Event-based Analysis of Timed Rebeca Models using SQL

B. Magnússon
Reykjavik University
Reykjavik, Iceland
brynjar@brynjar.is

E. Khamespanah
University of Tehran, Tehran, Iran
Reykjavik University, Reykjavik, Iceland
e.khamespanah@ut.ac.ir

M.Sirjani
Reykjavik University
Reykjavik, Iceland
marjan@ru.is

R. Khosravi
University of Tehran
Tehran, Iran
r.khosravi@ut.ac.ir

ABSTRACT

In this paper, we present a simulation-based approach for analysis of Timed Rebeca models to tackle the state space explosion problem. We present a simulation toolkit, TRSim, which uses McErlang as a back-end simulator and stores the occurrences of events while executing the model in a relational database. We also present TeProp, a timed event-based property language, which is designed to be easy-to-use for specifying and reasoning about timed occurrences of events in an actor system. One can check TeProp formulas against multiple simulation runs by transforming the formulas to SQL queries and executing the queries over the event database. Using this approach, the correctness of large models can be analyzed to a bounded degree of confidence. To illustrate the applicability of TRSim toolkit and TeProp we provide a number of case studies.

Keywords

Actor model, Timed-Rebeca, Verification, Realtime systems, Simulation, Database, Action based property, TeProp

1. INTRODUCTION

Model checking is a formal verification approach aiming to verify correctness of systems with a very high level of confidence. The importance of using model checking to assure reliability of reactive systems has long been acknowledged. The major limiting factor in applying model checking for verification of real world reactive systems is the huge amount of space and time required to store and explore the state space. Alternatively, simulation techniques do not guarantee the correctness of system, however, they are extremely useful for design exploration to obtain estimates of the correctness of the systems.

In this paper, we present an approach for analysis of actor models, using simulation and event-based property specifications. We generate numbers of simulation traces and store the traces in a relational database. We propose an event-based property language (TeProp) that can address interactions among components in a natural way, in comparison with state-based property languages. We map TeProp formulas to SQL queries using our tool. Analysis of systems takes place by executing the SQL queries over the database of simulation traces. Using database as the storage and SQL queries enable us to analyze complex reactive systems, modeled by Timed Rebeca, an actor based modeling language.

Reactive systems are inherently parallel systems in which ongoing communication and interaction between the system and its environment plays a key role [4]. A well-known paradigm for modeling the functional behavior of reactive systems with asynchronous communication is the actor model [15, 5]. In the actor model, *actors* are universal primitives of concurrent computation: in response to a message, an actor can make local decisions, create actors, send messages, and determine how to respond to the next message. They may also redirect communication links through exchange of actor identities [28]. All the actors in the system run concurrently and the message passing among them is asynchronous. There are some extensions to the actor model such as RT-synchronizer [24] and Timed Rebeca[3] for modeling real-time systems.

Timed Rebeca is proposed as an extension to *Rebeca* [29]. *Rebeca*, is an operational interpretation of the actor model with formal semantics, supported by model checking tools [28]. Rebeca is designed to bridge the gap between formal methods and software engineering. The formal semantics of Rebeca is a solid basis for its formal verification. Compositional and modular verification, abstraction, symmetry and partial-order reduction have been investigated for verifying Rebeca models. The theory underlying these verification methods is already established and is embodied in verification tools [28, 16, 26, 27]. With its simple, message-driven and object-based computational model, Java-like syntax, and a set of verification tools, Rebeca is an interesting and easy-to-learn model for practitioners. In Timed Rebeca, timing primitives are added to Rebeca to address *computation time*, *message delivery time*, *message expiration*, and *period of occurrence of events* [3]. More details about Timed Rebeca is presented in Section 2.

Timed Rebeca is supported by the Afra toolset for schedulability and deadlock freedom analysis [17]. In [17], the focus is on the key features of actors, being event-driven and isolated, and a highly efficient approach for generating state space of actors is proposed. This approach, in principle, can be used for model checking against any event-based property. At present the tool only supports schedulability and deadlock freedom check. Another tool is developed for mapping Timed Rebeca to Real-time Maude [25]. This provides a spectrum of analysis methods for Timed Rebeca, but the properties are all state-based and not event-based. The current work is a continuation of work presented in [20]. Here, the approach is using simulation and focusing on event-based properties. SQL is used to expand our ability in saving and analyzing simulation traces. Moreover, we propose an event based property language, TeProp, and define its formal semantics to facilitate specifying the required properties (Section 3). The stored traces can be analyzed using SQL queries generated from TeProp formulas (Section 4 and Section 5). We prepare a number of experimental results to illustrate the applicability of our approach (Section 6).

The contribution of this paper can be summarized as follows.

- Implementing a toolset for simulating Timed Rebeca models, using McErlang model checker, and storing simulation traces in a relational database for further analysis.
- Introducing TeProp as an event-based property language for Timed Rebeca models, including a tool for mapping TeProp formulas to SQL queries as well as a tool for analysis of the results of executing SQL queries on simulation traces.
- Providing experimental results which very well illustrate the applicability of the proposed technique and toolset.

2. BACKGROUND

Rebeca [29, 28] is an actor-based language for modeling concurrent and reactive systems with asynchronous message passing. Rebeca models have reactive objects with no shared variables, asynchronous message passing with no blocking send and no explicit receive, and unbounded buffers for messages. Objects in Rebeca are reactive and self-contained, called a *rebec*. In this paper we used actor and rebec interchangeably. Communication among rebecs takes place by message passing. Each rebec has an unbounded buffer, for its arriving messages. Computation is event-driven, meaning that each rebec takes a message that can be considered as an event from the top of its message queue and execute the corresponding message server (also called a method). The execution of a message server is an atomic execution of its body that is not interleaved with any other method execution.

As shown in example of Figure 1, a Rebeca model consists of a set of reactive class definitions – which are `TicketService`, `Agent`, and `Customer` in the example – and the main block (lines 42 to 46). In the main block the rebecs which are instances of the reactive classes are declared. In the example, `a`, `ts`, and `c` are three rebecs which are defined in the main block (lines 43 to 45). The body of the reactive class includes the declaration for its known rebecs, state variables, and message servers. The rebecs instantiated from a reactive class can only send messages to the known rebecs of that reactive class. For the reactive class `TicketService`, line 2 shows definition of its known rebecs, line 4 shows the definition of its state variables, and lines 10 to 14 shows the definition of one of its message servers. Message servers consist of the declaration of local variables and the body of the message server. The statements in the body can be assignments (e.g. line 13), conditional statements, enumerated loops, non-deterministic assignment, and method calls (e.g line 12). Method calls are sending asynchronous messages to other rebecs (or to itself). The operational semantics of Rebeca has been introduced in [29] in more details.

