SEVENTH FRAMEWORK PROGRAMME
THEME SECURITY
FP7-SEC-2009-1

Project acronym: *EMILI*
Project full title: Emergency Management in Large Infrastructures
Grant agreement no.: 242438

D4.2 Reactive Rules for Emergency Management

Due date of deliverable: 30/06/2010
Actual submission date: 31/08/2010

Revision: Version 1

Ludwig-Maximilians University Munich (LMU)

Project co-funded by the European Commission within the Seventh Framework Programme (2007–2013)		
Dissemination Level		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Author(s)	Simon Brodt, Steffen Hausmann, Francois Bry
Contributor(s)	

Index

Preface

The goal of the following survey on Event-Condition-Action (ECA) Rules is to come to a common understanding and intuition on this topic within EMILI. Thus it does not give an academic overview on Event-Condition-Action Rules which would be valuable for computer scientists only. Instead the survey tries to introduce Event-Condition-Action Rules and their use for emergency management based on real-life examples from the use-cases identified in Deliverable 3.1. In this way we hope to address both, computer scientists and security experts, by showing how the Event-Condition-Action Rule technology can help to solve security issues in emergency management. The survey incorporates information from other work packages, particularly from Deliverable D3.1 and its Annexes, D4.1, D2.1 and D6.2 wherever possible.

1 Reactivity in Emergency Situations

Emergency management in large infrastructures as examined in EMILI can basically be divided into three main tasks: First to continuously gather data about the state of the infrastructure, second to recognize critical and emergency situations out of this data and third to initiate an appropriate reaction. In all three fields human operators need support by technical systems, due to the size and complexity of the infrastructure, the amount of data, the number and severity of decisions to be taken in a short time, and the coordinated execution of a whole bunch of (re)actions.

Supervisory Control and Data Acquisition systems (SCADA) are build for the first of these tasks. The second task could be taken by Complex Event Processing (CEP) based on the data delivered by the SCADA system. These two aspects of an integrated system for emergency management are discussed in Deliverable D4.1. In this survey we want to focus on the third, the reactive part.

Remark: The above statement is not quite fair to SCADA systems. Most SCADA systems have at least some capabilities to detect situation and to initiate reactions. The keywords "Supervisory" and "Control" even suggest that they should have. The concrete capabilities on this field widely vary. Many SCADA systems provide comprehensive functionality for controlling the normal operation of the infrastructure. Some also care emergency treatment, but mostly not in the highly integrated and adaptive way which is striven for in EMILI. However all current SCADA systems have in common that the detection and reaction functionality is realized in a proprietary, non-generic way which tends to be hard to maintain and difficult to adapt to infrastructures which are not essentially identical with the one the SCADA system has been designed for. In EMILI complex event processing (CEP) and Event-Condition-Action rules (ECA) are used to realize situation detection and reactions in a generic way that is easily applicable to different kinds of infrastructures. This does not imply that existing functionality has to be re-implemented. EMILI will have a flexible architecture which allows for an easy integration and reuse of existing systems.[1] The combination of CEP and ECA rules with a current SCADA system used for data acquisition can be regarded as a new generation of (generic) SCADA systems. However we prefer the view of CEP and ECA rules working *on top* of (current) SCADA. There are two reasons: First CEP and ECA rules have a more general scope than SCADA systems. Second our view reflects the actual system architecture within EMILI. For a general introduction to SCADA systems see Deliverable D4.1, concrete SCADA systems are described in Deliverable D3.1 and Deliverable D5.2 contains detailed information on the interaction with sensors.

[1] Re-implementation is not required and the EMILI technology could be used to "fill the gaps" of an existing system. However it might be advisable to consider a stepwise re-implementation of existing functionality to increase the homogeneity, maintainability and reliability of the system.

2 Reactivity in the Use-Cases

All three use-cases described in Deliverable D3.1 share a need for reactions based on the detected events and situations. Under normal operation conditions the reactions are quite simple and usually local to some subsystem in an infrastructure. This is completely different in case of emergency. A comprehensive reaction to an emergency typically involves several subsystems of an infrastructure and a whole bunch of actions which have to be carried out in a coordinated way and with an precise timing.

The fire scenario in the airport use-case for instance mentions the following (non-primitive) actions as response to a fire detection [Deliverable D3.1 Annex A]:

- alarm messages (nearest officer, safety and security center, fire department, airlines . . .),
- activate fire suppression (sprinklers, water curtains, compressed gas)
- change ventilation regime (smoke extraction)
- close fire doors
- unlock doors on evacuation routes (access control system)
- inform people about emergency and available evacuation routes by public address system (acoustic announcement, info screens, personnel)
- activate emergency lightning on evacuation routes,
- redirect/cancel flights

The list of actions in the metro fire scenario is similar [Deliverable D3.1 Annex B]:

- alarm messages (local staff, emergency control center, external intervention services, traffic control center, . . .),
- activate fire suppression
- change ventilation regime (smoke extraction, fresh air supply)
- close fire doors, lower smoke curtains
- close entrances for public
- redirect/stop escalators and elevators
- inform people about emergency and available evacuation routes by public address system (acoustic announcement, info screens, personnel)
- activate emergency lightning on evacuation routes,
- redirect trains

Appropriate reactions are essential for the power-grid use-case as well. However the focus of the selected scenario is on the *detection* and *explanation* of complex events and situations. Particularly for the purely electronic part of the power-grid use-case, systems for planning and performing complex sets of actions are already in operation. They are realized in a very specific,

hard-coded fashion, in contrast to the generic approach striven for in EMILI. The detection and explanation of complex events and situations in a timely manner is the more urgent problem at this field, though.

In the following we recapitulate one aspect of reactivity in the airport and one in in the metro fire scenario. The examples in Sec. 3 and in Sec. 5 are based on these descriptions.

2.1 Airport Use-Case – First Phase Reactions to Fire Detection

A comprehensive reaction to a fire is very complex (see above). However the first phase between receiving the initial alarm and starting the full-scale reaction already involves a number of actions which are comparatively simple. Therefore they are well-suited for introducing Event-Condition-Action rules. The following description is adopted from Deliverable D3.1 Annex A Sec. 7.2 and is the basis for the examples in Sec. 3:

