Seamless Formal Verification of Complex Event Processing Applications

AnnMarie Ericsson
School of Humanities and Informatics
University of Skövde, Sweden
annmarie.ericsson@his.se

Paul Pettersson
Department of Computer Science and Electronics
Mälardalen University
SE-721 23, Västerås, Sweden
Paul.Pettersson@mdh.se

Mikael Berndtsson
School of Humanities and Informatics
University of Skövde, Sweden
mikael.berndtsson@his.se

Marco Seiriö
RuleCore, Sweden
marco@rulecore.com

ABSTRACT

Despite proven successful in previous projects, the use of formal methods for enhancing quality of software is still not used in its full potential in industry. We argue that seamless support for formal verification in a high-level specification tool enhances the attractiveness of using a formal approach for increasing software quality.

Commercial Complex Event Processing (CEP) engines often have support for modelling, debugging and testing CEP applications. However, the possibility of utilizing formal analysis is not considered.

We argue that using a formal approach for verifying a CEP system can be performed without expertise in formal methods. In this paper, a prototype tool REX is presented with support for specifying both CEP systems and correctness properties of the same application in a high-level graphical language. The specified CEP applications are seamlessly transformed into a timed automata representation together with the high-level properties for automatic verification in the model-checker Uppaal.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design tools and techniques

General Terms

CEP, CASE, Timed automata

Keywords

Design, Verification, CEP, Timed automata

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DEBS '07 June 20-22, 2007 Toronto, Ontario, Canada
Copyright 2007 ACM 978-1-59593-665-3/07/03 ...$5.00.

1. INTRODUCTION

Nowadays, many companies and organizations use event-driven applications as an approach to analyze and detect patterns in large volumes of data as they become available. Briefly, we can categorize these applications into: i) time series filtering, and ii) track and trace.

Applications related to time series filtering use one or more event streams as input and produce another event stream as output. Every real-time index at a stock market is a good example of this: multiple event streams with new prices of all the constituent stocks are used as input and the output is the index itself.

Events in track and trace applications carry information about state changes emitted from entities. For example, a door opens in building five, or an order changes state from unconfirmed to confirmed. In contrast to time series filtering, events in track and trace applications are not expected to arrive in a steady flow of events with an update on the latest value of an entity. Instead, the entity will generate an event whenever something interesting has changed.

The abilities to detect complex event patterns on the fly have led to a growing market of event and rule engines, e.g., Amit, ruleCore, Coral8, Aptsoft, Aleri, StreamBase. Regardless of whether they support time series filtering applications or track and trace applications, few if any of them have support for formally verifying that the events and rules do not contain design errors.

As the area of complex event processing (CEP) is still evolving and no wide spread development methodology is available for CEP systems, we argue that it is now time to introduce best practice methods for developing systems within the area.

Verifying that a set of events or rules behave as intended during runtime is not trivial due to the high complexity of event composition and rule execution models. Thus, such analysis should always be done by a tool, rather than by manual inspection.

We have designed and implemented a prototype tool REX (Rule and Event eXplorer) that acts as a front end to the timed-automata CASE-tool Uppaal[7]. REX provides support for specification of composite events, rules, and verification properties (e.g., termination, correctness) in a high-level language. REX automatically transforms events, rules,

and verification properties to a timed-automaton representation that can be used by UPPAAL. After transformation, REX seamlessly starts the model-checker in UPPAAL to test the verification properties and waits for the result from the model-checker. Thus, using systems like REX and UPPAAL together during specification and design of an CEP application facilitates tasks such as detecting design errors in events and rules.

Similar to events and rules in active databases [24], termination and confluence are two key issues when analyzing events and rules. An extensive body of knowledge already exist on termination and confluence within the active database field (e.g.[5, 11, 2, 6]). However, events and rules need to be analyzed beyond termination and confluence, i.e., they need to be analyzed with respect to application specific properties. For example, detecting whether event e_1 can occur in an interval started by event e_2 and ending with event e_3, is neither related to termination nor to confluence.

Due to the lack of tools for formally analyzing events and rules within the CEP area, we assume that many developers check termination, confluence, and correctness of their events and rules by testing, debugging and analyzing traces when errors have already occurred.

In this paper we present a novel approach for automatically verifying application specific properties of events and rules for CEP applications. We show how these properties can be described in the high-level property language in REX and how these properties are transformed to a timed-automata representation that can be used by UPPAAL. For in depth description of how the timed automaton representation of composite events is constructed we refer to [15, 16].

This paper is structured as follows; In section 2, preliminaries about the area of complex event processing and the CASE tool UPPAAL is presented. Section 3 describes the current status of our prototype tool REX followed by in depth information about how to specify and transform property specifications from REX to UPPAAL.

2. PRELIMINARIES

This section present background knowledge about complex event processing and UPPAAL.

2.1 Complex Event Processing

A characteristic feature of a CEP engine is finding patterns in an ordered (stream) or partially ordered (cloud) set of events. A complex event processing engine typically reacts to events occurring in predefined patterns. In this paper, we use the word *composite event* as a synonym for patterns possible to detect using a CEP engine.

Although no standard language exists for describing all types of composite events possible to detect in a CEP engine, some operators for combining events reappear in several engines. A common set of supported operators are conjunction ($\bigwedge$), disjunction ($\bigvee$), sequence (;) and non-occurrence. These operators are extensively used in previous work in active databases (e.g.[9, 18]). Additionally, functions performing calculation of, for example, average value, max and min value of parameter values arriving with events and counting occurrences of a specific type during a sliding time window interval are supported. Further, CEP engines may have support for rule based specifications where occurring events are triggering code (Action) if a specified condition is true.

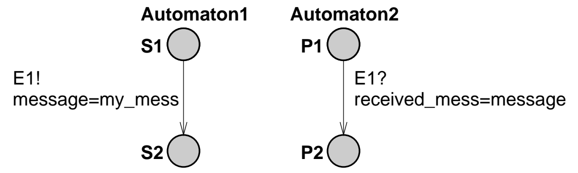

Figure 1: Synchronizing timed-automata

2.1.1 Consumption policy

The initiator of a composite event is the event occurrence initiating the composition of the composite event and the terminator is the event occurrence terminating the event composition. In this paper, an event occurrence is said to contribute to a composite event if the event is a terminator or initiator of the composite event.

If there are several occurrences of a contributing event type, the time and number of triggerings of the composite event depends on which instance of the contributing event types that are used for combining the composite event. Additionally, contributing event occurrences may carry parameters used in calculations implying that the choice of contributing event instances can affect the resulting outcome of running the CEP application.

The issue of combining contributing event occurrences are previously addressed in, for example, [9, 23] and [28]. In this paper, we use the chronicle and recent consumption policies defined in [9]. In chronicle consumption policy, events are composed in chronicle order (e.g. the first unused event occurrence of initiator type is combined with the first unused occurrence of terminator type). In recent consumption policy, composite events are composed by the most recent occurrence of each contributing event type.

2.1.2 Expiration time

The expiration time determines for how long an initiated composite event should wait for its terminator. It may be the case that the composite event is useless if it is not triggered within a predefined time span, for example, if the composite event triggers a rule whose action part has already expired. Additionally, it is beneficial to clean the system from semi-composed events in order to save memory[8].

2.2 UPPAAL

Timed automata are finite automata extended with a set of real-valued clocks[4]. Timed-automata are designed for specifying and verifying real-time systems. UPPAAL is a toolbox for modelling and analyzing specifications built on the theory of timed-automata extended with additional features. The tool is developed jointly by Uppsala University and Aalborg University[7]. A model in UPPAAL is built by a network of synchronizing timed automata. Each timed automaton simulates a process which is able to synchronize with other automata.