Timed Rebeca is an extension of Rebeca with time features for modeling and verification of time-critical systems. These primitives are added to Rebeca to address *computation time*, *message delivery time*, *message expiration*, and *period of occurrence of events*. In a Timed Rebeca model, each rebec has its own local clock. The local clocks can be considered as synchronized distributed clocks. Methods are still executed atomically, however passing of time while executing a method can be modeled. In addition, instead of queue for messages, there is a bag of messages for each rebec.

The timing primitives that are added to the syntax of Rebeca are *delay*, *deadline* and *after*. The *delay* statement models the passing of time for a rebec during execution of a message server (e.g line 11). The keywords *after* and *deadline* can only be used in conjunction with a method call (e.g. lines 38 and 23). The value of the argument of *after* shows how long it takes for the message to be delivered to its receiver. The *deadline* shows the timeout for the message, i.e., how long it will stay valid. The formal semantics of Timed Rebeca has been introduced in [3]. Some toolset are developed for verification of Timed Rebeca models which have been presented in [17, 25, 20].

3. EVENT-BASED PROPERTY LANGUAGE FOR TIMED REBECA

Computation in actor models is mainly derived by asynchronous communication. Therefore, reasoning about the relation among execution of message servers has a key role in analysis of actor models. So, a formalism to express correctness of properties of such models should be able to take events into account as well as atomic propositions over system states. As for the design of TeProp (Timed event-based Property language) our goal was to create a language that is capable of specifying properties about the timed occurrence of events in a natural way, and be easy to use by practitioners. The design of TeProp is based on Metric Temporal Logic (MTL) [19] which is an extension of Linear Temporal Logic (LTL) [22] adding optional real-time constraints to the temporal operators. As MTL uses relative intervals it is closer to Timed Rebeca needs, compared to other alternatives like Timed Computation Tree Logic TCTL [6], TILCO [21], and Timed Propositional Temporal Logic TPTL [7]. Our focus has restricted TeProp compared to MTL, which is in line with our goal to have an easy to use language. At the same time using timed property patterns, in the following we show that we can specify a wide range of properties using TeProp.

TeProp is influenced by the property patterns which are described in [2], [8] and [18], along with papers related to the untimed property languages such as [12]. We found that specifying *maximum, minimum, and exact distance between two events*, *periodic occurrence of events*, *bounded response to events*, and *precedence relation between events* are the widely used prop-

```
reactiveclass TicketService {
  knownrebecs {Agent a;}
  statevars {
    int issueDelay, nextId;
  }
  msgsrv initial(int myDelay) {
    issueDelay = myDelay;
    nextId = 0;
  }
  msgsrv requestTicket() {
    delay(issueDelay);
    a.ticketIssued(nextId);
    nextId = nextId + 1;
  }
}
reactiveclass Agent {
  knownrebecs {
    TicketService ts;
    Customer c;
  }
  msgsrv requestTicket() {
    ts.requestTicket()
       deadline(5);
  }
  msgsrv ticketIssued(byte id) {
    c.ticketIssued(id);
  }
}
reactiveclass Customer {
  knownrebecs {Agent a;}
  msgsrv initial() {
    self.try();
  }
  msgsrv try() {
    a.requestTicket();
  }
  msgsrv ticketIssued(byte id) {
    self.try() after(30);
  }
}

main {
  Agent a(ts, c):();
  TicketService ts(a):(3);
  Customer c(a):();
}
```

Figure 1: The model of ticket service system.

erty types in event based systems. The first five property types are mentioned in [19] and the sixth one is in [8]. Variations of these property types also appear in [2] and [18]. We designed TeProp to address these six types of properties. To this aim, we defined three temporal modalities **G**, **F**, **B** (pronounced "globally", "finally", and "before" respectively) and two operators $\rightarrow$ ("implies") and $\leadsto$ ("leads-to").

Differences with MTL. Since the standard **Until** operator in temporal logic is expressing that a state-proposition should hold until something happens and we are only concerned with the order and occurrence of instantaneous events we introduce the **Before** operator instead of including the **Until** operator. For example for stating that: e_1 precedes e_2 in the next 10 time units, we say e_1 $\mathbf{B}[0,10]$ e_2, while in MTL this would be $\neg((\neg e_1)\ \mathbf{U}[0,10]\ e_2) \wedge \mathbf{F}[0,10]\ e_2$. The other difference is defining the operator "leads-to" which is very similar to standard "implies" except for that if p is false in $p \leadsto q$ then $p \leadsto q$ is false (unlike for $p \rightarrow q$ which is true if p is false). We found this operator more intuitive when we express that occurrence of some event will finally cause some other events. Note that **G** can be paired by $\rightarrow$, and **F** can be paired with $\leadsto$. The syntax and semantics of TeProp are presented in the following subsections.

3.1 Syntax and Informal Semantics of TeProp

In TeProp temporal modalities are evaluated over time intervals. A time interval consists of two non-negative integers inside brackets [from, to]. The intervals are relative to the current time, i.e. the current time instant is represented by 0, positive integer represents the future, and the symbol *end* is used to refer to the occurrence time of the last event of a trace. Omitting an interval for an operator is the same as using the interval $[0, end]$.

As in this paper we define TeProp for Timed Rebeca models, events are defined based on the terminology of Timed Rebeca. An event in Timed Rebeca is assumed as starting the execution of a message server. An event is identified by the name of its receiver and the name of message server in form of *receiver.msgsrvName*(*condition*). Condition is optional and given as a Boolean expression over the event's parameters and its sender (using the keyword *sender*). The formal syntax of TeProp is depicted in Figure 2.

