



RebeCaos

José Proença¹(✉)  and Maurice H. ter Beek²(✉) 

¹ CISTER and University of Porto, Porto, Portugal

jose.proenca@fc.up.pt

² CNR-ISTI, Pisa, Italy

maurice.terbeek@isti.cnr.it

Abstract. We describe **RebeCaos**, a user-friendly web-based front-end tool for the **Rebeca** language, based on the **Caos** library for Scala. **RebeCaos** can simulate different operational semantics of (timed) **Rebeca**, thus facilitating the dissemination and awareness of **Rebeca**, providing insights into the differences among existing semantics for **Rebeca**, and supporting quick experimentation of new **Rebeca** variants (e.g., when the order of received messages is preserved). The tool also comes with initial reachability analyses for **Rebeca** models (e.g., the possibility of reaching deadlocks or desirable states). We illustrate the **RebeCaos** tool by means of a ticket service use case from the timed **Rebeca** literature.

Keywords: Rebeca · actors · time · web front-end · reachability analysis

1 Introduction

This tool paper accompanies a recent Festschrift contribution [5], which we refer to in particular for technical details. In this paper, we focus on describing some details of the implementation and of the tool at work.

The Reactive objects language (**Rebeca**) [18, 22, 23] is a high-level language designed for modelling and analysing concurrent and distributed systems based on the actor model of computation, which views systems as a collection of autonomous objects (actors) that communicate via asynchronous message passing.

Actors or reactive objects (**rebecs**) constitute its primary modelling components, which are particularly useful for modelling and analysing reactive systems in which components react to incoming messages. **Rebecs** communicate in a non-blocking fashion via asynchronous message passing between senders and receivers. Each **rebec** has a set of variables that store values, a set of methods (called message servers) and a message bag to store the received messages (along with their arrival times and their deadlines). Operationally, a **rebec** may take a message (with the least arrival time) from its message bag and execute the corresponding message server. Each **rebec** operates concurrently and can process one message at a time.

© IFIP International Federation for Information Processing 2025

Published by Springer Nature Switzerland AG 2025

C. Di Giusto and A. Ravara (Eds.): COORDINATION 2025, LNCS 15731, pp. 219–229, 2025.

https://doi.org/10.1007/978-3-031-95589-1_11

Throughout the years, several extensions of *Rebeca* have been introduced for the modelling and analysis of systems from specific domains, among which *pRebeca* [27] for probabilistic systems, *Timed Rebeca* [24] for real-time systems, *PTRebeca* [12] for probabilistic timed systems, and *Hybrid Rebeca* [14] for cyber-physical systems. In particular, *Timed Rebeca* extends core *Rebeca* with a global notion of time [1, 17, 20, 24], which is achieved by synchronisation of (local) time of the actors (rebecs) involved. In *Timed Rebeca*, the primitives *delay* and *after* are used to model the progress of time while executing a message server.

Timed Rebeca is supported by the tool *Afra* [16], which offers a comprehensive IDE for specifying and verifying *Rebeca* models. It unifies the Java artifacts from various *Rebeca*-related projects and offers tools for model creation, property specification, model checking, and counterexample visualisation. Besides *Afra*, there is a rich ecosystem of tools around *Rebeca* [22],¹ including generators for back-end model checkers such as *SMV* [25], *mCRL2* [11], and *McErlang* [9], a dedicated model checker *Modere* [13], and a more recent *Jacco* model checker for Java actors [28] based on *Rebeca*'s existing toolset.

We recently developed the *Caos* Scala framework (Computer aided design of structural operational semantics) [19]. *Caos* supports the creation of interactive JavaScript-based websites meant to animate operational semantics. These websites can provide both a quick feedback for developers and good insights to newcomers of a specific language. *Caos* has been used, e.g., to animate and guide the development of the semantics of choreographic languages [6–8, 15] and reactive systems [3, 4, 26], as well as to teach students about the semantics of C-like languages and of a concurrent process calculus.²

A survey from 2020 with the participation of 130 formal methods experts—including three Turing Award winners, all five FME Fellowship Award winners, and 17 CAV Award winners—acknowledges the importance of supporting tools for teaching formal methods, since “an overwhelming majority of answers judged the use of tools essential when teaching formal methods” (75.4% of the respondents answered “major role”) when asked “whether, and to which extent, students should be exposed to software tools when being taught formal methods” [10, Section 6: Formal Methods in Education]. Moreover, tools “allow students to quickly link theory with practice” [2].