2.2.1 Specifying models in UPPAAL

Each automaton in an UPPAAL model contains a set of *locations* S with an initial location $s_0 \in S$. A state of a timed automata is defined as a pair consisting of a location and an assignment mapping all clocks and variables to values in their domains.

Each location of a timed automaton can be labelled with

a *location invariant* described as a constraint on clocks that may force a transition from the location to be taken. If, for example, c_1 is a clock variable, and the invariant $c_1 <= 4$ is defined in location s_0, the automaton is not allowed to operate in location s_0 if $c_1 > 4$.

A transition can be associated with three parts; (i) constraints on clocks and variables specified by a guard g,(ii) reset clock values and change variable values with action a, and (iii) communicate with other automata by synchronizing on global channels.

To send data on channel x (! is the notation for send), another automaton must simultaneously receive the message on the same channel (? is the notation for receive). Synchronizing transitions imply parallel composition of two automata. If the channel is defined as a broadcast channel, several automata may receive a message sent by one automaton simultaneously.

In Figure 1, Automaton1 and Automaton2 synchronizes on channel E_1 when Automaton1 is in location S_1 simultaneously as Automaton2 is in location P_1. During the transition, Automaton1 sends the content of the variable *my_mess* to Automaton2 by assigning *my_mess* to the global variable *message* read by Automaton2.

In a network of timed-automata modelling a system in UPPAAL, the order in which different automata perform their transitions may be controlled by attaching different priorities to the automata.

2.2.2 *Analyzing models in* UPPAAL

Given an UPPAAL model of a system, model-checking can be performed by specifying the properties needed to be checked in CTL (computation tree logic) [3].

The syntax for describing that process (automaton) P has property i is *P.i* where i can be a location, a variable or a clock defined in automaton P. Given a state formula, for example, $P.i < 5$, the path formula $E <> P.i < 5$ (E<> is the syntax for $\exists\Diamond$ in UPPAAL) is used to check whether there exists a path where i is less than 5 in process P.

The result of querying the model is either that the property is satisfied or not satisfied. The property can quantify over specific states or over a trace of states. It is, for example, possible to ask if variable x will always have a value less than 5 in location S_1 in process P ($A[]P.S_1$ and $x < 5$) or if it is possible to reach location S_2 within 3 time units ($E <> P.S_2$ and $globalClock < 3$).

Uppaal supports a subset of CTL for expressing properties. In some cases, an additional test-automaton needs to be provided to be able to express the desired property [22]. The test-automaton must be constructed so that it is only possible to reach a designated state in the test-automaton if the questioned property is satisfied in the original model. For an in depth description and tutorial on the capacity of model-checking in UPPAAL we refer to [20].

3. REX

The Rule and Event eXplorer (REX) utilizes the capability of verifying systems in UPPAAL. REX automatically transforms specifications of rule based systems and property specifications to timed-automata and thereby allows seamless verification of composite events and rules in UPPAAL.

3.1 Rule specification

The rule paradigm supported in REX is event condition action (ECA) rules. The most expressive part of a rule in REX is the event part, allowing REX to be utilized for analyzing CEP specifications with or without support for triggering rules.

Properties

Name	Type
Name	E3
Operator	Conjunction
Initiator	E1
InitiatorFilter	par1<5
Terminator	E2
TerminatorFilter	
Parameters	par3=0
Function	par3=ACC(par1,par2)
Derived Parameters	par1=0,par2=0
Policy	Chronicle
Detection	
Expiration time	10
Group	
Is triggering rules:	Rule1,Rule2,

Figure 2: Example of a property table for an event.

In REX, an ECA rule is built up by four different items. Besides the event, condition and action items, an additional item is used named *DataObject*. A DataObject is an abstraction of, for example, a variable, a sensor or a tuple in a database. A DataObject can trigger an event, be read by a condition or changed in the action part of a rule.

The set of rules, events, conditions, actions and DataObjects in REX are graphically viewed in tree structures (one tree for rules, one for events etc.). When an item in a tree structure is selected, a property table is viewed showing the properties of the selected item. An example of a property table for an event is shown in Figure 2.

3.1.1 *Event*

When a new event is created, properties of the event are specified in the new events property table. Each event must have a unique name (the type of the event). When a new event type is created, it is available to be chosen as an initiator or terminator for a composite event or as the triggering event in a rule property table.

REX supports transformations of events from REX to UPPAAL of both primitive and composite events. Primitive events can be time events, triggered on a definite time or a time relative to some other occurrence, an external event (e.g. sensor event) or internal event (e.g. update of a tuple in a database). The operators currently supported for composite events are Times(n,E) (triggered after n occurrences of event type E), sequence, conjunction, disjunction, non-occurrence, delay, and a sliding window type which can be constructed to perform some operation and leverage result over a sliding time window.

The consumption policies currently supported are recent and chronicle with or without expiration times and with or without parameters and filters. Filters can be used to model parameter matching, i.e. that events are only seen as contributing events if a certain parameter has a specific value.

3.1.2 DataObject

The DataObjects in REX represent abstractions of anything that can be updated by an action, read by a condition or used as an event parameter. A DataObject in REX is a first class object keeping associations to the events it triggers when it is updated, conditions reading it and actions that can alter its value.

Since REX utilizes UPPAAL for model-checking, and the variable types supported in UPPAAL are integer, clocks, boolean and arrays of the previous types, DataObjects are also limited to be of these types.

A DataObject can be chosen to trigger an event when it is updated, be read by a condition or be updated by an action. For each DataObject, the property table shows which events are triggered by this DataObject, the conditions that read the DataObject and the actions that may alter the DataObject.

3.1.3 Condition

The different types of conditions that can be analyzed in the current version of REX is limited to expressions over integer values and data objects. (e.g. $dataobject1 < 4$). The set of operators supported in conditions are $\{<, >, ==, <=, >=, !=\}$.

3.1.4 Action

The action part of an ECA rule can often execute an arbitrary sequence of code. In REX, functions possible to write in UPPAAL is supported (e.g. for loops and if statements). A data object can be assigned an arbitrary integer or another data object. Additionally, simple functions can be written as actions, for example, starting a new task with a loop increasing the value of a data object each time unit during an interval.

If the execution time of the rules action part is known, it can be specified in REX and this information will be included in the model. If the execution time is not specified it is assumed that the time it takes to execute the action is irrelevant for the analysis.

3.2 Transformation approach

The result of transforming a specification from REX to UPPAAL is a timed-automaton model representing the behavior of the specified set of rules. The automaton can be divided into three different parts; environment, event composer and rule executer.

The environment is not a part of the specified rules, but it needs to be included in the timed-automaton model since events occurring in the environment affect system behavior. The different parts of the model are described in the following subsections.

3.2.1 Environment

The run-time behavior of the system is dependent on the time and order in which primitive events arrive from the environment. Independently of type of source (e.g. sensor activation, user input etc.), primitive events stemming from the environment are modelled as a channel in one or several dedicated environment automata.

The default environment automaton created by REX is an automaton where all specified primitive events occur in non-deterministic time and order and keep on occurring infinitely. However, REX allows users to tailor the environment automaton to specific needs by restricting the time and order of event occurrences or specify an upper limit for the number of rule triggerings for events. It may, for example, be the case that event E_1 can not occur before event E_2 and if the model-checker is not provided with this information, it may find combinations of events or rule executions that can never occur in the actual environment. A specific time-sequence can be specified for each primitive event containing a sequence of time points when the primitive event occur. Apart from the benefit of controlling the environment under test, restricting the order in which events can occur decrease the search space for the model-checker.

