



An LAGC Semantics for Timed Rebeca

Reiner Hähnle¹ , Einar Broch Johnsen² , and S. Lizeth Tapia Tarifa² 

¹ Technical University Darmstadt, Darmstadt, Germany

reiner.haehnle@tu-darmstadt.de

² University of Oslo, Oslo, Norway

{einarj, sltarifa}@ifi.uio.no

Abstract. Timed Rebeca is an actor-based language for modeling and analyzing timed reactive systems. Timed Rebeca has a formal SOS-style semantics, as well as one in terms of rewrite rules. While the latter is suitable for model exploration and bounded model checking, it is less so for the purpose of deductive verification. Since we believe there is great potential in deductive verification of Timed Rebeca programs, as a preparatory step, in the present paper we provide a locally abstract, globally concrete (LAGC) semantics. This is a new approach to the semantic foundation of programming languages. An LAGC semantics is a highly modular, incremental trace semantics, particularly suited to ensure soundness of global program analyses such as deductive verification. We provide the first LAGC-style semantics for Timed Rebeca and discuss possible future applications.

1 Introduction

With her work on Rebeca [20,21], Marjan Sirjani has been a driver for the usage of formal methods in a broad range of applications, based on the modeling and analysis of actor systems [1]. Rebeca has been used to analyze a broad range of systems, clearly demonstrating the usefulness of formal methods for real-world problems, and in particular of the actor perspective in modeling and analyzing real-world systems. In this line of work, Sirjani and her colleagues developed and adapted a range of analysis techniques to actor systems, such as simulation [18], model checking [12,15,23], statistical model checking [11], partial order reduction [2], and rewriting logic [19], in particular for timed systems with *Timed Rebeca* [18].

The development of formal analyses techniques for a modeling language relies on the semantics of the language. For timed actor languages, timing constraints introduce particular challenges related to the synchronization of parallel behavior in the distributed actors [3] and the allocation of time-sensitive resources [14]. For Timed Rebeca, significant effort has gone into formalizing its semantics to match different analysis techniques. At present, Timed Rebeca has an SOS-like semantics [18], as well as a formalization in the rewriting language Maude [19]. The latter is well-suited for bounded model checking and model exploration, however, less so for deductive verification and the analysis of global properties such as fairness.

$$\begin{aligned}
M \in Model &::= \overline{CD} \ msc \\
CD \in ClassDecl &::= \mathbf{reactiveclass} \ C \ \{ \overline{KR} \ \overline{SV} \ \overline{Ctr} \ \overline{MS} \} \\
KR \in KnownRebecs &::= \mathbf{knownrebecs} \{ d \} \\
SV \in stateVars &::= \mathbf{statevars} \{ d \} \\
Ctr \in constructor &::= C(d) \{ sc \} \\
MS \in MsgSrv &::= \mathbf{msgsrv} \ m(d) \{ sc \} \\
sc \in Scope &::= d \ s \\
d \in VarDecl &::= \varepsilon \mid T \ x; \ d \\
s \in Stmt &::= \mathbf{for-stm} \mid \mathbf{new-stm} \mid \mathbf{call-stm} \mid \mathbf{delay}(e) \mid s; s \mid \\
&\quad \mathbf{skip} \mid x = e \mid x = ?(\bar{e}) \mid \mathbf{if} \ e \ \{ s \} \\
\mathbf{for-stm} \in For &::= \mathbf{for} \ (s; e; s) \ \{ s \} \\
\mathbf{new-stm} \in New &::= x = \mathbf{new} \ C \ (\bar{e}) : (\bar{e}) \\
\mathbf{call-stm} \in Call &::= e.m(\bar{e}) \ [\mathbf{after}(\bar{e})] \ [\mathbf{deadline}(e)] \\
msc \in mainScope &::= \mathbf{main} \ \{ \overline{InDcl} \} \\
InDcl \in InstanceDcl &::= C \ x(\bar{e}) : (\bar{e})
\end{aligned}$$

Fig. 1: Syntax of Timed Rebeca. Overlined identifiers, such as \bar{e} , indicate lists or sets and square brackets [] indicate optional elements.

Recently, the authors were involved in the design of a denotational semantics, suitable for parallel and distributed systems, named *locally abstract, globally concrete* (LAGC) semantics [7]. In contrast to operational semantics such as SOS [17], which are well-suited to define execution aspects of a model needed for, e.g., simulation or model checking, denotational aspects shift the focus from *how* a model executes to *what* the resulting semantic object will be. LAGC-style semantics was developed to match with calculi for deductive verification [5, 7] and can also be used to analyze fairness in asynchronous languages [8]. In this paper, we provide an LAGC semantics for the asynchronous modeling language Timed Rebeca [18].

Paper overview. Section 2 gives a brief introduction to Timed Rebeca and Section 3 explains the basics of LAGC semantics to make the paper self-contained. Section 4 then presents our LAGC semantics for Timed Rebeca and Section 5 briefly reflects on our effort in developing this semantics, before Section 6 concludes the paper and suggests some lines of future work.

2 Timed Rebeca

Rebeca [21] models the behavior of a system as a set of active objects with encapsulated states, where communication happens via asynchronous message passing. Rebeca is an active object language [4] in the sense that it combines object-oriented features with actors, specifically actors are instances of classes that define how these instances react to messages, and messages are structured like method calls (using dot notation for sending messages). *Timed Rebeca* extends Rebeca with timed constructs.

Various versions of the syntax and semantics of Timed Rebeca [18,19,22] have been studied, for example, with timed semantics based on a global clock [19,22] versus distributed local clocks [18]. In this paper, we consider the syntax and semantics as given in [19].³ We first explain informally the syntax constructs of Timed Rebeca, then illustrate it with an example of a small model.

2.1 Syntax of Timed Rebeca

Figure 1 shows the syntax of Timed Rebeca as given in [19]. A model M consists of a number of *reactive class* declarations, specifying the behavior of the classes of the actors in the model, as well as a *main block* that statically defines the initial instances of the actor classes. An instance of a reactive class $InDcl$ is an actor x of type C in the system model, also called a *rebec*. The meta notation e is for expressions of the appropriate type.

Class declarations. The declaration of a reactive class CD starts with the keyword **reactiveclass**, followed by the reactive class name C . A reactive class declares known rebecs KR and state variables SV . The former are the statically known rebecs, declared with the keyword **knownrebecs** and may be the empty set. The keyword **statevars** declares state variables of a rebec with a basic (non-class) type. These are initialized to a default value and correspond to object fields. They are fully encapsulated and can only be read and set by sending messages to their owner. Each reactive class has one constructor Ctr , used to initialize instances of the class by initializing (some) state variables and possibly sending messages to other rebecs or to itself. The keyword **self** is reserved in the context of the body of a message to refer to the rebec currently executing this message.

Message servers. A rebec responds to an incoming message by executing the corresponding message server, each reactive class declares message signatures \overline{MS} , handled using message servers. Their declaration starts with the keyword **msgsrv**, followed by a name m , the formal parameters, and a message body. The body of a message server sc contains local variable declarations and a sequence of statements with standard control structures, including a **skip** statement, sequential composition,⁴ variable assignment, random variable assignment of the form $x = ?(\bar{e})$, where x is randomly assigned to one of the expressions in \bar{e} , an **if** statement, and a **for** loop of the form **for** ($s_1; e; s_2$) { s_3 }, where counters are initialized in s_1 , the loop guard is in e , the increment of the counters is according to s_2 , and the body of the loop is declared in s_3 .

³ The language constructs also differ slightly between these papers: [18] does not consider dynamic actor creation and the **for** loop, whereas a **now**-statement is featured in [22].

⁴ The Timed Rebeca grammar [19] has no explicit constructors for empty and concatenated statements. Instead, it admits possibly empty sequences of statements s^* . Since our local rules evaluate one statement at a time, it is more natural to base the grammar on an explicit empty statement **skip** and concatenation $s; s'$. Obviously, both grammars result in an equivalent language.

Actor configurations. Rebecs may be created statically in the main block, or dynamically using the **new** statement of the form $x = \mathbf{new} C (\bar{e}_1) : (\bar{e}_2)$, which dynamically creates an instance of class C , where the set of known rebecs is initialized according to \bar{e}_1 , and the constructor of C is immediately executed with the arguments in \bar{e}_2 . A constructor Ctr is very similar to a message server, with the exception that its name is identical to the name of the class.

