[No]
Generate C++ files
Compile and run C++ files
Report model checking result
[Check if the result is property violation]
[No]
[Yes]
Report counter example

Fig. 2. The main activities of a user with Afra for the model checking of a model

will provide more details about the classes of this diagram in Sects. 5 and 5.2. As illustrated in Fig. 3, `AbstractModelChecker` and `AbstractActor` are two core classes of this design. For the case of Core Rebeca, there is `AbstractCore RebecaAnalyzer` class which deals with actors of models, produces states based on the behavior of actors, and stores them in the state spaces storage (i.e. `CoreRebecaDFSHashmap` in the figure). As the model checker of Core Rebeca has to consider actor classes and model checking algorithm, it is inherited from both of `AbstractModelChecker` and `AbstractCoreRebecaAnalyzer`. The figure illustrates that `AbstractCoreRebecaAnalyzer` is also used for simulating Core Rebeca models[2]. The detailed description of this part of the diagram is provided in Sect. 5. The same condition is valid for the case of Timed Rebeca models. For the case of PTRebeca, inheritance takes place from the classes of Timed Rebeca classes as both the model checking and actors behaviors are developed based on the timed model checker. The detailed description of this part is provided in Sect. 5.2. The extensible hierarchy of Fig. 3 illustrates it can be easily extended to combine/modify actor behaviors and model checking algorithms to support future members of the Rebeca family.

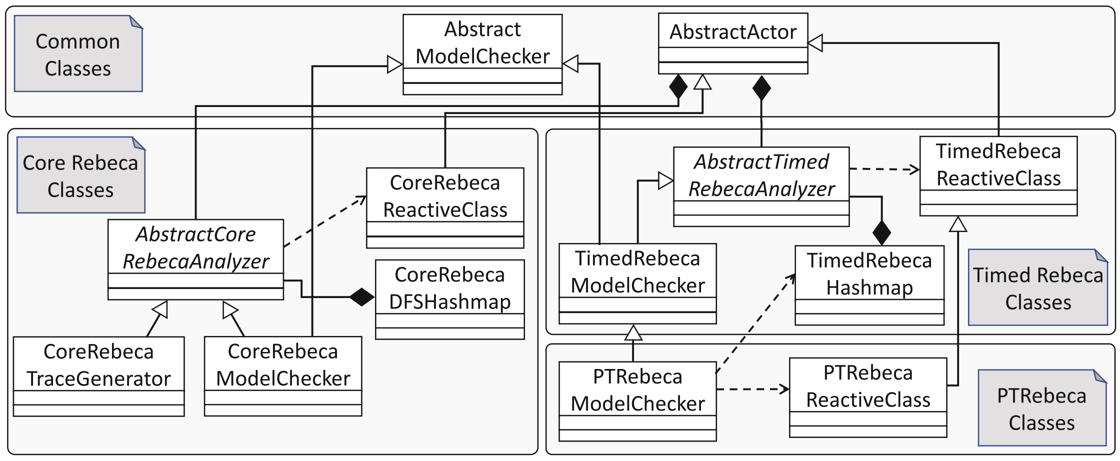


Fig. 3. The UML class diagram of model checkers in Afra

2 This feature is excluded from the current release of Afra.

4 Compiling Rebeca Family Models

Prior to dealing with the complexities of model checking engines of the Rebeca family members, we provide a short overview on how we developed an extensible compiler for them. Rebeca compiler component provides an interface which checks both syntax and semantics of given models and their corresponding property specifications, then publishes their *Abstract Syntax Tree (AST)* using predefined Java objects. It uses ANTLR toolset to parse the Rebeca family model and report syntax errors of the models. To improve the extensibility of the design compiler, we developed two grammar specifications for Core Rebeca: 1) expressions of Rebeca which also includes method calls and sending messages and 2) Rebeca constructs. Then, using the inheritance mechanism of ANTLR for parser specifications, we developed parser specifications of the other Rebeca family extensions. For example, there is a rule for specifying primary terms of expressions which can be an identifier or message sending:

```
primary : IDENTIFIER (LPAREN expressionList RPAREN)?
```