To this aim, we recently presented an animator of *Rebeca*, called *RebeCaos* [5].

Contribution. We describe *RebeCaos*, a user-friendly web-based front-end tool for *Rebeca* based on *Caos*, introduced in [5], to facilitate further dissemination and awareness of *Rebeca*, provide insights into the differences among existing *Rebeca* semantics, and support quick experimentation of new *Rebeca* variants (e.g., when the order of received messages is preserved). We describe some details of its implementation and also present some initial reachability analyses for *Rebeca* models (e.g., the possibility of reaching deadlocks or desirable states), which are novel contributions with respect to [5].

¹ <https://rebeca-lang.org/tools>.

² Many examples are listed here: <https://github.com/arcalab/caos>.

<pre> reactiveclass Example { // rebecs to who it can send knownrebecs { Example ex; } // internal state variables statevars { int counter; } } </pre>	<pre> // available methods msgsrv initial() { counter=0; ex.add(1); } msgsrv add(int a) { if (counter < 100) {counter = counter + a;} } } </pre>	<pre> // Starting point to // run the system main { Example ex1(ex2):(); Example ex2(ex1):(); } </pre>
--	--	---

Fig. 1. Simple toy example³ borrowed from Hojjat et al. [11]

Outline. After this Introduction, Sect. 2 informally recalls the syntax and semantics of core Rebeca, extended with time and dynamism, as described in detail in [5]. This is followed by the presentation of RebeCaos in Sect. 3, after which we conclude the paper and present ideas for future work in Sect. 4.

2 Rebeca in RebeCaos: Syntax and Semantics


This section provides a quick introduction to Rebeca’s syntax and semantics from [5], implemented in RebeCaos, by means of a simple toy example³, presented in Fig. 1, borrowed from Hojjat et al. [11],³ with a few adaptations. This program resembles a typical object-oriented one, where all objects are actors (called *rebecs*) and method invocation is asynchronous. In this concrete system there is a single **reactive class** **Example** that is instantiated twice in the **main** block on the right. The first instance is called **ex1** and the second **ex2**. Two groups of arguments can be passed: the first to provide other rebecs that can be used to call methods to, and the second to provide values to initialise the rebec (via the **initial** method).

In this system, both **Example** instances initialise their counter to zero and ask each other to increment their counter by one. The order in which they increment their counter is not fixed, and in the end both rebecs will have their counter set to one. The syntax and semantics is precisely described in [5, Sects. 2 and 3]. Below we briefly describe how these are represented in RebeCaos.

2.1 Syntax in RebeCaos

RebeCaos is implemented in Scala, compiled into JavaScript, which is compiled to JavaScript and loaded by a stand-alone HTML file. It uses our Caos library [19] to generate the web-based front-end and present the full state-space exploration.

The general structure of a Rebeca program in RebeCaos is given by the Scala data type **System** in Fig. 2.⁴ A system is a pair with (1) a table mapping names

³ The electronic version of this paper includes hyperlinks to examples that open in our online tool, marked with the symbol .

⁴ The full syntax code can be found at <https://github.com/FM-DCC/rebeaos/blob/v0.1/src/main/scala/rebeaos/syntax/Program.scala>.

<pre> case class System(classes: Map[String, ReactiveClass] , main: List[InstanceDecl]) case class ReactiveClass(qsize: Option[Int] , known: List[QVar] , state: List[QVar] , msgsrv: Map[String, Msgsrv]) </pre>	<pre> case class InstanceDecl(clazz: String , name: String , known: List[String] , args: List[Expr]) case class Msgsrv(vars: List[QVar] , stm: Statement) </pre>
---	--

Fig. 2. Structure of a Rebeca program in **RebeCaos**

to with known Rebeca classes, and (2) a list of instance declarations; a Rebeca class has (1) a possible queue size, (2) a set of variables for known rebecs, (3) a set of variables for the state, and (4) a set of methods. In turn, an instance declaration has (1) a class name, (2) a rebec name, (3) a list of known rebec names, and (4) a list of arguments; and a method has a list of variables and a statement. We omit the definitions of *QVar* (variable names qualified with a type name), expressions *Expr*, and statements *Statement*.

