Animating Rebeca

Maurice H. ter Beek[1(✉)] and José Proença[2(✉)]

[1] CNR–ISTI, Pisa, Italy
maurice.terbeek@isti.cnr.it
[2] CISTER and University of Porto, University of Porto, Porto, Portugal
jose.proenca@fc.up.pt

Abstract. Rebeca is 20+ years old. Introduced by Marjan Sirjani and colleagues for modelling and analysing actor-based systems, it comes with a variety of tool support, including dedicated model checkers, simulators, and code generators. When encountering Rebeca for the first time, either as a student, as a researcher, or as a practitioner from industry, one needs to grasp the subtleties of Rebeca's semantics, which includes variants with probabilities and time.

This paper presents a user-friendly web-based front-end, based on the Caos library for Scala, to animate different operational semantics of (timed) Rebeca. This can facilitate the dissemination and awareness of Rebeca, provide insights into the differences among existing semantics, and support quick experimentation of new variants (e.g., when the order of received messages is preserved). The tool is illustrated by means of a ticket service use case from the literature.

Keywords: Rebeca · operational semantics · web front-end · animation · actors · time

1 Introduction

The *Reactive objects language* (Rebeca) [21,24,25] 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. Rebeca has a 25-year history [27,28], which traces back to the agent-based language Planner proposed by Hewitt [13], further developed into a concurrent object-based language by Agha [2], and formalised into a theory of actor computation by Agha et al. [3,4].

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

E. A. Lee et al. (Eds.): Marjan Sirjani Festschrift, LNCS 15560, pp. 182–194, 2025.
https://doi.org/10.1007/978-3-031-85134-6_8

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.

Throughout the years, several extensions of Rebeca have been introduced for the modelling and analysis of systems from specific domains, among which pRebeca [31] for probabilistic systems, Timed Rebeca [26] for real-time systems, PTRebeca [15] for probabilistic timed systems, and Hybrid Rebeca [17] for cyber-physical systems. In particular, Timed Rebeca extends core Rebeca with a global notion of time [1,20,23,26], 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 [19], 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. Like other Eclipse-based plugins, the Afra interface is divided into a project browser, a model and property editor, and a view for model-checking results. Afra operates on the full space of global states (including local states and time) and transitions according to three types of possible actions (including taking a message, executing a message, and progressing time).

Besides Afra, there is a rich ecosystem of tools around Rebeca [24],[1] including generators for back-end model checkers such as SMV [29], mCRL2 [14], and McErlang [11], a dedicated model checker Modere [16], and a more recent Jacco model checker for Java actors [32] based on Rebeca's existing toolset.

As a result of this proliferation of extensions and tools for Rebeca, those who encounter Rebeca for the first time—either as a student, as a researcher, or as a practitioner from industry—need to grasp a number of subtleties of Rebeca's semantics, possibly including intricacies of probabilities and time.

We recently developed the Caos Scala framework (*Computer aided design of structural operational semantics*) [22], which can help ease these first contacts to Rebeca. 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 [8–10,18] and reactive systems [6,7,30], 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

[1] https://rebeca-lang.org/tools.
[2] Many examples are listed here: https://github.com/arcalab/caos.

which extent, students should be exposed to software tools when being taught formal methods" [12, Section 6: Formal Methods in Education]. Moreover, tools "allow students to quickly link theory with practice" [5].

Contribution. We present a user-friendly web-based front-end for Rebeca based on Caos, which we call RebeCaos, to animate different operational semantics of (Timed) Rebeca and take away a little chaos from those who are new to the Rebeca theory and its supporting tools. More generally, this may facilitate further dissemination and awareness of Rebeca, provide insights into the differences among existing semantics, and support quick experimentation of new variants (e.g., when the order of received messages is preserved). RebeCaos is illustrated by means of a ticket service use case from the literature.

Outline. After this Introduction, Sect. 2 provides the syntax and semantics of core Rebeca, extended with time and dynamism, followed by the introduction of RebeCaos, a novel tool animating Rebeca, in Sect. 3. Before a dedication to Marjan Sirjani, we conclude the paper and present ideas for future work.

2 Rebeca: Syntax and Semantics

This section provides a quick introduction to Rebeca's syntax and semantics, starting with a simple toy example, presented in Fig. 1, borrowed from Hojjat et al. [14],[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).

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

Fig. 1. Simple toy example borrowed from Hojjat et al. [14].