To develop the grammar specification of Timed Rebeca, we explicitly specified that the new grammar is an extension of the Core Rebeca grammar. Then, we overwrite the `primary` rule with the following, as a sending message may be followed by `after` or `deadline` specifiers in Timed Rebeca:

```
primary : IDENTIFIER (LPAREN expressionList RPAREN)? (AFTER LPAREN
    expression RPAREN)? (DEADLINE LPAREN expression RPAREN)?)?
```

For the case of Probabilistic Rebeca, both of the parser specifications are extended to add probabilistic expressions in the expression parser and `pAlt` in the language constructs. The compiler of Probabilistic Timed Rebeca is developed by inheriting from the parsers of both Timed Rebeca and Probabilistic Rebeca and no modification in parsing rules is needed. Compilers of property specifications are developed using the same approach for Core Rebeca and Timed Rebeca models.

The same as the compilers, for the semantic check of the models, we need to consider extensibility and future Rebeca extensions. To this end, we used picocontainer design pattern to manage semantic checker rules of each extension of the language. In addition, two sets of semantics checker are designed for the compiler of the Rebeca family models which check statements and expressions of models, As shown in Fig. 4. Implementing the `check` method in subclasses of `AbstractStatementSemanticsCheck` or `AbstractExpressionSemanticsCheck`, different semantics checkers for the Rebeca family constructs are Developed. Then, based on the Rebeca extension, a subset of these semantics checkers are put in the statements and expressions containers. Note that as Rebeca statements can be nested (e.g. nested loops of conditional statements) each semantic checker delegates semantics checking of its internal statements into the appropriate semantics checker object, which is accessible from the containers. In addition, for considering dynamic scoping of Rebeca variables `ScopeHandler` class in defined which keeps track of activation records, shown in Fig. 4.