The intuitive meaning of TeProp formulas constructed by valid combinations of temporal modalities and "implies" and "leads-to" operator, is depicted in the following.

$$\begin{aligned}
\phi &::= e \mid \neg\phi \mid \phi \wedge \phi \mid (\phi) \mid F_I\ e \mid F_I\ (e \leadsto \phi) \mid G_I\ (e \rightarrow \phi) \mid e\ B_I\ e \\
I &::= [\langle Integer\rangle,\ \langle Integer\rangle] \mid [\langle Integer\rangle,\ end] \\
e &::= receiver.messageName([condition])
\end{aligned}$$

Figure 2: The syntax of TeProp. In e (denoting an event), the "condition" is a boolean expression on the values of the parameters and sender of e which its evaluation results in a boolean value.

- **Finally**: $\mathbf{F}[i_1, i_2]\ e$. An event matching e will happen somewhere during the interval $[i_1, i_2]$.
  Examples: F X.call() - An event with instance name *X* and message server *call* will happen at some point.
  $\mathbf{F}[5,20]$ X.call(n==7) - An event with instance name *X* and message server *call* and the message parameter *n* with value 7 will happen at some point between 5 and 20 time units.

- **Before**: e_1 $\mathbf{B}[i_1, i_2]$ e_2. Within the interval $[i_1, i_2]$, an event matching e_1 happens at least once before an event matching e_2.
  Example: X.call() $\mathbf{B}[0,5]$ Y.call() - Within 0 and 5 time units an event with instance name *X* and message server *call* will happen before an event with instance name *Y* and message server *call*.

- **Globally** with **Implies**: $\mathbf{G}[i_1, i_2](e \rightarrow \phi)$. For all events matching e during the interval $[i_1, i_2]$, the formula ϕ must be satisfied when the time of occurrence of e is used as ϕ's current time. The formula is also satisfied if there is no event that matches e during $[i_1, i_2]$.
  Example: $\mathbf{G}(X.call() \rightarrow \mathbf{F}[0,10]\ Y.call())$ - Every occurrence of an event with instance name *X* and message server *call* is followed by an event with instance name *Y* and message server *call* within 10 time units of event *X*.
  $\mathbf{G}(X.call() \rightarrow Y.call()\ \mathbf{B}[0,9]\ Z.call())$ - Every occurrence of an event with instance name *X* and message server *call* is followed by an event with instance name *Y* and message server *call* before an event with instance name *Z* and message server *call* , both within 9 time units of event *X*.

- **Finally** with **Leads-to**: $\mathbf{F}[i_1, i_2](e \leadsto \phi)$. At least for one occurrence of an event matching e in the interval $[i_1, i_2]$

the evaluation of the formula ϕ using the timing of the event e as its current time must be satisfied.
Example: $\mathbf{F}(X.call() \rightsquigarrow \mathbf{F}[0,10]\ Y.call())$ - At least one occurrence of an event with instance name X and message server *call* is followed by an event with instance name Y and message server *call* within 10 time units of event X.
$\mathbf{F}(X.call() \rightsquigarrow Y.call()\ \mathbf{B}[0,9]\ Z.call())$ - At least one occurrence of an event with instance name X and message server *call* is followed by an event with instance name Y and message server *call* before an event with instance name Z and message server *call*, both within 9 time units of event X.

In the following we show how the six property patterns mentioned before can be specified in TeProp. For each property pattern we describe a scenario, for which we provide a TeProp formula.

- **Maximum distance between an event and its reaction:** every event e_1 is followed by its reaction e_2 within x units of time. TeProp formula: $\mathbf{G}(e_1 \rightarrow \mathbf{F}[0,x]\ e_2)$. This pattern also called as "bounded response between an event and its reaction".

- **Exact distance between an event and its reaction:** every event e_1 is followed by its reaction e_2 in exactly x units of time. TeProp formula: $\mathbf{G}(e_1 \rightarrow \mathbf{F}[x,x]\ e_2)$.

- **Minimal distance between an event and its reaction:** two consecutive events e are at least x units apart. TeProp formula: $\mathbf{G}(e_1 \rightarrow \neg\mathbf{F}[0,x]\ e_2)$.

- **Periodic occurrence of events:** event e occurs regularly with a period of x units of time. TeProp formula: $\mathbf{G}(e \rightarrow (\mathbf{F}[x,x]\ e \wedge \neg\mathbf{F}[0,x-1]\ e)) \wedge \mathbf{F}[0,\infty]\ e$.

- **Bounded response:** each occurrence of an event e is responded within a maximum number of time units. TeProp formula: $\mathbf{G}(e_1 \rightarrow \mathbf{F}[0,x]\ e_2)$.

- **Precedence relation between two events:** within the next x time units, the occurrence of e_1 precedes the occurrence of e_2. TeProp formula: $e_1\mathbf{B}[0,x]\ e_2$.

For the above patterns, except for the last one, the MTL formula is similar to the TeProp formula. Although it seems that TeProp and MTL have minor differences, when we have more complicated formula TeProp can be easier to use in the event based setting. For example, stating that at least once after an occurrence of e_1, within the next 8 time units, e_2 precedes e_3. In MTL it will be $\mathbf{F}(e_1 \rightarrow (\neg((\neg e_2)\ \mathbf{U}[0,8]\ e_3) \wedge \mathbf{F}[0,8]\ e_3 \wedge e_1))$, while in TeProp this would be $\mathbf{F}(e_1 \rightsquigarrow e_2\ \mathbf{B}[0,8]\ e_3)$. In the state space setting MTL is more expressive as not all MTL formulas can be expressed in TeProp, as it was designed for use in the event based setting.

3.2 Formal Semantics of TeProp

Consider an alphabet Σ of all events of a given model. Let π be a finite sequence (e_0,τ_0),(e_1,τ_1),... (e_t,τ_t) of timed events where $e_i \in \Sigma$ is an event name and $\tau_i \in \mathbb{N}$ increasing over time, shows the occurrence time of e_i. Satisfaction of a TeProp formula ϕ by a sequence of timed events π from position i (shown by $\pi, i \models \phi$) is defined inductively in Figure 3. Remind that by event at position i we refer to the i_{th} timed event in π, not the timed event at time i.