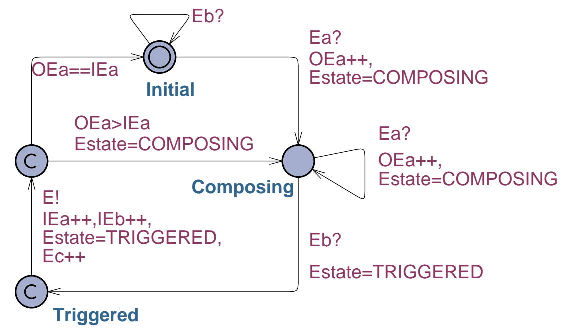

Figure 3: Example of composite event $E = E_a; E_b$ in chronicle consumption policy.

3.2.2 Event Composer

The event-composer consists of several automata where each automaton is modelling the behavior of a composite event type. Each composite event type possible to specify in REX has a corresponding timed-automaton operator pattern modelling the behavior of that composite event in a specific consumption mode. When a composite event is transformed to UPPAAL, a new automaton is created by instantiating the operator pattern modelling the composite event. A timed automata operator pattern listens to channels representing contributing event occurrences and sends on a channel representing the composite event occurrence when the event occurs.

An example of a timed automaton modelling the behavior of the composite event E $= E_a; E_b$ in chronicle consumption policy is shown in Figure 3. An event of type E is generated when there is an unused occurrence of type E_a followed by an unused occurrence of type E_b in the event history.

The automaton starts in location Initial. If an event of type E_b occurs in location Initial the automaton remains in the initial location since E_b is not initiator in E. However, if an event of type E_a occurs the automaton enters location Composing. The variable E_{state} is tracking the location changes of the composite event. The counter OE_a counts number of occurrences of the initiator type E_a since each terminator E_b must be matched with an initiator that has occurred but is not used yet.

If there is a new occurrence of type E_a when the automaton is in location Composing, the counter OE_a is increased, however the automaton remains in location Composing. If an event of type E_b occurs, the automaton enters location Triggered. In location Triggered there is an unused occurrence of both E_a and E_b in the event history and a new occurrence of the composite event E is generated (the au-

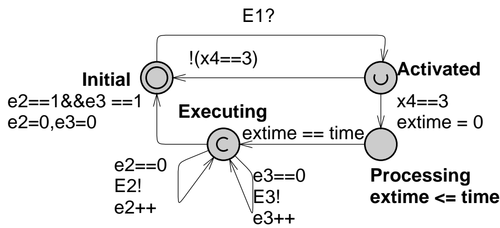

Figure 4: Example of a rule automaton.

tomaton is synchronizing on channel E!). The variables IE_a and IE_b counting the number of used occurrences of type E_a and E_b are increased since, in chronicle consumption policy, the generated composite event is consuming the contributing event occurrences. If there exist one or more unused occurrences of type E_a in the event history (guard $OE_a > IE_a$ is true) the automaton takes the transition to Composing waiting for a new event of type E_b to occur, else, the automaton takes the transition back to Initial.

Additional examples and in depth descriptions of timed automata patterns simulating the behavior of different composite events can be found in our previous work [15, 16].

3.2.3 Rule Executer

The rule execution model consists of at least one automaton for each rule in the rule set. UPPAAL provides the ability to set different priorities on different automata implying that a rule automaton gets the same priority as the rule it is modelling.

An example of a rule automaton is shown in Figure 4. The automaton starts in location Initial. When the event triggering the rule (event E_1) occurs, the rule automaton synchronizes on the events channel and enter location Activated. When a rule is activated, the condition is immediately evaluated ($x_4 == 3$). If the condition is evaluated to true, the action part of the rule is executed. The rule in the example has a specified execution time attached to its action. The execution time is modelled as a local clock which is reset (extime = 0) when the rule leaves location Activated. In location *Processing*, the invariant (extime <= time) and the guard on next transition (extime == time) ensure that the execution time is modelled properly.

If executing the action part of the rule is causing new events to be generated, the rule synchronizes on the channels representing the triggered events (E_2! and E_3! in Figure 4). The Executing location is marked as committed implying that no time passes while simulating that the new events are generated.

Depending on how rules are processed in the forthcoming platform for execution, the rule automaton may have to be constructed differently. If, for example, a rule can generate the same type of events that it is triggered by, it will not be triggered by that event in the previous model since the rule is in location Executing when the event is generated. In such cases, the action part of the rule is be modelled in a separate automaton that is started by a synchronization when the rules condition is true.

4. VERIFICATION PROPERTIES

Given a timed-automaton specification of a system, certain kinds of errors can be detected by verifying properties in a model-checker. However, the verification properties needed to be checked is often application specific. This makes it impossible to develop a set of predefined properties to run for verifying all different applications.

For example, in a scenario where the specification models an application with the aim of detecting fraud-attempts in a banking system, the application is required to generate a fraud alarm when a certain sequence of events has occurred. The requirement that the alarm will always be raised when a suspicious sequence of events occurs is one of many properties that needs to be verified in that system.

In a second scenario, the specification models a control application monitoring the temperature in a tank where it is paramount that the temperature remains within a specific interval. In such a system, practitioners need to verify that the system reacts correctly to a specific sequence of temperature readings.

4.1 Verification issues

REX supports automatic transformation of rule based specifications into timed automata. However, in order to utilize the generated timed-automaton specification for model-checking, verification properties specific to the specified set of rules must be formulated. Since the target tool for the transformation in REX is UPPAAL, the system requirements must be specified in the subset of CTL supported by UPPAAL.

Specifying a correct CTL question for a rule or event-based property is not trivial. First, it requires that the practitioner is familiar with CTL. Additionally, specifying correct verification questions requires detailed knowledge of the timed automaton model. Hence, in the case of REX, one of the main ideas of the tool is to be able to perform formal analysis without detail knowledge of the actual timed automaton model.

Our approach to overcome these difficulties is to support specification of properties in the same level of abstraction as the CEP specification. The properties possible to specify are based on the rules and events currently specified in the system together with predefined property patterns.

According to Dwyer et al.[14], most of the property specifications that practitioners write tend to reappear as patterns in different specifications. Based on that observation, Dwyer et al. [13] present a set of property patterns where application specific states can be included in predefined specification patterns.

Since, as far as we know, no previous work has addressed the issue of what properties are most common in a CEP application, we utilize the patterns in [13] to achieve a structural approach for specifying a useful set of verification properties in a high-level language. The transformation from properties to specific CTL questions is automated by REX. The patterns and transformation process is described in the following subsections.

4.2 Property Patterns

Dwyer et al. [13] propose a set of property patterns to facilitate the use of finite-state verification in practice. Their aim is to reuse property specifications for finite-state verification by defining a collection of simple patterns that can be transformed to, for example, CTL (Computation tree logic), LTL (Linear temporal logic) or regular expressions.