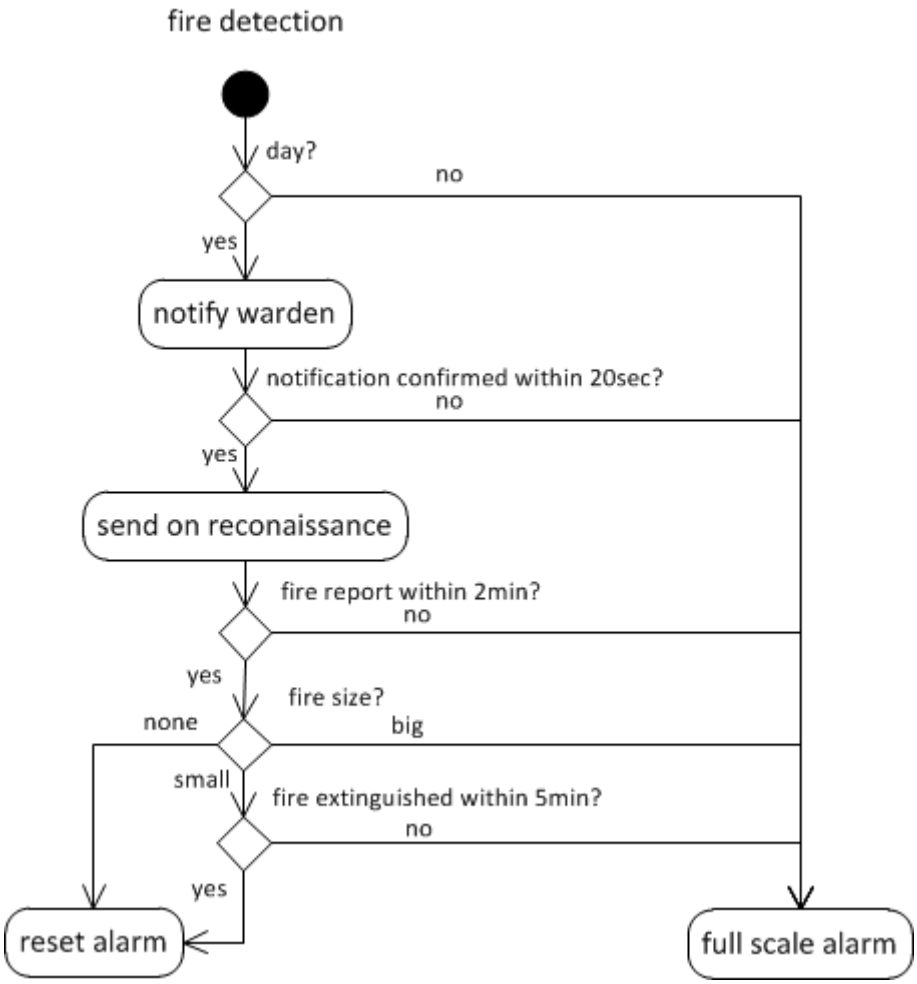

Figure 1: First reaction phase after fire detection

Things start when a (potential) fire is detected, e.g. by a smoke sensor or by somebody pressing an emergency button, and when the corresponding message/event enters the system.

At night, when nobody is present close to the scene and when false-alarms are unlikely, the fire detection event directly triggers a full-scale-alarm meaning that the Safety and Security Operations Center and the fire brigade are informed and fire suppression is (partially) activated.

During the day, when local staff is available and false-alarms happen more or less frequently, a full-scale-alarm is not triggered directly in order to limit the impact of false-alarms and small incidents to the operation of the airport. Instead a verification/reconnaissance and immediate response phase is initiated first.

This means that the responsible area warden is notified either by some acoustic and/or visual signal in a local control room or by a message to some mobile device. If the warden confirms the reception of the notification he/she is sent to the scene for assessing the fire (none/small/big) and for extinguishing the fire immediately if possible. If the warden does not confirm the notification within a few seconds (e.g. 20 s) or is not able to extinguish the fire quickly (e.g. 5 min) then full-scale-alarm is raised. The complete procedure is illustrated in Fig. 1.

The example above is a conscious over-simplification: Reactive rules are significantly more complex in emergency management. Simplified examples are appropriate here, though. Indeed, concepts are better explained on simple examples than on more complex ones that would convey too much application details.

2.2 Metro Use-Case – Ventilation Regime in Case of Fire

The above description from the airport use-case could probably be applied to the metro use-case with minor adaptions. However we want to look at another aspect of the metro fire scenario, the co-operation of the ventilation, the fire suppression and the public address system for the successful evacuation of a metro station in case of a fire. This example illustrates the need for precise timing/coordination of actions (see Sec. 3.8) and for adaptive reactions (see Sec. 3.14). The following description is adopted from Deliverable D3.1 Annex B:

In case of a fire in a metro station the most important and challenging task is the fast and safe evacuation of the passengers. The ventilation, the fire suppression and the public address system take a key role in this task: The ventilation system has to keep escape routes free of smoke, the fire suppression system should limit the fire intensity and stop or slow down the expansion of the fire and the public address system is needed to inform the passengers about the emergency, to communicate the escape routes and to guide the passenger flow along these routes. The coordination of all three systems is critical for the success of the evacuation.

The biggest problem is to choose the right combination of evacuation routes, ventilation regime and fire suppression measures. The choice depends on firstly the properties of the fire like location, intensity and smoke production rate, secondly the ability of the ventilation system to keep a route free of smoke for a sufficiently long period of time and thirdly the capability of the fires suppression system to prevent the fire of spreading towards an escape route too quickly.

The above decision is really complex in each of its dimensions.

The ventilation regime for example is essential for keeping the evacuation routes free of smoke. This is important as smoke is often toxic and furthermore dramatically reduces the sight which could result in passengers loosing their way and being trapped within the station. Unfortunately even the best ventilation system is unable to keep the whole station free af smoke during a fire. Thus a trade-off has to be made between parts of the station which can be kept free of smoke and parts which may even fill faster due to the ventilation (see Deliverable D3.1 Annex B Sec. 11.6 b).

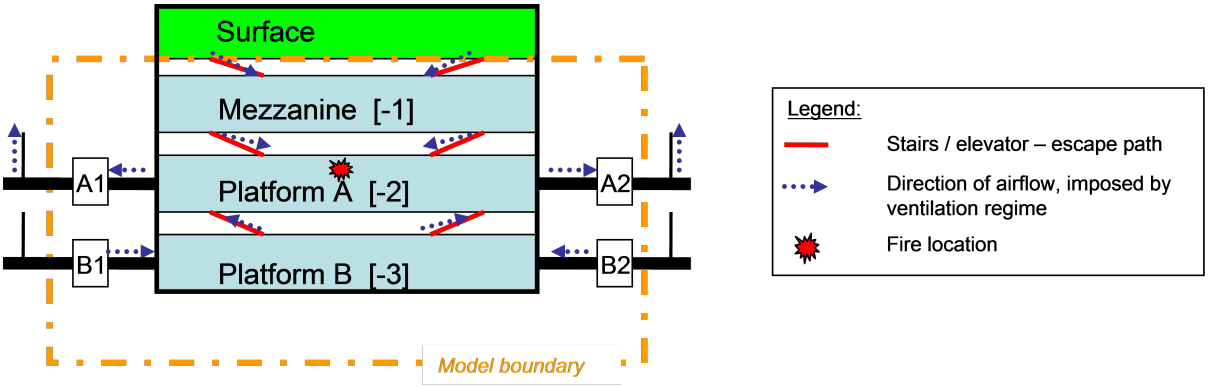

Figure 2: Effect of ventilation regime on airflow in station (from D3.1 Annex B)

Consider the situation in Fig. 2 (Deliverable D3.1 Annex B Fig. 8.3): There is a fire on level -2. Obviously smoke propagation in level -2 is inevitable. The propagation of smoke to the other levels can be prevented by activating the extraction fans at both sides of the tunnel of level -2. However activating these fans leads to a faster smoke propagation within level -2 and shortens the time available for the evacuation of this level. Thus on the one hand the fans should not be activated too early, i.e. not before people had enough time to leave level -2, but other hand necessarily have to be activated before smoke is able to enter the other levels.

The fire suppression system has to care about the progress of the evacuation, too. The activation of sprinklers for instance could help to control the fire, i.e. to limit its intensity and expansion for protecting the evacuation process and for preventing avoidable damage on the infrastructure. However activating the sprinklers too early could have serious consequences for the evacuation: People may tumble due to a slippery ground, the public address system may be damaged by the water or people do not follow the instructions as visual displays become hard to read and acoustic announcements may be difficult to understand. Thus activating the sprinklers at the right time is important.

3 Event-Condition-Action Rules by Example

The basic idea of Event-Condition-Action rules (ECA-rules) is to define reactive behavior by specifying which action has to be carried out when an specific event is detected and some additional condition holds. Therefore ECA-rules have the following general structure:

ON E V E N T **IF** C O N D I T I O N **THEN DO** A C T I O N **END**

The intended meaning is that each time when an event matching the event specification E V E N T is detected and the condition C O N D I T I O N is fulfilled at that time the corresponding action defined by A C T I O N is executed.