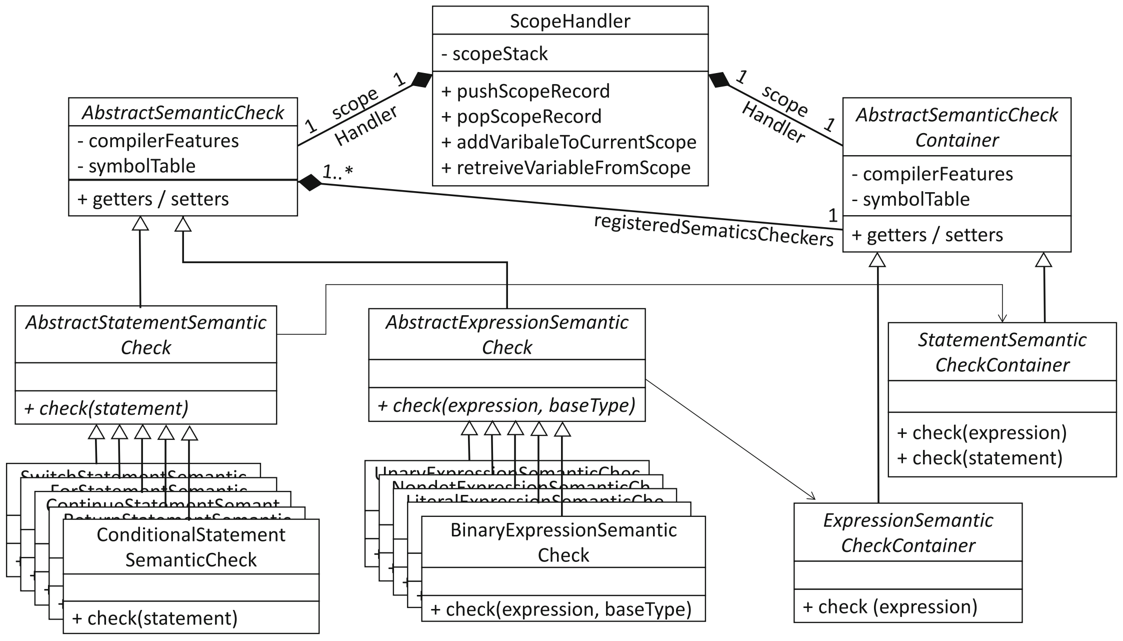

ScopeHandler
- scopeStack
+ pushScopeRecord
+ popScopeRecord
+ addVaribaleToCurrentScope
+ retreiveVariableFromScope
AbstractSemanticCheck
- compilerFeatures
- symbolTable
+ getters / setters
scope Handler
1
1..*
registeredSematicsCheckers
AbstractSemanticCheck Container
- compilerFeatures
- symbolTable
+ getters / setters
AbstractStatementSemantic Check
+ check(statement)
AbstractExpressionSemantic Check
+ check(expression, baseType)
StatementSemantic CheckContainer
+ check(expression)
+ check(statement)
ConditionalStatement SemanticCheck
+ check(statement)
BinaryExpressionSemantic Check
+ check(expression, baseType)
ExpressionSemantic CheckContainer
+ check (expression)

Fig. 4. The UML class diagram of the semantics checker of the Rebeca family

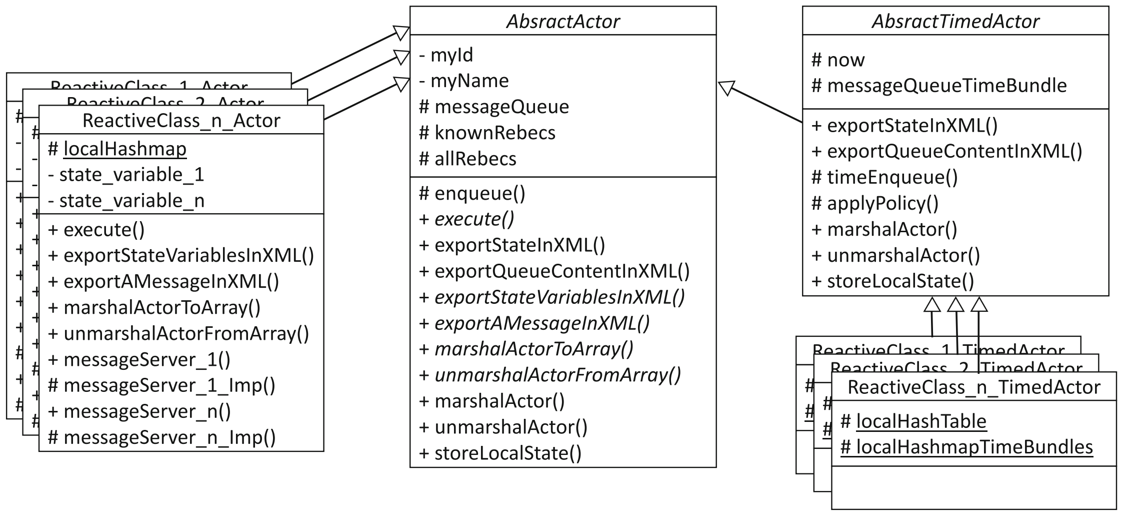


Fig. 5. The UML class diagram of actors classes in Modere

5 Model Checking of Rebeca Family Models

5.1 Model Checking of Core Rebeca

As mentioned before, the correctness properties of Core Rebeca models can be specified by assertions and LTL formulas. In Modere, the model and the negation of the correctness property are generated as two Büchi automata. The model satisfies the correctness property if and only if the synchronous product of these two automata does not accept any word. Otherwise, the accepted word has to be reported as the counterexample of the model. Modere uses Nested Depth

First Search (NDFS) algorithm for computing the product automata on-the-fly. This way, only one DFS is used to generate the product automaton and find the accepting states. To avoid stack overflow, Modere uses the non-recursive implementation of the NDFS and handles the search stack manually. Note that Modere only considers the fair sequences of execution. An infinite sequence is considered (weakly) fair when all the actors of the model are infinitely often executed or disabled. For generating the Büchi automata of the negations of property specifications we used LTL transformer of *Java PathFinder (JPF)* [18].