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

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 will be precisely described below.

2.1 Syntax

We provide the abstract syntax of Rebeca from [20] below, writing $\langle \ldots \rangle$ for (meta) parenthesis, superscript $*$ for (zero or more) repetitions, $\langle \ldots \rangle^*$ for (zero or more) repetitions separated by commas, $\langle \ldots \rangle^+$ for (one or more) repetitions separated by commas, $[\ldots]$ for indicating that the text within the brackets is optional, identifiers *className*, *rebecName*, *methodName*, *v*, *literal*, *delay*, and *type* for denoting class name, rebec name, method name, variable, integer number, delay method, and type, respectively, *e* for denoting an (arithmetic, Boolean, or nondeterministic choice) expression, and parameter t for an expression with natural number result. The special variables *self* holds a reference to the current rebec, and *sender* holds a reference to the rebec that called the ongoing method.

$$
\begin{aligned}
\mathit{Model} &::= \mathit{Class}^*\ \ \mathit{Main}\\
\mathit{Main} &::= \textbf{main}\ \{\ \mathit{InstanceDcl}^*\ \}\\
\mathit{InstanceDcl} &::= \mathit{className}\ \mathit{rebecName}(\langle \mathit{rebecName} \rangle^*) : (\langle \mathit{literal} \rangle^*);\\
\mathit{Class} &::= \textbf{reactiveclass}\ \mathit{className}\ \{\ \mathit{KnownRebecs}\ \ \mathit{Vars}\ \mathit{MsgSrv}^*\ \}\\
\mathit{KnownRebecs} &::= \textbf{knownrebecs}\ \{\ \mathit{VarDcl}^*\ \}\\
\mathit{Vars} &::= \textbf{statevars}\ \{\ \mathit{VarDcl}^*\ \}\\
\mathit{VarDcl} &::= \mathit{type}\ \langle v \rangle^+;\\
\mathit{MsgSrv} &::= \textbf{msgsrv}\ \mathit{methodName}(\langle \mathit{type}\ v \rangle^*)\ \{\ \mathit{Stmt}^*\ \}\\
\mathit{Stmt} &::= v = e;\ |\ v = ?(e, \langle e \rangle^+);\ |\ \mathit{Call};\ |\ \textbf{if}\ (e)\ \{\ \mathit{Stmt}^*\ \}\ [\textbf{else}\ \{\ \mathit{Stmt}^*\ \}]\\
&\quad |\ \ \textbf{delay}(t);\ |\ v = \textbf{new}\ \mathit{className}(\langle \mathit{rebecName} \rangle^*) : (\langle \mathit{literal} \rangle^*);\\
\mathit{Call} &::= \mathit{rebecName}.\mathit{methodName}(\langle e \rangle^*)\ [\textbf{after}(t)]\ [\textbf{deadline}(t)]
\end{aligned}
$$

Timed Extension. The syntactic constructs that address time in the syntax above are marked with a light green background. These include a **delay**(t), which is a statement that causes its rebec to wait t time before proceeding; an **after**(t), whose value shows how long it takes for a message to be delivered to its receiver; and a **deadline**(t), which provides the timeout for a message, i.e., stating that it will remain valid for t time. Furthermore, a special variable *now* holds the local time value of each rebec, incremented in the semantic rules below. An example of a system with time will be presented later in Fig. 3, borrowed from Khamespanah et al. [20], describing a a Ticket service.

Dynamic Extension. At execution time, new rebecs can be created with the `new` keyword, similar to object-oriented languages such as Java. This part is marked in the syntax above with a darker blue background. When using this extension, each rebec can know both a static set of rebecs, declared in the `knownrebecs` block, and a dynamic set of rebecs, used and modified like other regular variables. For instance, "`x = new Example (ex):(); x.add(3);`" creates a new rebec of the `Example` class and calls its `add` method. We do not include a larger example in this paper, but some examples can be found in RebeCaos.

2.2 Semantics

We follow mainly the semantics of Timed Rebeca programs presented by Reynosson et al. [23], also used by others [1,20]. This semantics uses both (1) an operational semantics to specify how the configuration of a system evolves when a pending message is processed, and (2) a natural semantics to evaluate a block of statements in a method. We start by describing the former, defining a configuration in a running system, followed by the evaluation of a given sequence of statements with both the timed and the dynamic extension.