The event specification E V E N T is a query to an event stream matching certain kinds or combinations of events. We refer to Deliverable D4.1 for this part. The C O N D I T I O N part is a query to non-event data for example in a database or a XML or RDF document. Typically this data is considered to be rather static, i.e. to be at most rarely update, as opposed to the very volatile event data delivered by the monitored system. Within EMILI it might become necessary to introduce more frequently changing data at this point for tracking the state of components or for workflow specifications (see Sec. 3.9). Finally A C T I O N specifies a single or a number of actions like sending an command to some external device, executing an script, or updating some data (e.g. the state of some object or an database entry). The event, the condition and the action part of an ECA-rule are able to share data for example by variable bindings. The variable bindings are generated by the the event and condition queries and can then be used in the action specification. Thus the event and the condition part have an double purpose: They define *when* to react and extract the necessary data for determining *how* to do it.

In this way ECA-rules are able to formalize executable knowledge like workflows or emergency plans. The following pages show a number of examples which introduce and motivate the use of ECA-rules for this purpose.

ECA-rules are inherently different from deductive rules like discussed in Deliverable 4.1 for example. ECA-rules define reactions, i.e. typically have side-effects. In contrast, deductive rules and ontologies focus on drawing conclusions (implicit facts, classification in terminologies), i.e. are free of side-effects. In the beginning of Complex-Event-Processing ECA-rules have often be misused to simulate deductive rules. However ECA-rules and deductive rules are actually complementary concepts [19].

The following examples are based on the first phase reactions to fire detection in the airport use-case as described in Sec. 2.1. As complex events (see Deliverable 4.1) are not in the focus of this document the event part of the ECA-rules in the examples consist only of a single event which actually would be the result of some complex event query. Thus the examples concentrate on the condition and the action part. Starting with extremely simple ones the complexity of the examples steadily increases introducing the different aspects of ECA rules one after the other while showing why each aspect is needed for an appropriate modelling of the description in Sec. 2.1.

3.1 Event-Action Rule

We start with the most simple form of a reactive rule: A single event triggers a single action. The pseudo-code below shows a rule where a full scale alarm is raised (action) as reaction to an unspecific fire detection (event).

```
ON fire_detection DO full_scale_alarm END
```

3.2 Event-Condition-Action Rule

Raising a full scale alarm as reaction to every (potential) fire detection is frequently not the appropriate reaction: A false alarm or a very small fire like a smouldering garbage can, would have the same impact on the operation of the infrastructure as a real emergency. During the day local staff could asses and possibly handle the situation without further help. Therefore an internal alarm instead of a full scale alarm would suffice as a first reaction. Using the two rules below a fire detection causes full scale alarm at night but only an internal alarm during the day.

```
ON
  fire_detection
IF
  is_night
THEN DO
  full_scale_alarm
END
```

```
ON
  fire_detection
IF
  is_day
THEN DO
  internal_alarm
END
```

3.3 Alternatives

The above rules look very independent. But actually they form an alternative. One rule with an explicit representation of the alternative in the syntax by some IF-THEN-ELSE construct is more convenient to write than two rules, as it avoids redundancies in the event and the condition specifications and is a good structuring feature increasing the understandability of the rules. Avoiding replications is also good for execution efficiency: The condition is only tested once.[2] Rules of this kind are sometimes called ECAA rules since they specify an action and an alternative action [10, 32]. An equivalent specification of the above example using IF-THEN-ELSE for the condition part could look like this:

[2] Testing a condition `C` only once in two ECA rules with `C` and **NOT** `C` is of course possible, but requires optimization techniques detecting and exploiting similarities in rules.

```
ON
  fire_detection
IF
  is_day
THEN DO
  internal_alarm
ELSE DO
  full_scale_alarm
END
```

3.4 Turning Rules *On* and *Off*

Intentionally the condition of an ECA rule is meant as additional specification restricting *when* to react and choosing the right reaction. In some applications however the condition has a more natural interpretation: It specifies the state of the infrastructure/system where some rule is *active*. This view is particularly useful in the case where there are significantly different rule sets for different states. The condition `is_day` or `is_night` for example may not only determine the reaction to fire detection but also the behavior of many other parts of the infrastructure like access control, ventilation regime and lighting. In this case a syntax emphasizing the state in which a rule is active may be more natural.

Consider the following example: The first rule specifies the behaviour of the system in night mode the second the behaviour in day mode. Note that in contrast to the specification in 3.2 where both rules are always active, only one of the rules below is active for each point of time.

```
WHEN
  is_night
THEN
  ON
    fire_detection
  DO
    full_scale_alarm
  END
END
```

```
WHEN
  is_day
THEN
  ON
    fire_detection
  DO
    internal_alarm
  END
END
```

The different ways of specifying the desired behavior do not only affect the syntax but also the evaluation strategy: A normal ECA rule is always waiting for a matching event and checks its condition only when such an event is detected. In contrast a ECA rule with activation condition (e.g. specifying some state of the infrastructure) is waiting for a matching event only when the activation condition is fulfilled (e.g. when the infrastructure is in the right state). Thus on the one hand using activation conditions can significantly reduce the number of rules waiting for events. On the other hand the activation conditions have to be monitored continuously, i.e. not only when a matching event arrives, for updating the set of active rules.

3.5 Variables

Up to that point the given examples neglected data which might be carried by the event(s). A fire detection event for instance usually carries information about the location of the fire (e.g. the affected area). The condition may contribute further data and its validity could depend on the data carried by the event(s). Finally the action is often parametrized by the data stemming from the event(s) and/or condition(s). Variables can be used for extracting data in the event and condition part and for sharing the data between all three parts of an ECA rule.

For the following examples lets assume that fire detection events carry an attribute indicating the area where the fire is detected. The variable for extracting this information is denoted `var Area`. In a large infrastructure like an airport different parts of the building may have different opening and closing hours and therefore the day/night mode settings may be area dependent. Therefore the validity of the `is_day` condition now depends on the value of variable `var Area`. Finally the internal alarm is only raised for the affected area. In this way only those people are informed which are responsible and familiar with that area and which are close enough to the scene to intervene quickly.

```
ON
  fire_detection(var Area)
IF
  is_day(var Area)
THEN DO
  internal_alarm(var Area)
END
```

The condition can also be used to query additional information. A warden responsible for some area for instance probably carries some mobile device. Beside the optical or acoustic internal alarm in the control room, sending an additional alarm message to this device might be a good idea. However, for doing so we first need the information which warden is actually responsible for the area. This information is not carried by the fire detection event and needs to be queried within the condition. The second condition `warden(var Area, var Warden)` in the pseudo-code below is used to find the warden (`var Warden`) responsible for the area (`var Area`) where the fire is detected.

```
ON
  fire_detection(var Area)
IF
  is_day(var Area),
  warden(var Area, var Warden)
THEN DO
  send_message(var Warden, "fire alarm in area " + var Area)
END
```

3.6 Multiple Actions

When considering the ECA rules in the last two examples one will notice that the event and the condition specifications are (almost) equal. This is a frequent phenomenon as the reaction to a situation (constituted by the detected combination of events and valid conditions) typically involves more than one action. Thus ECA rules should support specifications with multiple actions. In this way redundant event and condition specifications are avoided and the readability as well as the maintainability is significantly increased. The following (single) rule for example could be used to replace those of the last two examples.

```
ON
  fire_detection(var Area)
IF
  is_day(var Area),
  warden(var Area, var Warden)
THEN DO
  internal_alarm(var Area),
  send_message(var Warden, "fire alarm in area " + var Area)
END
```

3.7 Conditional Actions

Multiple actions greatly help to reduce redundant event and condition specifications, but are only applicable when the event and condition specifications of some rules are completely equal.[3] However there could be rules which are all meant as reaction to the same situation and share most of the event and condition specifications but have some additional condition specific to the corresponding action, like a test whether the action can be carried out for a certain parameter. In this case the rules could not be combined by means of multiple actions as described in Sec. 3.6 and the redundant specifications would remain.