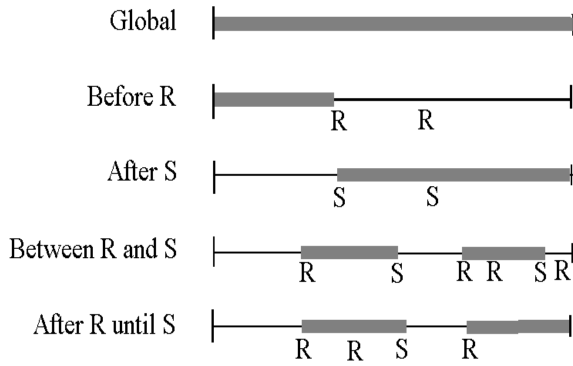

Figure 5: Scopes, i.e. duration of execution.

Each pattern has a scope, which is the extent of the program execution over which the pattern must hold. There are five scopes as shown in Figure 5. The scopes are defined as follows: *global* (property holds during entire program execution), *before* (property holds up to a given state), *after* (property holds after a given state), *between* (property holds in an interval between two states) and *after-until* (like between but the end state is not required).

The following patterns are presented in Dwyer et al.[13], (capital letters, e.g. P,Q,R,S represent states).

- *Absence* describes that the defined scope is free from state P
- *Existence* describes that a state P occur within the scope
- *Bounded Existence* describes that a state P must occur k times within the scope
- *Universality* describes that a state P is true throughout the scope
- *Precedence* describes that a state P must always be preceded by state Q in the scope
- *Response* describes cause-effect relationships. An occurrence of the state P must be followed by an occurrence of state Q.
- *Chain Precedence* sequence of state $P_1...P_n$ must always be preceded by sequence of state $Q_1...Q_m$ in the scope
- *Chain Response* sequence of state $P_1...P_n$ must always be followed by a sequence of state $Q_1...Q_m$ in the scope

4.3 Verification in REX

The property patterns are used for structuring the high-level property language in REX. One of the issues with transforming a property specified in REX to CTL is that the requirement properties for the REX system are likely to include both state information and event occurrences. It may, for example, be interesting to verify whether the event of type E shown in Figure 3 always is in state Composing (i.e. the transformed automaton is in location Composing) simultaneously as another event of type E_5 occurs. For the user of REX, defining states and event occurrences are performed by selecting a leaf in a tree structure. However, in the transformation process, event occurrences and states are treated differently. In the following, the lower-case letters (e.g. p, q, r, s) denotes predicates instead of states since event occurrences are not states, however, the fact that an event occur may be characterized by a predicate.

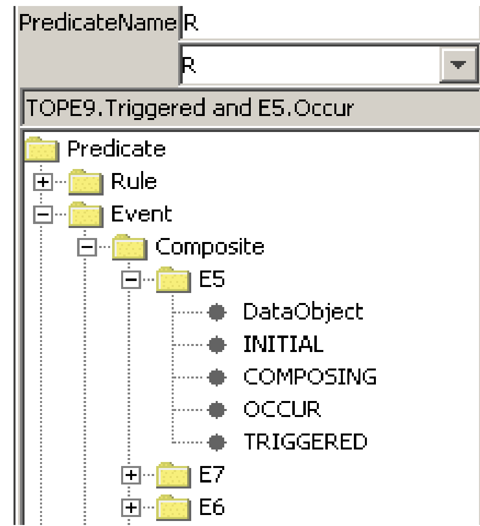

Figure 6: Example of Predicate Tree.

4.3.1 State definitions in REX

In REX, a rule can be in one of three different states; Initial, Activated and Executing. The rule states defined in REX are represented as locations in the automaton. A rule is said to be in Initial state in REX when its automaton model is in location Initial, in Activated state when its automaton is in location Activated etc. (see example of a rule automaton in Figure 4).

A composite event in REX may be in state Initial, Composing or Triggered. The event states in REX are represented as locations in the automaton representing the behavior of the composite event (see example of a transformed automaton in Figure 3).

The fact that an event occurs (e.g. when the automaton in Figure 3 is synchronizing on channel E!) is denoted *Occur* in Figure 6 and this is when a new instance of an event is generated.

A predicate can be defined by combining the states of different rules, states of events, event occurrences and values of DataObjects in the specification.

REX supports a graphical description where predicates are chosen in a tree structure. A tree structure for selecting predicates to a property specification is exemplified in Figure 6. The predicate tree supports a structured method to choose all different predicates available for the specified items. The available predicates are modelled as leaf nodes in Figure 8. For each DataObject leaf, the predicate is represented by an expression ($DataObject \sim x$ where $x \in \mathtt{N}$ and $\sim \in \{=, !=, <, >, <=, >=\}$). If DataObject is chosen in the predicate tree, a new dialog box appears, allowing the user to specify constraints on rule counters, event parameters or DataObjects included in rule-conditions or rule-actions.

Once the predicates are defined, a pattern is instantiated

Property	Value
PropertyName	P1
Pattern	Absence
Scope	Before R
P	E5Occur
R	E1inComposing

Figure 7: Example of Property selection table.

by selecting a set of choices in a pattern-property table. First, the pattern and scope of choice is selected. Based on the choice of pattern and scope, one, two, three or four defined predicates can be chosen to represent p, q, r and s in the pattern before the model-checking can start.

If, for example, pattern *P Absence* and scope *Before R* is selected as shown in Figure 7, then the predicates p and r are selected in comboboxes containing all previously specified predicates.

4.4 Transformation of properties

In Dwyer et al.[13] the property patterns are defined in CTL as well as LTL and regular expressions. However, the transformation to UPPAAL is not straightforward since the current version of UPPAAL does not support verification of liveness properties in models with priorities.

A test-automaton ($\mathcal{A}_{\mathcal{T}}$) must be generated by REX to model and verify liveness properties. In order to achieve a uniform transformation method, all property transformations are modelled with a test-automaton $\mathcal{A}_{\mathcal{T}}$.

Automaton $\mathcal{A}_{\mathcal{T}}$ contains specific locations named *Satisfied* and *Fail* which are only reachable if the timed-automata representation of the CEP application satisfies (or not satisfies for contradiction proofs) the specified property.

Automaton $\mathcal{A}_{\mathcal{T}}$ is decorated with guards modelling the predicates that must be true for the property to be satisfied. Additionally, $\mathcal{A}_{\mathcal{T}}$ has the highest priority of all generated automata to ensure that the guarded transition is always taken when the guard becomes true before some other automata can change the guard to false.

In the following subsections, transformations from predicates to guards is described followed by a description of how the patterns *Absence* and *Response* are transformed for the different scopes.

4.4.1 Transforming predicates to guards

The OCCUR predicate is transformed to a synchronization in the test automaton while all other predicates are transformed to conjuncts in guards.To represent the information that an item is in a specific state, a state variable is used for tracking the items state (see section 3.2.2). For example, the predicate E_n.INITIAL characterizing that event E_n is in state Initial is transformed to $En_{state} ==$ Initial.

DataObjects in REX can be global variables used in conditions or actions as well as parameters in events or counters counting the number of occurrences of specific items. All DataObjects are transformed to variables in UPPAAL, hence DataObject expressions are already in a valid format for guards.

The CTL experssion generated for the model-checker is

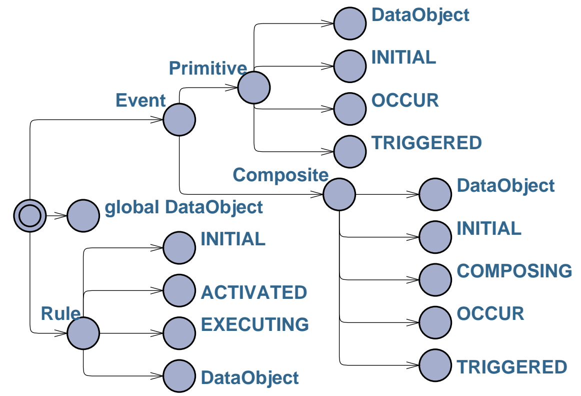

Figure 8: Tree modelling possible predicates to choose.

denoted φ. For properties that require $\exists\Box, \forall\Diamond$ or $\rightarrow$ a finite event sequence is required since these expressions can not be used with prioritized models.