Operational Semantics for the Scheduler. Assume a fixed set of rebec names R and let $r, s \in R$ range over these names. A configuration of a system is a pair $\langle Env, B \rangle$ where Env is a set of Rebeca states σ_r for each $r \in R$, and B is a multiset of pending messages. A *state* $\sigma_r \in Env$ of a rebec r is a mapping from variables to values, including the special variables *now* (representing the local time of a rebec) and *sender* (representing the rebec that called the ongoing method). A *message* in B is a tuple $\langle r, m(\overline{v}), s, TT, DL \rangle$ representing a pending call to a method m with arguments $\overline{v}$, called by a rebec s to a rebec r, sent at time TT and expiring at time DL. The SOS rule of Timed Rebeca is formalised below, specifying a scheduling of messages, i.e., the selection of the next pending message and consequently the next rebec to run:

$$[\text{SCHED}]\ \frac{(\sigma_r(m), \sigma_r[now = max(TT, \sigma_r(now))], [\overline{arg} = \overline{v}], sender = s], Env, B) \Downarrow (\sigma'_r, Env', B')}{\left\langle \begin{array}{c} \{\sigma_r\} \uplus Env, \\ \{(r, m(\overline{v}), s, TT, DL)\} \uplus B \end{array} \right\rangle \xrightarrow{[s]\ r.m(\overline{v})\ @\ TT..DL} \left\langle \begin{array}{c} \sigma'_r \uplus Env', \\ B' \end{array} \right\rangle} \; Cnd$$

where condition $Cnd = TT \leq min(B) \ \wedge \ \sigma_r(now) \leq DL$ and $\uplus$ denotes the disjoint union. The function $\Downarrow$ evaluates a sequence of statements, described below. The label of the transition $[s]\ r.m(\overline{v})\ @\ TT..DL$ is used in our prototype implementation but not included in the reference publications [1,20,23]. It states that rebec s called method $m(\overline{v})$ of rebec r, and the message must be processed between time TT and DL.

The initial configuration $\langle Env, B \rangle$ is given by the `main` block: for each instance declaration $C\ r(\overline{k}) : (\overline{v})$, the environment Env includes a state $\sigma_r = \{now \mapsto 0, self \mapsto r\} \cup \{ r_i \mapsto k_i \mid r_i \in C.\texttt{knownrebecs}, k_i \in \overline{k} \} \cup \{ x_i \mapsto v_i \mid x_i \in C.\texttt{statevars},$

$v_i \in \overline{v}\,\}$, and the messages B include $(\text{-}, \mathit{initial}(v), r, 0, +\infty)$. These *initial* methods are given higher priority within each rebec, which we leave out of the SCHED rule for simplicity, as is done in the above mentioned reference publications.

Natural Semantics for Statements. The effect of Timed Rebeca statements is given by the function $\Downarrow$, formalised below. This function receives a tuple (st, σ, Env, B) with a sequence of statements st, an initial state σ, the states of the neighbours Env, and the initial set of pending messages B. It returns a triple (σ', Env', B') with an updated state σ', environment Env', and messages B'. Here *eval* is a function that evaluates expressions in a given environment as expected, the value of special variable *self* is the name of the rebec, and σ is assumed not to be contained in *Env*.

$$[\text{SEND}]\ (x.m(\overline{v})\ \texttt{after}(d)\ \texttt{deadline}(DL), \sigma, Env, B)\ \Downarrow\ (\sigma, Env, \{(\sigma(x), m(\mathit{eval}(\overline{v}, \sigma)), \sigma(\mathit{self}), \sigma(\mathit{now}) + d, \sigma(\mathit{now}) + DL)\} \cup B)$$

$$[\text{DELAY}]\ (\texttt{delay}(d), \sigma, Env, B)\ \Downarrow\ (\sigma[\mathit{now} = \sigma(\mathit{now}) + d], Env, B)$$

$$[\text{ASSIGN}]\ (v = e, \sigma, Env, B)\ \Downarrow\ (\sigma[v = \mathit{eval}(e, \sigma)], Env, B)$$

$$[\text{COND1}]\ \frac{\mathit{eval}(e, \sigma) = \mathit{true} \quad (S_1, \sigma, Env, B)\ \Downarrow\ (\sigma', Env', B')}{(\texttt{if}\ (e)\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \sigma, Env, B)\ \Downarrow\ (\sigma', Env', B')}$$