Communication structure. Each rebec has a separate FIFO message queue containing its pending received messages. The effect of sending a message is appending the message to the FIFO message queue of the receiving rebec. In Rebeca, the execution of a message server is non-preemptive: The executing rebec does not take the next message from its queue before the running message server is finished. The general behavior of each rebec is a loop which first takes a message from the queue and then executes it in the corresponding message server. The actor is idle when there is no enabled message in the FIFO queue. The order of execution of enabled concurrent rebecs in Timed Rebeca is arbitrary.

Time and timed message passing. The **delay** statement models computation time. Timed Rebeca assumes that all statements other than delays are executed instantaneously, and non-zero computation time must be specified via the delay statement of the form **delay**(t), which indicates the current rebec will be blocked (i.e., it is unable to perform any action) within the next t units of time. Rebecs communicate with each other by sending messages of the form $o.m(\bar{e})$, where o specifies the recipient of the message. The execution of the message is non-blocking, meaning that the corresponding message m along with its arguments \bar{e} is put in the receiver’s message queue and the sender continues executing the subsequent statements in the message server. Timed Rebeca’s call-statement may include *timing constraints*:

- The **after**(t) tag may be attached to a message and defines the earliest time that the message can be served as t time units after the time when the message was sent. This means that the receiver can take the message from its queue only after t units of time elapsed.
- The **deadline**(t) tag may be attached to a message and defines the expiration time of the message, the last time the message can be served, as t time units after the time when the message was sent. This means that the message remains at most t units of time in the receiver’s queue, and is ignored⁵ later if its processing has not already started.

A deadline parameter can have the special value **Inf**, representing a deadline that never expires. These **after** and **deadline** tags need to be considered for the enabledness of rebecs. In Timed Rebeca, a rebec is enabled if it is not busy handling a message and its message queue has a message with **after**(t) such that t is less than the time in an **after** tag of any messages of any other actor. Such a message is also called an enabled message. This corresponds to an *implicit* model of global time which will later be made *explicit* in our semantics.

⁵ An implementation may or may not garbage collect expired messages.

2.2 Example: A Thermostat and a Heater in Timed Rebeca

```

1  reactiveclass Thermostat {
2    knownrebecs { Heater heater; }
3    statevars { int period; int temp; }
4    Thermostat(int p, int t) {
5      self.period = p;
6      self.temp = t;
7      self.checkTemp();
8    }
9    msgsrv checkTemp() {
10     if (self.temp >= 30) self.heater.off() deadline(20);
11     if (self.temp <= 25) self.heater.on() deadline(20);
12     self.checkTemp() after(self.period);
13   }
14   msgsrv changeTemp(int delta) { self.temp = self.temp + delta; }
15 }
16 reactiveclass Heater {
17   knownrebecs { Thermostat thermostat; }
18   statevars { boolean on; int delta; }
19   Heater() {
20     self.on = false;
21     self.run();
22   }
23   msgsrv on() { delay(2); self.on = true; }
24   msgsrv off() { delay(2); self.on = false; }
25   msgsrv run() {
26     self.delta = ?(1,2,3);
27     if (self.on == false) self.delta = -1 * self.delta;
28     self.thermostat.changeTemp(self.delta);
29     self.run() after(10);
30   }
31 }
32 main { Thermostat t(h):(5, 25); Heater h(t):(); }

```

Fig. 2: A model of a thermostat and a heating system in Timed Rebeca.

We use a simple example, originally presented in [19], that models a thermostat and a heater. The goal of the system is to keep the temperature between 25 and 30 degrees. The system consists of two objects or *rebecs*, a thermostat and a heater. The thermostat checks the temperature periodically. If the temperature is outside the desired range, the thermostat sends to the heater the appropriate “on” or “off” message. It takes two time units for the heater to turn on or off. The temperature is non-deterministically changed every 10 time units, depending on whether the heater is on or off.

The Timed Rebeca model, shown in Figure 2, consist of two classes `Thermostat` (lines 1–15) and `Heater` (lines 16–31). The main block (line 32) instantiates one

rebec for each reactive class. The main block starts the system with an initial temperature of 25 and specifies that the period of checking the temperature by the thermostat is 5.

A `Thermostat` rebec knows a rebec of type `Heater`, and has two integer state variables `period` and `temp`. The latter models the temperature sensor. Hence, the temperature can be changed simply by setting the `temp` variable. The `Thermostat` constructor initializes the two state variables from the values received as arguments, and sends to itself a `checkTemp` message to initiate its periodic behavior: When `temp >= 30` it sends a message to `heater` to turn it off, when `temp <= 25` it sends a message to `heater` to turn it on, both messages have `deadline(20)`. It finally calls itself again, so the periodic behavior restarts after at least `period` time units, using the tag `after`. Observe that the fields of a rebec (i.e., the state variables and known rebecs) are accessed via a preceding the keyword `self`.

A `Heater` rebec knows a rebec of type `Thermostat`, and has two state variables `on` and `delta`. The constructor initializes the variable `on = false`, assigns the default value 0 to `delta`, and calls itself to `run` its periodic behavior: it randomly selects a value for `delta`, then sends a message to the `Thermostat` with the effect to decrease (increase) the temperature in case the heater is off (on) by `delta`. It finally calls itself again, so the periodic behavior starts again after 10 time units, using the tag `after`. A `Heater` receives messages to turn itself on or off which takes two time units, using the `delay` statement.

3 Basics of LAGC Semantics

We briefly introduce the ideas underlying LAGC semantics. For space reasons, we do not give full technical definitions of LAGC, but introduce essential notions in a compact manner (for a fully precise account, see [7]). Remark that this section may be skipped on first reading and consulted if clarifications are needed; it is included to make the paper more self-contained (furthermore, the text contains some excerpts from [8]).

The main principle of LAGC semantics [7] is to strictly separate two phases: The first evaluates statements to sets of parameterized, *symbolic* local traces. The second instantiates the parameters and composes local traces. Composed traces must respect a well-formedness predicate derived from the semantics of the underlying programming language, without referring to program syntax or intermediate structures. Together, well-formed traces over states and events avoid complex data structures in configurations of reduction rules as it is the case in SOS rules [17]. LAGC configurations contain not only the current state, but the whole trace leading to it, including any events that occurred. This richer structure makes it easy to extract information. The modular separation of progress and composition is also crucial for our presentation. It permits an incremental presentation of LAGC for Timed Rebeca.

3.1 States

To permit *symbolic* expressions (i.e., containing variables) occurring as *values* in program states, we use the star expression $*$ to represent an unknown value that cannot be further evaluated. The $*$ symbol does not occur in programs, only in the semantics. We adopt the notational convention of using capital letters for symbolic variables; but symbolic and non-symbolic variables belong to the same syntactic category and some operations transform a symbolic variable into a non-symbolic one. We keep the syntax and semantics of expressions abstract (as does [19]). In the examples we use standard arithmetic and boolean expressions with their canonical semantics informally.

Definition 1 (Symbolic State, State Update). *A symbolic state σ is a partial mapping $\sigma : Var \rightarrow Sexp$ from variables to symbolic expressions $Sexp = Exp \cup \{*\}$. A symbolic variable is a variable X bound to an unknown value $\sigma(X) = *$. $Sexp$ are expressions that contain symbolic variables. The notation $\sigma[x \mapsto se]$ expresses the update of state σ at x with symbolic expression se .*

In a symbolic state σ , its symbolic variables $\text{symb}(\sigma)$ act as parameters, relative to which a local computation is evaluated. They are used to represent, for example, call parameters that cannot be known locally. We assume there are no dangling references.⁶ States without symbolic variables are called *concrete*. We denote by $\sigma \subseteq \sigma'$ that state σ' extends (as a mapping) state σ .

Example 1. $\sigma = [x \mapsto Y + 42, Y \mapsto *]$ is a symbolic state with $\text{symb}(\sigma) = \{Y\}$.

Timed Rebeca requires a notion of time that extends the languages that were given an LAGC semantics in [7]. For the semantics, this means that evaluation must track the (global) time when time evolves. In the sequel, the parameter N over non-negative integers serves this purpose.

Timed Rebeca classes are instantiated into rebecs o . These are represented by the domain *RId*. In Timed Rebeca, each rebec is an actor with its own processor; therefore, it is necessary to keep track of the object where semantic evaluation takes place. We use the parameter O for this purpose.

