Complex Actions for Event Processing

Steffen Hausmann
Institute for Informatics,
University of Munich
http://www.pms.ifi.lmu.de/
hausmann@pms.ifi.lmu.de

Maximilian Scherr
Institute for Informatics,
University of Munich
http://www.pms.ifi.lmu.de/
scherr@cip.ifi.lmu.de

François Bry
Institute for Informatics,
University of Munich
http://www.pms.ifi.lmu.de/
bry@pms.ifi.lmu.de

ABSTRACT

Automatic reactions triggered by complex events have been deployed with great success in particular domains, among others, in algorithmic trading, the automatic reaction to real-time analysis of marked data. However, to date, reactions in complex event processing systems are often still limited to mere modifications of internal databases or are realized by means similar to remote procedure calls.

In this paper, we argue that expressive complex actions with support for composite workflows and integration of so called external actions are desirable for a wide range of real-world applications among other emergency management. This article investigates the particularities of external actions needed in emergency management, which are initiated inside the event processing system but which are actually executed by external actuators, and discuss the implications of these particularities on composite actions. Based on these observations, we propose versatile complex actions with temporal dependencies and a seamless integration of complex events and external actions. This article also investigates how the proposed integrated approach towards complex events and complex actions can be evaluated based on simple reactive rules. Finally, it is shown how complex actions can be deployed for a complex event processing system devoted to emergency management.

Keywords

Complex Actions, Reactive Rules, Workflows, Complex Event Processing

1. INTRODUCTION

The research reported about in this article has been motivated by new approaches for innovative technologies that improve emergency management in critical infrastructures as it is today.

In emergency management it is crucial to quickly determine the cause and effects of a yet or possibly emerging emergency in a reliable fashion. Therefore, complex event processing is perfectly suited to these kind of applications, as it provides means to quickly analyze the high loads of data which are caused by various sensors of highly technically equipped infrastructures in a timely and continuous fashion. The abstraction provided by complex events is especially desirable during emergency situations, as the number of sensor readings and alarms drastically increases and the large quantity of usually highly redundant information makes it barely possible for humans to grasp the important messages from the avalanche of alarms or to retain a concise overview of the situation. However, complex event processing still lacks expressive capabilities for complex actions which enable sophisticated reactions to the detected situations beyond basic actions that are triggered by reactive rules. But especially during emergency situation it is crucial to support operators by automatically executing suited reactions, so that they have the room needed to consider decisions that cannot be taken by computers. This article aims at filling this gap.

The contributions of this paper are the following: We propose versatile complex actions for event processing that are designed to deal with the particularities of emergency management application, in particular external actions, a rich notion of success and failure of actions, homogeneous integration of queries for events, static and dynamic data, and support of rich conditional actions that enable the specification of versatile workflows as they are needed in emergency management. Moreover we demonstrate how the proposed concepts can be realized on top of a common event processing system.

2. ILLUSTRATING USE CASE

The following example introduces a challenging and visionary scenario for emergency management which is based on the description of a use case scenario proposed in [24].

A metro train is just leaving a station when a fire sensor issues a pre-alarm that indicates the presence of fire in the rear part of the train. The alarm message is immediately sent to the control center of the metro network where it is presented to the operator in charge. By means of the surveillance cameras in the train, the operator can verify that there is indeed a fire in the train and hence enters a confirmation into the emergency management system which leads to a series of automatic reactions. The operation modes of the platform the train on fire is heading to and the tunnels that are leading to the platform are set to "critical" and trains are redirected away from the threatened station.

Meanwhile, the system aggregates alarms from adjacent sensors which, bit by bit, also confirm the presence of the fire from independent measurement methods. The characteristics of the aggregated values are eventually used to determine the location and the size of the fire, which turns out to be rather small in this case. The collected information is then fed into a simulator in order to determine how the toxic smoke that is caused by the fire, the major threat to passengers in this scenario, will propagate in the station once the train has arrived there. Concurrently, the operation mode of the platform is propagated within the station and to affected tracks leading to this station. Moreover, the evacuation of the station's intermediate exchange level is initiated.

When the simulation data is available, the system determines an evacuation strategy for the threatened platform that provides safe escape routes and guides passengers to these routes. Hence, the system proposes to the operator to use the escalator in the front and the stairs in the middle of the platform for the evacuation of passengers. However, the system recommends to avoid the stairs in the rear part of the station for the evacuation, as, according to the simulation, smoke will quickly spread from the rear part of the burning train to the rear part of the platform and eventually to the corresponding evacuation route. Within 1.5 minutes, the operator accepts the proposal of the system and confirms the announcement of the evacuation routes in the station and the adaptation of the ventilation regime to keep the selected evacuation routes free of smoke as long as possible.

Accordingly, the system activates the emergency lightning and uses the public address system to direct passengers to the safe evacuation routes in the front and the middle of the platform. However, when the system tries to put the escalator in the front in an upward running direction it receives events indicating a malfunctioning of the escalator which is interpreted as failure of the inversion of the escalator's running direction. Therefore, the system asks the operator to check whether the escalator can be used for the evacuation anyhow, as the escalator may now be actually standing still, but because the operator is currently busy coordinating the arrival of the fire fighters, the request is neither confirmed nor rejected. Eventually, the system autonomously adapts the evacuation routes based on the results of the simulation, so that passengers are directed away from the possibly faulty escalator towards a safe evacuation route.

When the train on fire finally enters the station 2.2 minutes after the fire detection, the emergency lighting and effective ventilation are already activated. Passengers from the train are directed to the remaining safe evacuation route and can quickly leave the platform, so that within 3 minutes the train is evacuated and 8 minutes later the entire platform is evacuated as well. The system also provides the arriving fire fighters with aggregated information of the current conditions of the infrastructure and the simulated future development of the scenario.