If a finite event sequence is required, the user of REX is asked to define an upper limit of the number of generated primitive events in the model. The stop state named θ is a CTL property specifying that all rules specified in the system under test are in location Initial, all primitive events in the environment have reached their maximum number of triggerings and none of the composite events are in location Triggered. Hence, the stop state is a deadlock where no more events will occur and all rules have finished executing.

Most of the presented properties are modelled as a contradiction proof. This means that a satisfied result from UPPAAL is a non-satisfied result on the property asked in REX and vice versa. The user of REX is not concerned of whether the proof is transformed to a contradiction proof or not.

In the following sections, the $\mathcal{A}_{\mathcal{T}}$ and φ for verifying properties combined by the Absence and Responds pattern in all different scopes are presented. All predicates p, q, r, s are transformed to conjuncts in guards $p\prime$, $q\prime$, $r\prime$, $s\prime$.

4.4.2 Example application

A subset of an example application for controlling pressure and temperature in a tank is used for exemplifying the use of property verification. The tank has a valve that can be opened or closed to regulate pressure and heaters and coolers for regulating temperature. The valve is opened by assigning value OPEN to variable *valve* and closed by assigning CLOSE to the same variable.

The event stream read by the application contains events of type $E_t(tmp)$ and $E_p(press)$. When an event of type $E_t(tmp)$ or $E_p(press)$ occurs, a composite event $E_{tp}(tmp, press)$ is generated, forwarding the parameters *tmp* and *press* retrieved from its contributing events.

Events of type $E_{tp}(tmp, press)$ trigger rules of type R_{vOpen}. When R_{vOpen} is triggered, the condition $tmp * press > MAX$ is evaluated where MAX is a predefined maximum value. If the condition evaluates to true, variable *valve* is set to OPEN implying that a valve should be opened to decrease the pressure in the tank. A sensor is attached to the valve signaling $E_{vOpened}$ when the valve is opened and $E_{vClosed}$ when the valve is closed.

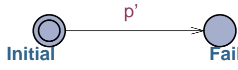

Figure 9: $\mathcal{A}_\mathcal{T}$ for Pattern: P Absence, Scope: Global

The rule R_{vClose} is triggered by $E_{tp}(tmp, press)$. The condition in R_{vClose} checks if $tmp * press < MIN$. If the condition evaluates to true, variable *valve* is set to CLOSE implying that the valve should be closed to increase the pressure in the tank.

The rules $R_{decrease}$ and $R_{increase}$ are triggered each time an event of type $E_t(tmp)$ occurs. If the temperature is above 10000 and the valve is set to open, the temperature should be decreased. However, if the temperature is less than 5000 and the valve is closed, the temperature should be increased.

The application is specified in REX and transformed to UPPAAL. All rules are transformed according to the method exemplified in Figure 4 and the composite event E_{tp} is transformed to a specific automaton modelling conjunctions in recent consumption policy. The example application consists of the following events and rules:

$E_{vOpened}$ Signalled by sensor when valve is opened.

$E_{vClosed}$ Signalled by sensor when valve is closed.

$E_t(tmp)$ Triggered by sensor reading temperature periodically. Parameter *tmp* represent current temperature.

$E_p(press)$ Triggered by sensor reading pressure periodically. Parameter *press* represent current pressure.

$E_{tp}(tmp, press)$ is a conjunction in recent policy
$E_{tp}(tmp, press) = E_p(press) \bigwedge E_t(tmp)$

R_{vOpen} is an ECA rule defined as follows:
Event: $E_{tp}(tmp, press)$,
Condition: $tmp * press > MAX$,
Action: $valve = OPEN$

R_{vClose} is an ECA rule defined as follows:
$E_{tp}(tmp, press)$,
Condition: $tmp * press < MIN$,
Action: $valve = CLOSE$

$R_{decrease}$ is an ECA rule defined as follows:
$E_t(tmp)$,
Condition: $tmp > 10000$ and $valve == OPEN$,
Action: $decreaseTmp()$

$R_{increase}$ is an ECA rule defined as follows:
$E_t(tmp)$,
Condition: $tmp < 5000$ and $valve == CLOSE$,
Action: $increaseTmp()$

4.4.3 Pattern: P Absence, Scope: Global

If *P Absence* is satisfied in scope *Global* it implies that predicate p is never satisfied during the entire execution.

The automaton $\mathcal{A}_\mathcal{T}$ shown in Figure 9 is generated together with the specified system and the CTL expression $\varphi = \exists\Diamond\mathcal{A}_\mathcal{T}.Fail$ is run to verify the property.

If guard p' becomes true, $\mathcal{A}_\mathcal{T}$ takes the transition to location Fail. However, if p' remain false, $\mathcal{A}_\mathcal{T}$ remains in location

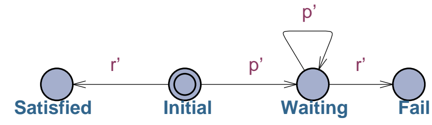

Figure 10: $\mathcal{A}_\mathcal{T}$ for Pattern: P Absence, Scope: Before R

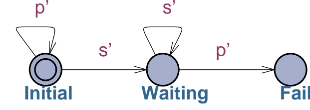

Figure 11: $\mathcal{A}_\mathcal{T}$ for Pattern: P Absence, Scope: After S

Initial. Hence, if it is possible to reach $\mathcal{A}_\mathcal{T}.Fail$, predicate p is not absent in the global scope.

If, for example, p is defined as $R_{increase}$.EXECUTE and $tmp > 6000$, then P Absence Global is satisfied if the automaton modelling $R_{increase}$ never is in location Executing simultaneously as the value of the DataObject representing temperature is above 6000.

4.4.4 Pattern: P Absence, Scope: Before R

Selecting the *P Absence* pattern together with the *Before R* scope creates a property that is satisfied if predicate p is never satisfied before predicate r.

The generated $\mathcal{A}_\mathcal{T}$ for property P Absence Before R is shown in Figure 10. $\mathcal{A}_\mathcal{T}$ is created as a contradiction proof where $\varphi = \exists\Diamond\mathcal{A}_\mathcal{T}.Fail$. A non-satisfied result from the model-checker implies that p is absent before r is satisfied. Note that the property only test whether p is absent before r is satisfied for the first time during execution.

If p is defined as $E_{tp}(tmp, press)$.COMPOSING and r is defined as $E_t(tmp)$.OCCUR, a satisfied result on the property P Absent Before R imply that $E_{tp}(tmp, press)$ will not reach state Composing before $E_t(tmp)$ occur. In the example application, this implies that $E_t(tmp)$ is never the initiator of the first event occurrence of type $E_{tp}(tmp, press)$.

4.4.5 Pattern: P Absence, Scope: After S

Selecting the *Absence* pattern together with the *After S* scope creates a property that is satisfied if p is never satisfied after s is satisfied.

The generated test-automaton $\mathcal{A}_\mathcal{T}$ for property P Absence After S shown in Figure 11 together with $\varphi = \exists\Diamond\mathcal{A}_\mathcal{T}.Fail$ forms a contradiction proof. Note that the property does not check whether p is satisfied before s or not, only if p is satisfied after s is satisfied for the first time.