We assume an evaluation function $\text{val}_\sigma^{O,N} : Sexp \rightarrow Sexp$ for symbolic expressions se in the context of a state σ , an object O and execution time N , defined as usual for concrete expressions, and defined as $\text{val}_\sigma^{O,N}(X) = X$ for symbolic variables $X \in \text{symb}(\sigma)$. The evaluation of most expressions simply ignores the parameters O, N , with the exception of the **self** reference, where $\text{val}_\sigma^{O,N}(\mathbf{self}) = O$, and fields **self.v**, where $\text{val}_\sigma^{O,N}(\mathbf{self.v}) = \sigma(O.v)$.

It is always possible to evaluate expressions without symbolic variables to values and one could define a set of simplification rules on symbolic expressions, but they are not needed in the context of this article. The evaluation function is trivially extended to sets of expressions.

⁶ A dangling reference is a reference to a variable that is not in the symbolic store.

3.2 Traces and Events

Traces are sequences over states and structured events. For example, the presence of synchronization events makes it possible to express restrictions on call sequences via well-formedness conditions. Symbolic states imply symbolic traces, which motivates to constrain traces by path conditions:

Definition 2 (Path Condition). A path condition pc is a finite set of Boolean expressions. A fully evaluated concrete pc is exactly one of \emptyset , $\{\text{ff}\}$, $\{\text{tt}\}$, $\{\text{ff}, \text{tt}\}$. It is consistent when it does not contain ff . For any concrete state σ , path condition $\text{val}_\sigma^{O,N}(pc)$ is fully evaluated.

Definition 3 (Event Marker, Conditioned Symbolic Trace). An event marker over expressions \bar{e} is a term of the form $ev(\bar{e})$. A symbolic trace τ is defined inductively by the following rules (ε denotes the empty trace):

$$\tau ::= \varepsilon \mid \tau \curvearrowright \nu \qquad \nu ::= \sigma \mid ev(\bar{e}) \quad .$$

A conditioned symbolic trace has the form $pc \triangleright \tau$, where pc is a path condition and τ is a symbolic trace. If pc is consistent, we simply write τ for $pc \triangleright \tau$.

Traces can be finite or infinite. Let $\langle \sigma \rangle$ denote the singleton trace $\varepsilon \curvearrowright \sigma$. Concatenation of two traces τ_1, τ_2 is written as $\tau_1 \cdot \tau_2$ and defined when τ_1 is finite. The final state of a non-empty, finite trace τ is selected with $\text{last}(\tau)$, the first state with $\text{first}(\tau)$, respectively.

Example 2. A conditioned symbolic trace is $\tau = \{Y > 0\} \triangleright \langle \sigma \rangle \curvearrowright \sigma[w \mapsto 17]$, where σ is as in Example 1.

Traces semantically model sequential composition of program statements. Assume τ_r, τ_s are traces of statements r, s , respectively. To obtain the trace corresponding to sequential composition $r; s$, traces τ_r and τ_s must be concatenated, but the first state of the second trace should be identical to (or more precisely an extension of) the final state of the first trace. The chop operator gets rid of the redundant intermediate state.

Definition 4 (Chop on Traces [10, 16, 25]). Let pc_1, pc_2 be path conditions and τ_1, τ_2 be symbolic traces, and assume that τ_1 is a non-empty, finite trace. The semantic chop $(pc_1 \triangleright \tau_1) ** (pc_2 \triangleright \tau_2)$ is defined as follows:

$$(pc_1 \triangleright \tau_1) ** (pc_2 \triangleright \tau_2) = (pc_1 \cup pc_2) \triangleright \tau \cdot \tau_2 \quad \text{where } \tau_1 = \tau \curvearrowright \sigma, \tau_2 = \langle \sigma' \rangle \cdot \tau' \text{ if } \sigma \subseteq \sigma' \quad .$$

Chop is well-defined when the first argument is a finite non-empty trace, we only use it this way.

Events are uniquely associated with that state in a trace, where they occur. Events do not update a state, but may extend it with new symbolic variables. To do so, an event $ev(\bar{e})$ is inserted into a trace after a state σ , which can be extended by fresh symbolic variables \bar{V} , using an event trace $ev_\sigma^{\bar{V}}(\bar{e})$ of length three:

$$ev_\sigma^{\bar{V}}(\bar{e}) = \langle \sigma \rangle \curvearrowright ev(\bar{e}) \curvearrowright \sigma', \text{ where } \sigma' = \sigma[\bar{V} \mapsto *].$$

Given a trace of the form $\tau_1 \curvearrowright \sigma$ and event $ev(\bar{e})$ with fresh symbolic variables \bar{V} , appending the event is achieved by the trace $\tau_1 \cdot ev_{\sigma}^{\bar{V}}(\bar{e})$. Def. 4 ensures that events in traces are joinable: $\tau ** ev_{\sigma}^{\bar{V}}(e)$ is well-defined whenever $last(\tau) = \sigma$. If \bar{V} is empty then the state is unchanged, in this case we omit the set of symbolic variables: $ev_{\sigma}(\bar{e}) = ev_{\sigma}^{\emptyset}(\bar{e})$.

Example 3. To insert event $ev(Z)$, introducing symbolic variable Z , into trace τ from Example 2 at σ we use the event trace $ev_{\sigma}^{\{Z\}}(Z) = \langle \sigma \rangle \curvearrowright ev(Z) \curvearrowright \sigma[Z \mapsto *]$. The result is: $\{Y > 0\} \triangleright \langle \sigma \rangle \curvearrowright ev(Z) \curvearrowright \sigma[Z \mapsto *] \curvearrowright \sigma[Z \mapsto *, w \mapsto 17]$.

Traces are assumed to be well-formed, for example the domains of their states match, variables in events are defined, events $ev_{\sigma}(\bar{e})$ in traces must be aligned with the states before and after, and so on [7].

To enable tracking of the actor where a statement is evaluated, we will tag traces with objects.

Definition 5 (Tagged Trace). Let $ev(\bar{e})$ be an event, τ a trace, and $o \in RId$ an object. A tagged trace τ^o is defined inductively as follows:

$$\begin{aligned} (ev(\bar{e}))^o &= ev^o(\bar{e}) \\ \sigma^o &= \sigma \\ (\tau \curvearrowright \nu)^o &= \tau^o \curvearrowright \nu^o \end{aligned}$$

3.3 Making Traces Concrete

Traces with symbolic variables model program executions relative to an unknown context. The symbolic variables in such traces become instantiated when the execution they represent is scheduled in a concrete context. At this point a symbolic trace is *concretized* by instantiating all of its symbolic variables. This results in a concrete trace with a path condition that is either consistent or not. Technically, we use the notion of a *concretization mapping*. A concretization mapping is defined relative to a state. It associates a concrete value to each symbolic variable of the state.

Definition 6 (State Concretization Mapping). A mapping $\rho : Var \rightarrow Val$ is a concretization mapping for a state σ if $dom(\rho) \cap dom(\sigma) = symb(\sigma)$.

A concretization mapping ρ may also define the value of variables not in the domain of σ . Concretization mappings are canonically extended to events and conditioned traces [7].

Example 4. Consider σ of Example 1 with $symb(\sigma) = \{Y\}$. We define a concretization mapping $\rho = [Y \mapsto 3]$ for σ with $\rho(\sigma) = [x \mapsto 45, Y \mapsto 3]$. Applying ρ to the trace in Example 2, we obtain $\rho(\tau) = \{3 > 0\} \triangleright \langle \rho(\sigma) \rangle \curvearrowright \rho(\sigma)[w \mapsto 17]$. We adopt the convention to strip away consistent path conditions such as here.

3.4 Continuations

The LAGC semantics evaluates one single statement “locally”. Obviously, it is not possible to fully evaluate composite statements in this manner. Therefore, local LAGC rules perform one evaluation step at a time and defer evaluation of the remaining statements, which are put into a *timed continuation*, to be subjected to subsequent rule applications at a later time. Timed continuations extend the continuations of the original LAGC paper [7] to accommodate timed semantics. Syntactically, timed continuations are simply statements s wrapped in the symbol K and tagged by the time when they can be executed: $K^t(s)$. To achieve uniform definitions we allow the case that no further evaluation is required (i.e., the evaluation has been completed) and use the “empty bottle” symbol (\emptyset) for this purpose.