Based on the design strategies that we introduced in Sect. 3, in Modere, reactive classes are transformed into C++ classes and actors are instantiated from them, shown in Fig. 5. Each class that corresponds to a reactive class has a local hash table (its name if `localHashtable` in Fig. 5) for storing its local states and the index of each local state in the hashtable is assumed as its id. Note that as one hashtable is enough for storing states of all of the instances of a reactive class, it is defined as a `static` field. The global state of the model is the composition of the local states of all of the actors of the model and it is stored in another hashtable as one state of the transition system of the model. Dividing state of the system into inter-process and intra-process hashtables is similar to the method used in SPIN, causes up to 60% reduction in the memory usage for storing transition systems [17]. State exploration in Modere takes place by calling the `execute` method of all the enabled actors from each state. Actors in Modere have an `execute` method that picks a message from the head of its message queue and execute a method which corresponds to that message. As calling `execute` results in delegating the execution to one of the methods of the actor classes, its implementation is different for each actor class; so, it is defined as an `abstract` method in `AbstractActor` and is overwritten in its inherited classes.

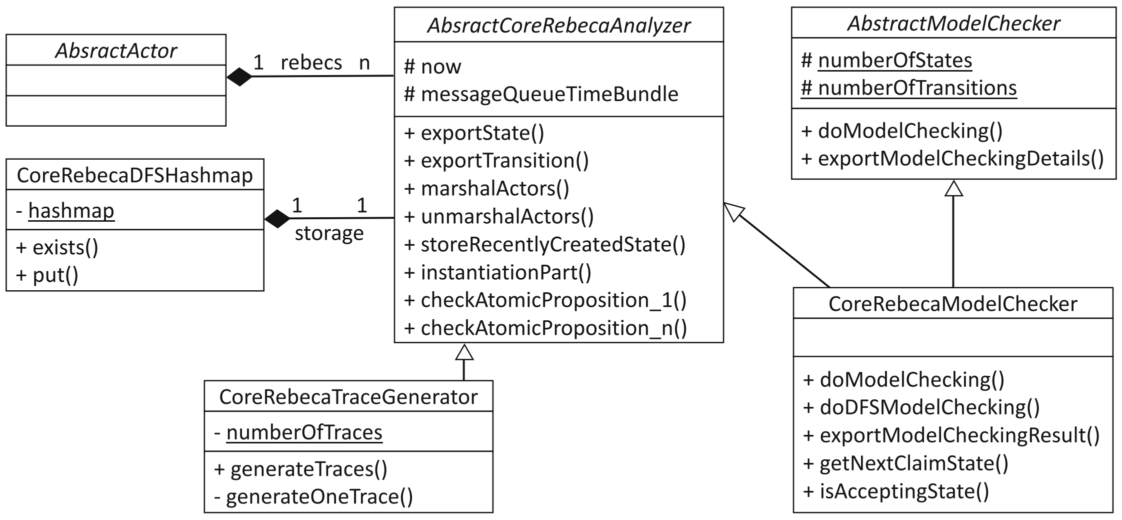


Fig. 6. The UML class diagram of Modere

As shown in Figs. 5, `AbstractActor` provides methods `marshalActor` and `unmarshalActor` for putting/restoring the state of an actor into/from its local hashtable, including the values of state variables and the queue content. But, as different actors have a different set of state variables, the implementation of these two methods are actor dependent. To break this dependency, we defined two abstract helper methods in `AbstractActor`, i.e. `marshalActorToArray` and `unmarshalActorFromArray`, which put/restore the state of an actor into/from a byte array. The actor classes implement these two methods based on their state variables and queues configuration. So, `marshalActor` and `unmarshalActor` methods consider dealing with the local hashtable and use the helper methods to deal with the actor state variables and queues contents. As we will discuss later, this strategy made the implementation of actor classes which correspond to Timed and Probabilistic Rebeca easier. The same strategy is followed in implementing methods which correspond to exporting the state of actors in XML. As shown in Fig. 5, `exportStateInXML` and `exportQueueContentInXML` methods are implemented in `AbstractActor`; but, `exportStateVariablesInXML` and `exportAMessageInXML` are defined as abstract methods and are implemented in the actor classes to consider state variables and message structure of actors.

To implement the provided services of reactive classes, two types of methods are defined in `ReactiveClass_x_Actor` classes, as shown in Fig. 5 (note that `x` in the name of classes are replaced with the name of reactive classes which are defined in the given model). The public methods are called by the other actors and put a message in the queue of actors. The protected methods (which have `_Imp` suffix) are called by the `execute` method of the actors to perform the expected behavior of executing message server.