$\pi,i \models true$	always
$\pi,i \models e$	iff the name of the instance and message server in e_i and e are the the same and the values of parameters and sender of e_i satisfy the optional condition of e
$\pi,i \models \neg\phi$	iff $\pi,i \not\models \phi$
$\pi,i \models \phi_1 \wedge \phi_2$	iff $\pi,i \models \phi_1 \wedge \pi,i \models \phi_2$
$\pi,i \models \mathbf{F}_I\ e$	iff $\exists\ j \in [i,\lvert\pi\rvert] \cdot \pi,j \models e \wedge \tau_j - \tau_i \in I$
$\pi,i \models e_1\ \mathbf{B}_I\ e_2$	iff $(\exists\ k \in [i,\lvert\pi\rvert] \cdot \pi,k \models e_2 \wedge \tau_k - \tau_i \in I) \rightarrow (\exists\ j \in [i,k] \cdot \pi,j \models e_1 \wedge \tau_j - \tau_i \in I)$
$\pi,i \models \mathbf{G}_I\ (e\ \rightarrow \phi)$	iff $\forall\ j \in [i,\lvert\pi\rvert] \cdot (\pi,j \models e \wedge \tau_j - \tau_i \in I) \rightarrow \pi,j \models \phi$
$\pi,i \models \mathbf{F}_I\ (e\ \rightsquigarrow \phi)$	iff $\exists\ j \in [i,\lvert\pi\rvert] \cdot \pi,j \models e \wedge \tau_j - \tau_i \in I \wedge \pi,j \models \phi$

Figure 3: The formal description of TeProp.

4. DATABASE DESIGN AND MAPPING TEPROP FORMULAS TO SQL QUERIES

Storing the results of Timed Rebeca simulator in a relational database and creating a mapping from TeProp formulas to SQL queries is a convenient way to achieve our goal in analyzing Timed Rebeca models. Using a relational database, we get an established solution for storing the simulation traces as well as taking the advantage of the industrial standard technology for data analysis. Besides, we allow modelers to write their own SQL queries in addition to checking TeProp formulas.

To increase the performance of analyzing TeProp formulas, we decide that for each rebec instance we create a separate table per message server. The general form of "SimulationID_InstanceName_MessageServerName" is used for the naming of the tables to illustrate that each table corresponds to which simulation run, rebec, and message server. The table has three columns, called "id", "time", and "sender". The column for "id" stores a global auto-generated number used in the simulator to indicate the total order of events. The columns "time" and "sender" store time of the event and sender of the event respectively. In case of message servers with parameters additional columns are added to the table for each parameter.

To give a top-down view of the mapping from TeProp formulas to SQL queries we start from the base SQL query. The basic query is a `select` query. The `where` clause of the `select` query is filled with the results of mapping from TeProp formulas to SQL queries. The general form of the base query is depicted below.

```
select 'satisfied' from [basic information table]
  where ([the outcome of the mapping from TeProp
  formulas to SQL queries])
```

This way, if a TeProp formula is satisfied one record, contains "satisfied", is returned.

In TeProp formulas, the time in which the inner formulas are to be satisfied is determined by the outer formula. For example, for TeProp formula $\mathbf{F}[0,10](e_1 \rightarrow \mathbf{F}[0,5]e_2)$ the outer formula is $\mathbf{F}[0,10](e_1 \rightarrow \phi)$, the outer event is e_1 and the inner formula is $\mathbf{F}[0,5]e_2$. For each occurrence of e_1 in time interval $[0,10]$, the inner formula $\mathbf{F}[0,5]e_2$ must be satisfied assuming that the occurrence time of e_1 is time 0 for $\mathbf{F}[0,5]e_2$. In this section, the information related to the outer formula are shown by variables with *parent* subscript.

Two basic TeProp formulas using conjunction and negation operators are mapped to the SQL queries using "and" and "not" operators as described below.

$$\begin{aligned} \neg\phi &\quad \rightarrow \textbf{not}(\phi) \\ \phi_1 \wedge \phi_2 &\quad \rightarrow (\phi_1)\ \textbf{and}\ (\phi_2) \end{aligned}$$

In the other four types of TeProp formulas, looking for the set of different occurrences of event e in a time interval is required. This set is retrieved from simulation traces using the following SQL query. The sequence number and the timing of event e is considered to be related to the sequence number and timing of its outer formula, obtained by the following query. We refer to this query by $e[i_1, i_2]$.

$$\begin{aligned} &\textbf{select}\ \mathtt{alias_e.id}\ \textbf{from}\ \mathtt{event_e}\ \mathtt{alias_e}\ \textbf{where}\ \mathtt{alias_e.id} \\ &\quad > \mathtt{alias_{parent}.id}\ \textbf{and}\ \mathtt{alias_e.time}\ \textbf{between} \\ &\quad \mathtt{alias_{parent}.time} + \mathtt{i_1}\ \textbf{and}\ \mathtt{alias_{parent}.time} + \mathtt{i_2} \end{aligned}$$

In the above formula, $\mathtt{event_e}$ is the name of the table storing occurrences of e, $\mathtt{alias_e}$ is an alias to refer to e, and $\mathtt{alias_{parent}}$ is an alias to refer to the parent of $e[i_1, i_2]$. This query returns the set of ids.

The event occurrence, "globally", and "finally" TeProp formulas are directly mapped to SQL using the definition of $e[i_1, i_2]$ as described bellow.

$$\begin{aligned} &e && \rightarrow \textbf{exists}(\mathtt{e[0,0]}) \\ &\mathbf{F}[i_1,i_2]\ e && \rightarrow \textbf{exists}(\mathtt{e[i_1,i_2]}) \\ &\mathbf{F}[i_1,i_2]\ (e \rightsquigarrow \phi) && \rightarrow \textbf{exists}(\mathtt{e[i_1,i_2]})\ \textbf{and}\ (\phi) \\ &\mathbf{G}[i_1,i_2]\ (e \rightarrow \phi) && \rightarrow \textbf{not exists}((\mathtt{e[i_1,i_2]})\ \textbf{and}\ \textbf{not}(\phi)) \end{aligned}$$

Finally, "before" formula $e_1\ \mathbf{B}[i_1, i_2]\ e_2$ is mapped to SQL query as shown in the following.