Reconsider the examples in Sec. 3.5 and Sec. 3.6 assuming that not every warden has a mobile device. In the below code a new condition `has_mobile_device(var Warden)` that is testing whether the warden has a mobile device or not is added to the second rule of Sec. 3.5. The effect is that a message is only send when the warden has a mobile device.

```
ON
  fire_detection(var Area)
IF
  is_day(var Area)
THEN DO
  internal_alarm(var Area)
END
```

[3] Reordering of the event and/or condition specifications could be possible depending on the properties of the used language.

```
ON
  fire_detection(var Area)
IF
  is_day(var Area),
  warden(var Area, var Warden),
  has_mobile_device(var Warden)
THEN DO
  send_message(var Warden, "fire alarm in area " + var Area)
END
```

The following extension of the rule from Sec. 3.6 is a *incorrect* combination of the two rules above, though. The rule does not raise an internal alarm if the affected warden does not have a mobile device, which is a quite stupid behavior as the internal alarm should be absolutely independent from any mobile device. In this way there is actually no reaction if the warden of the area where the fire has been detected does not have a mobile device.

```
ON
  fire_detection(var Area)
IF
  is_day(var Area),
  warden(var Area, var Warden),
  has_mobile_device(var Warden)
THEN DO
  internal_alarm(var Area),
  send_message(var Warden, "fire alarm in area " + var Area)
END
```

A solution for this problem are conditional actions also referred to as EC^nA^n rules [32]. They allow a optional preceding condition for each action which serves as a kind of guard for the action: The action is only carried out if the condition is fulfilled. Using conditional actions it is possible to group all reactions to a situation into one rule and thus avoiding redundant specifications for events and conditions. The following rule is a *correct* combination of the first two rules in this section using a conditional actions.

```
ON
  fire_detection(var Area)
IF
  is_day(var Area),
  warden(var Area, var Warden)
DO
  internal_alarm(var Area),
  IF
    has_mobile_device(var Warden)
  THEN DO
    send_message(var Warden, "fire alarm in area " + var Area)
  END
END
```

3.8 Timing of Actions

The reaction to a detected (complex) situation usually consists of a number of actions not only of a single one. This was already addressed in Sec. 3.6 and Sec. 3.7 where multiple and conditional actions are introduced. Those two sections prepare the basis for comprehensive reactions in emergency management but do not account for one very important aspect: The coordination, particularly the timing of the various actions. In other words all actions specified as reaction to a particular combination of events and conditions are so far executed independently and at the same time, namely immediately after the detection of the situation. This is not necessarily the intended behavior as the following two examples show:

First consider once again the example of informing the local warden about a fire detection in his/her area. Beside raising an internal alarm inside the control room and sending a message to the mobile device of the warden (if available) one might want to additionally announce the warden using the public address system. The corresponding action might look like this: `announce(var Warden + " to the control room please!")` . Usually such an announcement would be repeated at least once because in the first time it might not have the full attention of the warden. However simply duplicating the action does not lead to the desired result, at least when the public address system does not have an intelligent internal solution, as either both announcements had to be made at the same time or one of the announcements had to be skipped. Introducing an additional operator for specifying sequences of actions instead of sets of independently executed actions is a common way for approaching this problem [41, 9, 29, 2]. In the current example a sequence specification could be used to play the two announcement one after the other. Using sequence specification defining a short break of 5 seconds between the two announcement, for increasing understandability, is not possible, though. The pseudo code below employs time constraints for this purpose. `a1` and `a2` are names for the two actions defining the announcements and `(a1 + 5 sec)` **`before`** `a2` states that action `a2` should be executed 5 sec after action `a1`.

```
ON
  fire_detection(var Area)
IF
  is_day(var Area),
  warden(var Area, var Warden)
DO
  internal_alarm(var Area),
  IF
    has_mobile_device(var Warden)
  THEN DO
    send_message(var Warden, "fire alarm in area " + var Area)
  END
  a1: announce(var Warden + " to the control room please!")
  a2: announce(var Warden + " to the control room please!")
WHERE
  (a1 + 5 sec) before a2
END
```

Thus specifying only the sequence or execution order of actions is still too coarse to allow a precise timing of actions. The need for timing in emergency management is further explained and justified in the second more complex example. The point to be made here is that timing is an issue for action specification and that it means more than just defining a sequence or execution order.

The example above is of course very simple and one can obviously argue that this particularly problem should not be addressed on the level of ECA rules but should be left to the public address system. Current public address systems actually have capabilities to repeat messages as the ones in the example. However the mentioned issues are essential for emergency management in general. This can be demonstrated by another example, the coordination of the ventilation, the fire suppression and the public address system in case of a fire inside a metro station as described in Sec 2.2.

The public address system could probably start informing the passengers immediately after the evacuation routes and the corresponding ventilation and fire suppression regimes have been determined.[4] The guidance of the evacuation might profit from a proper timing of messages and signals announced or displayed by the public address system. However we want to focus on the ventilation and the fire suppression system which frequently should not start their actions immediately. As explained in Sec. 2.2 activating smoke extraction and/or sprinklers too early can have a disastrous impact on the evacuation and could even cause casualties. It is extremely important to start the smoke extraction and fire suppression at exactly the right time. Thus timing is essential for a successful coordination of evacuation, ventilation and fire suppression. Consequently a ECA language for emergency management necessarily needs to support means for specifying the timing of actions.

3.9 Workflows

Emergency plans, procedures and guidelines often define workflows which should be followed in certain situations. The workflows are typically described in natural language (possibly accompanied by a formal definition) and are usually visualized by some workflow diagram. Particularly due to their visualisations, workflows are easy to understand for humans. Each edge in such an workflow actually corresponds to a rule telling what to do next depending on the position in the workflow and current observations (i.e. events, system conditions). Thus workflows can naturally be transformed into (flat) ECA rule sets [14]. The resulting rule set however, has an serious disadvantage compared with the original workflow specification: The relations between the rules, e.g. which one comes after which one or which ones are available at some point in the workflow, are not explicitly represented anymore. In other words the “flow” through the rules becomes invisible. Compare for example the workflow diagram from Sec. 2.1 (in a more precise version) with the set of rules implementing it:

[4] Sec. 3.14 targets the point of determining/planning the combination of evacuation routes, ventilation regime and fire suppression and the need for dynamic adoptions.