Using these classes, the model checker of Core Rebeca can be implemented using the classes of Fig. 6. The common behavior of model checking and simulation are put in `AbsractCoreRebecaAnalyzer`. This class is able to handle instantiation of actors (as described in the `main` part), marshal or unmarshal the global state of the system, export the global state of the system in XML, and check atomic propositions in a global state. `CoreRebecaModelChecker` uses these methods to implement the model checking algorithm. The NDFS algorithm of Modere is implemented in `doDFSModelChecking` and two methods `getNextClaimState` and `isAcceptingState` are used to traverse the property Büchi automata and check its accepting states respectively.

As mentioned in [17], Modere has been used for the model checking of models from networking, distributed systems, an some other models from different domains and handles state spaces of up to 10 million states. Also, two reduction techniques have been implemented for it which made it applicable for the analysis of more complicated models.

5.2 Model Checking of Timed and PTRebeca Models

As depicted in Fig. 3, the structure of the Timed Rebeca model checking classes is the same that of in Core Rebeca. Two major semantics have been proposed considered for Timed Rebeca: coarse-grained semantics which is a natural event-based semantics for actors, and fine-grained semantics which is a standard state-based semantics [19]. Using the coarse-grained semantics, in each state, the local time of each actor can be different from the others, i.e., the execution of actors is not synchronized over their local times. The state space which is generated using this semantics is called Floating Time Transition System (FTTS). In contrast, using the fine-grained semantics, the local time of all actors is the same. Note that when we talk about synchronized local clocks we are explaining the concept of time in the model, while fine-grained semantics respects this synchrony, in the coarse-grained we relax the time synchronization constraint. Comparing to the fine-grained semantics, using FTTS can be considered as a reduced state transition system where the event-based properties are preserved.

In addition to differences in the semantics, the mechanism of detecting repeated states in Core Rebeca and Timed Rebeca are different. In Core Rebeca, two states are the same if the valuation of state variables of all actors are the same, together with the content of their message queues. In Timed Rebeca this condition is needed but progress in time does not allow states to be the same as it goes to infinity. It because of the fact that there is no explicit time reset operator in Timed Rebeca. However, reactive systems which generally show periodic or recurrent behaviors are modeled using Timed Rebeca. In other words, they perform periodic behaviors over infinite time. Based on this fact, in [20] we proposed a new notion for equivalence relation between two states to make the transition systems finite, called *shift equivalence relation*. Intuitively, in shift equivalence relation two states are equivalent if and only if they are the same except for the parts related to the time and shifting the times of those parts in one state makes it the same as the other one. To make detecting the shift equivalence relation possible, we divided the content of states into two parts in Timed Rebeca. The first part contains values of state variables and untimed part of the message bag. This part is stored in the local hashtable of actors the same as what we described for Core Rebeca actors. A list of time-bundles is associated with the states of the local hashtable of actors which stores the second part, i.e. the local time of the actor and timed specifier of messages of its message bag. This way, a newly generated state is repeated if it corresponds to an existing item in the local hashtable and shifting the values of its time-bundle make it equal to one of the existing time-bundles associated to that item. These changes require inheriting `AbstractTimedActor` from `AbstractActor` to decorate methods which are responsible for marshaling, unmarshaling, and storing the state of timed actors. In addition, methods which export the state of the timed actors have to be overwritten to put timing specification of actors and its received messages in exported data. As mentioned in [16], behaviors of actors and generating the state space of PTRebeca is very similar to that of in Timed Rebeca. So, `PTRebecaReactiveClass` and `PTRebecaModelChecker` are directly inherited from their corresponding classes in Timed Rebeca.

Combining the mentioned techniques, Timed Rebeca and PTRebeca is used in modeling, model checking, and performance evaluation of NoC designs, WSAN applications, network protocols, and transportation planning, which results in state spaces of up to 10 million states.