$$\begin{aligned} &\textbf{exists}\ (\textbf{select}\ \mathtt{alias_{e_1}.id}\ \textbf{from}\ \mathtt{event_{e_1}}\ \mathtt{alias_{e_1}}\ \textbf{where} \\ &\quad (\mathtt{alias_{e_1}.id} > \mathtt{alias_{parent}.id}\ \textbf{and}\ \mathtt{alias_{e_1}.time}\ \textbf{between} \\ &\quad \mathtt{alias_{parent}.time} + \mathtt{i_1}\ \textbf{and}\ \mathtt{alias_{parent}.time} + \mathtt{i_2}) \\ &\ \ \textbf{and} \\ &\quad (\textbf{exists}\ (\textbf{select}\ \mathtt{alias_{e_2}.id}\ \textbf{from}\ \mathtt{event_{e_2}}\ \mathtt{alias_{e_2}}\ \textbf{where} \\ &\quad \mathtt{alias_{e_2}.id} > \mathtt{alias_{e_1}.id}\ \textbf{and}\ \mathtt{alias_{e_2}.time}\ \textbf{between} \\ &\quad \mathtt{alias_{e_1}.time}\ \textbf{and}\ \mathtt{alias_{parent}.time} + \mathtt{i_2})) \\ &) \end{aligned}$$

In the above formula the first term makes sure that there is at least one occurrence of e_1 and the second term makes sure that all the occurrences of e_2 are after e_1 in the interval of $[i_1, i_2]$.

4.1 SQL Examples

To give a clear idea how the SQL queries look like we have included below some TeProp formulas and their SQL counterpart. The formulas are for a simple communication protocol model that can be found at [1]

Listing 1: SQL for $\mathbf{G}$(senderAgent.start() $\rightarrow$ $\mathbf{F}[0,10]$receiverAgent.send())

```
select 'satisfied' from "base" t0_0 where (
  not exists(
    select t1_0.ID from "senderAgent_start"
        t1_0 where t1_0.ID > t0_0.ID and
      t1_0.time >= t0_0.time and not (exists(
        select t1_1.ID from
            "receiverAgent_send" t1_1 where
            t1_1.ID > t1_0.ID and
          t1_1.time between t1_0.time and
              t1_0.time + interval '10' second
      ))
  )
)
```

Listing 2: SQL for $\mathbf{F}$(senderAgent.start() $\rightsquigarrow$ $\mathbf{F}[0,8]$receiverAgent.send())

```
select 'satisfied' from "base" t0_0 where (
  exists(
    select t1_0.ID from "senderAgent_start"
        t1_0 where t1_0.ID > t0_0.ID and
      t1_0.time >= t0_0.time and (exists(
        select t1_1.ID from
            "receiverAgent_send" t1_1 where
            t1_1.ID > t1_0.ID and
          t1_1.time between t1_0.time and
              t1_0.time + interval '8' second
      ))
  )
)
```

Listing 3: SQL for $\mathbf{F}$ receiverAgent.ack()

```
select 'satisfied' from "base" t0_0 where (
  exists(
    select t1_0.ID from "senderAgent_ack" t1_0
        where
      t1_0.ID > t0_0.ID and t1_0.time >=
          t0_0.time
  )
)
```

Listing 4: SQL for senderAgent.start() $\mathbf{B}[0,5]$senderAgent.ack()

```
select 'satisfied' from "base" t0_0 where (
exists(
  select t1_0.ID from "senderAgent_start" t1_0 where
t1_0.ID > t0_0.ID and t1_0.time between t0_0.time and
t0_0.time + interval '5' second and exists(
  select t1_1.ID from "senderAgent_ack" t1_1 where
  t1_1.ID > t1_0.ID and t1_1.time between t1_0.time and
  t0_0.time + interval '5'
)
)
)
```

5. TRSIM TOOLKIT

We implemented a set of tools, TRSim toolkit, to support the analysis of Timed Rebeca models using TeProp. TRSim toolkit uses the new version of the translator of Timed Rebeca models to Erlang codes and PostgreSQL database [23]. PostgreSQL is an open source database system available for different operating systems with support for nested queries which are essential for the mapping of TeProp expressions to SQL queries.

5.1 Transformer of Timed Rebeca to Erlang

A toolset for transforming Timed Rebeca models to Erlang codes (timedreb2erl) is previously presented in [3]. Authors transform Timed Rebeca model to Erlang code, then they use McErlang [14] to simulate it. Later, an extension of McErlang which supports discrete-time semantics is released [13], results in development of a new transforming toolset for Timed Rebeca models [20]. For the simulation based analysis purposes choosing the right time to stop the simulation and the number of simulation traces must be considered to achieve credible analysis.

The Right Time to Stop the Simulation. As Timed Rebeca is used for modeling and verification of reactive systems with non-terminating behavior, simulation of Timed Rebeca models results in infinite simulation traces. But we need finite simulation traces of Timed Rebeca models to be able to use simulation based analysis. Therefore, each simulation run of Timed Rebeca models should be stopped at some point to have finite simulation traces. However, the best place to stop the simulation is not always clear and depends heavily on the model and the properties. Changing the model such that it stops after few iterations is an ideal solution for reactive systems that have a non-terminating behavior, this can be done with a local counter inside a reactive class. Another alternative for achieving bounded simulation traces is using the time limit option in McErlang. It allows specifying how long the simulation should be in seconds.

Number of Simulation Traces. No matter how simple our models are, their behavior can be complex as a result of non-deterministic choices which model concurrency of the elements. Therefore, for a given model it is important to run as much as possible simulation traces to try to cover all the behaviors of the model and compute μ_z as mean value of correctness of the model against a given property. But deciding on the right number of simulation traces that is needed to cover all the cases for all the models is not clear. Therefore, $\tilde{\mu}_z$ is computed as (ϵ, δ)-approximation of μ_z. We say $\tilde{\mu}_z$ is an (ϵ, δ)-approximation of μ_z if $Pr[|\mu_z - \tilde{\mu}_z| < \epsilon] \geq 1 - \delta$. ϵ is the error value and δ is the confidence value of the approximated value of $\tilde{\mu}_z$. Dagum et al. in [10] presented a general optimal approximation algorithm to provide N as an upper bound on the number of execution traces and compute $\tilde{\mu}_z$, as the following.

$$N = \frac{\Upsilon_2 \times \epsilon}{\hat{\mu}_z}$$

$$\Upsilon_2 = 2(1 + \sqrt{\epsilon})(1 + 2\sqrt{\epsilon})(1 + \frac{ln3/2}{ln2/\delta})\Upsilon$$

$$\Upsilon = \frac{4}{\epsilon^2}(e - 2)ln(2/\delta)$$

In this formula, the value of N depends on the value of $\hat{\mu}_z$ which is raw estimates of μ_z. Here $\hat{\mu}_z = \frac{1+(1+\epsilon)\Upsilon}{N}$, where N is the number of traces which are needed to be analyzed to at least $\lfloor 1+(1+\epsilon)\Upsilon \rfloor$ number of them satisfies the given property. For the raw estimation of μ_z, values of $\min\{1/2, \sqrt{\epsilon}\}$ and $\delta/3$ are used to compute the value of Υ.