$$[\text{COND2}]\ \frac{\mathit{eval}(e, \sigma) = \mathit{false} \quad (S_2, \sigma, Env, B)\ \Downarrow\ (\sigma', Env', B')}{(\texttt{if}\ (e)\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \sigma, Env, B)\ \Downarrow\ (\sigma', Env', B')}$$

$$[\text{SEQ}]\ \frac{(S_1, \sigma, Env, B)\ \Downarrow\ (\sigma', Env', B'),\ (S_2, \sigma', Env', B')\ \Downarrow\ (\sigma'', Env'', B'')}{(S_1; S_2, \sigma, Env, B)\ \Downarrow\ (\sigma'', Env'', B'')}$$

$$[\text{CREATE}]\ \frac{r \text{ is a fresh rebec name}}{\begin{array}{c}(v = \texttt{new}\ C(\overline{x}), \sigma, Env, B)\ \Downarrow \\ (\sigma[v = r],\ \sigma_r[\mathit{now} = \sigma(\mathit{now}),\ \mathit{self} = r] \cup Env, \\ (r, \mathit{initial}(\mathit{eval}(\overline{x}, \sigma)), \sigma(\mathit{self}), \sigma(\mathit{now}), +\infty) \cup B)\end{array}}$$

3 RebeCaos by Example

This section describes RebeCaos, a pedagogical tool that animates the semantics of Rebeca as just presented in Sect. 2.2. RebeCaos is implemented in Scala, compiled into JavaScript, and it uses the Caos library [22] to generate the web-based front-end and to present the full state space exploration. The resulting 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 two examples: a dynamic variation of the simple toy example, listed in Fig. 1, and a timed example of a ticket service, borrowed from Reynisson et al. [23].

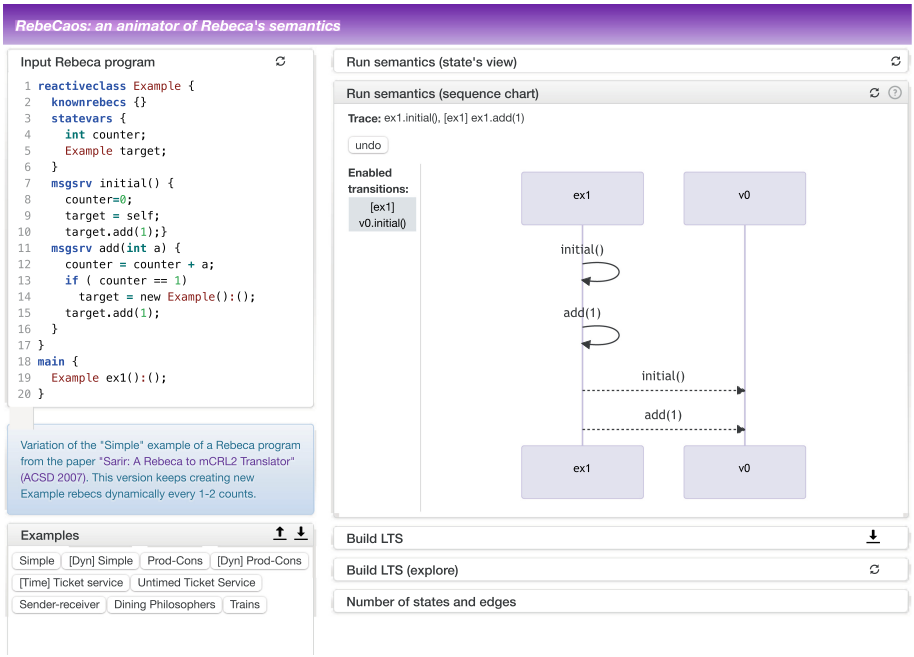

Fig. 2. Screenshot of the interface of RebeCaos.

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. 2. 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 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 the screenshot, we provide the above mentioned variation of the simple toy example in the widget Input Rebeca Program, as well as an interactive step-by-step execution of its semantics using sequence charts. Other examples, including the ones mentioned throughout the paper, can be loaded from the Examples widget.

3.2 Running Step-by-Step

The widget Run semantics (sequence chart) in Fig. 2 supports a guided step-by-step execution of an input Rebeca program, following closely the semantics presented in 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, i.e., called but not yet processed.

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 Fig. 4 to capture the state space of the Ticket Service program in Fig. 3.[4] This concrete program was used, e.g., by Khamespanah et al. [20], and it produces the infinite state space partially represented in Fig. 4 on the left side.