This simplified and idealized scenario demonstrates how intelligent systems can substantially support operators during emergency situations and give them the needed room to consider complex decisions that, at least for legal reasons, cannot be left to fully automatized systems. It is worth stressing, that so far, emergency management in such a situation is based on predefined and static and therefore barely adaptable emergency scenarios that, in some cases like in that of a fire in the Düsseldorf airport in 1996 [1], might result in casualties that even a relatively simple system as sketched above would suffice to avoid. However, in order to provide the desired support, the envisioned system needs to satisfy several requirements that are rarely fully addressed, if addressed at all, by todays event processing systems:

- prevention of so called avalanches of alarms [27] by aggregating alarms referring to the same incident to fewer and more abstract events
- fast and reliable situation assessment by means of complex events
- integration of simulations to enable reactions based on a likely development of the scenario
- support of static data and states to keep track of the dynamic properties of the infrastructure
- integration of external actuators, such as escalators and displays, with support for indirect feedback
- reactive rules integrating queries for events and stateful objects and the execution of actions
- versatile complex actions with temporal dependencies and support of compensating and fallback actions

Listing 1: Schema for `temp-pre-alarm` Events

```
temp-pre-alarm{
 id{identifier},
 reception-time{time-interval}
 area{long},
 sensor-id{long}
}
```

Listing 2: A Complex Event Query

```
and{
 event e: temp-pre-alarm{ area{var A} },
 event f: smoke-pre-alarm{ area{var A} }
} where { {e,f} within 2 min }
```

3. PRELIMINARIES

3.1 Streams of Structured Data

In our system, the content of a stream is represented by means of structured data that is similar to structs known from `C` and `C#`. For emergency management applications, structs are a desirable compromise between the expressiveness of semistructured data and the efficiency of flat tuples: they represent data in a way that is convenient for humans, but at the same time they can be efficiently mapped to flat tuples in a database.

Listing 1 contains a sample schema for `temp-pre-alarm` events. In general, each event is associated with a unique id and a time interval referring to the reception time of the event. In this example, the actual payload of the event is represented by the attributes `area` and `sensor-id`.

For querying data, we have adapted the pattern based querying approach from Xcerpt [8], where queries resemble the data that is queried: a potentially incompletely specified query pattern is matched with the data and variables in the pattern are used to extract relevant portions of the data.

Consider, for instance, the following event query pattern: `temp-pre-alarm{ area{var A} }`. It matches the previously described `temp-pre-alarm` events and simultaneously binds the corresponding area id to the variable `var A`. The same pattern is subsequently used in the complex event query from listing 2 to match the stream whenever smoke and temperature pre-alarms occur in the same area within a time window of two minutes. Note that the variable `var A` appears in both query patterns and therefore only `temp-pre-alarm` and `smoke-pre-alarm` events that occurred in the same area are joined. Moreover, the so called event identifiers `e` and `f` are used in the **`where`** part of the query to concisely constrain the reception times of the according events.

As it is common for event query languages, further temporal relationships between events can be specified by means of Allen's temporal relations [2], such as, `e` **`before`** `f`, etc. They are used in logical formulas along with conjunctions, disjunctions and negation to obtain expressive relationships that support arbitrary combinations of all relations. A detailed description of the employed query language called Dura, which is inspired by the reactive language XChange [7] and the rule based complex event processing language XChange$^{\text{EQ}}$ [6] but has been simplified so as to better meet the emergency management needs, can be found in [14].

3.2 Event Driven Feedback Loop

The architecture of our system relies on an alternating pattern of event detection and action execution: Events are detected by sensors of the infrastructure and are subsequently forwarded to the event processing system by means of an enterprise service bus. The system derives higher level events in a continuous fashion which finally triggers reactive rules and initiates external actions, respectively. The initiated actions are then distributed from the event processing system to the corresponding actuators which actually execute them. Eventually, the effect of the actions modifies the physical state of the infrastructure which is detected by sensors and hence according events are sent to the event processing system.

It is worth stressing that all information is exclusively communicated by means of events. When an action is initiated inside the event processing system, a corresponding *action*`$initiated` event is derived by the system which actually represents the action.[1] Likewise, the success and failure of an action entails corresponding *action*`$succeeded` and *action*`$failed` events. This holds also for states and stateful objects, which are introduced in section 4: updates of states entails *object*`$updated` events, which can be used in event queries to indirectly react to the state change.

4. STATEFUL OBJECTS FOR NON-VOLATILE DATA

As it is motivated in the introduction, having some notion of state or stateful object is desirable for emergency management, as well as for other application which need to keep track of properties that are subject to change in a non-volatile way.

At first glance one might consider using either plain event queries or queries to a database to obtain the desired means to model dynamic data. However, both approaches reveal some limitations which makes them inappropriate for our desired purposes. Events are inherently volatile and they are usually detected when they end whereas states should be detected when they begin.[2] Moreover, emulating states by means of event queries is cumbersome and requires joins over unbounded time window, as all events from the past that characterize the current state need to be taken into account to reconstruct its current values. Entries in a database are also rather unsuitable to model states, as they only represent a snapshot of the current state without any history of past or even future states. Moreover, updates of database tables are highly non-declarative which raises issues for the semantic of queries: the execution order of updates influences their result and in addition the overwritten data may still be needed for further processing. Furthermore, considering states that are valid over a certain time period are natural and desirable, as it makes for instance a difference whether the state of an area has just been updated to "exceptional" or whether the area has been in an exceptional operation mode within the last couple of minutes.

[1]*action* and *object* are just wildcards for the actual names.
[2]Consider, for instance, a state representing an emergency which can only be "detected" when the emergency is over.

To obtain a declarative approach towards states and non-volatile dynamic data, we decided to adopt an idea that is similar to fluents proposed by Kowalski et. al. [15]. Accordingly, stateful objects are data terms that carry a payload of structured data which is only valid over a right open time interval, the so called valid time, that is determined by the time point of the creation and the termination of the stateful object, respectively. The values of stateful objects can be updated by means of internal actions, which actually terminates the currently valid stateful object and creates a new one with the updated values. In this way, the old values of the updated object are not lost, they are just not currently valid anymore. However, they remain valid for the time before the update.

