Event-based Analysis of Timed Rebeca Models using SQL

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

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

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

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

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 backend 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.

Categories and Subject Descriptors D.2.4 [SOFTWARE ENGINEERING]: Software/Program Verification - Assertion checkers, Formal methods, Statistical methods.

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 [3]. A well-known paradigm for modeling the functional behavior of reactive systems with asynchronous communication is the actor model [5, 15]. 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 ex-
change of actor identities [27]. All the actors in the sys-
tem 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
Timed Rebeca is proposed as an extension to Re-
beca [28]. Rebeca, is an operational interpretation of
the actor model with formal semantics, supported by
model checking tools [27]. Rebeca is designed to bridge
the gap between formal methods and software engi-
eering. 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 meth-
ods is already established and is embodied in verifica-
tion tools [16, 26, 27, 29]. 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, mes-
sage expiration, and period of occurrence of events [4]. More
details about Timed Rebeca is presented in Section 2.
Timed Rebeca is supported by the Afa 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 ap-
proach 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 fo-
cusing on event-based properties. SQL is used to ex-
and our ability in saving and analyzing simulation
taces. 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 Sec-
tion 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 lan-
guage 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 il-
lustrate the applicability of the proposed technique
and toolset.

2. Background
Rebeca [27, 28] is an actor-based language for model-
ing concurrent and reactive systems with asynchronous
message passing. Rebeca models have reactive objects
with no shared variables, asynchronous message pass-
ing 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. Communi-
cation 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
the corresponding message server (also called a method).
The execution of a message server is an atomic execu-
tion 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 [28] 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 [4]. Some toolset are developed for verification of Timed Rebeca models which have been presented in [17, 20, 25].

Figure 1. The model of ticket service system.

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 [4]. Some toolset are developed for verification of Timed Rebeca models which have been presented in [17, 20, 25].

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 [7], TILCO [21], and Timed Propositional Temporal Logic TPTL [6]. 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 property 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 → ("implies") and ← ("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 \( e_1 \mathcal{B}[0, 10] e_2 \), while in MTL this would be \( \neg((\neg e_1) \cup[0, 10] e_2) \land F[0, 10] e_2 \). The other difference is defining the operator “leads-to” which is very similar to standard “implies” except for that if \( p \equiv q \) then \( p \rightarrow q \equiv q \) 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 and we want to make sure that there is at least one occurrence of the first event. Note that \( G \) can be paired by \( \rightarrow \), and \( F \) can be paired with \( \rightarrow \). 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 \( \text{end} \) 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 \([0, \text{end}]\).

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 \( \text{receiver}.\text{msgsrvName}(\text{condition}) \). Condition is optional and given as a Boolean expression over the event’s parameters and its sender (using the keyword \( \text{sender} \)). The formal syntax of TeProp is depicted in Figure 2.

\[
\phi ::= e \mid \neg \phi \mid \phi \land \phi \mid (\phi) \mid F_I e \mid F_I (e \rightarrow \phi) \mid G_I (e \rightarrow \phi) \\
I ::= [(\text{Integer}), (\text{Integer})] \mid [(\text{Integer}), \text{end}] \\
e ::= \text{receiver}.\text{msgsrvName}[\text{condition}]
\]

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.

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

- **Finally:** \( F[i_1, i_2] e \). An event matching \( e \) will happen somewhere during the interval \([i_1, i_2]\).
  
  **Examples:** \( F X.\text{call()} \) - An event with instance name \( X \) and message server \( call \) will happen at some point. \( F[5, 20] X.\text{call}(n=7) \) - 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:** \( e_1 \mathcal{B}[i_1, i_2] e_2 \). Within the interval \([i_1, i_2]\), an event matching \( e_1 \) happens at least once before an event matching \( e_2 \).
  
  **Example:** \( X.\text{call()} \mathcal{B}[0, 5] Y.\text{call()} \) - 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:** \( G[i_1, i_2] (e \rightarrow \phi) \). For all events matching \( e \) during the interval \([i_1, i_2]\), the formula \( \phi \) must be satisfied when the time of occurrence of \( e \) is used as \( \phi \)'s current time. The formula is also satisfied if there is no event that matches \( e \) during \([i_1, i_2]\).
  
  **Example:** \( G(X.\text{call()} \rightarrow F[0, 10] Y.\text{call()} \) - 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 \( X \).

- **Finally with Leads-to:** \( F[i_1, i_2] (e \Rightarrow \phi) \). At least for one occurrence of an event matching \( e \) in the interval \([i_1, i_2]\) the evaluation of the formula \( \phi \) using the timing of the event \( e \) as its current time must be satisfied.
  
  **Example:** \( F(X.\text{call()} \Rightarrow F[0, 10] Y.\text{call()} \) - 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 \( X \).