6 Conclusion and Future Work

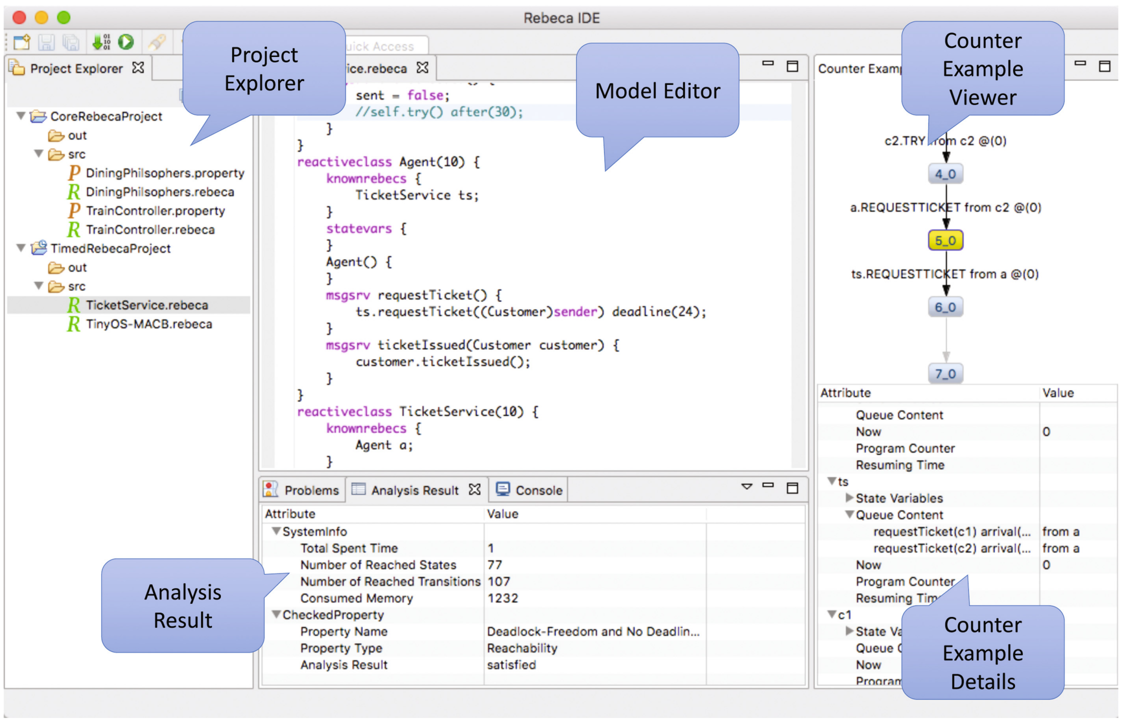
In this paper, we addressed the problem of designing an extensible toolset for modeling and model checking of a family of languages. We showed that how Rebeca family models are defined and how an extensible compiler can be developed for the existing and future extensions of it, in Afra. Using the proposed approach, developing syntax and semantics checkers of the future extension of Rebeca only requires rewriting the compiler specification rules of the modified parts and their semantics checker observers. At the second step, we proposed an extensible design for developing the model checkers of a subset of Rebeca family extensions. Separating actors behavior from the state space generation mechanism, we illustrated that how a new model checker can be developed for a new extension of Rebeca.

We have used the proposed approaches for developing model checkers for Core Rebeca, Timed Rebeca, and Probabilistic Timed Rebeca and integrate them in Afra as an *Eclipse*-based standalone toolset. Afra provides an enriched editor, easy to trace counter-example viewer, and exports models and their transition systems to some other formats. As a future work, we planned to integrate compilers and model checkers of more members of Rebeca family in Afra to benefit from the mentioned facilities. We also want to enrich transformation from models and state spaces to other formalism to allow modelers to use them for analyzing their actor models.

Acknowledgments. The work on this paper has been supported in part by the project "Self-Adaptive Actors: SEADA" (nr. 163205-051) of the Icelandic Research Fund.

A Graphical User Interface of Afra

An overview of Afra user interface is depicted in Fig. 7. Afra user interface consists five main sections which are, projects browser, model and property editor, model-checking result view, and counter example and its details views. The demo of how to work with the toolset is available from the address http://rebeca-lang.org/assets/tools/Afra/Afra-3.0-Demo.mov.



```
            sent = false;
            //self.try() after(30);
        }
}
reactiveclass Agent(10) {
    knownrebecs {
        TicketService ts;
    }
    statevars {
    }
    Agent() {
    }
    msgsrv requestTicket() {
        ts.requestTicket((Customer)sender) deadline(24);
    }
    msgsrv ticketIssued(Customer customer) {
        customer.ticketIssued();
    }
}
reactiveclass TicketService(10) {
    knownrebecs {
        Agent a;
    }
```