If s is defined as R_{vOpen}.EXECUTING and p is defined as $E_{vOpened}$.OCCUR, a satisfied result on the property P Absent After S imply that $E_{vOpened}$ is not triggered by executing the action of R_{vOpen}.

4.4.6 Pattern: P Absent, Scope: Between R and S

Selecting the Absence pattern together with the scope *Between R and S* creates a property that is satisfied if p is never satisfied in an interval starting when r is satisfied and ending when s is satisfied.

The generated $\mathcal{A}_\mathcal{T}$ is shown in Figure 12 and $\varphi = \exists\Diamond$-$\mathcal{A}_\mathcal{T}$.Fail. The verification property is a contradiction proof.

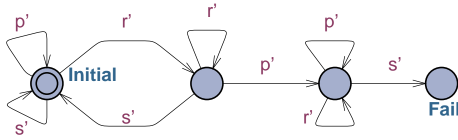

Figure 12: $\mathcal{A}_{\mathcal{T}}$ for Pattern: P Absent, Scope: Between R and S

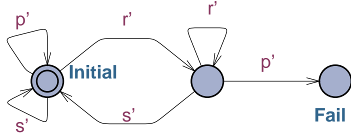

Figure 13: $\mathcal{A}_{\mathcal{T}}$ for Pattern: P Absent, Scope: After R Until S

To reach $\mathcal{A}_{\mathcal{T}}$.Fail, $p\prime$ must be true between $r\prime$ and $s\prime$. The property does not concern whether $p\prime$ is true before $r\prime$ or after $s\prime$, only if it is possible for $p\prime$ to be true in an interval starting with $r\prime$ and ending with $s\prime$.

If p is defined as $E_t(tmp)$.OCCUR, r is defined as R_{vOpen}.-EXECUTE and s is defined as R_{vClose}.EXECUTE, a satisfied result received by running the property P Absent Between R and S imply that $E_t(tmp)$ never occur in an interval started when R_{vOpen} execute its action and ending when R_{vClose} execute its action.

4.4.7 Pattern: P Absent, Scope: After R Until S

The scope After R until S is similar as the between scope, the difference is that the end of the scope is not closed, i.e. S is not required to be satisfied.

The $\mathcal{A}_{\mathcal{T}}$ for P Absent After R Until S shown in Figure 13 together with $\varphi = \exists\Diamond\mathcal{A}_{\mathcal{T}}$.Fail forms contradiction proof. The property is satisfied if $p\prime$ is true after $r\prime$ and before $s\prime$ is true. However, unlike the between scope, $s\prime$ is not required to be satisfied after $p\prime$ for the property to be satisfied.

4.4.8 Pattern: P Responds to Q, Scope: Global

The P Responds to Q Global property is satisfied if whenever Q is satisfied, P is eventually satisfied. Note that P can be satisfied even if Q is not satisfied and that one P can be responding to several Q.

The generated $\mathcal{A}_{\mathcal{T}}$ is shown in Figure 14 and the $\varphi = \exists\Diamond(\theta$ and $\mathcal{A}_{\mathcal{T}}$.Fail) is a contradiction. If it is possible to reach $\mathcal{A}_{\mathcal{T}}$.Fail simultaneously as the model is in the stop state θ, p has not always responded to q.

An alternative approach is to require p to respond to q within a limited time period. The requirement of bounded response times are previously modelled in [21]. If p is required to answer q within a bounded response time, a finite

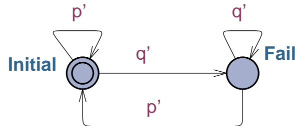

Figure 14: $\mathcal{A}_{\mathcal{T}}$ for Pattern: P Responds to Q, Scope: Global with a stop state θ.

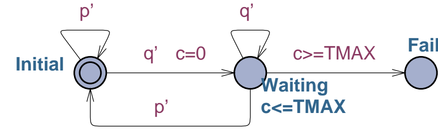

Figure 15: $\mathcal{A}_{\mathcal{T}}$ for Pattern: P Responds to Q, Scope: Global within time TMAX.

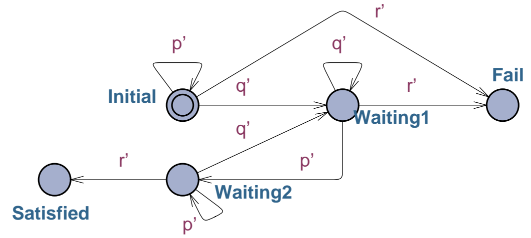

Figure 16: $\mathcal{A}_{\mathcal{T}}$ for Pattern: P Responds to Q, Scope: Before R

sequence of primitive events in the environment is not required. The $\mathcal{A}_{\mathcal{T}}$ modelling bounded response time shown in Figure 15 together with $\varphi = \exists\Diamond\mathcal{A}_{\mathcal{T}}$.Fail forms a contradiction proof.

The $\mathcal{A}_{\mathcal{T}}$ for modelling bounded response times is an extension of the automaton modelling P responds to Q with a stop state. The new automaton is extended with a clock, c and a new location Fail which is reached when the maximum time between Q and R has expired.

The clock c is reset when $q\prime$ is satisfied, the invariant $c <= TMAX$ in location Waiting ensure that the model will leave location Waiting within TMAX time units. If $p\prime$ has not responded to $q\prime$ within TMAX time units, location $\mathcal{A}_{\mathcal{T}}$.Fail is entered.

If q is defined as $E_{tp}(tmp)(press)$.OCCUR and p is defined as R_{vOpen}.ACTIVATED, p responds to q within 10 time units is satisfied if R_{vOpen} is always activated within 10 time units after $E_{tp}(tmp)(press)$ occurs.

4.4.9 Pattern: P Responds to Q, Scope: Before R

Figure 16 shows the $\mathcal{A}_{\mathcal{T}}$ for the pattern P Responds to Q Before R. This property is satisfied if predicate p responds to predicate q before predicate r is satisfied for the first time. Note that a satisfied result on this property does not guarantee that R holds after P has responded to Q, only that R never holds before P has responded to Q and that P always respond to Q before R is satisfied for the first time.

The automaton $\mathcal{A}_{\mathcal{T}}$ for modelling P Responds to Q Before R is shown in Figure 16. It is an extension of the $\mathcal{A}_{\mathcal{T}}$ shown in Figure 14. A new location Fail is introduced which is reached if property r is satisfied before q is satisfied for the first time or if r is satisfied between q and r is satisfied. The location Satisfied is reached if p has responded to q before r is satisfied for the first time. The CTL property to run is a contradiction proof where $\varphi = \exists\Diamond\mathcal{A}_{\mathcal{T}}$.Fail.

If q is defined as R_{vOpen}.EXECUTING and p is defined as $E_{vOpened}$.OCCUR and r is defined as R_{vClose}.ACTIVATED. Property P responds to Q before R is satisfied if the event $E_{vOpened}$ occurs in response to R_{vOpen}.EXECUTING before R_{vClose} is activated for the first time.

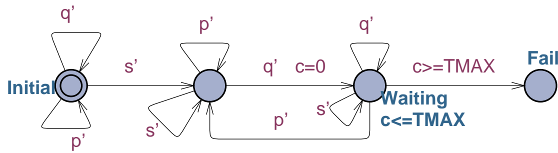

Figure 17: $\mathcal{A}_\mathcal{T}$ for Pattern: P Responds to Q, Scope: After S within T time units.