As described in [20], the modeler defines number of simulation traces which are required to achieve a given (ϵ, δ)-approximation. In most cases time limits are more important than the number of traces and affects the values of ϵ and δ. In other words, time limit is set for simulation and the confidence interval is computed based on the generated traces.

5.2 TRSim Components

TRSim includes three stand alone applications which are *PrepareDB*, *Logger*, and *Query Tool*. Figure 4 shows the architectur of TRSim, including the life cycle from the modeling to the analysis of results. The detailed activities of the mentioned applications are described below.

PrepareDB is a command line Java program that creates the database tables which are required for the Logger based on a given Timed Rebeca model. A separate table is created for every message server of each rebec instance using the naming rule "SimulationID_InstanceName_MessageServerName".

The tool also creates a table for storing information about the simulation characteristics such as the environment variables. It also creates a database view that selects the lowest time in any of the tables to find the starting time of the simulation.

Logger is a command line Java program that stores all the logs of the message server calls in a database. Logger uses the Erlang Jinterface to communicate with Erlang, allowing Logger to send and receive messages like an Erlang actor. To achieve a better performance during the simulation, for each table of the database, a data file is created on disk to store the logs. The data files are compatible with the schema of their corresponding tables. At the end of the simulation, the content of each data file is inserted into the database using bulk insert.

The logger also detects potential **Zeno** behavior by counting the number of messages received in a row tagged with the same time. If this number exceeds a threshold, defined by the modeler, a warning of possible Zeno behavior is shown in the application.

Query Tool is a Java desktop application for the analysis of simulation traces stored in the database, using TeProp formulas. The modeler can select a database which contains simulation traces, write a TeProp formula, and check its sat-

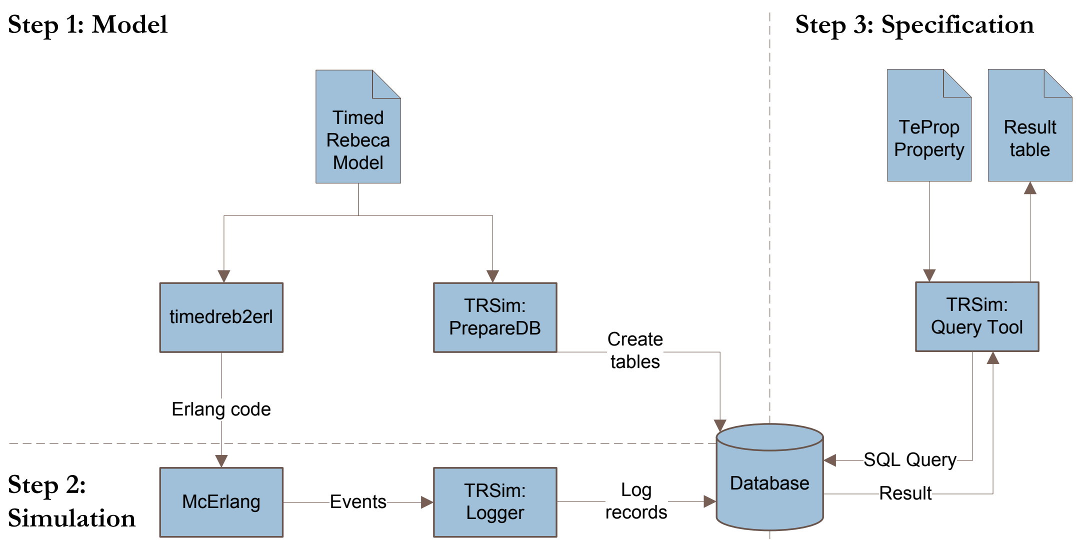

Figure 4: The architectural overview of the TRSim tool-set.

isfiability over all the simulation traces in the database. For checking a TeProp formula against simulation traces in the database, mapping from TeProp formulas to SQL queries is used (described in Section 4). The results of a query is displayed in a tabular format, showing the number of simulation traces, simulation environment parameters, and whether the property is satisfied. For further uses, the result table can be exported as a LaTeX table.

6. EXPERIMENTAL RESULTS

Using TRSim toolkit and TeProp specifications, we have analyzed many case studies and reported two of them in this paper. For each case study, we have run multiple simulations and checked various TeProp formulas. The source files of these models and TRSim toolkit are available from the Rebeca home page [1]. The experiments have been performed on a server with Intel(R) Xeon(TM) Quad CPU 2.66GHz processor and 32GB of RAM running 64-bit Ubuntu 10.04.4 LTS with Linux kernel 2.6.32-38-server, PostgreSQL 9.1.3, and Erlang R13B03.

6.1 Sensor Network

The sensor network model is introduced and modeled in [3]. The model consists of two sensors, a scientist, an admin, and a rescue team. The sensors are set up to measure the levels of toxic gas in the scientist's environment. Each sensor sends the measured values periodically to the admin with a period of *sensor0period* time units for the first sensor and *sensor1period* for the second sensor. the Admin checks the received values periodically with a period of *adminCheckDelay* time units. When the measured value exceeds dangerous toxic level, the admin immediately notifies the scientist by a message. If the scientist does not acknowledge the message within *scientistDeadline* time units, the admin sends the rescue team to save the scientist. If the scientist can not be rescued within *rescueDeadline* time units, it will die. There is a communication delay of *netDelay* time units between all the elements of the model.

For the analysis of the model we have used 700 simulation runs, using seven different settings, depicted in Table 1 (100 simulation runs for each setting). The results of analysis of the model against its related TeProp formulas are shown in Table 2. The percentages in the cells of Table 2 are computed based on the number of traces satisfying the given property.

In the first step, safety of the scientist is evaluated in different settings of the environment (shown in the first row of Table 2). To this aim, "$\neg\mathbf{F}\,[\mathtt{0}, \mathtt{end}]$ admin.scientistDead()" TeProp formula is designed to make sure that the admin never receives and executes "scientistDead" message. The other properties have similar interpretations.