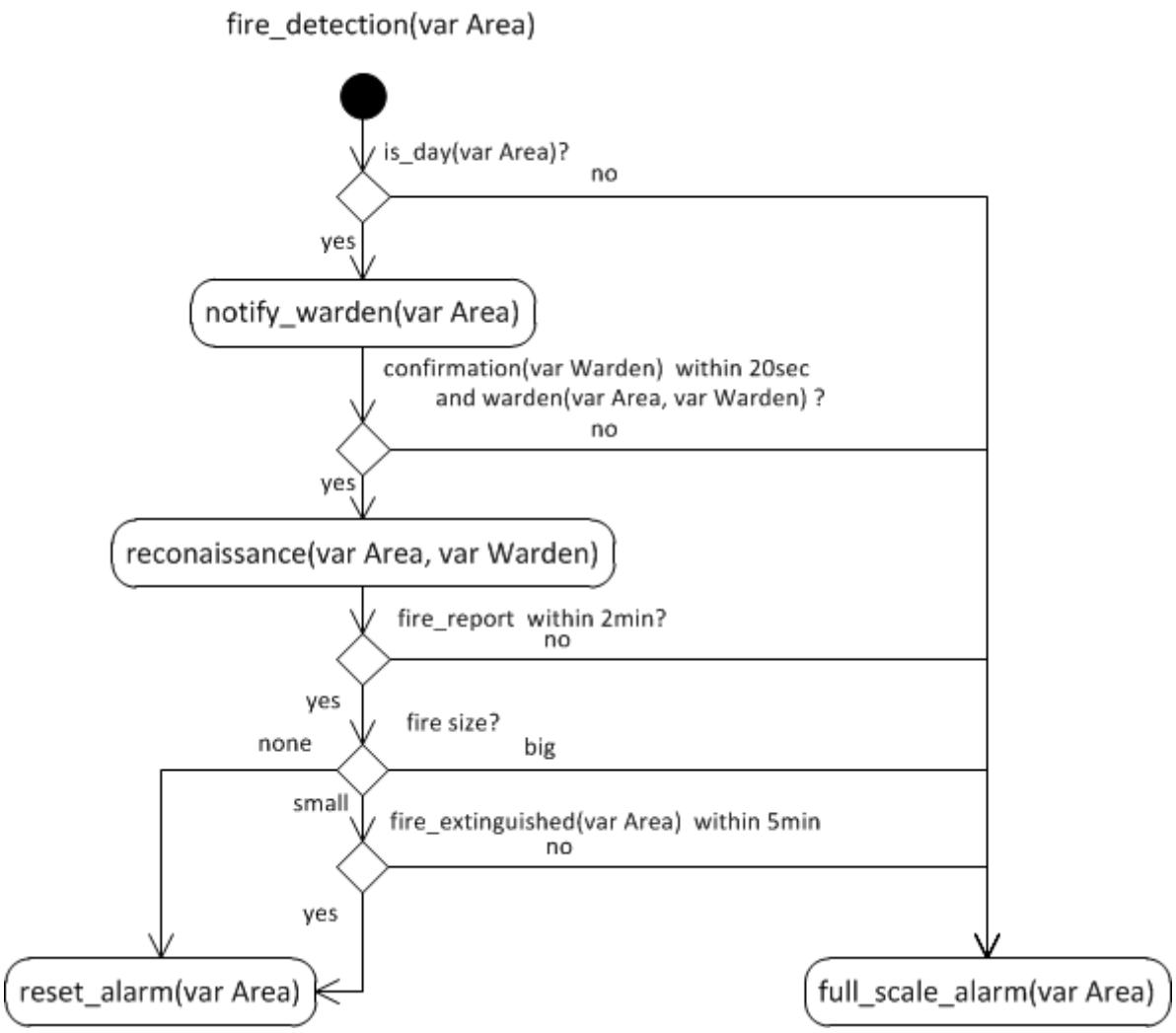

Figure 3: First reaction phase after fire detection

```
ON
  fire_detection(var Area)
IF
  is_day(var Area)
THEN DO
  notify_warden(var Area)
END
```

```
ON
  fire_detection(var Area)
IF
  NOT is_day(var Area)
THEN DO
  full_scale_alarm(var Area)
END
```

```
ON
  n: notify_warden(var Area),
  c: confirmation(var Warden)
  WHERE c after n within 20sec
IF
  warden(var Area, var Warden)
THEN DO
  reconnaissance(var Area, var Warden)
END
```

```
ON
  n: notify_warden(var Area),
  NOT c: confirmation(var Warden)
  WHERE c after n within 20sec
IF
  warden(var Area, var Warden)
THEN DO
  full_scale_alarm(var Area)
END
```

```
ON
  r: reconnaissance(var Area, var Warden),
  NOT f: fire_report(var Area, var _)
  WHERE f after r within 2min
DO
  full_scale_alarm(var Area)
END
```

```
ON
  r: reconnaissance(var Area, var Warden),
  f: fire_report(var Area, BIG)
  WHERE f after r within 2min
DO
  full_scale_alarm(var Area)
END
```

```
ON
  r: reconnaissance(var Area, var Warden),
  f: fire_report(var Area, NONE)
  WHERE f after r within 2min
DO
  reset_alarm(var Area)
END
```

```
ON
  r: reconnaissance(var Area, var Warden),
  f: fire_report(var Area, SMALL),
  e: fire_extinguished(var Area)
  WHERE f after r within 2min
    AND e after r within 5min
DO
  reset_alarm(var Area)
END
```

```
ON
  r: reconnaissance(var Area, var Warden),
  f: fire_report(var Area, SMALL),
  NOT e: fire_extinguished(var Area)
  WHERE f after r within 2min
    AND e after r within 5min
DO
  full_scale_alarm(var Area)
END
```

Obviously reading and understanding the rules is harder than understanding the diagram representation of the workflow. Of course this is partially due to the fact that rules inevitably form a linearized representation of the workflow whereas a diagram can show its graph structure. There is a more significant reason for the decrease of readability, though. (Basic) ECA rules do not support two major concepts of workflows, namely choice points and (intermediate) states. Both can (partially) be simulated but are actually not represented explicitly. We take a closer look to both points in the following two sections.

3.10 States

States are an essential concept of workflow specifications and a requirement for expressive ECA-languages [48, 15] as the specification of event-driven workflows or other automaton-like definitions of activities are "the" application field of ECA rules. The necessity of states may not be too obvious from the above example as states are identified with actions are, i.e. somehow there are no explicit states.[5] This is possible under the special conditions of the example, as each non-final state is only reachable by a single, unique action. However determining the state implicitly by the preceding action is only convenient in this special situation, not in general. If a state is reachable by more than one action, then determining the state would have to look at the disjunction of these actions, i.e. whether one of these actions has occurred. Even worse: An action may occur at two or more points in the workflow and might lead to different states (i.e. the action is not unique). In this case the current state can only be determined by considering

[5]The example has been chosen this way as there is no ECA-rule system or approach which currently supports or has examined states and thus a convenient (even pseudo code) syntax was hard to find spontaneously. Deliverable D4.3 will examine a syntax for states more closely.

the preceding history of the action, in the worst case up to the start of the workflow.

A way to work around this problem is to carry the state information in additional attributes of the events and actions. Generating new events carrying the state information may be another one. Both approaches however feel artificial and unintuitive, they still miss an explicit representation of states within the language and are not generally applicable. Thus the ECA-language for EMILI needs support for states not only on a global level, i.e. "What is the state of some system component?"[6], but also for states local to some ECA rule (or a set of rules), i.e. "What is the current position in the workflow and what happened before?".

3.11 Complex choices – Nested Rules

Workflow or other activity definitions have to define choices/decisions to find the appropriate reaction to a situation. More complex decisions are usually not made within a single choice and sometimes have to evolve over time depending on previously taken decisions and new observations. Explicit states are a convenient way to remember major decisions, e.g. actions, for the future. However finding the way from one state to the next one, e.g. finding the next action involves a number of minor choices based on arriving or not arriving events and other conditions. These choices often form a kind of decision tree. It could be useful not to represent each intermediate state in this decision tree explicitly, but to concentrate on major states with a clear intuitive semantic within the workflow. An important reason for this is that the concrete form of a decision tree is an issue for (manual) optimisation and thus intermediate states may be rather unstable. A second reason is keeping the code compact.

A common way for defining decision trees, e.g. in imperative or functional programming languages, are nested case distinctions e.g. using "if-then-else". A similar approach can be followed for ECA rules. Sec. 3.3 introduced "if-then-else" specifications of alternatives on the top level of a rule. The conditional actions from Sec. 3.7 are a first form of nesting, as a conditional action is considered to be a action itself performing the action part *if* the condition is fulfilled and *else* doing nothing. Extending conditional actions to support the "else" case is natural.