In our use case scenario, the entire metro station is split into a variety of areas which are represented by stateful objects with a unique area id and properties for operation mode (normal, exceptional or emergency), the current smoke concentration in that area, etc. Initially the operation mode for all areas is normal and the smoke concentration is set to 0 percent. When fire is detected, for instance at time t_1, the operation mode of the according area is updated from normal to exceptional. Hence, after the update, there are two stateful objects that are associated with the regarded area. The one stateful object is valid from time t_0 to t_1 exclusively and has its operation mode attribute set to normal, whereas the other stateful object is valid from time t_1 until it is changed and has its operation mode attribute set to exceptional.

Stateful objects are queried in a way that resembles event queries, though the keyword in front of queries is substituted by **`state`**: a query pattern is used to select relevant portions of the data and the according stateful object identifier is used to relate the valid time of the matched stateful objects to, for instance, the reception time of an event. Besides the temporal relations that also can be used in combination with event identifier, two additional relations, which are solely dedicated for stateful objects, are available, namely **`valid-at`** which tests whether the stateful object is valid at a certain time point and **`valid-during`** which tests whether the stateful object is valid sometime in a given time interval.

The query from listing 3 matches whenever an alarm event occurs in an area which is currently, more precisely, when the alarm is actually detected, in normal operation mode. This query can be easily extended to implement a simple filter that sorts out alarms from areas that are already known to be in a critical condition.

Listing 3: Unified Event and Stateful Object Query

```
and{
 event e: alarm{ area{var A} },
 state s: area{ aid{var A}, op-mode{"normal"} }
} where { s valid-at end(e) }
```

It is worth to stress that, in contrast to other approaches, like common ECA rules [21], queries for stateful objects are homogeneously integrated into event queries making queries easier to express and to understand. This is desirable, as queries of both kinds can be arbitrarily combined and the valid time of queried stateful objects can be freely restricted, whereas it is always fixed to the time the event query matches in case of ECA rules. Accordingly, reactive rules in Dura have consists only of two parts, namely a body that contains a combined query for events and stageful objects and a head that contains a complex action. Thus, they resemble EA rules from [13].

5. ACTIONS IN EVENT PROCESSING

5.1 Properties of External Actions

External actions, as they need to be considered for emergency management, have quite different properties than functions from common programming languages that are just executed to accomplish arbitrary computations and, more or less, immediately return their result.

By contrast, external actions are just initiated in the event processing system, they are then sent to and eventually executed by the corresponding actuators. Therefore, the event processing system has at first no knowledge about how long the execution will take, if it was successful or failed, etc. Instead, the system needs to rely on events that are either sent directly by the actuator or that are sent by independent sensors whose events allow to indirectly draw conclusions on the execution result, to determine the actual execution result and the duration of an action. Moreover, the duration of external actions cannot be described by a single point in time, as the equipment needs to actually make a physical reaction which inherently takes a certain amount of time. Therefore, the execution time of an action is indeed a proper time interval.

Another important characteristic of external actions that needs to be accounted for is that they often have no, or no obvious, compensating actions. Think, for instance, of a sprinkler system that has been activated. The water that has been released cannot be simply pushed back into the tube by means of an inverse action. Accordingly, transactions and backtracking cannot be used in conjunction with actions as they are required in our use case.

5.2 Rich Specifications for Success and Failure of Actions

When actions are just initiated in the system but are actually executed by some external actuator, the system needs to rely on some kind of feedback to determine whether the actions have been executed successfully or not. In common approaches, the feedback is directly provided by the actuator in form of positive or negative confirmation messages. However, in our case a more expressive approach is desirable, as some actuators cannot provide the desired feedback directly, the success of an action needs to be interpreted differently in

Listing 4: Specifying Success with Temp. Relations

```
concurrent{
 action a: smoke-extraction{ platform{var P} },
 action b: air-supply{ airflow-to{var P} }
} succeeds on { {a,b} within 20 sec }
```

Listing 5: Specifying Success with Event Queries

```
concurrent{
 action a: smoke-extraction{ platform{var P} },
 action b: air-supply{ airflow-to{var P} }
} fails on and{
 event e: a$initiated{ platform{var P} },
 not event f: airflow{ pf{var P}, vel{var V} }
} where {var V>3, f during extend(e, 1 min) } }
```

different situations, and as the success of composite actions depends on more than the success of its sub-actions.

To illustrate the importance of this issue, consider the following scenario. When fire is detected on a platform of a station the ventilation regime of the entire station is immediately adapted to keep the propagation of smoke limited to the platform where the fire is actually located. Therefore, the fresh air supply of all platforms, except for the one being on fire, is increased to its maximum and ventilators on the platform on fire are activated to extract the released smoke.

Even in this relatively simple scenario it is unclear when the described complex action should be considered as having been executed successfully: should it be successful when all of its sub-actions have succeeded or when the desired airflow is detected that leads to the extraction of smoke? The first interpretation of success is more technical as it checks whether the equipment is working properly whereas the second interpretation is more abstract as it checks whether the action has the desired effect, namely that smoke is extracted from the platform in the intended way. Both interpretations are sound, it just depends on the context of the action to decide which of them is the right one.

To deal with the varying degree of freedom and complexity that is required to specify the success and failure of actions, there are two different ways available, each of them suited to a certain situation. First of all, the temporal relations known from queries for events and stateful actions can be used in the according specifications. In this case, the according action identifier refers to the time interval determined by the initiation and the detection of the success of the referenced action. Therefore, only temporal relation between successful sub-actions can be specified. Alternatively, a common event query can be used to specify the execution result of actions which enables the specification of arbitrary complex conditions for the success and failure of actions which can even incorporate queries of stateful objects. But as it is still desirable to incorporate at least the result of some subactions in this case, special event queries, such as, queries for `a$succeeded` and `a$failed` with `a` being an action identifier, are used to establish relationships to the duration and execution result of the composite action. Moreover, the query `action$initiated` can be use refer to the initiation of the entire complex action.