Definition 7 (Timed Continuation Marker). *Let s be a program statement or the symbol \emptyset , and let t_0 be a non-negative integer. Then a timed continuation marker has the form $K^{t_0+N}(s)$, where N is the time parameter.*

4 LAGC Semantics for Timed Rebeca

Local evaluation defines sets of parameterized, symbolic traces that can later be composed into global executions. Local evaluation is defined such that for each statement s and state σ , the result of $\text{val}_\sigma^{O,N}(s)$ is a set of tagged, conditioned, symbolic traces, so-called *continuation traces*, of the form

$$pc \triangleright \tau \cdot K^t(s') \in \mathbf{CTr} ,$$

where τ is a *finite* symbolic trace and s' the remaining statements to be evaluated. Let Θ be the set of traces of s' and ρ any concretization mapping; then the expression $pc \triangleright \tau \cdot K^t(s')$ is used to describe the set of traces: $\{\rho(\tau) ** \tau' \mid \rho(pc) \text{ consistent, } \tau' \in \Theta\}$. In other words, the traces in this set can be consistently instantiated from τ , and extended by executing s' at time t .

4.1 LAGC Semantics of For

We now define the LAGC semantics of **For**, the imperative fragment of Timed Rebeca.

The rule for **skip** generates an empty path condition, returns the state σ it was called in, and produces the empty continuation, resulting in a single trace:

$$\text{val}_\sigma^{O,N}(\mathbf{skip}) = \{\emptyset \triangleright \langle \sigma \rangle \cdot K^N(\emptyset)\} .$$

The assignment rules generate an empty path condition and traces from the current state σ to a state updating σ at x , and produce the empty continuation.

$$\text{val}_\sigma^{O,N}(x = e) = \{\emptyset \triangleright \langle \sigma \rangle \curvearrowright \sigma[x \mapsto \text{val}_\sigma^{O,N}(e)] \cdot K^N(\emptyset)\} .$$

$$\text{val}_\sigma^{O,N}(x =?(e_1, \dots, e_m)) = \bigcup_{1 \leq j \leq m} \text{val}_\sigma^{O,N}(x = e_j) .$$

The conditional statement is a complex statement and cannot be evaluated locally in one step, so we expect it to produce a non-empty continuation. The rule branches on the value of the condition, resulting in two traces with complementary path conditions. The first trace is obtained from the current state and the continuation corresponding to the **if**-branch, and the second trace corresponds to an empty else-branch:

$$\text{val}_\sigma^{O,N}(\mathbf{if}(e) \{ s \}) = \{ \{ \text{val}_\sigma^{O,N}(e) \triangleright \langle \sigma \rangle \cdot \mathbf{K}^N(s), \{ \text{val}_\sigma^{O,N}(!e) \triangleright \langle \sigma \rangle \cdot \mathbf{K}^N(\emptyset) \} \}$$

Timed Rebeca's conditional has an optional **else** branch which we do not model here. It is obvious and not needed in the examples.

The rule for sequential composition $r; s$ is obtained by first evaluating r to traces of the form $pc \triangleright \tau \cdot \mathbf{K}^N(r')$ with continuation r' , and then adding s to r' :

$$\text{val}_\sigma^{O,N}(r; s) = \{ pc \triangleright \tau \cdot \mathbf{K}^t(r'; s) \mid pc \triangleright \tau \cdot \mathbf{K}^t(r') \in \text{val}_\sigma^{O,N}(r) \} .$$

If r' is the empty continuation \emptyset , it must be ignored. To achieve this the rewrite rule “ $\emptyset; s \rightsquigarrow s$ ” is exhaustively applied to statements inside continuations.

Example 5. We start evaluation of the statement $s_{seq} = (x := 1; y := x + 1)$ in an arbitrary symbolic state σ . The rule for sequential composition yields $\text{val}_\sigma^{O,N}(s_{seq}) = \{ \emptyset \triangleright \langle \sigma \rangle \curvearrowright \sigma[x \mapsto 1] \cdot \mathbf{K}^N(y := x + 1) \}$. It uses the result of evaluating the first assignment in the context of σ : $\text{val}_\sigma^{O,N}(x := 1) = \{ \emptyset \triangleright \langle \sigma \rangle \curvearrowright \sigma[x \mapsto 1] \cdot \mathbf{K}^N(\emptyset) \}$.

We use the semantics of **for** to illustrate that the semantics of a statement can be expressed in terms of the semantics of other statements (here **if** and sequence), without having to expose intermediate states:

$$\begin{aligned} \text{val}_\sigma^{O,N}(\mathbf{for}(s_1; e; s_2) \{ s_3 \}) \\ = \text{val}_\sigma^{O,N}(s_1; \mathbf{if}(e) \{ s_3; s_2; \mathbf{for}(\mathbf{skip}; e; s_2) \{ s_3 \} \}) . \end{aligned}$$

Note that this definition is not circular because the evaluation of **if** puts the **for** statement inside a continuation. In practice, loops are rarely used in actor languages such as Timed Rebeca.

4.2 Event Structure

Events in LAGC traces record behavioral aspects that cannot be recovered from a sequence of states alone. In Timed Rebeca, this concerns the creation of rebecs (actors), the invocation and scheduling of messages, and the evolution of time.

Definition 8. *The following events are used in the LAGC semantics for actor and timed constructs. All events but the last are tagged by the object o on which the event is observed.*

- $newEv^o(\text{createdRebec}, \text{knownRebecs}, \text{constrParams})$ observes the creation of a new rebec createdRebec with known rebecs knownRebecs and constructor parameters constrParams .
- $invEv^o(\text{callParams}, \text{callee}, \text{msgName}, \text{callId}, \text{after}, \text{deadline})$ observes the invocation of a message msgName from caller o to callee with call identifier callId and arguments callParams . The message cannot be called before time after is reached and it expires after time deadline . The call identifier is chosen in a way such that it uniquely identifies a message call.
- $invREv^o(\text{callParams}, \text{msgName}, \text{callId})$ observes that a called message can be scheduled for execution and stands for “invocation reaction event” [6]. The parameters are a subset of those for invocation events.
- $delayEv^o(\text{time})$ observes that the currently executed message on rebec o takes non-zero time to execute and it can continue at the earliest at time .
- $timeEv(\text{time})$ observes an advance of the global system clock to time . Since the clock does not belong to any specific actor, it is not tagged.

Since a call identifier uniquely determines a message call, one can ask why the message name and call arguments are needed as parameters of invocation reaction events. This allows the evaluation of all statements the LAGC semantics to be completely *local*. Rule (1) below needs not only to guess the call identifier, but also the matching call (i.e., the message name and arguments).

To enhance readability, event parameters in the previous definition have mnemonic names. To save space in the semantic rules we use o, o', \dots for rebecs, e, e', \bar{e}, \dots for (sequences of) expressions, v, v', \bar{v}, \dots for (sequences of) concrete semantic values, m for message names, i for call identifiers, and t, t', \dots for time points.

4.3 LAGC Semantics of Dynamic Object Creation

The local evaluation rule for dynamic object creation emits a *new* event and extends the given state σ with the variables declared in the object’s class C .

$$\begin{aligned} \text{val}_\sigma^{O,N}(x = \mathbf{new} C(\bar{e}_1) : (\bar{e}_2)) = \\ \{ \emptyset \triangleright newEv_\sigma^{\{X\}}(X, \bar{v}_1, \bar{v}_2) \} \curvearrowright \sigma[x \mapsto X, X \mapsto *, \overline{X.kr} \mapsto v_1] \cdot K^N(X.C(\bar{v}_2)) \mid \\ X \in RId, \text{class}(X) = C, kr \in \text{knownRebecs}(C), X \notin \text{dom}(\sigma), \\ \bar{v}_1 = \text{val}_\sigma^{O,N}(\bar{e}_1), \bar{v}_2 = \text{val}_\sigma^{O,N}(\bar{e}_2) \} . \end{aligned}$$

The rule creates a fresh symbolic variable X to represent the unknown object identity that is returned. The name of the new object is simply guessed; the well-formedness predicate in the composition rules will later check that the name is indeed fresh (see Section 4.7). A *new actor* creation event is issued that records the new object, its known rebec parameters, and its initial value parameters. Next, the current state σ is extended with the as yet unknown object X as well as with the known rebec arguments. The continuation of this rule is a call to the constructor function of class C with the values of parameters \bar{e}_2 . The path condition is empty.