Attribute	Value
▼SystemInfo	
Total Spent Time	1
Number of Reached States	77
Number of Reached Transitions	107
Consumed Memory	1232
▼CheckedProperty	
Property Name	Deadlock-Freedom and No Deadlin...
Property Type	Reachability
Analysis Result	satisfied

Attribute	Value
Queue Content	
Now	0
Program Counter	
Resuming Time	
▼ts	
▶State Variables	
▼Queue Content	
requestTicket(c1) arrival(...	from a
requestTicket(c2) arrival(...	from a
Now	0
Program Counter	
Resuming Tim	
▼c1	
▶State Va	
Queue C	
Now	
Program	

Fig. 7. Afra graphical user interface

References

1. Agha, G.A.: ACTORS - A Model of Concurrent Computation in Distributed Systems. Artificial Intelligence, MIT Press, Cambridge (1990)
2. Hewitt, C.: Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot. MIT Artificial Intelligence Technical report 258, Department of Computer Science, MIT, April 1972
3. Sirjani, M., Jaghoori, M.M.: Ten years of analyzing actors: Rebeca experience. In: Formal Modeling: Actors, Open Systems, Biological Systems, pp. 20–56 (2011)
4. Jaghoori, M.M., Sirjani, M., Mousavi, M.R., Khamespanah, E., Movaghar, A.: Symmetry and partial order reduction techniques in model checking Rebeca. Acta Inf. **47**(1), 33–66 (2010)
5. Sabouri, H., Sirjani, M.: Slicing-based reductions for Rebeca. Electr. Notes Theor. Comput. Sci. **260**, 209–224 (2010)
6. Sirjani, M., de Boer, F.S., Movaghar-Rahimabadi, A.: Modular verification of a component-based actor language. J. UCS **11**(10), 1695–1717 (2005)
7. Reynisson, A.H., et al.: Modelling and simulation of asynchronous real-time systems using timed Rebeca. Sci. Comput. Program. **89**, 41–68 (2014)
8. 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
9. Jafari, A., Khamespanah, E., Sirjani, M., Hermanns, H.: Performance analysis of distributed and asynchronous systems using probabilistic timed actors. In: ECEASST 70 (2014)

10. Razavi, N., Behjati, R., Sabouri, H., Khamespanah, E., Shali, A., Sirjani, M.: Sysfier: Actor-based formal verification of systemc. ACM Trans. Embedded Comput. Syst. **10**(2), 19:1–19:35 (2010)
11. Yousefi, B., Ghassemi, F., Khosravi, R.: Modeling and efficient verification of broadcasting actors. In: Dastani, M., Sirjani, M. (eds.) FSEN 2015. LNCS, vol. 9392, pp. 69–83. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24644-4_5
12. Yousefi, B., Ghassemi, F., Khosravi, R.: Modeling and efficient verification of wireless ad hoc networks. CoRR abs/1604.07179 (2016)
13. Parr, T.J., Quong, R.W.: ANTLR: a predicated-LL(k) parser generator. Softw. Pract. Exp. **25**(7), 789–810 (1995)
14. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997)
15. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using Rebeca. Fundam. Inform. **63**(4), 385–410 (2004)
16. 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)
17. Jaghoori, M.M., Movaghar, A., Sirjani, M.: Modere: the model-checking engine of Rebeca. In: Haddad, H. (ed.) Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Dijon, France, April 23–27, 2006, pp. 1810–1815. ACM (2006)
18. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. **10**(2), 203–232 (2003)
19. Khamespanah, E., Sirjani, M., Viswanathan, M., Khosravi, R.: Floating time transition system: more efficient analysis of timed actors. In: Braga, C., Ölveczky, P.C. (eds.) FACS 2015. LNCS, vol. 9539, pp. 237–255. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-28934-2_13
20. Khamespanah, E., Sirjani, M., Sabahi-Kaviani, Z., Khosravi, R., Izadi, M.: Timed Rebeca schedulability and deadlock freedom analysis using bounded floating time transition system. Sci. Comput. Program. **98**, 184–204 (2015)