Our implementation performs a breath-first traversal, stopping after a finite number of steps. By using an adaptation of this program without time references , included in the examples of our tool, the state space becomes finite, depicted in Fig. 4 on the right side. The highlighted node on top corresponds to the highlighted node in the state space of the timed version, since the initial parts of these two graphs are identical. Interestingly, the finite state space on the right side for the untimed version has the same shape as the state space produced by Khamespanah et al. [20] 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.

[4] The widget Build LTS (explore) (cf. Fig. 2) 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.

```
reactiveclass TicketService {
  knownrebecs {
    Agent a;
  }
  statevars {
    int issueDelay;
  }
  msgsrv initial(int d) {
    issueDelay = d;
  }
  msgsrv requestTicket() {
    delay(issueDelay);
    a.ticketIssued (1);
  }
}
```

```
reactiveclass Customer {
  knownrebecs {
    Agent a;
  }
  msgsrv initial() {
    self.try();
  }
  msgsrv try() {
    a.requestTicket();
  }
  msgsrv ticketIssued(byte id) {
    self.try() after(30);
  }
}
```

```
reactiveclass Agent {
  knownrebecs {
    TicketService ts;
    Customer c;
  }
  msgsrv requestTicket() {
    ts.requestTicket()
      deadline (5);
  }
  msgsrv ticketIssued(byte id) {
    c.ticketIssued(id);
  }
}
main {
  Agent a(ts,c):();
  TicketService ts(a):(3);
  Customer c(a):();
}
```

Fig. 3. Ticket Service with time extensions borrowed from Khamespanah et al. [20].

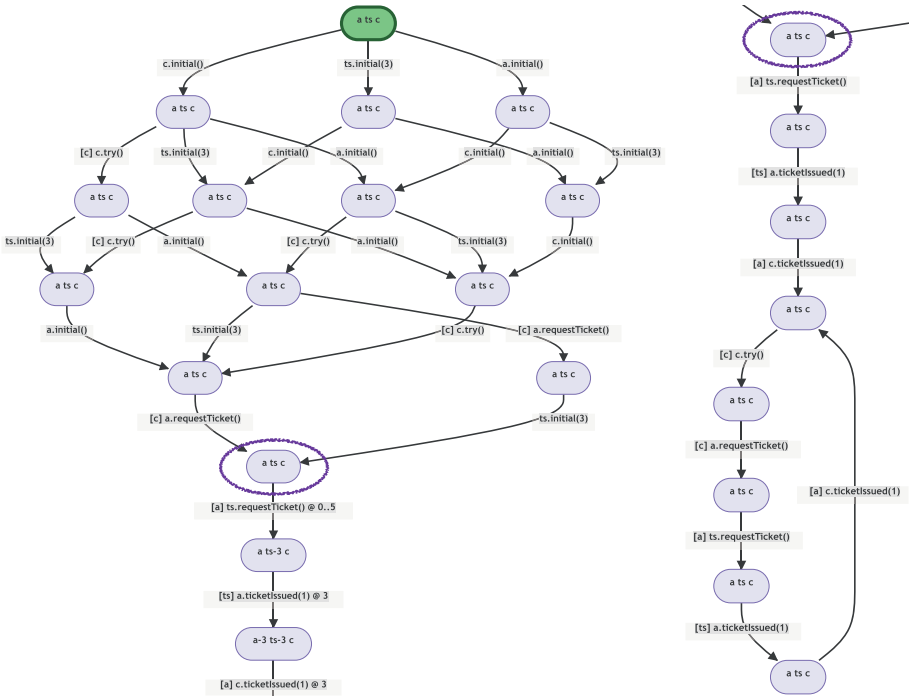

Fig. 4. Full state exploration of the ticket service (left) and an untimed version (right); the former continues infinitely below and the latter shows how this infinite sequence becomes a loop by showing the different behaviour from the highlighted state "a ts c".

3.4 Beyond RebeCaos

Existing tools for Rebeca, such as Afra [19], use an extended version of the syntax presented in Sect. 2.1. These extensions have not yet been implemented in RebeCaos, since most of them are not needed for many of the examples from the literature. These extensions include, e.g., arrays, syntactic macros (with the env keyword), and type casting.

Furthermore, as mentioned in the Introduction, Rebeca has been equipped with a probabilistic semantics (cf., e.g., [15,31]) 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.