Alternatives are not only common for conditions but also for events, though. For example the workflow from Sec. 2.1 specifies an action if a `confirmation` event for a notification arrives within 20 seconds and another action if it does not.[7] Introducing alternatives (if-then-else) and consequently of conditions on events (only if-then) leads to a nesting of ECA rules. In other words, an ECA rule can be considered to be an action/activity itself and can be used wherever a basic action would be allowed in the language.

A specification of the workflow form Sec. 2.1 making maximum use of nesting could look as shown in the following pseudo code specification. In this specification all intermediate states of the workflow are implicit which represents the fact that even the diagram shows no explicit states but identifies the states with the preceding action. This is very similar to the set of basic

[6] This information might come directly from the employed SCADA system or from a reflection of the data in SCADA system which is more appropriate to the specifics of evaluation.

[7] Note that the time bound is essential for evaluating the negative case.

ECA rules shown in Sec. 3.9. Actually the pseudo code almost results from a proper nesting of these basic rules. In contrast to this rule set however, the relations between the (nested) rules remain clear and the "flow" through the rules is obvious.

```
ON
  fire_detection(var Area)
IF
  is_day(var Area)
THEN DO
  n: notify_warden(var Area),
  IF
    c: confirmation(var Warden),
    warden(var Area, var Warden),
    IF
      f: fire_report(var Area, IG)
    THEN DO
      full_scale_alarm(var Area)
    ELSE IF
      f: fire_report(var Area, NONE)
    THEN DO
      reset_alarm(var Area)
    ELSE IF
      f: fire_report(var Area, SMALL)
    THEN DO
      IF
        e: fire_extinguished(var Area)
      THEN DO
        reset_alarm(var Area)
      ELSE DO
        full_scale_alarm(var Area)
      WHERE e after r within 5min
    ELSE DO
      full_scale_alarm(var Area)
    WHERE f after r within 2min
  THEN DO
    reconnaissance(var Area, var Warden)
  ELSE DO
    full_scale_alarm(var Area)
  WHERE c after n within 20sec
ELSE DO
  full_scale_alarm(var Area)
END
```

3.12 Complex Action Specifications

The specifications of reactions in emergency management can become extremely complex. One main goal of EMILI is the integrated treatment of all subsystems of an infrastructure. Therefore some ECA rules work on a highly integrative and thus more abstract level. The actions specified in these rules affect more than one subsystem and involves a number of actions in each of the

affected subsystems. A action specification resolving this complex reaction from high-level actions like "evacuate the station" to basic actions like "turn on emergency light XY" would probably be infeasible. Furthermore different ECA rules may cause similar reactions, i.e. at least part of the reactions are equal or equal up to some parameters. The ECA rules should be able to share (parts of) the action specifications instead of having multiple copies of the specifications. Thus a procedure mechanism, where an complex action is specified once and given a name is required [10].

An example for a complex action which is likely to be used in a number of rules is the `full_scale_alarm(var A)` action from the Airport use-case example in Sec. 2.1. It involves alarming the Airport Fire & Rescue Department, the Airport Medical Services, the Safety and Security Operations Center, the Emergency Operations Control, and others (see Deliverable D3.1 Annex A). Each of these information tasks can consist of a number of actions. The full scale alarm also triggers the evacuation of the area which is a very complex action again.

3.13 Grouping of Rules

A flat, unstructured set or list of rules offers no guidance where a certain functionality is to be found and which rules interact. Virtually all wide-spread programming languages offer modules, packages, or similar constructs to structure programs. Grouping rules into separate, named rule sets and possibly also building hierarchies of rule sets exposes the structure of a rule program and eases considerably human understanding. Also, rule sets could introduce scopes for identifiers, alleviating the danger of unwanted interaction of rules due to name-clashes of identifiers. These structuring mechanisms aim primarily at the organization of a program, i.e., keep related pieces of code together and unrelated code separate in the program layout. This eases authoring and maintenance for human programmers [10].

Beside general structuring features which can be useful to separate the rules local to different subsystems of the infrastructure there exits another concept with a highly structuring effect: states (see Sec. 3.10). Many ECA-rules, particularly those formalizing a workflow, will be local to some state, i.e. will only be active in this state and thus can be considered as "belonging" to this state. Thus these rules should be grouped by state. An example for a high-level grouping which could be achieved using states are the different threat scenarios in the Airport Use-Case (Deliverable D3.1 Annex A). Each of the scenarios could correspond to a state. The specific rules of the scenario would be assigned to this state and would become active only when the state, i.e. the scenario, is detected.

3.14 Adaptive Reactions

Emergency procedures are often defined with respect to a number of emergency scenarios (or categories of scenarios) which reflect the experiences and expectations on what could possibly happen. Emergencies matching these scenarios are therefore well treated when following

the procedures. Unfortunately the most serious emergencies tend to cause extremal situations which are hard to foresee. A static treatment using predefined emergency procedures can lead (and has led) to suboptimal and also disastrous reactions. Thus an appropriate emergency management system needs a more adaptive way of choosing a reaction than predefined procedures. The system should be validated against a set of scenarios which represent the experiences and expectations on emergencies. However it must not be limited to these scenarios, i.e. the decision algorithm needs to be flexible enough to have a real chance of coping with unexpected situations.

The coordination of the ventilation, the fire suppression and the public address system for the evacuation of a metro station in case of a fire as described in Sec. 2.2 is a good example for the need of adaptive reactions. As type, size and location of the fire determine the availability of evacuation paths, i.e. mainly whether the paths are/remain free of smoke, evacuation routes have to be chosen based on the properties of the fire. Static evacuation routes and procedures which do not account for e.g. smoke propagation are highly dangerous and could guide people into death.

The goal for EMILI is a system that adapts dynamically to situations. A number of requirements have been identified for this goal:

Combining simulation and real-life data: Quick and adequate reactions not based on predefined reactions are, according to the EMILI project's Description of Work, core objectives of EMILI. These objectives call for real-time simulations generating predictions for the close future (time ranges of a few minutes are necessary). Reactions proposed by the system, like the choice of an evacuation path, should be "marked" with their so-called "provenance", making it possible to track the simluation data they are based upon and to validate (invalidate, resp.) them once the simulation data used turn out to be realized (not to be realized, resp.).

Ranking suggested actions: In some cases, more than one choice (for example for evacuation paths) might be suggested by the system. Therefore, a ranking of these choices (based on cost functions to be defined by the stakeholders) will be needed.

Physical models amenable to fast computation: The simulations mentioned above call for physical models that can be computed extremely rapidely. The evacuation of a a tunnel in fire must be completed within a few ten minutes making it necessary to predict the smoke distribution for this duration within at most a few seconds. Traditional software based on computational fluid dynamics designed for building optimization during the architectual planing phase do not fullfil such requirements.

Complex Goals and Actions: The safe evacuation from a tunnel in fire is a goal that can only be realized through coordinated complex actions like setting ventilators in a given manner so as to keep the chosen evacuation path free of smoke, coordinating the further routes of inbound trains, etc. So far CEP an ECA rules languages have no notions of goals and rather limited notions of complex actions.