4.4.10 Pattern: P Responds to Q, Scope: After S

The After S scope for P Responds to Q require a stop state or time limit to be specified. Figure 17 shows the $\mathcal{A}_\mathcal{T}$ for the pattern P Responds to Q After S within TMAX time units. The $\mathcal{A}_\mathcal{T}$ in Figure 17 is an extension of the $\mathcal{A}_\mathcal{T}$ in Figure 15. The automaton in Figure 17 is extended with a new initial location requiring s to be satisfied for the property to reach location Fail.

A non-satisfied result from running the CTL property $\varphi = \exists\Diamond\mathcal{A}_\mathcal{T}$.Fail ensure that q became true after s and p failed to respond to q within the specified time limit.

If p is defined as R_1.ACTIVATED, q is defined as E_1.-OCCUR and s is defined as E_2.OCCUR, the property P Responds to Q After S with time limit $TMAX = 10$ is satisfied if after the first occurrence of an event of type E_2, R_1 always reach state Activated within 10 time units after E_1 has occurred.

4.4.11 Pattern: P Responds to Q, Scope: Between R and S

The property P Responds to Q Between R and S require that P responds to Q within a closed interval, i.e. it requires that in an interval starting with R and ending with S, P always respond to Q.

The $\mathcal{A}_\mathcal{T}$ in Figure 18 for P Responds to Q Between R and S is an extension of the $\mathcal{A}_\mathcal{T}$ modelling P responds to Q Before R.

The Between R and S scope is specified with a stop state and the generated CTL property is a contradiction proof for $\varphi = \exists\Diamond(\mathcal{A}_\mathcal{T}$.Fail) or (($\mathcal{A}_\mathcal{T}$.Waiting1 or $\mathcal{A}_\mathcal{T}$.Waiting2) and θ).

If r is defined as $tmp > 10000$ and s is defined as $tmp < 10000$ and q is defined as $E_t(tmp)$.OCCUR and p is defined as $E_{valveOpen}$, a satisfied result on the property P responds to Q between R and S imply that if a temperature reading occur when the temperature is above 10000, the valve is always opened before the temperature is below 10000.

If p is defined as R_{vOpen}.EXECUTE, q is defined as E_t.-OCCUR, r is defined as $press > MAX/tmp$ and s is defined as $press < MAX/tmp$ a satisfied result on property P Responds to Q between R and S imply that if the relation between pressure and temperature is above its max value (r), an occurrence of the event carrying the temperature value (q) is responded by R_{vOpen} executing its action opening the valve reducing the relation between temperature and pressure below MAX value (r).

The *P Responds to Q After R Until S* property is almost similar as the P Responds to Q Between R and S property. The properties are modelled with identical $\mathcal{A}_\mathcal{T}$, however, the scope After R Until S does not require that S holds after P has responded to Q, and hence, the disjunction $\mathcal{A}_\mathcal{T}$.Waiting2 is removed from φ.

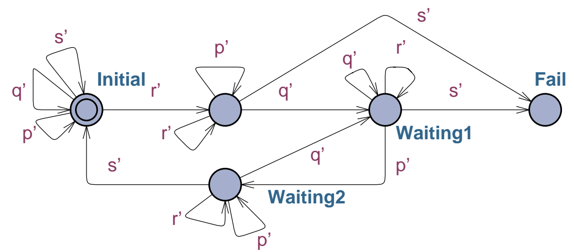

Figure 18: $\mathcal{A}_\mathcal{T}$ for Pattern: P Responds to Q Scope: Between R and S and Scope: After R Until S

4.4.12 General properties

Previous research in analysis of rule based systems addresses general properties that are not covered by the property patterns. For example, confluence and termination have achieved big attention from researchers within the area of active databases (e.g.[2, 5, 29]). A set of rules is confluent if the outcome of simultaneously triggered rules is unique and termination analysis is aiming to ensure that the set of rules will not continue to trigger each other infinitely.

Verifying confluence in REX is ongoing work. For termination analysis, an extra automaton is created, initializing rule-conditions to all different permutations of both true and false values. The extra automaton is run before the automata modelling rules and events implying that termination is checked for all different values on conditions. Verifying termination can be performed in one of two ways using REX. i) An upper limit of the number of allowed rule triggerings is specified in REX and the verification property to ask is whether the maximal number of rule triggerings can be exceeded by any rule. ii) The reachability property to ask is whether the model will always reach a state where all rules and events are in their initial locations. The first approach is useful if the rules are specified for a system with a maximum number of consecutive triggerings or if the search space for the model-checker must be reduced.

Termination analysis for a set of rules specified in REX is related to previous work performed in termination analysis for active databases. The issues of detecting whether the action of one rule triggers other rules and if actions affect condition evaluations of other rules are similar; however, REX does not deal with the state of a database or SQL queries. On the contrary, REX provides a rich set of composite events to be specified decorated with both filters, parameters and time constraints, a combination that is previously not considered in the area of termination analysis for active databases.

4.5 Discussion

REX supports combinations of the property patterns presented in [13] and application specific predicates. The approach provides a structured method for defining application specific verification properties for rule based systems. However, only relying on the patterns for property verification has a price, the ease of use is a trade-off against expressiveness of properties. If practitioners write CTL properties directly in UPPAAL and construct test automata themselves, practitioners may construct properties that are more complex and expressive than what is supported by REX. How-

ever, writing CTL questions and test-automata is an error prone task [22] which requires that the practitioner fully understands the generated timed-automaton model.

According to [14], the property patterns cover 92 percent of the properties that practitioners tend to write for finite-automata models. However, the defined patterns are not enough for the purpose of verifying all useful properties of CEP systems. Previous work has been performed in both extending [25, 19, 10] and fine tune [27] the patterns defined in [13] for specific needs. For CEP systems with ability to define time properties such as expiration times and delays, the patterns obviously need to be extended with time properties, for example as proposed in [19].

The event language supported by REX is to a large extent inspired by Snoop[9]. The aim of REX is to serve as a platform for implementing ideas for automatic verification subsuming a large set of event languages. Snoop serves as a starting point since it contains a set of commonly used operators and well defined consumption policies. However, the language supported by REX can be extended with any operator whose behavior can be expressed as a separate timed automaton.

5. STATE OF PRACTICE AND RELATED WORK

The amount of support for modelling and analyzing design of CEP specification differs between engines. TIBCO, for example, supports an UML based modelling approach where the relationships between events are captured in an UML based model. The behavior of events are captured by a state model representing interaction between applications and services.

Some engines, such as Amit[1], supports developers with a wizard based authoring tool and ability to simulate the definitions for testing purposes while RulePoint have a graphical interface for modelling rules together with support for logging and process monitoring facilities.

We are aware of that the list of exemplified engines is far from complete, however, they represent CEP engines with different approaches for supporting correctness analysis of the CEP specification. The support currently available for modelling debugging and testing in CEP engines are complementary to our approach. It is, for example, not possible to reveal performance of a specific system or monitor ongoing event streams by utilizing REX. Additionally, modelling a system in REX requires some abstractions to allow formal analysis. On the contrary, it is not possible to perform exhaustive tests on a system where all different paths of execution are tested using available test methods. Hence, REX provides means for detecting relations between events and rules that would not be detected using other methods and to check properties that are hard to verify using other approaches.

Current state of practice use various forms of modelling techniques, such as UML, and simulations for detecting design errors. However, this can be an error prone task, since it can be problematic to detect a design error in a complex event pattern simply by visualize and test it. Thus, in addition to being able to visualize and test the complex event patterns, it should also be possible to formally check for design errors. Our approach does not replace, but complements existing modelling and testing approaches.