Listings 4 and 5 exemplary demonstrate by means of the given example both forms of execution result specifications. Note that for the sake of readability in both listings only one of **`succeeds on`** and **`fails on`** is given. Just interpret the missing **`fails on`** as fails when it does not succeed and

Listing 6: Composition with Temporal Constraints

```
concurrent{
 action a: ...
 action b: ...
 action c: ...
 action d: ...
} where { b before c, a before d }
```

likewise interpret the missing `succeeds on` as succeeds when it does not fail. In listing 4, temporal relations are used to specify that the complex action is successful when both subactions are successfully executed within 20 seconds. This can be regarded as a temporally restricted conjunction of both actions. By contrast, listing 5 defines failure of the same action by an event query that matches when up until one minute after the initiation of the `air-supply` action no airflow with a velocity of more than 3 m/s could be detected on the platform. Note that by querying `a$initiated` instead of `smoke-extraction$initiated` it is ensured that only `smoke-extraction` actions are matching the query that have actually been caused by the execution of this complex action. Thus, events that are referring to `smoke-extraction` actions which been initialized by unrelated reactive rules are ignored. Moreover, note that the payload of `a$initiated` contains the values of the parameters that have been passed to the according action at its execution.

5.3 Action Composition with Temporal Dependencies

Composition of actions with temporal dependencies between them is arguably the most common and important requirement for complex actions. Many approaches rely on some kind of composition based operators, such as, sequences of actions and concurrent execution of actions, denoted ; and ||, to realize composite actions as they have been described, for instance, in [3].

Eckert et. al. [6] argue that a clear separation of different query dimensions, such as, event composition and temporal conditions, is desirable for event queries to obtain a highly expressive query language. Likewise, we argue that different aspects of action execution should be kept separate as well. However, composition based operators for composite actions combine several dimensions and hence should be avoided, as it reduces the expressiveness of the language.

Consider, for example, the following complex action: four actions, namely `a`, `b`, `c`, and `d`, should be executed such that `d` is be executed after `a` succeeded and `c` is executed after `b` succeeded. This can be easily realized with composition operators as well as with complex actions as we envision them. Listing 6 illustrates our approach: the execution of the four actions is specified inside the `concurrent` statement whereas the temporal dependencies that impose constraints on the execution order of actions are given in an independent `where` part. In comparison, the same action specified by means of composition operators seems to be more concisely and straightforward: $\big((a;d) \parallel (b;c)\big)$.

However, if the temporal constraints are slightly modified, for instance, by adding an additional constraint so that `c` is only executed after `a` succeeded, the drawbacks of the concise representation of composition operators become apparent. As the execution of actions and the specification of temporal constraints are clearly separated in our approach, one just needs to add the constraints `a before c` to the `where` part in listing 6 to achieve the desired effect. However, the same effect cannot be achieved with composition operators due to their syntactical limitations which are caused by the combined specification of the actual actions and their temporal dependencies.

Listing 7: Complex Action Rules

```
FOR
 adapt-ventilation{ platform{var P} }
DO
 concurrent{
  action a: air-supply{ airflow-to{var P} },
  action b: smoke-extraction{ platform{var P} }
 } succeeds on { {a,b} within 20 sec }
END
```

Beyond this kind of separation of concerns, it is also important to separate specifications that affect the execution order of actions, and can actually be guaranteed by the system, from tests that just determine whether an action was successful. Constraints such as `a before b` can be easily satisfied by the system, as the execution of `b` just needs to be delayed until `a` has been actually successfully executed. By contrast, without additional knowledge on the duration of actions, the constraint `{a,b} within 20 sec` can only be tested after both actions have finished, as the system has no mean to influence the duration of external actions.

Although both kind of constraints could be specified in a common place of the query, we argue that they should be strictly separated as well, in order to achieve clear and unambiguous semantics for complex actions. If specifications for the execution order of actions are mixed with test that just determine the execution result of an action, programmers might easily confuse them and assume that conditions which can only be tested are actually guaranteed by the system. For instance, conditions like `{a,b} within 20 sec` may lead to the conclusion that the according actions are always executed within 20 seconds. Misinterpretations like this can easily cause serious implementation flaws, although they could have been easily avoided by clearly separating specifications from tests.

Accordingly, the `where` part of an action may only contain specifications that influence the execution order of actions which can actually be guaranteed to hold for every arbitrary actions. Moreover, specifications are always conjunctions of formulas in order to obtain clear and deterministic semantics for the execution of composite complex actions. By contrast, the `succeeds on` and `fails on` part may contain arbitrary test, either in the form of formulas or in form of generic event queries, that are used to determine the execution result of the action.

5.4 Complex Action Rules

Complex action rules are similar to declarative rules, which derive new events based on a complex event query. Complex action rules are a mean to give names to complex actions so that the same action can be used in multiple reactive rules or different complex actions by just referring to its name. The advantages of having a notion of complex action rules are similar to those of function, procedure, or method definitions in common programming languages: the development and reuse of code is simplified and thus programs become more robust and easier to maintain.

Listing 7 contains a complex action rule that defines the action `adapt-ventilation` which is derived from the previous example in listing 4. In order to refer to the initiation and the parameters of the defined action, the special event `action$initiated` and the special action identifier `action` can be used. Accordingly, in the context of the complex action rule from listing 7, `init(action)` refers to the initiation time of the complex action `adapt-ventilation` and the event query `action$initiated{ platform{var P} }` can be used, for instance in the `succeeds on` part of the action, to get information on the parameters that have been passed to the action.

5.5 Conditional Actions with Integration of Arbitrary Queries

Although temporal dependencies between action are highly desirable for complex actions, they can only relate actions according to their execution results. However, as already mentioned, the interpretation of the success of an action varies depending on its purpose and context. Moreover, it is also desirable to relate actions that are going to affect the infrastructure, to the current state of the infrastructure or the surrounding area which they are about to affect. To this end, further means are required which allow for more versatile and expressive relations between actions and the information that is available in the event processing system.