The results of the verification shows that the model is behaving as intended; the rescue team is sent when the scientist does not acknowledge within the time limit and sadly the scientist dies in the case the rescue team does not reach him in time. The results show well how some timings end in tragic results. It is interesting to compare the results for property 1 and 2, where for settings 2 and 6 the scientist does not die as the rescue team is sent for him. But for setting 7 the scientist does not die although no rescue team is sent, as the scientist acknowledges the warning message in time. We also observe in property 3 that the timings for setting 7 avoids sending the rescue team because scientist acknowledges the admin's message.

6.2 Multi Flight Booking

Multi flight booking model is an extended version of the ticket service (see Figure 1) where timing is crucial. Here, there are related requests which have to be served by two different servers in an atomic transaction. Because of the distribution without synchronization, undesirable behavior may happens when ending up with one successful request while the other has been unsuccessful.

The model consists of two airlines and a customer. The customer starts by finding flights; that involves finding the first flight and requesting a reservation of a ticket from the website of the first airline and then finding and reserving the second flight from the website of the second airline. In the model it takes *findFlightTime1* units of time to find an appropriate ticket from the first web site and *findFlightTime2* time units for the second website. The airline websites keep

Setting	Network delay	Admin period	Sensor 0 period	Sensor 1 period	Scientist deadline	Rescue deadline
1	1	4	2	3	2	3
2	1	4	2	3	2	4
3	2	1	1	1	4	5
4	2	1	1	1	4	6
5	2	1	1	1	4	7
6	2	4	1	1	4	7
7	2	4	1	1	5	7

Table 1: Environment settings used for the simulation of the sensor network model. The first six settings are the same as the ones in [3].

Property	Setting / Result						
	1	2	3	4	5	6	7
The scientist will not die: $\neg\mathbf{F}\ [0,\mathit{end}]$ admin.scientistDead()	0%	100%	0%	0%	0%	100%	100%
The rescue team never went to rescue: $\neg\mathbf{F}\ [0,\mathit{end}]$ rescue.go()	0%	0%	0%	0%	0%	0%	100%
Admin never misses an acknowledgment as result of ordering of events within a time unit: $\mathbf{G}$(admin.checkScientistAck() $\rightarrow \neg\mathbf{F}\ [0,0]$ admin.ack())	0%	0%	0%	0%	0%	0%	100%

Table 2: Overview of TeProp formulas checked for the sensor network. The setting column refers to the environment variables in Table 1.

the reservations of each flight for either *reservationTimeout1* or *reservationTimeout2* time units. After receiving the messages that both flights have been reserved, the customer starts to book the flights by sending a request to the airlines websites. It takes *bookingTime1* and *bookingTime2* units of time to prepare the booking order before sending it. If an airline website receives the booking order before the reservation, it sends a confirmation that the flight has been successfully booked, otherwise a message is sent indicating that the flight was not booked. There is a communication delay of *networkDelay* time units between all the actors of this model.

For the analysis of the model, we have used 700 simulation runs, using 7 different settings, depicted in Table 3 (100 simulation runs for each setting). The results of the analysis of the model against its related TeProp formulas are shown in Table 4.

The results in Table 4 show that the model can behave as intended; similar to the ticket service model in which valid tickets are booked if the timing parameters are set properly. Comparing the results of properties 1 to 3 clearly shows the impact of an unsuccessful booking for a flight, as both flights must be booked for a successful multi-flight booking. Even though our success rate of booking a ticket for each flight is close to 50% we can end up with only 30% of successful multi-flight bookings. Property 4 then gives us an indication on whether the time for keeping the reservations is too generous. This is important since in reservation systems the goal is to minimize the this time (keeping the reservation) to fulfill as many requests as possible.

7. RELATED WORK

Böhlen et al. introduced a transformation from Linear Temporal Logic (LTL) to TSQL2, a temporal extension to the SQL language [9]. Our transformation from TeProp to SQL differs from theirs in the way that we are also working with time but use a standard SQL database (not a temporal extension).

TLtoSQL is a tool-set for rapid post-mortem verification of systems using temporal logic to SQL [11]. Post-mortem execution log files are read into the toolkit and automatically converted into JUnit test cases. The JUnit test cases are then executed and the event sequence during the run is stored in a database along with the events, their relative order, and time. The toolkit then offers a graphical editor for Linear Temporal Logic (LTL) and Metric Temporal Logic (MTL) formal specifications. The output of the editor is SQL query code that the user can then execute on the database. While both TRSim and TLtoSQL make use of a database and conversion from a property language to SQL, they differ in their intended use. TLtoSQL is meant to be a verification framework for verifying system implementations using execution logs, where TRSim is an integrated environment for simulation and verification of Timed Rebeca models; making it easy to run verification queries over multiple simulations. LTtoSQL also stores all the information in one database table, while TRSim uses a separate table for each message server. This way, the performance of the analysis of SQL queries is increased significantly.

8. CONCLUSION

In this paper, we proposed an approach for verification of timed actor models, using database and simulation technique; and we illustrated its applicability with a number of case studies. We used relational database as a repository of simulation traces and SQL queries for the analysis of traces. To analyze the relation among the events occurred during the execution of actor systems, we introduced an event based property language (TeProp). We also proposed a mapping from TeProp formulas to SQL queries. The database system executes SQL queries corresponding to TeProp formulas over simulation traces and reports the verification results. This way we can use distributed SQL query analysis or any other solutions proposed for database systems for analysis of big data sets.

Acknowledgement

The work on this paper was supported by the project "Timed Asynchronous Reactive Objects in Distributed Systems: TARO" (nr. 110020021) of the Icelandic Research Fund.

9. REFERENCES

[1] *Rebeca Home Page*. http://www.rebeca-lang.org.

Setting	Network delay	findFlight Time1	findFlight Time2	booking Time1	booking Time2	reservation Timeout1	reservation Timeout2
1	1	1	2	1	2	2	4
2	1	2	2	2	2	5	10
3	2	1	3	0	0	2	5
4	2	2	1	0	3	4	8
5	2	2	1	2	1	2	4
6	2	3	2	1	1	8	10
7	2	3	2	1	1	15	16

Table 3: Environment settings used for the simulation of the multi flight booking model.