In the creation of new actors, care must be taken to avoid name clashes between the fields of different actors. Concretely, this problem can be solved by systematically prefixing fields by the symbolic identifier X of the new actor, to be concretized later. Here, the names of the known rebec declarations are prefixed by the symbolic actor identifier and mapped to the actual parameter values \bar{e}_1 . We assume that fields are similarly prefixed by the object identifier when evaluating the constructor function and that field accesses are always prefixed by **self** (for example, **self.v**).

4.4 LAGC Semantics for the Timed Constructs

We now define evaluation rules for the timed constructs of Timed Rebeca. The rule for delay extends the trace with a *delay* event that has as parameter a time $t = N + \text{val}_\sigma^{O,N}(e)$, the absolute time relative to the current global time N , at which the currently executing message can continue. The continuation is empty.

$$\text{val}_\sigma^{O,N}(\mathbf{delay}(e)) = \{\emptyset \triangleright \text{delayEv}_\sigma(t) \cdot \mathbf{K}^t(\emptyset) \mid t = N + \text{val}_\sigma^{O,N}(e)\} .$$

The rule for asynchronous message passing of m to a receiver e_1 with arguments \bar{e} , as well as time constraints **after**(e_2) and **deadline**(e_3), extends the trace with an *invocation* event $\text{invEv}_\sigma(\text{val}_\sigma^{O,N}(\bar{e}), \text{val}_\sigma^{O,N}(e_1), m, i, t_1, t_2)$, then ends with the empty continuation. It has an empty path condition. This corresponds to a *non-blocking* semantics of message calls: The code following the message call can continue executing. Observe that the event includes absolute time point t_1 relative to the current global time N when the message that is appended to the recipient $\text{val}_\sigma^{O,N}(e_1)$ can be earliest scheduled, as well as the absolute expiration time t_2 after which the message cannot longer be scheduled. Messages have a unique identifier i which is guessed during trace composition. The well-formedness rules will ensure that i is unique.

$$\begin{aligned} & \text{val}_\sigma^{O,N}(e_1.m(\bar{e}) \mathbf{after}(e_2) \mathbf{deadline}(e_3)) \\ &= \{\emptyset \triangleright \text{invEv}_\sigma(\text{val}_\sigma^{O,N}(\bar{e}), \text{val}_\sigma^{O,N}(e_1), m, i, t_1, t_2) \cdot \mathbf{K}^N(\emptyset) \\ & \quad \mid t_1 = N + \text{val}_\sigma^{O,N}(e_2), t_2 = N + \text{val}_\sigma^{O,N}(e_3), i \in \text{Mid}\} . \end{aligned}$$

The time constraints **after** and **deadline** are optional. When they are not present, they are implicitly added as follows:

$$\begin{aligned} \text{val}_\sigma^{O,N}(e.m(\bar{e})) &= \text{val}_\sigma^{O,N}(e.m(\bar{e}) \mathbf{after}(0) \mathbf{deadline}(\mathbf{Inf})) \\ \text{val}_\sigma^{O,N}(e_1.m(\bar{e}) \mathbf{after}(e_2)) &= \text{val}_\sigma^{O,N}(e_1.m(\bar{e}) \mathbf{after}(e_2) \mathbf{deadline}(\mathbf{Inf})) \\ \text{val}_\sigma^{O,N}(e_1.m(\bar{e}) \mathbf{deadline}(e_2)) &= \text{val}_\sigma^{O,N}(e_1.m(\bar{e}) \mathbf{after}(0) \mathbf{deadline}(e_2)) \end{aligned}$$

The time model of Timed Rebeca assumes a global clock which has always a positive integer value starting at 0. Observe that when calculating the **after** value t_1 of an invocation event as $N + 0$, this means that the message can potentially be scheduled immediately in the receiver's (FIFO) queue. Similarly, when calculating the **deadline** value t_2 of an invocation event as $N + \mathbf{Inf}$, it will evaluate to **Inf** which is greater than any value of N . This means that a message with **deadline**(**Inf**) never expires.

4.5 Message Servers and Classes

The semantics of a message server is a trace that starts with an invocation reaction event corresponding to a previous invocation event. The values of the formal parameters cannot be known locally, so we introduce symbolic variables for them. These are instantiated during trace composition such that the invocation reaction event matches a previous invocation event. At this time also the correct call identifier is guessed.

$$\begin{aligned} \text{val}_\sigma^{O,N}(\mathbf{msgsrv} \ m(\overline{T} \ \overline{x}) \ \{sc\}) = \\ \{ \emptyset \triangleright \text{invREv}_\sigma^{\overline{Z}}(\overline{Z}, m, i) \curvearrowright \sigma[\overline{z} \mapsto \overline{Z}, \overline{Z} \mapsto *, \overline{z}' \mapsto d_{\overline{x}'}] \cdot \mathbf{K}^N(sc[\overline{x}, \overline{x}' \leftarrow \overline{z}, \overline{z}']) \} \quad (1) \\ | \overline{z}, \overline{Z} \notin \text{dom}(\sigma), i \in \mathbf{MId} \} . \end{aligned}$$

There is a subtlety in Timed Rebeca concerning the formal parameters of messages: Not all state variables need to occur among them, some may be initialized with default values. Let \overline{x}' be the state variables in m 's class that do not occur in \overline{x} . To avoid name clashes in formal parameters $\overline{x}, \overline{x}'$ of different message executions, these are renamed to fresh names $\overline{z}, \overline{z}'$, where the \overline{z} are given symbolic values \overline{Z} and the \overline{z}' default values in the new state $\sigma[\overline{z} \mapsto \overline{Z}, \overline{Z} \mapsto *, \overline{z}' \mapsto d_{\overline{x}'}]$. We repeat the renaming trick for local variables in the following rule:

$$\text{val}_\sigma^{O,N}(T \ x; d \ s) = \{ \emptyset \triangleright \langle \sigma \rangle \curvearrowright \sigma[x' \mapsto 0] \cdot \mathbf{K}^N(\{ d \ s[x \leftarrow x'] \}) \mid x' \notin \text{dom}(\sigma) \} .$$

Although we elide the exact expression syntax and its semantics, it may be useful to remind the reader of how to evaluate the self reference **self**:

$$\text{val}_\sigma^{O,N}(\mathbf{self}) = O .$$

4.6 The Trace Composition Rules

We observed that local evaluation returns an empty continuation \emptyset when the execution is complete. Therefore, in the composition rules below, the empty continuation may occur as an input and needs to be locally evaluated. The evaluation of the empty continuation yields the empty set of traces:

$$\text{val}_\sigma^{O,N}(\emptyset) = \{ \} .$$

Local traces are instantiated and composed into concrete global ones. Since the code in the body of a message server is non-blocking, sequential and deterministic, there is exactly one trace, provided that the execution starts in a concrete state that assigns values to all the variables of a program [7]. Consequently, no scheduler needs to be defined for the execution of a message, only for *when* messages are executed. However, in general different actors process messages simultaneously at any given time. This is reflected in the semantic configurations.

Definition 9 (Configuration). *An LAGC configuration for semantic evaluation is a pair sh, Σ , where sh is a concrete, tagged trace and Σ is a partial mapping from rebe identifiers in \mathbf{RId} to timed continuations of the form $\mathbf{K}^N(s)$.*

The task of the composition rule for message servers is to evaluate statements in a concrete state until the next continuation, then stitch the resulting concrete traces together. Given a configuration with *concrete* trace sh having final state σ and a continuation $\Sigma(o) = K^N(s)$, we can evaluate s starting in σ . The auxiliary function $\text{now}(sh)$ returns the current global time at the end of trace sh , which we need to schedule and evaluate the continuation $K^N(s)$. The result is a set of conditioned traces from which one trace with a consistent path condition and a trailing continuation $K^N(s')$ is chosen for the execution on o (via tagging the resulting concrete trace $\rho(\tau)$).