Many approached, among others ECA rules [21] and more sophisticated ones like [20], allow the specification of some kind of preconditions for actions. However, these approaches are limited, as the conditions need to be specified in a reactive rule and are hence not part of the action itself. Although this does not seem to make a difference at first sight, it limits the capability of nesting complex actions, which is further discussed in section 5.6. Moreover, it seems to be desirable to integrate not only queries for static data, but also for stateful objects and complex events.

Accordingly, a coherent integration of actions with queries for events and stateful objects is desirable, as it not only enables test of static and dynamic conditions before the execution of an action, but as it is also a natural extensions of common preconditions that enables very expressive yet concise complex action specifications. To this end, we introduce so called conditional actions which are denoted `IF` *query* `THEN` *action* `ELSE` *action* `END`. The event *query* delays the execution of the subsequently given actions until it matches the stream of events and stateful objects. To be well defined, the *query* needs to be timely bounded with respect to other actions, as otherwise the optional else part could would never be executed. As conditional action are considered to be action as well, a concise specification of nested actions that integrate queries for events and stateful action are feasible.

Note that conditional actions are also suited to realize compensating actions and the abortion of composite workflows. In the first case, a conditional action is used that queries the failure of an action and executes the corresponding compensating action if the query matches. In the second case, conditional actions are used to query if the current workflow has been aborted and only execute the next action if the query does not match the event stream.

Listings 8 and 9 demonstrate how the statement is applied by means of examples from our use case scenario. In listing 8, a conditional action is used to check whether the smoke concentration in a certain area is below a certain threshold,

Listing 8: Using Stateful Objects as Precondition

```
IF state s: area{ aid{var Id}, smoke{var C} }
 where {var C < 0.3, s valid-during ... }
THEN
 action: ...
END
```

Listing 9: Using Events as Precondition

```
concurrent{
 action a: adapt-ventilation{ platform{var P} },
 IF not {
   event e: airflow{ pf{var P}, vel{var V} }
  } where {var V>3, e during extend(a, 1 min) }
 THEN
  action: ...
 END
}
```

before an action is executed. This is valuable, for instance, to check whether there will be smoke in a certain area within the next several minutes before an evacuation route is announced that crosses the area.

Listing 9 is dedicated to the different interpretations of success for the `adapt-ventilation` action. It refers to the complex action rule from listing 7 which defines the success of the action by means of the execution result of its sub-actions. However, in this context, we would like to consider the actual effect of the `adapt-ventilation` instead, and thus the success as it has been defined in listing 7 is only of limited suitability, as it is only indirectly related to the effect of the action. Therefore, instead of relying on the predefined success of the action, a conditional action is used that determines whether the effect of the action has been achieved, that is, whether the airflow has developed as expected, and triggers some kind of compensating action when necessary. Note that, adapting the complex action rule for `adapt-ventilation` actions may be undesirable or even impossible, as it may break other rules or as the rule may be located in a module of the program that is not accessible for the programmer.

5.6 Specifying Composite Workflows

The strength of our approach lies in the integration of events, stateful objects and actions which enables natural and concise specifications of sophisticated workflows. In the following, a more extensive and challenging example from our use case is described, that combines the different constructs that have been discussed so far.

Listing 10 specifies an action that is responsible for the announcement of a given evacuation path. In doing so, the action should first check if the path is actually safe, that is, whether it is free of smoke, before it announces it to the passengers. However, when the system distinguishes that there is or will be smoke, it should request a decision from the operator that determines whether the path should be used anyway.

To this end, the stateful object `ev-path` is queried to obtain the areas that are traversed by the evacuation path with the id `var Path`. Moreover, the stateful object `area` is queried to determine whether there *will be* smoke within the next three minutes. Note that `area` contains not only data that is currently valid, but moreover data that will be valid according to the simulation. Accordingly, the semantic time, which determines when the stateful object is deemed

Listing 10: Specification of Nested Workflows

```
FOR
 evacuate-area{ path-id{var Path} }
DO
 IF and{
   state s: ev-path{ id{var Path}, successor{var AreaOnPath} },
   state t: area{ aid{var AreaOnPath}, smoke{var Conc}, semantic-time{var SemTime} }
  } where {
   s valid-at init(action),
   var SemTime valid-during extend(init(action), 3 min), var Conc >= 0.3
  }
 THEN
  action a: request-route-confirmation{ path{var Path}, hazardous-area{var AreaOnPath} },
  IF event e: route-confirmation{ request-id{action a} }
     where { {e,a} within 1 min }
  THEN
   action: announce-ev-path{ id{var Path} }
  END
 ELSE
  action: announce-ev-path{ id{var Path} }
 END
END
```

to be valid according to the simulation, is considered for this purpose. If there seems to be no smoke on the evacuation route in the near future, it is subsequently announced to the passengers. Otherwise, a decision is requested from the operator by means of the `request-route-confirmation` action and if the operator nevertheless acknowledges the announcement of the route within one minute, it is eventually announced.

6. IMPLEMENTING COMPLEX ACTIONS

6.1 Assumptions on the Underlying System

The properties of the event processing system that is used under the hood to implement complex actions are substantially influencing the properties and boundaries of complex actions.

Recall from section 3.2 that by design, all information is communicated by means of events. So when an action is executed by a reactive rule, internally an *action*`$initiated` event is derived instead which actually represents the action. This event carries a unique id for the action, the payload of the action, that is, the parameters that have been passed to it, and the initiation time which corresponds to the reception time of the event. Likewise, the success and failure of an action is represented by corresponding events as well.

An influential property of the system is that there cannot be made any assumptions about when a certain query will be evaluated. Note that this limitation also hold for most, if not all, event processing system, as sophisticated methods known from real time systems are required to obtain accurate guarantees for the latency of single event types. As a consequence, upper bounds for the time when a certain event will be derived, and hence when a certain action is initiated, cannot be made, as the required latency estimation is inherently unavailable in the underlying event processing system. Accordingly, only lower bounds for the initiation time of actions can be guaranteed, such as, action `a` is not initiated before a certain time point has passed.