Property	Setting / Result						
	1	2	3	4	5	6	7
The first ticket is successfully booked: **F** [0,*end*] customer.flightBooked(f == "1" ∧ successful == "true")	7%	52%	13%	28%	0%	90%	100%
The second ticket is successfully booked: **F** [0,*end*] customer.flightBooked(f == "2" ∧ successful == "true")	9%	54%	50%	48%	0%	100%	100%
All tickets are successfully booked: ¬**F** [0,*end*] customer.flightBooked(successful == "false")	2%	31%	7%	19%	0%	90%	100%
Booking occurred 3 or more time units before the reservation ran out: **F** [0,*end*] (ws1.bookFlight() ⇝ **F** [3, *end*] ws1.reservationExpired()) ∨ **F** [0,*end*] (ws2.bookFlight() ⇝ **F** [3, *end*] ws2.reservationExpired())	0%	75%	0%	30%	0%	57%	100%

Table 4: Overview of TeProp properties checked for the multi flight booking. The setting column refers to the environment variables in Table 3.

[2] N. Abid, S. Dal Zilio, and D. Le Botlan. A Real-Time Specification Patterns Language. Technical Report LAAS 11364, 2011.

[3] L. Aceto, M. Cimini, A. Ingólfsdóttir, A. H. Reynisson, S. H. Sigurdarson, and M. Sirjani. Modelling and Simulation of Asynchronous Real-Time Systems using Timed Rebeca. In M. R. Mousavi and A. Ravara, editors, *FOCLASA*, volume 58 of *EPTCS*, pages 1–19, 2011.

[4] L. Aceto, A. Ingólfsdóttir, K. G. Larsen, and J. Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, New York, NY, USA, 2007.

[5] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[6] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 414 –425, jun 1990.

[7] R. Alur and T. A. Henzinger. A really temporal logic. *J. ACM*, 41(1):181–203, Jan. 1994.

[8] P. Bellini, P. Nesi, and D. Rogai. Expressing and organizing real-time specification patterns via temporal logics. *J. Syst. Softw.*, 82(2):183–196, Feb. 2009.

[9] M. H. Böhlen, J. Chomicki, R. T. Snodgrass, and D. Toman. Querying TSQL2 Databases with Temporal Logic. In P. M. G. Apers, M. Bouzeghoub, and G. Gardarin, editors, *Advances in Database Technology - EDBT'96, 5th International Conference on Extending Database Technology, Avignon, France, March 25-29, 1996, Proceedings*, volume 1057 of *Lecture Notes in Computer Science*, pages 325–341. Springer, 1996.

[10] P. Dagum, R. M. Karp, M. Luby, and S. M. Ross. An optimal algorithm for monte carlo estimation. *SIAM J. Comput.*, 29(5):1484–1496, 1995.

[11] D. Drusinsky. TLtoSQL: Rapid post-mortem verification using temporal logic to sql code generation in the Eclipse PDE. In *SoSE*, pages 1–5, 2009.

[12] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 411–420, New York, NY, USA, 1999. ACM.

[13] C. B. Earle and L.-Å. Fredlund. Verification of timed erlang programs using mcerlang. In *FMOODS/FORTE*, pages 251–267, 2012.

[14] L.-Å. Fredlund and H. Svensson. McErlang: a model checker for a distributed functional programming language. *SIGPLAN Not.*, 42(9):125–136, Oct. 2007.

[15] C. Hewitt. Description and Theoretical Analysis (Using Schemata) of PLANNER: a Language for Proving Theorems and Manipulating Models in a Robot. Technical Report 258, MIT AI Laboratory, 1972.

[16] M. M. Jaghoori, M. Sirjani, M. R. Mousavi, E. Khamespanah, and A. Movaghar. Symmetry and Partial Order Reduction Techniques in Model Checking Rebeca. *Acta Inf.*, 47(1):33–66, 2010.

[17] E. Khamespanah, Z. S. Kaviani, R. Khosravi, M. Sirjani, and M.-J. Izadi. Timed-Rebeca Schedulability and Deadlock-Freedom Analysis Using Floating-Time Transition System. In *AGERE!@SPLASH*, pages 23–34, 2012.

[18] S. Konrad and B. Cheng. Real-time specification patterns. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 372 – 381, may 2005.

[19] R. Koymans. Specifying Real-Time Properties with Metric Temporal Logic. *Real-Time Systems*, 2(4):255–299, 1990.

[20] H. Kristinsson, A. Jafari, E. Khamespanah, B. Magnusson, and M. Sirjani. Model Checking and Performance Evaluation of Timed Rebeca Models Using McErlang. In *AGERE!@SPLASH*, 2013.

[21] R. Mattolini and P. Nesi. An interval logic for real-time system specification. *Software Engineering, IEEE Transactions on*, 27(3):208 –227, mar 2001.

[22] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46 –57, 31 1977-nov. 2 1977.

[23] PostgreSQL. http://www.postgresql.org, 2012.

[24] S. Ren and G. Agha. RTsynchronizer: Language Support for Real-Time Specifications in Distributed Systems. In R. Gerber and T. J. Marlowe, editors, *Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pages 50–59. ACM, 1995.

[25] Z. Sabahi-Kaviani, R. Khosravi, M. Sirjani, P. C. Ölveczky, and E. Khamespanah. Formal semantics and analysis of Timed Rebeca in Real-Time Maude. In *FTSCS*, pages 178–194, 2013.

[26] H. Sabouri and M. Sirjani. Slicing-Based Reductions for Rebeca. In *Proceedings of FACS 2008*. ENTCS, 2008.

[27] M. Sirjani, F. S. de Boer, and A. Movaghar-Rahimabadi. Modular Verification of a Component-Based Actor Language. *J. UCS*, 11(10):1695–1717, 2005.

[28] M. Sirjani and M. M. Jaghoori. Ten Years of Analyzing Actors: Rebeca Experience. In G. Agha, O. Danvy, and J. Meseguer, editors, *Formal Modeling: Actors, Open Systems, Biological Systems*, volume 7000 of *Lecture Notes in Computer Science*, pages 20–56. Springer, 2011.

[29] M. Sirjani, A. Movaghar, A. Shali, and F. S. de Boer. Modeling and Verification of Reactive Systems using Rebeca. *Fundam. Inform.*, 63(4):385–410, 2004.