4 Advantages

- ECA rules are easy to write and understand for humans as they reflect human thinking. This is particularly true in event driven applications like emergency management because events are naturally represented by an *explicit* event specification [10].
- Requirements are frequently specified in the form of rules expressed in either a natural or formal language, for example legislative rules, or (emergency) reaction plans. Ideally, a one-to-one mapping between rules used for requirements specifications and (executable) ECA-rules used for workflow enactment can be achieved [10, 14].
- ECA rule based specifications have a flexible and modular nature. Therefore they easy to adapt, alter, and maintain [10, 14]. Safety and security requirements and regulations for large infrastructures are usually quite stable, i.e. changes do not happen frequently. Furthermore rules for emergency management will probably have to pass extensive tests and reviews before they are approved for practical usage. Therefore the rule-sets currently used in an emergency management system will not change very often. However during development, testing and refinement the flexibility, adaptivity and maintainability of rule-based specifications are invaluable. Furthermore these properties of rule-based specifications allow an easy transfer of a proven, reliable rule-set from one infrastructure to another similar one.
- Reactive rules, especially ECA rules, easily integrate with other kinds of rules such as deductive rules (rules expressing views over data or rules used for reasoning with data) and normative rules (rules expressing conditions that data must fulfill; also called integrity constraints) [14]. Methods for automatic verification, validation and transformation of rule sets have been well- studied and applied successfully in the past [10, 14].
- ECA rules can be managed in a single rule base as well as in several rule bases possibly distributed over a network, because of the explicit specification of events which allows also for massage based communication [10, 14].
- ECA rules allow an easy handling of errors and exceptional situations that can conveniently be expressed as (special) events [10, 14].

5 Challenges: Disadvantages and Deficiencies of Current Systems

Most disadvantages mentioned below are not disadvantage of the ECA approach in principle but of current systems. Actually in preparing this overview we were rather surprised that the current results on ECA rules for complex event processing (CEP) are much behind our expectations and insufficient for emergency management in large infrastructures. This is partially due to the origin of CEP and ECA from active databases that were developed for low level system management. Whereas CEP has emancipated from its origin, ECA rules have stagnated on a crude level. This might sound surprising as ECA rules have extensively been examined in active databases. The reason are a number of fundamental differences between ECA in databases and ECA for CEP, particularly when applied to emergency management.

5.1 Interaction With the Physical World

Active database systems are concerned about (internal) events reporting about queries/changes to the database and support (internal) action, e.g. updates, for changing the data. Many CEP systems stay close to this configuration and only account for (event-based) message exchange and data manipulation. Thus they are working on data only and do not interact with the physical world by means of sensors and actuators.[8] However (external) actions interacting with the physical world have fundamental differences to (internal) actions changing a database only:

No rollback. Changes on data are easily reversible in principle and thus internal actions only working on data can be retracted. This is never true for actions affecting the physical world. Such actions can at most be compensated but not be retracted because they already had an effect on the physical state and time can not be turned back. In many cases it is even impossible to compensate external action. Consider for example the evacuation of a metro station in case of a fire: After fire detection an evacuation route is chosen and people follow this route for 2 minutes. Then the system notices that the choice of the evacuation route was wrong. In this case the system is not able to restart the evacuation, i.e. to reload the situation 2 minutes ago.

This property of actions affecting the physical world has serious consequences on how planning and particular conflict resolution can be carried out. The "Let's try and see" approach[9] frequently used for actions only changing data is not applicable. Actions to the physical world can only be tried out virtually using physical models, e.g. simulation.

[8] A email notification is of course an interaction with the physical world but (almost) without feedback cycle. The purpose of the system is fulfilled when the action is executed, i.e. the message is send at the right time. This is different for emergency management. Here the purpose of the system is only fulfilled when an action has the intended effect in the physical world, i.e. changes the physical state of the infrastructure in the desired way. This state change is surveyed by a sensor. Thus there is a strong feedback-cycle over the physical world.

[9] "Let's try and see" does not mean that you do not care about the exact result. However the basis for e.g. transaction management in databases is to first run queries/actions until a conflict is detected and then to resolve the conflict using rollbacks. This backtracking-like approach of "let's try" until a conflict is detected and resolving the conflict by stepping back, is not applicable for actions interacting with the physical world.

No immediate effect – Indirect feedback over success. Actions are always carried out for achieving some kind of effect or goal. The effect or goal of (internal) actions working on data is the change of data. When carrying out such an action the effect is immediately visible. The data has changed. The success of the action can easily be confirmed by checking the data.[10]

Again things are different for (external) actions affecting the physical world. Changing the state of a light from off to on may work almost immediately and the success of the action can also be checked directly. Opening a fire damper or powering up the smoke extraction fans in a metro station takes some time, though. But at least the basic success of the actions, i.e. fire damper open and fans on, can still be confirmed directly.

Unfortunately this is not true for the success of the main goal standing behind these actions. The main goal of opening a fire damper and powering on a fan, i.e. of activating a ventilation regime, is the extraction of smoke. Thus the success of the goal is defined by the fact whether the amount of smoke in the station is reduced and not by the fact whether all components reach the intended state. Therefore the success of the goal can only be confirmed indirectly, e.g. using smoke sensors. This confirmation will need some time even all actions have basically succeeded because (basically) successful activation of an ventilation regime does not magically remove (or even only reduce) the smoke in the station from one moment to the other.

The above example shows a second difference of actions on the physical world to actions on data. The basic effect of an action affecting the physical world and the goal of the action may differ. Consequently it may not be possible to directly confirm the success of an action with respect to its goal. The success of of the action could only be determined indirectly by sensors observing whether the physical world responds in the desired way.

Fuzzy effect. Actions changing data have a clearly defined effect. The same holds for actions in action languages (see Sec. 7.1) which are intended for planning. For actions on the physical world only the basic effects are clear. The basic effect of opening a smoke damper is obvious: The smoke damper is open afterward. The effect on the aerodynamics of the station is fuzzy, though. But the aerodynamic effect is the interesting one for planning a ventilation regime. The basic effect of some announcement is clear, too. The specific sound of the announcement will leave the loudspeakers. But the more interesting effect for planning person flows is, how many people will move in the according direction afterwards. This effect is fuzzy again.

Unfortunately no current CEP system with ECA rules has even rudimentarily considered these issues. The maximum support for external action affecting the physical world is the execution of some script. Thus external actions to the physical world are mostly out of the scope of the systems an reasoning about actions is possible in no way.

[10] In a distributed environment actually checking the success of an executed actions could be come more difficult, e.g. due to unreliable communication. But this is another dimension of the problem, which is independent from the kind of action (data/physical world). The main point is, that the information about the success is at least available somewhere though not necessarily at the right place.

5.2 Time and Timing

Where time (or at least the reception order) for events is a common feature for most CEP systems, time has never been considered for the condition and the action part. This is probably another relict from the active database origin of ECA rules in CEP. Sec. 3.8 explained why timing is important issue for actions. Current ECA languages do not support timing for actions, though. At most, they offer sequence and other operators for determining the execution order of actions.

The realizing of execution patterns for actions using ECA rules do not always reflect the procedural, imperative way of thinking familiar to many people from imperative or object-oriented programming [14]. This is particularly obvious when looking at the realization of the sequence pattern with ECA rules:

- If only basic ECA rules without sequence operator and timing are available, for sequencing A and B, B is triggered by a separate rule which reacts on a finish event of A.
- If a sequence operator is available, sequencing A and B is obviously trivial and looks familiar from imperative programming.
- When using timing for actions, sequencing A and B can be achieved by stating first that A and B should be both executed and second by giving a constraint telling that the finishing time of A has to be smaller than the starting time of B. This looks more complicated than the specification with a sequence operator and it is for this primitive example. But as soon as time plays a role and particular when non-linear temporal patterns between more than two actions should be realized then temporal constraints have a clear advantage compared with any kind of composition operators.

5.3 States

Recent ECA rule languages do not explicitly support states at all. Global system states could be simulated using the condition part of the rule and explicit updates. States that are local to an ECA rule or a rule set specifying a workflow, i.e. states formalizing the current position in the execution of an action specification or a workflow and which store information from previous steps, are hardly realizable [48]. In some cases the implicit state can either be determined by the history of preceding events and actions or the state information is carried in additional, artificial attributes of events and action or in special events generated for this purpose [14]. Both ways are neither natural nor generally applicable. The Sections 3.9, 3.10 and 3.11 further discuss the need for workflow specifications and an explicit language support states.