RebeCaos includes a parser written with parser combinators that produces a **System** from the input text, available at <https://github.com/FM-DCC/rebecaos/blob/v0.1/src/main/scala/rebecaos/syntax/Parser.scala>. It does not include type-checking or other simple analyses, hence some errors are caught only at runtime.

2.2 Semantics in **RebeCaos**

RebeCaos uses the operational semantics from the literature [5], by evolving a *configuration* that captures the state of a program. At each operational step, a pending message is *selected*, triggering *execution* of the body of the associated method by the receiving rebec, and resulting in an update of the set of pending messages. **RebeCaos** supports a timed extension of Rebeca [1] (with delays and deadlines) and a dynamic extension [21] (with runtime creation of new rebecs).

Some snippets of the implementation are presented in Fig. 3. The full implementation can be found online at <https://github.com/FM-DCC/rebecaos/blob/v0.1/src/main/scala/rebecaos/backend/Semantics.scala>.

Configuration. The left of Fig. 3 presents the general structure for a configuration **St** of a running program. This state includes the full Rebeca program (**System**), the local state of each rebec (**Rebecs**), and a bag of messages (**Msgs**). In turn, each message describes the receiver *rcv*, the method name *m*, the actual arguments *args*, the sender *snd*, and possible time restrictions (a waiting time before execution *tt* and an optional deadline *dl*).

Evolution. The right of Fig. 3 presents the signature of the main functions that define the evolution of a configuration. The initial configuration is built by *initSt*, which creates empty states for each rebec declared in the main body, and initialises the pending messages with the *initial* methods of each rebec. The operational semantics is encoded by the *next* function which, given the current

<pre> type St = (System, Rebecs, Msgs) type Rebecs = Map[String, RebecEnv] type Msgs = Bag[Msg] case class Msg(rcv:String, m:String, args:List[Data], snd:String, tt:Int, dl:Option[Int]) </pre>	<pre> def initSt(s: System): St = //... def next(st: St): Set[(Act, St)] = //... def enabled(m: Msg, initials: Set[String], smallestTT: Int) : Boolean = //... def evalStm(stm: Statement)(using reb: RebecEnv, syst: System) : Set[(RebecEnv, Msgs, Rebecs)] = //... </pre>
---	--

Fig. 3. Snippet of the implementation of the semantics of Rebeca in **RebeCaos**

```

val widgets = List(
  //...
  "Run_semantics_(state's_view)" -> steps((e:St)=>e, Semantics, Show.apply, a=>Show(a._1), Text),
  "Run_semantics_(sequence_chart)" -> steps(..., Mermaid)
  //...
)

```

Fig. 4. Snippet of the configuration file that describes some widgets in **RebeCaos**

configuration `st`, produces a set of possible next configurations labelled by an action of type `Act` (describing the message being executed). This `next` function is part of the CAOS framework, and exploited to generate the visual analysis in **RebeCaos**. Its definition uses auxiliary functions such as `enabled`, which checks if a given message can be selected (i.e., it has no pending initial method and it has the smallest delay to start), and `evalStm`. The latter evaluates the semantics of the body of a method using a big-step semantics [5]. Given its non-deterministic nature, the result is a set of possible outcomes, each of which has an updated rebec state, a set of outgoing messages, and set of newly produced rebecs.

3 RebeCaos at Work

This section describes how to use **RebeCaos**, a tool for animating the semantics of Rebeca, as first introduced in our Festschrift contribution [5].

RebeCaos is a Scala implementation with a web-based front-end to explore the state-space of Rebeca programs. The tool consists of an interactive webpage that does not rely on a server, and that can easily be used in any browser with JavaScript support.