Rule (2) models *progress* in the execution of an activated message. A subtle point is that the rule is not applicable when $s = \emptyset$, because then, by definition, the evaluated trace set is empty.

$$\frac{\begin{array}{l} \Sigma(o) = K^n(s) \quad \sigma = \text{last}(sh) \quad n = \text{now}(sh) \quad pc \triangleright \tau \cdot K^T(s') \in \text{val}_\sigma^{o,n}(s) \\ s \neq \emptyset \quad \rho \text{ concretizes } \tau, T \quad \rho(pc) \text{ consistent} \quad \text{wf}(sh ** \rho(\tau)^o) \end{array}}{sh, \Sigma \rightarrow sh ** \rho(\tau)^o, \Sigma[o \mapsto K^{\rho(T)}(s')]} \quad (2)$$

Rule (3) models the *activation* of a new message on o . This is only possible, when these conditions are fulfilled: (i) the actor o is idle, i.e. $\Sigma(o) = K^t(\emptyset)$ (or it has not yet executed anything); (ii) since the semantics is denotational, there is an invocation event corresponding to the activation in sh ; and (iii) the time constraints given at invocation time must be satisfied. The rule “guesses” values for an invocation reaction event fulfilling these conditions and the well-formedness of global traces (Section 4.7) will ensure that they hold. In this semantic model, actors are only created once they start processing the first message.

$$\frac{\begin{array}{l} n = \text{now}(sh) \quad (\Sigma(o) = K^t(\emptyset) \text{ or } o \notin \text{dom}(\Sigma)) \quad \sigma = \text{last}(sh) \\ \text{lookup}(m, \mathcal{G}) = m(\bar{x}) \ sc \quad pc \triangleright \tau \cdot K^n(s) \in \text{val}_\sigma^{o,n}(m(\bar{x}) \ sc) \\ \rho \text{ concretizes } \tau \quad \rho(pc) \text{ consistent} \quad \text{wf}(sh ** \rho(\tau)^o) \end{array}}{sh, \Sigma \rightarrow sh ** \rho(\tau)^o, \Sigma[o \mapsto K^n(s)]} \quad (3)$$

Rule (4) models *time advance*. In this rule, $\text{pending}(sh, o)$ is an auxiliary function over traces sh (defined in Section 4.7) that expresses that there is a pending message ready to be executed on rebec o . Thus, the premises of the rule express two constraints: (i) there is no non-empty enabled continuation for o at the current time n , and (ii) there is no executable message at current time n for any o . Rule (4) can then be expressed as follows:

$$\frac{\begin{array}{l} \forall o. (\Sigma(o) = K^t(s) \wedge s \neq \emptyset \implies n < t) \\ n = \text{now}(sh) \quad \sigma = \text{last}(sh) \quad \forall o. |\text{pending}(sh, o)| = 0 \end{array}}{sh, \Sigma \rightarrow sh ** \text{timeEv}_\sigma(n+1), \Sigma} \quad (4)$$

4.7 Well-formedness

We use the well-formedness predicate $\text{wf}(sh)$ on concrete, tagged traces to ensure that only traces conforming to the Timed Rebeca semantics can be produced

by the trace composition rules (2)–(4). The relevant information is contained in the various events emitted during local evaluation, therefore, we define well-formedness inductively over the final event in a trace. The correct evolution of states and delay events is ensured by local evaluation, so no restriction is necessary here. The correctness of time events is guaranteed by a separate condition in the premise of rule (4), so we have four trivial cases:

$$\begin{aligned} \text{wf}(\epsilon) &= \text{true} \\ \text{wf}(sh \curvearrowright \sigma) &= \text{wf}(sh) \\ \text{wf}(sh \curvearrowright \text{delayEv}(t)) &= \text{wf}(sh) \\ \text{wf}(sh \curvearrowright \text{timeEv}(t)) &= \text{wf}(sh) \end{aligned}$$

When creating a new rebec on actor o , two conditions must be ensured. First, there cannot be another rebec associated with the name o ; second, the known rebecs passed to the new rebec must be known to the caller o' .

$$\begin{aligned} \text{wf}(sh \curvearrowright \text{newEv}^{o'}(o, \bar{o}, \bar{v})) &= \\ \text{wf}(sh) \wedge \nexists o'', \bar{o}', \bar{v}'. \text{newEv}^{o''}(o, \bar{o}', \bar{v}') \in sh \wedge \bar{o} \subseteq \text{knows}(sh, o') &. \end{aligned}$$

Known rebecs are tracked in the function `knows`. It extracts from the “new” events in a tagged concrete trace the rebecs known by o : Whenever o is created, by definition it knows the known rebecs \bar{o} passed as the second argument of the event (first equation). Whenever o creates a new rebec o' this is added to its known rebecs (second equation). The third equation deals with initialization and is explained in the subsequent section.

$$\begin{aligned} \text{knows}(sh \curvearrowright \text{newEv}^{o'}(o, \bar{o}, \bar{v}), o) &= \bar{o} \\ \text{knows}(sh \curvearrowright \text{newEv}^{o'}(o', \bar{o}, \bar{v}), o) &= \{o'\} \cup \text{knows}(sh, o) \\ \text{knows}(sh_{\text{init}}, \text{main}) &= O_R \\ \text{knows}(sh \curvearrowright \nu, o) &= \text{knows}(sh, o) \text{ otherwise} \end{aligned}$$

Interestingly, the tracking of known rebecs is not modeled in the existing Timed Rebeca semantics [18, 19]. It is a clear advantage of the compositional design of the LAGC semantics that this can be added simply by extending well-formedness by one more predicate.

Turning to message calls, the central property that invocation events need to ensure is that each call identifier is unique in a given trace. In addition, the rebec o on which the message is invoked must exist and be known to the caller o' :

$$\begin{aligned} \text{wf}(sh \curvearrowright \text{invEv}^{o'}(\bar{v}, o, m, i, t_1, t_2)) &= \\ \text{wf}(sh) \wedge o \in \text{knows}(sh, o') \wedge \exists o'', \bar{o}, \bar{v}'. \text{newEv}^{o''}(o, \bar{o}, \bar{v}') \in sh & \\ \wedge \nexists o'', \bar{v}', o''', m', t'_1, t'_2. \text{invEv}^{o''}(\bar{v}', o''', m', i, t'_1, t'_2) \in sh &. \end{aligned}$$

The most complex definition is that for invocation reaction events, because two properties must be ensured: The call with identifier i to be executed has not

already been selected earlier (second conjunct) and the message can actually be scheduled at the current time. The formula in the third conjunct first retrieves an invocation event with matching callee, identifier, message name, and call arguments, then makes sure that the current time is within the specified bounds. Observe that $\text{now}(sh) \leq \mathbf{Inf}$ is always true.

$$\begin{aligned} \text{wf}(sh \curvearrowright \text{invREv}^o(\bar{v}, m, i)) &= \text{wf}(sh) \wedge \nexists o', \bar{v}', m'. \text{invREv}^{o'}(\bar{v}', m', i) \in sh \\ &\wedge \left(\exists o', t_1, t_2. \text{invEv}^{o'}(\bar{v}, o, m, i, t_1, t_2) \in sh \wedge t_1 \leq \text{now}(sh) \leq t_2 \right) . \end{aligned} \quad (5)$$

Remark 1. By inspection of the well-formedness equations, one can see that delay events are not required in the LAGC semantics. However, they are often useful to reconstruct the full timed behavior from a given trace.

To find the current clock time in a given trace, we simply look for the most recent time event and take its argument:

$$\begin{aligned} \text{now}(sh \curvearrowright \text{timeEv}(n)) &= n \\ \text{now}(sh \curvearrowright \nu) &= \text{now}(sh) \text{ otherwise} \end{aligned}$$

It remains to define the pending function used in rule (4). Let us first make precise what we mean by pending:

Definition 10 (Pending Invocation Event). *A tagged concrete trace sh contains a pending invocation event if:*

1. sh contains an invocation event for message m , arguments \bar{v} , call identifier i , and time constraints t_1, t_2 ;
2. sh contains no subsequent invocation reaction event with call identifier i ;
3. such an invocation reaction event could be scheduled now.

The first and third condition correspond to the third conjunct of equation (5), the second condition to its second conjunct. This means we can define the pending function with the help of well-formedness as follows:

$$\text{pending}(sh, o) = \{i \mid \exists m, \bar{v}. \text{wf}(sh \curvearrowright \text{invREv}^o(\bar{v}, m, i))\} . \quad (6)$$

This definition implies that the sh argument is well-formed itself, but since well-formedness of sh is an invariant guaranteed by the trace composition rules, this is no restriction.