We believe that 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 core Rebeca to experiment 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. Performing such experiments with RebeCaos may provide quicker feedback to developers, and help convince others that the new experiments make sense.

4 Conclusion

We presented RebeCaos, a user-friendly web-based front-end tool based on the Caos library for Scala and we illustrated by means of several examples how to animate Rebeca extended with time and dynamism. As future work, we might extend the syntax and semantics of Rebeca currently supported by RebeCaos, e.g., with a means to deal with probabilistic extensions of Rebeca, or with entirely novel extensions such as an experimental type-checking algorithm for verifying conformity with a given behavioural type.

Dedication

This paper is dedicated to Marjan Sirjani, the mother of Rebeca. While so far neither of us has written a paper together with Marjan, our paths have crossed several times.

Maurice recalls her expertise and kind professionalism during a European project review earlier in his career as well as their smooth and pleasant collaboration as PC co-chairs of COORDINATION 2022 and as co-editors of the associated special issue in Logical Methods in Computer Science. Since then many meetings have followed.

José recalls meeting Marjan for the first time at FOCLASA 2008, in Reykjavik, at the beginning of her Icelandic teaching position and his early work as a PhD student. Marjan visited many times his small group in CWI, in Amsterdam, as part of the warm Iranian cluster that brought him many wonderful moments.

As a tribute to Marjan, we decided to reanimate Rebeca. We hope that Marjan finds her child Rebeca as attractive as ever!

Acknowledgements. 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, ISEP/IPP 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., Ingólfsdóttir, 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'11). EPTCS, vol. 58, pp. 1–19 (2011). https://doi.org/10.4204/EPTCS.58.1
2. Agha, G.A.: ACTORS: A Model of Concurrent Computation in Distributed Systems. MIT Press Series in Artificial Intelligence, MIT Press (1986). https://doi.org/10.7551/mitpress/1086.001.0001
3. Agha, G., Mason, I.A., Smith, S., Talcott, C.: Towards a theory of actor computation. In: Cleaveland, W.R. (ed.) CONCUR 1992. LNCS, vol. 630, pp. 565–579. Springer, Heidelberg (1992). https://doi.org/10.1007/BFb0084816
4. Agha, G.A., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. J. Funct. Program. **7**(1), 1–72 (1997). https://doi.org/10.1017/S095679689700261X
5. 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
6. 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
7. 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
8. 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'22). LIPIcs, vol. 222, pp. 27:1–27:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.ECOOP.2022.27
9. 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

10. 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
11. 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
12. 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
13. Hewitt, C.: Description and Theoretical Analysis (Using Schemata) of Planner: A Language for Proving Theorems and Manipulating Models in a Robot. Technical report AITR-258, MIT (1972), http://hdl.handle.net/1721.1/6916
14. 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
15. 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
16. 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
17. Jahandideh, I., Ghassemi, F., Sirjani, M.: Hybrid Rebeca: modeling and analyzing of cyber-physical systems. In: Chamberlain, R., Taha, W., Törngren, M. (eds.) CyPhy/WESE -2018. LNCS, vol. 11615, pp. 3–27. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-23703-5_1
18. 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
19. 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
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). https://doi.org/10.1016/J.SCICO.2014.07.005
21. 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
22. 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

23. Reynisson, A.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
24. 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, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74792-5_5
25. Sirjani, M., Jaghoori, M.M.: Ten years of analyzing actors: Rebeca experience. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 20–56. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24933-4_3
26. Sirjani, M., Khamespanah, E.: On time actors. In: Ábrahám, E., Bonsangue, M., Johnsen, E.B. (eds.) Theory and Practice of Formal Methods. LNCS, vol. 9660, pp. 373–392. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30734-3_25
27. Sirjani, M., Movaghar, A.: An Actor-Based Model for Reactive Systems: Rebeca. Technical report CS-TR-80-01, Sharif University of Technology (2001)
28. Sirjani, M., Movaghar, A., Mousavi, M.R.: Compositional verification of an actor-based model for reactive systems. In: Proceedings of the 1st Workshop on Automated Verification of Critical Systems (AVoCS 2001), pp. 114–118. Oxford University (2001). https://rebeca-lang.org/assets/papers/2001/CompositionalVerificationOfAnObject-BasedModelForReactiveSystems.pdf
29. 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
30. Tinoco, D., Madeira, A., Martins, M.A., Proença, J.: Reactive graphs in action. In: Marmsoler, 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
31. 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
32. 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