We describe how to use **RebeCaos** guided by an example: a dynamic variation of the [simple toy example](#)[↗], listed in Fig. 1. In Sect. 3.4, we present some initial reachability analyses for Rebeca models (e.g., the possibility of reaching deadlocks or desirable states), which is a novel contribution with respect to [5].

3.1 Interface of **RebeCaos**

RebeCaos can be opened by navigating to <https://fm-dcc.github.io/rebecaos>, where the user can view an interface similar to the screenshot in Fig. 5. Here we use the second example, called “[Dyn] Simple”[↗], which is a variation of the simple toy example from Fig. 1 that dynamically creates new instances. This

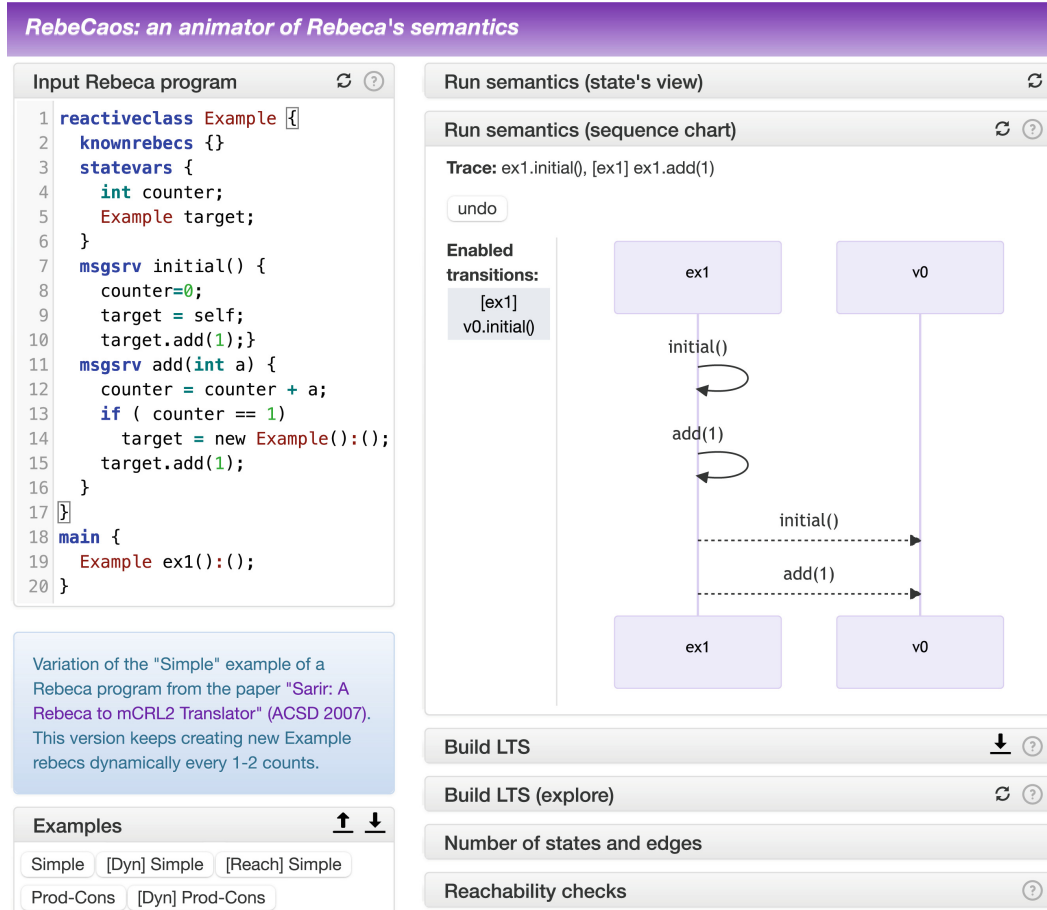


Fig. 5. Screenshot of the interface of RebeCaos with a simple dynamic example⁵

interface includes several *widgets*, each of which can be collapsed (such as Run semantics (state's view)) or expanded (such as Run semantics (sequence chart)). Clicking the title of a widget toggles between these modes and reloads its content.