- **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: \( G(e_1 \rightarrow \)
For each occurrence of event $e$ 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.

$$\text{select} \, \text{'satisfied'} \, \text{from} \, \text{[basic information table]}$$

$$\text{where} \, \left( \text{[the outcome of the mapping from TeProp formulas to SQL queries]} \right)$$

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 $F[0,10][e_1 \rightarrow F[0,5]e_2]$ the outer formula is $F[0,10][e_1 \rightarrow \phi]$, the outer event is $e_1$ and the inner formula is $F[0,5]e_2$. For each occurrence of $e_1$ in time interval $[0,10]$, the inner formula $F[0,5]e_2$ must be satisfied assuming that the occurrence time of $e_1$ is time 0 for $F[0,5]e_2$. 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.

$$\neg \phi \quad \rightarrow \, \text{not}(\phi)$$

$$\phi_1 \land \phi_2 \quad \rightarrow \, (\phi_1) \, \text{and} \, (\phi_2)$$

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 $e[i_1,i_2]$.

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.
Figure 3. The formal description of TeProp.

| \( \pi, i \models e \) | always if the name of the instance and messageserver in \( e \), and \( e \) are the same and the values of parameters and senderof \( e \) satisfy the optional condition of \( e \) |
| \( \pi, i \models \neg \phi \) | iff \( \pi, i \not\models \phi \) |
| \( \pi, i \models \phi_1 \land \phi_2 \) | iff \( \pi, i \models \phi_1 \land \pi, i \models \phi_2 \) |
| \( \pi, i \models F_i e \) | iff \( \exists j \in [i, |\pi|] \cdot \pi, j \models e \land \tau_j - \tau_i \in I \) |
| \( \pi, i \models e_1 B_i e_2 \) | iff \( \exists k \in [i, |\pi|] \cdot \pi, k \models e_2 \land \tau_k - \tau_i \in I \rightarrow (\exists j \in [i, k] \cdot \pi, j \models e_1 \land \tau_j - \tau_i \in I) \) |
| \( \pi, i \models G_i (e \rightarrow \phi) \) | iff \( \forall j \in [i, |\pi|] \cdot \pi, j \models e \land \tau_j - \tau_i \in I \rightarrow \pi, j \models \phi \) |
| \( \pi, i \models F_i (e \leadsto \phi) \) | iff \( \exists j \in [i, |\pi|] \cdot \pi, j \models e \land \tau_j - \tau_i \in I \land \pi, j \models \phi \) |

The formal description of TeProp.

```sql
select alias_e.id from event_e alias_e where alias_e.id > alias_par.id and alias_e.time between alias_par.time + i1 and alias_par.time + i2
```

In the above formula, `event_e` is the name of the table storing occurrences of `e`, `alias_e` is an alias to refer to `e`, and `alias_par` is an alias to refer to the parent of `e[i1,i2]`. 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[i1,i2]` as described bellow.

```sql
e \rightarrow \exists(e[0,0])
F[i1,i2] e \rightarrow \exists(e[i1,i2])
F[i1,i2] (e \leadsto \phi) \rightarrow \exists(e[i1,i2]) \land (\phi)
G[i1,i2] (e \rightarrow \phi) \rightarrow \neg \exists((e[i1,i2]) \land \neg(\phi))
```

Finally, “before” formula \( e_1 B[i1,i2] e_2 \) is mapped to SQL query as shown in the following.

```sql
exists (select alias_e.id from event_e alias_e where (alias_e.id > alias_par.id and alias_e.time between alias_par.time + i1 and alias_par.time + i2) and (exists (select alias_e2.id from event_e alias_e2 where alias_e2.id > alias_e.id and alias_e2.time between alias_e.time and alias_par.time + i2)))
```

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 \([i1,i2] \).
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 nondeterministic 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 $\mu_z$ 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 \((\epsilon, \delta)\)-approximation of \(\mu_z\). We say \(\tilde{\mu}_z\) is an \((\epsilon, \delta)\)-approximation of \(\mu_z\) if \(\Pr[|\mu_z - \tilde{\mu}_z| < \epsilon] \geq 1 - \delta\). \(\epsilon\) is the error value and \(\delta\) 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.

\[
N = \frac{\Upsilon_2 \times \epsilon}{\tilde{\mu}_z}
\]

\[
\Upsilon_2 = 2(1 + \sqrt{\epsilon})(1 + 2\sqrt{\epsilon})(1 + \frac{\ln 3/2}{\ln 2/\epsilon})\Upsilon
\]