Timed Rebeca imposes FIFO order on messages sent to the same actor. As shown in [7, Section 6.2], it is possible to add such constraints to the well-formedness predicate in a compositional manner. With the help of the pending function, the definition becomes succinct. Let the notation $i \prec_{sh} i'$ express that call identifier i appears the first time syntactically before i' in a trace sh . Then we can express FIFO simply by adding to well-formedness of invocation reaction events the constraint that the scheduled event must be the syntactically first event that can be scheduled:

$$\begin{aligned} \text{wf}_{\text{fifo}}(sh \curvearrowright \text{invREv}^o(\bar{v}, m, i)) &= \\ &\text{wf}(sh \curvearrowright \text{invREv}^o(\bar{v}, m, i)) \wedge (i = \min_{\prec_{sh}} \text{pending}(sh, o)) . \end{aligned}$$

Remark 2. This definition of FIFO embodies a subtle semantic choice: Assume now(sh) = 1 and $invEv^{o'}(\bar{v}, o, m, i, 2, 3) \in sh$, $invEv^{o''}(\bar{v}', o, m, i', 1, 2) \in sh$ such that $i \prec_{sh} i'$. Then the call with identifier i' is scheduled, because the call with identifier i is not yet pending. But the call with identifier i' might cause a delay beyond time 3, in which case call i is never scheduled, even though $i \prec_{sh} i'$. We implemented *first in first schedulable out*, i.e., time constraints take precedence over FIFO constraints. Other semantics, where a queue is blocked until the first pending call is available, can be obtained by adjusting the definition of pending.

4.8 Initialization

For a given Timed Rebeca program, let us denote the rebecs declared in the main block by R . We assign a fixed, unique object o_r to each $r \in R$ and define the set of all initial objects to be $O_R = \{o_r \mid r \in R\}$.

The semantic evaluation of a Timed Rebeca program initially sets the global time to 0 and assigns the object o_r to each $r \in R$:

$$sh_{init} := timeEv_{\sigma_\epsilon}(0) \curvearrowright \langle \sigma_\epsilon^{main}[\bar{r} \mapsto o_r] \rangle$$

The third clause of the definition of the knows predicate in Section 4.7 ensures that at this point the *main* object knows about all initial rebecs. The semantic evaluation starts in the initial concrete trace $sh_{init} \curvearrowright newEv_\sigma^{main}(main, O_R, \epsilon)$. Observe that this trace is well-formed.

To define the initial configuration, we transform each instance declaration in the main block of the form $C \ r(\bar{r});(\bar{e})$; into a regular object declaration of the form $r = \mathbf{new} \ C(\bar{r});(\bar{e})$;. This saves us from defining different mechanisms for static and dynamic object creation. Let s_{main} be the sequence of instance declarations in the main block transformed in this way. Now we start semantic evaluation with the configuration

$$sh_{init} \curvearrowright newEv^{main}(main, O_R, \epsilon), [main \mapsto K^0(s_{main})] .$$

For each $r = \mathbf{new} \ C(\bar{r});(\bar{e})$;, according to the semantics of **new**, a new object is created with a fresh abstract identifier X_r which must be concretized and assigned to r . The concretization must match the initial state where r has value o_r . Hence X_r , and therefore r , will be bound correctly to o_r . The semantic rule for **new** also adds $o_r \mapsto K^0(r.C(\bar{e}))$ to the tasks in the configuration, i.e., a call to the constructor C of r 's class which initiates program execution.

Let $sh, \Sigma \xrightarrow{*} sh', \Sigma'$ denote the transitive closure of applying rules (2)–(4), expressing that sh', Σ' can be reached from sh, Σ in zero or more steps. We can now formally define the trace set of a Rebeca model as follows:

Definition 11 (Trace semantics of Rebeca models). *Given a Rebeca model $M = \overline{CD} \ \mathbf{main} \ \{ \overline{InDcl} \}$ with initially known rebecs R ; the traces of M , denoted $\mathbf{Tr}(M)$ is the set of reachable traces from the initial trace of M :*

$$\mathbf{Tr}(M) = \{ sh \mid sh_{init} \curvearrowright newEv^{main}(main, O_R, \epsilon), [main \mapsto K^0(s_{main})] \xrightarrow{*} sh, \Sigma \} .$$

Observe that the trace set given in Definition 11 is prefix closed. On the other hand, executions in our semantics are infinite; in particular, rule (4) can be applied indefinitely when no further tasks (either progress or messages) can be processed. Even finite computations end in an unbounded sequence of time advance events. Hence, it might be useful to define *final* configurations:

Definition 12 (Final Configuration). *A configuration sh, Σ is final when*

- (i) $\nexists o. (\Sigma(o) = K^t(s) \wedge s \neq \emptyset \wedge \text{now}(sh) < t)$ and
- (ii) $\nexists o, t. (\text{now}(sh) < t \wedge |\text{pending}(sh \curvearrowright \text{timeEv}(t), o)| > 0)$

Whether a given configuration is final can be effectively computed, because it suffices to consider t until the maximal **after** constraint in sh . Now we can adapt the trace semantics of Rebeca in Definition 11 to finitely terminating traces by choosing traces in $\mathbf{Tr}(M)$ that reach final configurations.

5 Discussion

This paper has proposed an LAGC style [7] semantics for Timed Rebeca [19]. We here analyze our effort. Typically for LAGC semantics, there is exactly one local evaluation rule for each kind of statement, which characterizes its behavior in a succinct manner, independent of context. Rules without events are completely *modular*; i.e., they can be freely modified or added without affecting the remaining definitions. For example, all rules of the For language in Section 4.1 are either identical to or minor modifications of existing rules [7, Section 3.1]. Events are used in LAGC semantics to characterize non-local behavior. Of the five event types introduced in Section 4.2, three are variations of events in [7, Section 7.2] and one is not strictly required. Also two of the three trace composition rules are variations: the rules for progress (2) and for scheduling message calls (3). Likewise, traces tagged with a parameter O for the caller object as well as the object-to-task mapping Σ are taken from [7].

The novel aspects in the LAGC semantics of Timed Rebeca concern (i) the handling of time, (ii) initialization including the main block, and (iii) keeping track of known rebecs. Of these, the most interesting is the first. The central concepts we needed are *timed local evaluation* and *timed continuations*. We believe it is natural that evaluation and continuations not only express *what* to execute, but also *when*. Timed continuations for an LAGC-style semantics were first introduced by Tapia Tarifa [24] to capture time and time-sensitive resources in Real-Time ABS [3, 14]. Real-Time ABS is a real-time extension of ABS [13], an active object language that extends Timed Rebeca with non-preemptive suspension points via *await* statements and futures, but has an unordered message queue. Timed Rebeca differs from Real-Time ABS by combining message delay and expiry with a FIFO message queue. This combination requires additional care in capturing the enabledness of pending messages, and its interaction with time advance. In this paper, we have considered a discrete time domain for Timed Rebeca; however, a dense time domain can be realized in a straightforward manner, by forcing the time advance rule (4) to a advance to maximum

elapsed time (see [24]). We expect that timed continuations will also be useful in other language settings with timed semantics.

With timed continuations and delay events, the semantic rule for the **delay** statement becomes obvious. The global clock is modeled by time events that are inserted into the current trace by the trace composition rule (4), whenever a program is in a *quiescent* state; thus we opted for a maximal progress semantics, where all pending message calls have been processed before time can advance. Using time events, it is easy to extend invocation (reaction) events to handle delayed message calls, but the advance-time-when-quiescent approach makes executions infinite. For this purpose, we suggest a notion of final configuration (Definition 12) to stop the semantic evaluation.

Handling main blocks uniformly requires the dedicated setup described in Section 4.8, because the Timed Rebeca designers chose to have both static (in the main block) and dynamic (with **new**) rebec creation. This issue was not addressed formally in previous semantics for Timed Rebeca (e.g., [18, 19]). To define the “knows” function and use it to check correct usage of known rebecs is a straightforward addition to the well-formedness rules in our setting. As far as we know, this is the first formal semantics which incorporates the **knownrebecs** mechanism in the language semantics.

To summarize, we needed five event types of which two are new (one of them essential), eleven local rules (one for each kind of statement) of which one is new, three trace composition rules of which one is new, and six well-formedness rules of which one is new and one (FIFO) is formulated differently than in [7]. We stress that all additions are *conservative extensions* of the LAGC framework: None of the fundamental definitions needed to be changed, the well-formedness rules merely add conjuncts for the checks of time and known rebecs.

6 Conclusion and Future Work

This paper contributes a LAGC semantics for Timed Rebeca, which is a highly modular, trace-based denotational semantics. We believe this is the first denotational, trace-based semantics for Timed Rebeca. Additionally, the semantics in this paper addresses certain corner-cases of Timed Rebeca that were left open in previous work [19]: the main block and the semantic representation of **knownrebecs**. The latter is particularly tricky for variations of Timed Rebeca that include dynamic creation of rebecs, which allows dynamic topologies, and deserves a semantic treatment.