In Fig. 5, we provide the above mentioned variation of the simple toy example in the widget Input Rebeca Program, plus an interactive step-by-step execution of its semantics using sequence charts. Other examples, including the ones mentioned throughout the paper and in [5], can be loaded from the Examples widget.

Widgets are specified in an object that describes the layout of the website (available at <https://github.com/FM-DCC/rebecaos/blob/v0.1/src/main/scala/rebecaos/frontend/CaosConfig.scala>). A snippet of the code building the widgets that run the semantics (using the semantics from Fig. 3) is presented in Fig. 4. The widget constructor `steps` receives a pre-processing function, an object that implements the `next` function over states in `St` (Semantics), two functions to present textual representations of the actions and the states, and a marker (Text or Mermaid) describing how to interpret the textual representation of the states (as pure text or as a Mermaid diagram,⁵ respectively).

⁵ See <https://mermaid.js.org/>.

3.2 Running Step-by-Step

The widget Run semantics (sequence chart) in Fig. 5 supports a guided step-by-step execution of an input Rebeca program, following closely the semantics presented in [5, Sect. 2.2]. The sequence diagram depicts the sequence of *called* and *processed* methods; in this case, the rebec `ex1` already processed the methods `initial()` and `add(1)`, marked with a solid line, and also created a new rebec `v0` and called its method `add(1)`. On the other hand, neither of the `initial` and the `add` methods have been processed yet, marked with a dashed arrow.

The column on the left of the sequence chart lists the *enabled transitions*; in this case there is only one enabled transition named “[`ex1`] `v0.initial()`”, representing a pending method call `initial` to rebec `v0` by triggered by rebec `ex1`. The method `add(1)` is not yet enabled because initial methods have precedence over all other methods of the same rebec. By iteratively clicking on an enabled transition we can grow the sequence chart, producing a trace in which solid arrows are ordered based on when they are processed, while dashed arrows are randomly ordered at the bottom, representing the multiset of pending messages.

An alternative widget to build a trace of a Rebeca program is Run semantics (state’s view), not depicted in the screenshot. This has the same buttons to select enabled transitions, but instead of the sequence chart it depicts the precise state of each rebec and the multiset of pending messages.

3.3 Running All Steps

It is often convenient to automatically traverse all possible states, instead of manually creating possible traces. This is performed by the widget Build LTS, illustrated in [5, Figs. 3 and 4].⁶ This concrete program was used, e.g., by Khamespanah et al. [17], and it produces an infinite state space partially represented in [5, Fig. 4]. Our implementation performs a breath-first traversal, stopping after a finite number of steps. By using an [adaptation of this program without time references](#)⁷, included in the examples of our tool, the state space becomes finite, also depicted in [5, Fig. 4]. The initial parts of these two graphs are identical. Interestingly, the finite state space for the untimed version has the same shape as the state space produced by Khamespanah et al. [17] for the timed version, where they use an optimisation that collapses states that are similar, i.e., with a behaviour that keeps on repeating itself after some time. This optimisation is not currently implemented in RebeCaos.

3.4 Reachability Analysis

The widget Reachability checks is a novel contribution to be used as follows. An input Rebeca program can end with some lines with reachability queries, written as “reaches `EXPR`;” (cf. Fig. 6, left side). This new widget traverses the state

⁶ The widget Build LTS (explore) (cf. Fig. 5) is similar to Built LTS but the state space is drawn iteratively, requiring the user to click the states that (s)he wants to expand.