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

In this formula, the value of \(N\) depends on the value of \(\tilde{\mu}_z\), which is raw estimates of \(\mu_z\). Here \(\tilde{\mu}_z = \frac{1 + (1 + \epsilon)\Upsilon}{N}\), where \(N\) is the number of traces which are needed to be analyzed to at least \([1 + (1 + \epsilon)\Upsilon]\) number of them satisfies the given property. For the raw estimation of \(\mu_z\), values of \(\min(1/2, \sqrt{\epsilon})\) and \(\delta/3\) are used to compute the value of \(\Upsilon\). For example, the raw value of \(\Upsilon\) for the case of \((2, 0.01)\)-approximation is computed the same as below.

\[
\Upsilon = \frac{4}{\min(1/2, \sqrt{\epsilon})}(\epsilon - 2)\ln(2/0.01/3)
\]

\[
= 16 \times 0.71 \times 6.40 = 72.7
\]

As described in [20], the modeler defines number of simulation traces which are required to achieve a given \((\epsilon, \delta)\)-approximation. In most cases time limits are more important than the number of traces and affects the values of \(\epsilon\) and \(\delta\). 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 8 shows the architect 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 satisfiability 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 [4]. 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 \(\text{sensor0period}\). The admin checks the received values periodically with a period of \(\text{adminCheckDelay}\) 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, TeProp formula $\neg F[0, \infty] \text{admin.scientistDead()}$ 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 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
Table 1. Environment settings used for the simulation of the sensor network model. The first six settings are the same as the ones in [4].

<table>
<thead>
<tr>
<th>Setting</th>
<th>Network delay</th>
<th>Admin period</th>
<th>Sensor 0 period</th>
<th>Sensor 1 period</th>
<th>Scientist deadline</th>
<th>Rescue deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

The scientist will not die:
\[ \neg F[0, end] \text{admin.scientistDead()} \]

The rescue team never went to rescue:
\[ \neg F[0, end] \text{rescue.go()} \]

Admin never misses an acknowledgment as result of ordering of events within a time unit:

\[ G(\text{admin.checkScientistAck()} \rightarrow \neg F[0, 0] \text{admin.ack()}) \]

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Setting / Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>The scientist will not die:</td>
<td></td>
</tr>
<tr>
<td>[ \neg F[0, end] \text{admin.scientistDead()} ]</td>
<td>0%</td>
</tr>
<tr>
<td>The rescue team never went to rescue:</td>
<td>100%</td>
</tr>
<tr>
<td>[ \neg F[0, end] \text{rescue.go()} ]</td>
<td>0%</td>
</tr>
<tr>
<td>Admin never misses an acknowledgment as result of ordering of events</td>
<td>0%</td>
</tr>
<tr>
<td>within a time unit:</td>
<td>100%</td>
</tr>
<tr>
<td>[ G(\text{admin.checkScientistAck()} \rightarrow \neg F[0, 0] \text{admin.ack()}) ]</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Setting</th>
<th>Network delay</th>
<th>findFlight Time1</th>
<th>findFlight Time2</th>
<th>booking Time1</th>
<th>booking Time2</th>
<th>reservation Timeout1</th>
<th>reservation Timeout2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>15</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

Table 4. Environment settings used for the simulation of the multi flight booking model.

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 TSOL2, 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
Table 4. Overview of TeProp properties checked for the multi flight booking. The setting column refers to the environment variables in Table 3.

<table>
<thead>
<tr>
<th>Property</th>
<th>Setting</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>The first ticket is successfully booked:</td>
<td>1</td>
<td>7%</td>
</tr>
<tr>
<td>$F[0,end] \text{customer.flightBooked}(f == &quot;1&quot; \wedge $successful == &quot;true&quot;)$</td>
<td>2</td>
<td>52%</td>
</tr>
<tr>
<td>The second ticket is successfully booked:</td>
<td>3</td>
<td>13%</td>
</tr>
<tr>
<td>$F[0,end] \text{customer.flightBooked}(f == &quot;2&quot; \wedge $successful == &quot;true&quot;)$</td>
<td>4</td>
<td>28%</td>
</tr>
<tr>
<td>All tickets are successfully booked:</td>
<td>5</td>
<td>0%</td>
</tr>
<tr>
<td>$\neg F[0,end] \text{customer.flightBooked}(\text{successful} == &quot;false&quot;)$</td>
<td>6</td>
<td>90%</td>
</tr>
<tr>
<td>Booking occurred 3 or more time units before the reservation ran out:</td>
<td>7</td>
<td>100%</td>
</tr>
<tr>
<td>$F[0,end] (\text{ws1.bookFlight() \sim F[3,end] ws1.resExpired()} \lor $</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>$F[0,end] (\text{ws2.bookFlight() \sim F[3,end] ws2.resExpired()}$</td>
<td>2</td>
<td>75%</td>
</tr>
</tbody>
</table>

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.

References