In Ray and Ray [26] a method is proposed for reasoning about active database applications using a model checker. An example is shown where a small set of rules is transformed to a representation of a finite automaton. In short, each cell in the database is transformed to two variables in the automaton and each rule is transformed to a Boolean variable.

The idea of Ray and Ray [26] is similar with the ideas motivating part of this work, namely utilizing an existing model checker for verifying a set of rules. However, Ray and Ray only address a specific execution model with primitive events stemming from updates of cells in an active database. The transformation from rules to the finite automata is made manually and the resulting finite automata representation is very specific for the current set of rules. In this project a more general approach to the formal verification is taken where different execution models are considered together with composite events and time constraints.

In the area of active databases, previous research has suggested the use of petri-nets for analyzing rule termination (e.g.[29, 17]). In this work, we take an additional step towards performing model-checking on a set of rules. The rules possible to express in our work may contain filters, parameters and composite events with different operators in different contexts. Additionally, in this paper a method for defining properties to check is presented.

The correctness properties possible to express in REX are based on the property patterns defined in [13]. The researchers behind the property patterns has extended the work to a language framework for expressing correctness properties of dynamic Java programs [12]. Although part of this work is based on the patterns specified in [13], this work is tailored to serve rule based languages and allows practitioners to express their correctness properties in terms of rules and events.

6. CONCLUSION

The novelty of this work is an approach for describing verification properties for CEP systems in a high-level specification language. Additionally, mappings from the high-level language to a property possible to formally verify in the timed-automata based CASE tool Uppaal is presented.

Previous work in rule analysis is focused on general properties, such as, termination and confluence. The contribution of this work contains an approach for seamlessly utilizing the model-checker in Uppaal for formal verification of both general and application specific properties.

The high-level property specification is implemented in REX. REX is a research tool supporting graphical specification of rule based applications and automatic transformation of the specification from REX to Uppaal. The ability to specify and transform CEP applications to Uppaal together with the ability to specify verification properties in REX enable non-experts in formal methods to utilize the power of model-checking.

Our approach aims to lower the threshold for non-experts in formal methods to analyze specifications in a model-checker in order to increase software quality. If the application is modelled in REX, the extra time and expertise needed to perform formal analysis is decreased. We argue that the approach of seamlessly including formal verification in a high-level CASE tool is a feasible approach for increasing the utilization of formal verification in practice.

7. ACKNOWLEDGEMENT

This research has been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. http://rewerse.net), CUGS (the national graduate school in computer science) and ARTES (A network for Real-Time research and graduate Education in Sweden)

8. REFERENCES

[1] A. Adi and O. Etzion. Amit the situation manager. *The VLDB Journal*, 13:177–203, 2004.

[2] A. Aiken, J. Hellerstein, and J. Widom. Static analysis techniques for predicting the behavior of active database rules. *ACM Transactions on Database Systems*, 20:3–41, 1995.

[3] R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking for real-time systems. In *5th Symposium on Logic in Computer Science (LICS'90)*, pages 414–425, 1990.

[4] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 26:183–235, 1994.

[5] J. Bailey, G. Dong, and K. Ramamohanarao. Decidability and undecidability results for the termination problem of active database rules. In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 264–273, New York, NY, USA, 1998. ACM Press.

[6] E. Baralis, S. Ceri, and S. Paraboschi. Compile-time and runtime analysis of active behaviors. *Knowledge and Data Engineering, IEEE Transactions on*, 10(3):353–370, 1998.

[7] J. Bengtsson, K. G. Larsen, P. Pettersson, and Y. Wang. Uppaal - a tool suite for automatic verification of real-time systems. *In Hybrid Systems III: Verification and Control*, pages 232–243, 1996.

[8] A. P. Buchmann, J. Zimmermann, J. A. Blakeley, and D. L. Wells. Building an Integrated Active OODBMS: Requirements, Architecture, and Design Decisions. In *Proceedings of the 11th International Conference on Data Engineering*, pages 117–128. IEEE Computer Society Press, 1995.

[9] S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data Knowledge Engineering*, 14(1):1–26, 1994.

[10] M. Chechik and D. O. Paun. Events in property patterns. In *SPIN*, pages 154–167, 1999.

[11] S. Comai and L. Tanca. Termination and confluence by rule prioritization. *IEEE Transactions on knowledge and data engineering*, 15(2):257–270, 2003.

[12] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 205–223, London, UK, 2000. Springer-Verlag.

[13] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In M. Ardis, editor, *Proc. 2nd Workshop on Formal Methods in Software Practice (FMSP-98)*, pages 7–15, New York, 1998. ACM Press.

[14] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 1999 International conference on software engineering (ICSE '99)*, pages 411–421. ACM, 1999.

[15] A. Ericsson and M. Berndtsson. Detecting design errors in composite events for event triggered real-time systems using timed automata. In *First International Workshop on Event-driven Architecture, Processing and Systems (EDA-PS 06) Chicago*, pages 39–47, 2006.

[16] A. Ericsson, R. Nilsson, and S. Andler. Operator patterns for analysis of composite events in timed automata. In *WIP Proceedings : 24th IEEE Real-Time Systems Symposium, Cancun, Mexico.*, 2003.

[17] S. Gatziu and K. R. Dittrich. Detecting composite events in active database systems using petri nets. In *RIDE-ADS*, pages 2–9, 1994.

[18] N. Gehani, H. Jagadish, and O. Shmueli. Compose: A system for composite event specification and detection, 1994.

[19] V. Gruhn and R. Laue. Patterns for timed property specifications. *Electr. Notes Theor. Comput. Sci.*, 153(2):117–133, 2006.

[20] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[21] M. Lindahl, P. Pettersson, and W. Yi. Formal design and analysis of a gear controller. *Lecture Notes in Computer Science*, 1384:281–297, 1998.

[22] A. B. Luca Aceto and K. G. Larsen. Model checking via reachability testing for timed automata. In B. Steffen, editor, *In Proceedings of the 4th International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 263–280. LNCS 1384, 1998.

[23] A. K. Mok, P. Konana, G. Liu, C.-G. Lee, and H. Woo. Specifying timing constraints and composite events: An application in the design of electronic brokerages. *IEEE Trans. Softw. Eng.*, 30(12):841–858, 2004. Member-Aloysius K. Mok.

[24] N. W. Paton and O. Diaz. Active database systems. *ACM Comput. Surv.*, 31(1):63–103, 1999.

[25] D. O. Paun and M. Chechik. Events in linear-time properties. In *Requirements Engineering, 1999, Proceedings. IEEE International Symposium on.*

[26] I. Ray and I. Ray. Detecting termination of active database rules using symbolic model checking. *Lecture Notes in Computer Science*, 2151:266–279, 2001.

[27] R. L. Smith, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil. Propel: an approach supporting property elucidation. In *Software Engineering, 2002, ICSE 2002 Proceedings of the 24rd International Conference on*, pages 11–21. IEEE, 2002.

[28] D. Zimmer, A. Meckenstock, and R. Unland. A general model for event specification in active database management systems. In *Deductive and Object-Oriented Databases*, pages 419–420, 1997.

[29] D. Zimmer, A. Meckenstock, and R. Unland. Using petri nets for rule termination analysis. In *CIKM '96: Proceedings of the workshop on on Databases*, pages 29–32, New York, NY, USA, 1997. ACM Press.