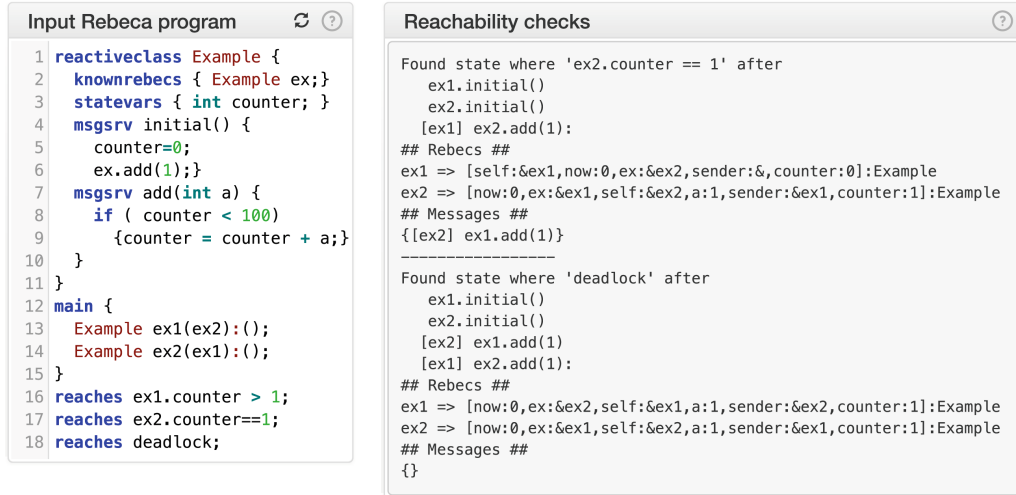


Fig. 6. Reachability feedback using our [toy example with reachability queries](#)

space while searching for states where property **EXPR** holds. It terminates when all queries have been validated, the state space has been fully traversed, or after a bound on the number of transitions has been reached (whatever happens first).

An example is displayed in Fig. 6, where we search for the three properties appended at the end of the input Rebeca program depicted on the left. Reachability queries are written using the same syntax as expressions in Rebeca, with the exception that variable names must be qualified. For example, `ex1.counter` represents the `counter` variable of rebec `ex1`. Furthermore, the special keyword `deadlock` is a predicate that holds if and only if the state has no outgoing transitions. In the displayed example, the Reachability checks widget of **RebeCaos** finds states for only two of the reachability queries (cf. Fig. 6, right side). For each of these two queries, it provides both a sequence of actions and a description of the state reached (including variables of the rebecs and the messages in the queue).

3.5 Beyond **RebeCaos**

Existing tools for Rebeca, like **Afra** [16], use an extended version of the syntax presented in [5, Sect. 2.1]. These extensions—including, e.g., arrays, syntactic macros (with the `env` keyword), and type casting—are not yet implemented in **RebeCaos**, as they are typically not needed for the examples from the literature.

Furthermore, as mentioned in the Introduction, **Rebeca** has been equipped with a probabilistic semantics (cf., e.g., [12, 27]) and this also has not been implemented in **RebeCaos** so far. **RebeCaos** moreover provides neither support for model checking nor for code generation, mainly because **RebeCaos** is not meant to substitute an IDE for **Rebeca**, but rather to provide an easy entry point.

RebeCaos can be useful not only to help teaching and explaining the insights of **Rebeca**, but also as a relatively easy and intuitive tool for quick feedback on experiments with new extensions or with variations. One could, for instance, implement a semantics that preserves the order of method calls, replacing the

multiset of pending messages B by a queue, or attempt to define a type-checking algorithm that verifies conformity with a given behavioural type.

4 Conclusion

We presented **RebeCaos**, a user-friendly web-based front-end tool based on our **Caos** library for Scala, and illustrated by means of examples how to animate **Rebeca** extended with time and dynamism. In particular, we presented some initial reachability analyses for **Rebeca** models (e.g., the possibility of reaching deadlocks or desirable states), which is a novel contribution. We expect to include a generalisation of this reachability search in a future version of **Caos**.

In the future, we might extend the syntax and semantics of **Rebeca** currently supported by **RebeCaos** with a means to deal with probabilistic extensions of **Rebeca**, with support for the symbolic time representation presented by Khamespanah et al. [17], or with entirely new extensions such as an experimental type-checking algorithm for verifying conformity with a given behavioural type.

Acknowledgments. This work was funded by the MUR PRIN 2020TL3X8X project T-LADIES (Typeful Language Adaptation for Dynamic, Interacting and Evolving Systems) and by the CNR project “Formal Methods in Software Engineering 2.0”, CUP B53C24000720005. This work is also supported by the CISTER Research Unit (UIDP/UIDB/04234/2020), financed by National Funds through FCT/MCTES (Portuguese Foundation for Science and Technology), and by the EU/Next Generation, within the Recovery and Resilience Plan, within project Route 25 (TRB/2022/00061 – C645463824-00000063).