A remarkable difference from other approaches is the way in which non-monotonic operators, such as negation, can be used in queries. As usual in the area of complex event processing and relational databases, negation is implemented by negation as failure. However, in contrast to many other approaches [10, 16, 11], negation does not need to be restricted to time windows of a predefined width. It is indeed sufficient to specify just an upper time bound to enable the evaluation of the negation. The companion paper [5] addresses the detail of the system-level implementation.

6.2 Temporal Dependency Analysis

In the `where` part of a query versatile constraints for the execution order of events can be specified by the programmer. However, at compile time we need to examine whether the specified constraints can be actually guaranteed by the runtime system.

To this end, we distinguish time points that refer to the occurrence of events and are merely observable from time points of action initiations which can be influenced by the system within the boundaries that are tolerated by the constraints on the action. Accordingly, for any action identifier `a`, `init(a)` refers to the initiation time of the action which can be influenced by the system, `succ(a)` and `fail(a)` refer to the time when the success and failure of the action is detected which can therefore just be observed, and for any event or stateful objects identifier `i`, `begin(i)` and `end(i)` refers to the begin and end of the event or stateful object which can just be observed either. Note that values of time points that can just be observed are determined by the occurrence of events and therefore their exact value cannot be known in advance.

Based on these notions we can elaborate a simple yet effective syntactical criteria that determines whether the conditions of a `where` part can actually be guaranteed by the system. In the following, we will consider formulas which are build from atoms that have the form $tp_1 + d \leq tp_2$ whereby tp_1 and tp_2 denote time points that are specified by the means described above and d is a positive duration. Note that Allen's relations can be transformed into conjunctive formulas that comply with this form of atoms.[3] For instance, `a before b` can be represented by $\texttt{succ(a)} < \texttt{init(b)}$, `a during b` by $\texttt{init(b)} < \texttt{init(a)} \wedge \texttt{succ(a)} < \texttt{succ(b)}$, etc. Accordingly, specifications from the **`where`** part of an action can be represented by a conjunction of such formulas. Naturally, not every formula represents a valid condition that can be guaranteed by the system.

Now, consider the following formulas that describe certain characteristics of actions and their relations:

$$\texttt{init(a)} \leq \texttt{succ(a)} \quad (1)$$
$$\texttt{init(a)} \leq \texttt{fail(a)} \quad (2)$$
$$tp_1 + d \leq \texttt{init(a)} \quad (3)$$

Equations 1 and 2 specify that actions may only succeed or fail after they have been initialized and equation 3 specifies that the initiation of an action may be an upper bound of an arbitrary time point tp. This seems to characterize the properties of the underlying event processing system pretty well, as only the initiation of an action may be a (positive) upper bound, except for the success and failure of actions which inherently occur after the initialization of the corresponding action and can therefore be guaranteed as well.

One could further relax the conditions and allow negative durations for d. However, this does not have any implications for practical purposes, as one cannot know the value of tp_1 before the corresponding event actually occurs and hence one cannot initiate the action in advance, although the constraint would allow it.

One could furthermore argue that $\texttt{succ(a)} \leq \texttt{succ(b)}$ can be guaranteed although it does not correspond to the given formulas, as one just needs to wait for the success of `a` before `b` is initialized. However, this implicitly changes the condition to $\texttt{succ(b)} \leq \texttt{init(b)} \wedge \texttt{init(b)} \leq \texttt{succ(b)}$ which indeed conforms to the form of the given formulas but additionally constraints the initiation of action `b`. However, as this implicitly modifies the semantics of the complex actions and furthermore can be indeed explicitly specified if desired we do not consider such kind of constraints.

Thus, the three equations actually precisely describe the conditions that can be guaranteed by the event processing system. Therefore, each atom of the according conjunctive formula needs to resemble one of the equations to be a valid constraint for the **`where`** part of an action.

A similar problem known as simple temporal problem with uncertainty [25, 19] also investigates under which conditions plans for the execution of actions with unknown duration can be guaranteed. However, approaches from this area focuses on the dynamic execution and adaption of execution plans during runtime and often require upper bounds for the duration of actions. Nevertheless, it seems to be desirable to investigate how result from their work can be applied to our issues. For instance, if the duration of actions can be determined by the form of the success and failure specifications, it seem feasible to transfer some result. However, this is subject to future investigations.

6.3 Rewriting Complex Actions to Reactive Rules

By design, our system only supports reactive rules that concurrently initiate several actions when the queries in their body match. So in order to obtain support for complex actions as they have been described in this paper we need to express them by these rather limited means. Note that in the following we will just focus on complex action rules, as anonymous complex actions that are specified in reactive rules can be easily transformed to complex action rules and the same methods as they are discussed below can be applied to translate conditional actions as well.

[3]In addition, formulas like $tp_1 + d < tp_2$ are also allowed and actually required, but for the sake of simplicity, this further kind of formulas is not explicitly described in this article.

Listing 11: An Abstract Complex Action Rule

```
FOR
 ca{ x{var X}, y{var Y} }
DO
 concurrent{
  action a: a{ param{var X} },
  action b: b{ param{var Y} }
 } where { succ(a)+30 sec <= init(b) }
   succeeds on { succ(b)-init(a) <= 2 min }
   fails on and {
    event e: action$initialized{}
    not event f: action$succeeded{}
   } where {end(f)-end(e) <= 2 min}
END
```

The basic idea to obtain support for complex action rules is rather simple: as all information related to actions is communicated by means of events, we can split complex action rules into several independent reactive rules. Appropriate queries for events entailed by the execution of the corresponding actions are then used to make sure that the constraints that were specified for the complex actions are actually satisfied. So basically, each action from the **`concurrent`** part is moved to its own reactive rule and the **`succeeds on`** and **`fails on`** parts are translated to declarative rules that derive the according *action*`$succeeded` and *action*`$failed` events.