In general, denotational semantics are challenging to achieve for concurrent languages because scheduling between different parallel activities goes against compositionality. The framework of LAGC semantics addresses this challenge by means of a locally symbolic, denotational semantics that include events, combined with global composition rules to address synchronization between these events. Timed languages add an additional level of complexity to this synchronization; in this paper we have in particular addressed the issue of how constraints on message passing (i.e., delay and deadline) combine with FIFO mes-

sage ordering and time advance within the LAGC semantic framework. This paper shows how the LAGC semantic framework naturally extends to timed actors in Timed Rebeca.

We believe that the modular nature and succinct formulation of the LAGC semantics for Timed Rebeca makes it easy to understand it and to extend it when further language features are added in the future. The semantics presented in this paper enables the design a calculus for deductive verification [9] for Timed Rebeca programs, following [5, 7], and prove its soundness for the LAGC semantics. Furthermore, the non-deterministic scheduling rule (3) of the proposed semantics could be refined into a set of deterministic rules and show some variation of *weak fairness* for it, following [8]. This requires to define a suitable *timed* version of weak fairness, because messages with a **deadline** do not stay enabled.

References

1. Agha, G.A.: ACTORS - a model of concurrent computation in distributed systems. MIT Press series in artificial intelligence, MIT Press (1990)
2. Bagheri, M., Sirjani, M., Khamespanah, E., Hojjat, H., Movaghar, A.: Partial order reduction for timed actors. In: Bloem, R., Dimitrova, R., Fan, C., Sharygina, N. (eds.) Proc. 13th International Conference on Software Verification (VSTTE 2021). LNCS, vol. 13124, pp. 43–60. Springer (2021), https://doi.org/10.1007/978-3-030-95561-8_4
3. Björk, J., de Boer, F.S., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: User-defined schedulers for real-time concurrent objects. *Innov. Syst. Softw. Eng.* **9**(1), 29–43 (2013), <https://doi.org/10.1007/s11334-012-0184-5>
4. de Boer, F., Din, C.C., Fernandez-Reyes, K., Hähnle, R., Henrio, L., Johnsen, E.B., Khamespanah, E., Rochas, J., Serbanescu, V., Sirjani, M., Yang, A.M.: A survey of active object languages. *ACM Computing Surveys* **50**(5), 76:1–76:39 (Oct 2017), <https://doi.org/10.1145/3122848>
5. Bubel, R., Gurov, D., Hähnle, R., Scaletta, M.: Trace-based deductive verification. In: Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning. EPiC Series in Computing, vol. 94, pp. 73–95 (2023). <https://doi.org/10.29007/vdfd>
6. Din, C.C., Dovland, J., Johnsen, E.B., Owe, O.: Observable behavior of distributed systems: Component reasoning for concurrent objects. *J. Log. Algebraic Methods Program.* **81**(3), 227–256 (2012), <https://doi.org/10.1016/j.jlap.2012.01.003>
7. Din, C.C., Hähnle, R., Henrio, L., Johnsen, E.B., Pun, V.K.L., Tapia Tarifa, S.L.: Locally abstract, globally concrete semantics of concurrent programming languages. *Transactions on Programming Languages and Systems* **46**(1) (2024). <https://doi.org/https://doi.org/10.1145/3648439>
8. Hähnle, R., Henrio, L.: Provably fair cooperative scheduling. *The Art, Science, and Engineering of Programming* **8**(2) (2024), <https://doi.org/10.22152/programming-journal.org/2024/8/6>
9. Hähnle, R., Huisman, M.: Deductive verification: from pen-and-paper proofs to industrial tools. In: Steffen, B., Woeginger, G. (eds.) *Computing and Software Science: State of the Art and Perspectives*, LNCS, vol. 10000, pp. 345–373. Springer, Cham, Switzerland (2019). https://doi.org/10.1007/978-3-319-91908-9_18

10. Halpern, J.Y., Manna, Z., Moszkowski, B.C.: A hardware semantics based on temporal intervals. In: Díaz, J. (ed.) *Automata, Languages and Programming*, 10th Colloquium, Barcelona, Spain. LNCS, vol. 154, pp. 278–291. Springer, Berlin, Heidelberg (1983). <https://doi.org/10.1007/BFb0036915>
11. Jafari, A., Khamespanah, E., Kristinsson, H., Sirjani, M., Magnusson, B.: Statistical model checking of timed rebecca models. *Comput. Lang. Syst. Struct.* **45**, 53–79 (2016), <https://doi.org/10.1016/j.cl.2016.01.004>
12. Jaghoori, M.M., Movaghar, A., Sirjani, M.: Modere: the model-checking engine of Rebeca. In: Haddad, H. (ed.) *Proc. Symposium on Applied Computing (SAC 2006)*. pp. 1810–1815. ACM (2006), <https://doi.org/10.1145/1141277.1141704>
13. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F., Bonsangue, M.M. (eds.) *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011), https://doi.org/10.1007/978-3-642-25271-6_8
14. Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Integrating deployment architectures and resource consumption in timed object-oriented models. *J. Log. Algebraic Methods Program.* **84**(1), 67–91 (2015). <https://doi.org/10.1016/j.jlamp.2014.07.001>
15. Khamespanah, E., Mechitov, K., Sirjani, M., Agha, G.A.: Schedulability analysis of distributed real-time sensor network applications using actor-based model checking. In: Bosnacki, D., Wijs, A. (eds.) *Proc. 23rd International Symposium on Model Checking Software (SPIN 2016)*. LNCS, vol. 9641, pp. 165–181. Springer (2016), https://doi.org/10.1007/978-3-319-32582-8_11
16. Nakata, K., Uustalu, T.: A Hoare logic for the coinductive trace-based big-step semantics of While. *Logic Methods in Computer Science* **11**(1) (2015). [https://doi.org/10.2168/LMCS-11\(1:1\)2015](https://doi.org/10.2168/LMCS-11(1:1)2015)
17. Plotkin, G.D.: A structural approach to operational semantics. *J. Log. Algebr. Program.* **60–61**, 17–139 (2004)
18. Reynisson, A.H., Sirjani, M., Aceto, L., Cimini, M., Jafari, A., Ingólfssdóttir, A., Sigurdarson, S.H.: Modelling and simulation of asynchronous real-time systems using Timed Rebeca. *Science of Computer Programming* **89**, 41–68 (2014). <https://doi.org/10.1016/J.SCICO.2014.01.008>
19. Sabahi-Kaviani, Z., Khosravi, R., Ölveczky, P.C., Khamespanah, E., Sirjani, M.: Formal semantics and efficient analysis of Timed Rebeca in Real-Time Maude. *Science of Computer Programming* **113**, 85–118 (2015). <https://doi.org/10.1016/J.SCICO.2015.07.003>
20. Sirjani, M.: Rebeca: Theory, applications, and tools. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) *Proc. 5th International Symposium on Formal Methods for Components and Objects (FMCO 2006)*. LNCS, vol. 4709, pp. 102–126. Springer (2006), https://doi.org/10.1007/978-3-540-74792-5_5
21. 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 - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*. LNCS, vol. 7000, pp. 20–56. Springer (2011), https://doi.org/10.1007/978-3-642-24933-4_3
22. Sirjani, M., Khamespanah, E.: On time actors. In: Abraham, E., Bonsangue, M.M., Johnsen, E.B. (eds.) *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*. LNCS, vol. 9660, pp. 373–392. Springer (2016). https://doi.org/10.1007/978-3-319-30734-3_25

23. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using Rebeca. *Fundam. Informaticae* **63**(4), 385–410 (2004), <http://content.iospress.com/articles/fundamenta-informaticae/fi63-4-05>
24. Tapia Tarifa, S.L.: Locally abstract globally concrete semantics of time and resource aware active objects. In: Ahrendt, W., Beckert, B., Bubel, R., Johnsen, E.B. (eds.) *The Logic of Software. A Tasting Menu of Formal Methods*. LNCS, vol. 13360, pp. 481–499. Springer (2022), https://doi.org/10.1007/978-3-031-08166-8_23
25. Venema, Y.: A modal logic for chopping intervals. *J. of Logic and Computation* **1**(4), 453–476 (1991). <https://doi.org/10.1093/LOGCOM/1.4.453>