References

1. Aceto, L., Cimini, M., Ingólfssdóttir, A., Reynisson, A.H., Sigurdarson, S.H., Sirjani, M.: Modelling and simulation of asynchronous real-time systems using timed rebeca. In: Mousavi, M.R., Ravara, A. (eds.) Proceedings of the 10th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2011). EPTCS, vol. 58, pp. 1–19 (2011). <https://doi.org/10.4204/EPTCS.58.1>
2. ter Beek, M., Broy, M., Dongol, B.: The role of formal methods in computer science education. *ACM Inroads* **15**(4), 58–66 (2024). <https://doi.org/10.1145/3702231>
3. ter Beek, M.H., Cledou, G., Hennicker, R., Proença, J.: Can we communicate? Using dynamic logic to verify team automata. In: Chechik, M., Katoen, J.P., Leucker, M. (eds.) FM 2023. LNCS, vol. 14000, pp. 122–141. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-27481-7_9
4. ter Beek, M.H., Hennicker, R., Proença, J.: Team automata: overview and roadmap. In: Castellani, I., Tiezzi, F. (eds.) COORDINATION 2024. LNCS, vol. 14676, pp. 161–198. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-62697-5_10

5. ter Beek, M.H., Proença, J.: Animating Rebeca. In: Lee, E.A., Mousavi, M.R., Talcott, C. (eds.) *Rebeca for Actor Analysis in Action*. LNCS, vol. 15560, pp. 182–194. Springer, Cham (2025). https://doi.org/10.1007/978-3-031-85134-6_8
6. Cledou, G., Edixhoven, L., Jongmans, S.S., Proença, J.: API generation for multiparty session types, revisited and revised using Scala 3. In: Ali, K., Vitek, J. (eds.) *Proceedings of the 36th European Conference on Object-Oriented Programming (ECOOP 2022)*. LIPIcs, vol. 222, pp. 27:1–27:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). <https://doi.org/10.4230/LIPIcs.ECOOP.2022.27>
7. Edixhoven, L., Jongmans, S.S.: Realisability of branching pomsets. In: Tapia Tarifa, S.L., Proença, J. (eds.) *FACS 2022*. LNCS, vol. 13712, pp. 185–204. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-20872-0_11
8. Edixhoven, L., Jongmans, S.S., Proença, J., Cledou, G.: Branching pomsets for choreographies. In: Aubert, C., Di Giusto, C., Safina, L., Scalas, A. (eds.) *Proceedings of the 15th Interaction and Concurrency Experience (ICE 2022)*. EPTCS, vol. 365, pp. 37–52 (2022). <https://doi.org/10.4204/EPTCS.365.3>
9. Fredlund, L., Svensson, H.: McErlang: a model checker for a distributed functional programming language. In: *Proceedings of the 12th International Conference on Functional Programming (ICFP 2007)*, pp. 125–136. ACM (2007). <https://doi.org/10.1145/1291151.1291171>
10. Garavel, H., ter Beek, M.H., van de Pol, J.: The 2020 expert survey on formal methods. In: ter Beek, M.H., Ničković, D. (eds.) *FMICS 2020*. LNCS, vol. 12327, pp. 3–69. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58298-2_1
11. Hojjat, H., Sirjani, M., Mousavi, M.R., Groote, J.F.: Sarir: a Rebeca to mCRL2 translator. In: *Proceedings of the 7th International Conference on Application of Concurrency to System Design (ACSD 2007)*, pp. 216–222. IEEE (2007). <https://doi.org/10.1109/ACSD.2007.62>
12. 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). <https://doi.org/10.1016/J.SCICO.2016.03.004>
13. Jaghoori, M.M., Movaghar, A., Sirjani, M.: Modere: the model-checking engine of Rebeca. In: *Proceedings of the 21st Symposium on Applied Computing (SAC 2006)*, pp. 1810–1815. ACM (2006). <https://doi.org/10.1145/1141277.1141704>
14. Jahandideh, I., Ghassemi, F., Sirjani, M.: Hybrid Rebeca: modeling and analyzing of cyber-physical systems. In: Chamberlain, R.D., Taha, W., Törngren, M. (eds.) *WESE 2018*. LNCS, vol. 11615, pp. 3–27. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-23703-5_1
15. Jongmans, S.S., Proença, J.: ST4MP: a blueprint of multiparty session typing for multilingual programming. In: Margaria, T., Steffen, B. (eds.) *ISoLA 2022*. LNCS, vol. 13701, pp. 460–478. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-19849-6_26
16. Khamespanah, E., Sirjani, M., Khosravi, R.: Afra: an eclipse-based tool with extensible architecture for modeling and model checking of Rebeca family models. In: Hojjat, H., Ábrahám, E. (eds.) *FSEN 2023*. LNCS, vol. 14155, pp. 72–87. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-42441-0_6
17. 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). <https://doi.org/10.1016/J.SCICO.2014.07.005>
18. Khosravi, R., Khamespanah, E., Ghassemi, F., Sirjani, M.: Actors upgraded for variability, adaptability, and determinism. In: de Boer, F.S., Damiani, F.,