Listing 11 contains a complex action rule which is intentionally kept very abstract in order to obtain concise examples. It executes two actions `a` and `b`, whereby `b` should be initiated at least 30 seconds after `a` has been successful. Moreover, the specified action `ca` is considered successful when `b` is detected to be successful at most 2 minutes after `a` has been initialized and it fails if it has not been successful within 2 minutes. In the following, this complex action rule will be translated into four new rules: a reactive rule that executes `a` when `ca` has been initialized, another reactive rule that executes `b` when `a` has been successful and 30 seconds have passed, and finally two declarative rules, one that derives that `ca` has been successful when `a` was initiated and in addition `b` has been successful within at most two minutes and a second one that derives that `ca` failed if it has not been successful within two minutes.

The first reactive rule executes `a` and hence requires the value of the parameter `x`, which has been passed to the action on the execution of `ca`. Therefore, the reactive rule queries the `ca$initiated` event in its body in order to obtain the required value. The second reactive rule needs, in addition to the parameter `y`, also information on the exact time point of a's success, as the condition `succ(a)+30 sec <= init(b)` specifies a lower bound for the initiation of `b`. Therefore, a query for `a$succeeded` events is added to the body of the rule whereby the time point of the actions success can be reconstructed by means of the occurrence of the according event, that is `end(a-succ)`. Moreover, the delayed execution of `b` is realized by means of a well chosen negated

Listing 12: Translation of the `concurrent` Part

```
ON
 event: ca$initiated{ id{var Id}, x{var X} },
DO
 action a: a{ param{var X}, prov{var Id} }
END

ON
 and{
  event: ca$initiated{ id{var Id}, y{var Y} },
  event a-succ: a$succeeded{ prov{var Id} },
  not event b-init: b$initiated{ prov{var Id} }
 } where { end(b-init) < end(a-succ)+30 sec }
DO
 action: b{ param{var Y}, prov{var Id} }
END
```

Listing 13: Translation of the `succeeds/fails on` Part

```
DETECT
 ca$succeeded{ id{var Id}, var Params }
ON
 and{
  event: ca$initiated{id{var Id}, var Params},
  event a-init: a$initiated{ prov{var Id} },
  event b-succ: b$succeeded{ prov{var Id} }
 } where { end(b-succ)-end(a-init) <= 2 min }
END

DETECT
 ca$failed{ id{var Id}, var Params }
ON
 and {
  event e: ca$initialize{id{var Id},var Params}
  not event f: ca$succeeded{ id{var Id} }
 } where {end(f)-end(e) <= 2 min}
END
```

query which cannot match any event between the success of `a` and the following 30 seconds, but nevertheless blocks the evaluation of the query within this period and thus also blocks the execution of `b`. Note that the constraint `end(b-init) < end(a-succ)+30 sec` which determines the time for the delay of `b`'s execution is directly derived by negating the constraint in the where part of the according action and substituting time points referring to the execution of actions by event identifiers of the according events.

However, one issue still remains as `b` is executed whenever an `a` action succeeds which satisfies the given time constraints regardless of whether the action has been caused by the complex action rule for `ca` or by any other unrelated rule which just happend to execute `a`. Therefore, additional provenance information that allows to reconstruct the actual cause of an action is considered. Accordingly, the id of the `ca` instance is added to the payload of `a` and hence `ca$initiated` and `a$succeeded` events can be joined to determine whether `a` has been actually caused by the same action instance of `ca`.[4] The respective reactive rules can be found in listing 12.

The declarative rule for deriving the according success events for `ca` is straight forward. An event is derived when the `ca` action has actually been initiated and the events for the initiation of `a` and the success of `b` that have been caused by this very action occur within two minutes. Again, time points referring to the execution of actions are substituted by event identifiers and the according event queries are added to the body of the rule. Moreover, the available provenance information is used to ensure that all events have actually been caused by the same action instance of `ca`.

[4]The additional parameters can be omitted from the according events when they are actually sent to the actuators.

The failure specifications for the complex action from listing 11 is given by means of generic event query. Therefore, the complete query is just copied to a new declarative rule which derives the according `ca$failed` event. Moreover, for all atomic event queries that use action identifier to refer to actions of the corresponding complex action the actual name of the referred action is inserted and the provenance information on these events is joined with the id of the initiation of the complex action. The respective declarative rules are given in listing 13.

There are some technical particulars that have been simplified or left out from the description, such as, the treatment of several equally actions inside a complex action rule, etc. For further details on the transformation of complex actions refer to [22].

7. RELATED WORK

The need for reactive extensions of event processing approaches has already been identified by other authors, most notably in [23] and [20].

In their work on event-driven reactivity, Schmidt et. al. [23] envision a holistic approach which uniformly integrates events, actions, conditions, context and situations. The authors emphasise the need for a notion of context and situations for reactive event processing systems, which can indeed be obtained by means of stateful objects.

There have been efforts in the field of active databases towards transactions that enable composite reactions which are triggered by ECA rules [17, 9]. Lately, this work has been extended to incorporate particular characteristics of event streams into the notion of transactions [26]. However, work in this area mainly focuses on composite database updates that do not incorporate a notion of time for actions.

Engel et. al. [12] investigate extensions for complex event processing towards proactive event-driven applications. The authors propose to integrate prediction and automatic decision making technologies to enable proactive reactions based on future uncertain events. However, simulations as they are required for emergency management are far too complex to be specified and efficiently computed within the event processing system.

Behrends et. al. [4] propose to specify composite actions in ECA rules by means of the CSS process algebra [18]. In this way, reasoning on the properties and the effect of concurrently executed actions is enabled. However, although CSS is a powerful formalism, its plainly formal nature limits the relevance for practical purposes.

A homogeneous reaction rule language with versatile pre/-postconditions for reactive rules, transactional knowledge base updates and means for asynchronous message exchange that enables reactive behavior of the event processing systems is proposed by Paschke et. al. [20].