- Hähnle, R., Johnsen, E.B., Kamburjan, E. (eds.) *Active Object Languages: Current Research Trends*, LNCS, vol. 14360, pp. 226–260. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-51060-1_9
19. Proença, J., Edixhoven, L.: Caos: a reusable Scala web animator of operational semantics. In: Jongmans, S.S., Lopes, A. (eds.) *COORDINATION 2023*. LNCS, vol. 13908, pp. 163–171. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-35361-1_9
 20. Reynisson, Á.H., et al.: Modelling and simulation of asynchronous real-time systems using Timed Rebeca. *Sci. Comput. Program.* **89**, 41–68 (2014). <https://doi.org/10.1016/J.SCICO.2014.01.008>
 21. Sirjani, M.: *Formal Specification and Verification of Concurrent and Reactive Systems*. Ph.D. thesis, Sharif University of Technology (2004)
 22. Sirjani, M.: Rebeca: theory, applications, and tools. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) *FMCO 2006*. LNCS, vol. 4709, pp. 102–126. Springer, Cham (2006). https://doi.org/10.1007/978-3-540-74792-5_5
 23. Sirjani, M., Jaghoori, M.M.: Ten years of analyzing actors: Rebeca experience. In: Agha, G., Danvy, O., Meseguer, J. (eds.) *Formal Modeling: Actors, Open Systems, Biological Systems*. LNCS, vol. 7000, pp. 20–56. Springer, Cham (2011). https://doi.org/10.1007/978-3-642-24933-4_3
 24. Sirjani, M., Khamespanah, E.: On time actors. In: Ábrahám, E., Bonsangue, M.M., Johnsen, E.B. (eds.) *Theory and Practice of Formal Methods*. LNCS, vol. 9660, pp. 373–392. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30734-3_25
 25. Sirjani, M., Shali, A., Jaghoori, M.M., Iravanchi, H., Movaghar, A.: A front-end tool for automated abstraction and modular verification of actor-based models. In: *Proceedings of the 4th International Conference on Application of Concurrency to System Design (ACSD 2004)*. pp. 145–150. IEEE (2004). <https://doi.org/10.1109/CSD.2004.1309125>
 26. Tinoco, D., Madeira, A., Martins, M.A., Proença, J.: Reactive graphs in action. In: Marmosler, D., Sun, M. (eds.) *FACS 2024*. LNCS, vol. 15189, pp. 97–105. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-71261-6_6. cf. Reactive graphs in action (extended version), [arXiv:2407.14705](https://arxiv.org/abs/2407.14705) [cs.PL], <https://doi.org/10.48550/arXiv.2407.14705>
 27. 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
 28. Zakeriyan, A., Khamespanah, E., Sirjani, M., Khosravi, R.: Jacco: more efficient model checking toolset for Java actor programs. In: Boix, E.G., Haller, P., Ricci, A., Varela, C.A. (eds.) *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE! 2015)*, pp. 37–44. ACM (2015). <https://doi.org/10.1145/2824815.2824819>