All presented approaches lack an explicit notion of external actions as they are desirable for emergency management and hence do not provide means that deal with their particularities. Consequently, implementing emergency management applications, although sometimes possible, is very cumbersome in the according systems.

8. CONCLUSION

In this paper we proposed versatile complex actions for emergency management. Although building up from relatively simple components, the overall integration into a ingle language for complex event and complex action processing is innovative and original, extends over previous contributions, and fulfills a need, especially for emergency management applications, so far largely neglected.

The approach to complex actions described in this article contributes to substantially improve emergency management as it is today and enables the design and implementation of innovative new strategies for emergencies in public infrastructures.

Even though the approach to complex actions proposed in this article has been motivated by an emergency management use case, it is more generally applicable to large classes of reactive event processing applications that require both complex events and complex actions of the kind considered in this article.

9. ACKNOWLEDGEMENTS

We thank Marco Bettelini and Nikolaus Seifert for introducing us to their vision of novel and innovative emergency management. We also thank Simon Brodt for countless discussions and his valuable feedback.

This work has been partly founded by the European Commission within the project "EMILI — Emergency Management in Large Infrastructures" under grant agreement number 242438.

10. REFERENCES

[1] Fire Investigation Summary Düsseldorf. Technical report, National Fire Protection Association, 1998.

[2] J. F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, Nov. 1983.

[3] J. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2-3):131–146, May 2005.

[4] E. Behrends, O. Fritzen, W. May, and F. Schenk. Combining ECA Rules with Process Algebras for the Semantic Web. In *Proc. Int. Conf. on Rules and Rule Markup Languages for the Semantic Web*, pages 29–38. IEEE Press, 2006.

[5] S. Brodt and F. Bry. Temporal Stream Algebra. submitted for publication, 2012.

[6] F. Bry and M. Eckert. Rule-based composite event queries: the language XChangeEQ and its semantics. In *Proc. Int. Conf. on Web Reasoning and Rule Systems (RR)*, pages 16–30. Springer, 2007.

[7] F. Bry and P.-L. Pătrânjan. Reactivity on the web: paradigms and applications of the language XChange. In *Proc. Symp. on Applied Computing (SAC)*, pages 1645–1649. ACM, 2005.

[8] F. Bry and S. Schaffert. Towards a declarative query and transformation language for XML and semistructured data: Simulation unification. *Proc. Int. Conf. on Logic Programming (ICLP)*, 2401:255–270, 2002.

[9] S. Chakravarthy, V. Krishnaprasad, Z. Tamizuddin, and R. H. Badani. ECA Rule Integration into an OODBMS: Architecture and Implementation. In *Proc. Int. Conf. on Data Engineering (ICDE)*, pages 341–348. IEEE Press, 1995.

[10] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White. Cayuga: A General Purpose Event Monitoring System. In *CIDR*, pages 412–422, 2007.

[11] M. Eckert, F. Bry, S. Brodt, O. Poppe, and S. Hausmann. Two Semantics for CEP, no Double Talk: Complex Event Relational Algebra (CERA) and its Application to XChangeEQ. In *Reasoning in Event-based Distributed Systems*, volume 347 of *Studies in Computational Intelligence*, pages 71–98. Springer, 2011.

[12] Y. Engel and O. Etzion. Towards proactive event-driven computing. In *Proc. Int. Conf. on Distributed Event-Based Systems (DEBS)*, pages 125–136. ACM, 2011.

[13] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite Event Specification in Active Databases: Model and Implementation. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 327–338. Morgan Kaufmann, 1992.

[14] S. Hausmann, S. Brodt, and F. Bry. Dura - Concepts and Examples. Deliverable d4.3, Institute for Informatics, University of Munich, 2011.

[15] R. Kowalski and M. Sergot. A logic-based calculus of events. *New generation computing*, 4(1):67–95, 1986.

[16] J. Krämer and B. Seeger. Semantics and implementation of continuous sliding window queries over data streams. *ACM Trans. Database Syst.*, 34(1), Apr. 2009.

[17] D. McCarthy and U. Dayal. The architecture of an active database management system. In *Proc. int. Conf. on Management of Data (SIGMOD)*, pages 215–224. ACM, 1989.

[18] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.

[19] P. H. Morris and N. Muscettola. Execution of Temporal Plans with Uncertainty. In *Proc. Nat. Conf. on Artificial Intelligence and Conf. on Innovative Applications of Artificial Intelligence*, pages 491–496. AAAI Press, 2000.

[20] A. Paschke, A. Kozlenkov, and H. Boley. A Homogeneous Reaction Rule Language for Complex Event Processing. *Proc. Int. Workshop on Event Driven Architecture and Event Processing Systems (EDA-PS)*, 2007.

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

[22] M. Scherr. *Desugaring Dura: Compiling a High-Level Event Processing Language.* Diploma thesis, University of Munich, 2011.

[23] K.-U. Schmidt, D. Anicic, and R. Stühmer. Event-driven Reactivity: A Survey and Requirements Analysis. In *Proc. Int. Workshop on Semantic Business Process Management*, pages 72–86, 2008.

[24] N. Seifert, M. Bettelini, and S. Rigert. Emergency management and rules in control systems of critical infrastructures. Deliverable d3.2 annexe a, ASIT Ltd., 2011.

[25] T. Vidal and H. Fragier. Handling contingency in

temporal constraint networks: from consistency to controllabilities. *Journal of Experimental and Theoretical Artificial Intelligence*, 11(1):23–45, 1999.

[26] D. Wang, E. A. Rundensteiner, and R. T. E. Iii. Active Complex Event Processing over Event Streams. *Proc. VLDB Endowment*, 4(10):634–645, 2011.

[27] D. D. Woods and E. S. Patterson. How Unexpected Events Produce An Escalation Of Cognitive And Coordinative Demands. In P. A. Hancock and P. A. Desmond, editors, *Stress, Workload, and Fatigue*, pages 290–304. Lawrence Erlbaum Associates, 2001.