Explicit states also determine which ECA rules are active for some point of time. Thus they introduce a notion of which events are expected next. Without states every ECA rule is considered to be active and is triggered by every incoming event matching the event specification, regardless of whether this event is expected or not. This might entail unexpected behavior, especially if events are generated “out-of-order” by faulty or malicious behavior of systems[14].

5.4 Development, Exploration and Visualisation Tools

Rule sets for emergency management in large infrastructures are likely to become quite extensive. The different rules interact, simple ones contribute to more complex rules, some rules are only active in certain system states or belong to some workflow. Although language features like alternatives, conditional actions and nested ECA rules, states and modules help to keep the code organized and readable, tools for exploring an visualizing the relations between rules, states, events, actions and infrastructure components are highly desirable. The SOMAL ontology (see Deliverable D2.1) holds the information on these relations and the SITE GUI is intended to provide exploration tools (see Deliverable 6.2).

6 The State-of-the-Art

The state-of-the-art of ECA rules has been surveyed for a number of times [7, 48, 42] in the past three years. As ECA rules have not been in the focus of research and development, the surveys are still up-to-date. One of them, *Reactive rules on the Web* [7] originates from activities in the REWERSE Network of Excellence ("6th Framework Programme", Information Society Technologies (IST), project reference number 506779) finally leading to EMILI and was written with contribution of current members of the EMILI consortium[11]. It contains an extensive overview on ECA and production rules. Thus we felt that another academic overview would not contribute to EMILI. For this reason we decided to write our survey from the emergency management point of view. In Sec. 3 we identify a number of requirements for ECA rules in emergency management using simple examples. In the following existing ECA systems and languages are briefly examined with respect to these requirements.

ECA rules originate from artificial intelligence and have been adopted for many technical systems particularly for active databases. From there they found their way to complex event processing (CEP). However the focus of most CEP languages and systems is on the detection of complex events and situations (see Deliverable D4.1, [48]) and on their distribution [42, pp. 23-26]. The reactive capabilities of the current languages and system are often very restricted. Frequently they are limited to notification rules, updates and external function calls.[12] To the best of our knowledge none of the existing languages supports timing for actions (Sec. 3.8). There is no explicit support for states (particular local to an ECA rule, (Sec. 3.10)) in the languages, at best other features of the language allow a more or less comfortable simulation of states. The interaction with the physical world (Sec. 5.1 is at most considered by enabling external function calls. Features like conditional actions, workflow specifications and nesting of rules are not available.

Existing Systems

ECA-LP [41] and its XML representation Rule ML [44, 43, 42] claims to have a homogeneous approach for deductive and reactive rules and enables complex action specifications. ECA-LP has a kind of declarative semantics, for both events and actions, however this semantics is not really meaningful[13] because it is just inherited from the underlying rule/inference system PROVA [41, 34, 33]. ECA-LP supports (transactional) updates and external actions and can

[11] Particularly Paula-Lavinia Kroner (former Pătrânjan) at SKYTEC and François Bry at LMU.

[12] CEP systems focusing on event detection an distribution only are not considered in this section. See Deliverable D4.1. for such systems.

[13] The semantics is only given by means of a translation into PROVA rules. Thus the "declarative" semantics of ECA-LP is mostly a "operational" semantics of ECA-LP with respect to the declarative semantics of PROVA rules. The "declarative" semantics does not clarify *what* the meaning of an ECA-LP rule should be but states *how* to evaluate an ECA-LP using PROVA.

provide a kind of state[14] using the production rule features of ECA-LP.

The ruleCore Markup Language (rCML) [49, 40] supports ECA-rules and actions like script execution and sending an event.The official documentation on [40] suggests that sending (action-)events has become the only kind of actions in the mean time. The documentation is not very comprehensive, though.

XChange [12, 13, 46, 18] also supports ECA rules. Basic actions are updates, transactions and notifications. It enables complex update definitions, i.e. some kind of complex actions, using conjunctions and disjunctions for actions. XChange has a declarative and operational semantics for event queries. The event querying part of the language and its semantics has been further refined by the event query language XChangeEQ [11, 19].

The basic idea of MARS [4, 5, 6] is to realize ECA functionality by choosing one language for the event specifications, one language for the condition specifications and one language for the action specifications of the ECA rule and plugging them together. The languages are able to share information using variables. The semantics of ECA rules are inherited from the semantics of the languages used in the event, condition and action specifications. The languages for the event and condition specifications are expected to have algebraic nature. The action specifications can base on a process algebra like CCS [38] (see 7.2) or on "classical" programming language.

[14] This does not mean that there is a language support for states in ECA-LP. But using the stateful fact base of production rules, a state-like behavior can be achieved more easily than with plain ECA rules.

7 State-of-the-Art Beside Event-Condition-Action Rules

Beside Event-Condition-Action rules the specification of actions and the reasoning about these specifications has been approached from a number of directions and on different fields. Some of those fields are somehow ancestors of CEP; others are quite independent from the notion of event. In the following we will give a rather comprehensive list of directions and fields where actions have been examined. Each direction or field will be briefly characterized w.r.t. its background, goals, advantages and deficiencies. The overview is mainly based on [42]. We refer to this survey for more details.

7.1 Action Languages

Action Languages [25, 26, 50, 28, 27, 20, 17] are intended to provide a logic framework for declarative reasoning about action and change. They are particularly useful for some kinds of planning problems. A brief illustration for some of them can be found in [42, p. 13f]. The greatest disadvantage of action languages from the CEP point of view is that there is no notion of events. The only way "events" are somehow considered is in the sense of effects caused by actions. This implies that there is no way to treat events as input.

7.2 Process Calculi/Process Algebra

Process Calculi or Process Algebras [2, 39, 3, 29] are basically used to study the behavior of parallel or distributed systems by algebraic means. Their focus is the interaction of different processes. Some of them can be used to (automatically) proof safety and liveliness properties of a parallel system. Process Calculi and Process Algebras do not have a notion of time. Advanced process calculi are used in modern ECA rule approaches to specify complex actions [5, 43].

7.3 Transaction Logics

Transaction Logic [9, 8] is a general logic of state changes. It provides a logical language for programming transactions and updates of views in databases. It can also be used to specify active rules, including procedural knowledge and object-oriented method execution with side effects. Furthermore definitions of complex actions as combinations of simple actions are supported. Transaction Logics comes with an own proof theory. However Transaction Logic has no representation for events.

7.4 Update Languages

Update languages [37, 1, 21, 36, 41] enable updates to rules and define a declarative and operational semantics of sequences of dynamic Logic Programs. The focus lies on updates to derivation or/and reactive rules (in contrast to updates on data or states). They are able to deal

with knowledge base updates and transactions. Most of them are not concerned about events or states.

7.5 Production Rules

Production rule systems [23, 22, 31, 30, 47] (see also Deliverable D4.1) consist of a knowledge base containing a set of facts and rules of the kind "when condition is fulfilled then add/delete fact". A rule is fired, i.e. the add/delete action is performed when the condition of the rule is fulfilled. The knowledge base is updated incrementally using the RETE algorithm [24]. Beside primitive add/delete actions neither events nor states or actions are explicitly represented. However facts representing events, states or actions can be used to implement a reactive system by means of production rules. The greatest deficiency of production rules is the lack of an explicit representation for events, states and actions and the absence of clear declarative semantics.

7.6 Active Databases

Active databases [16, 52, 35, 45, 54, 53, 51] are intended to add active functionality to a database system. This is usually realized by some kind of ECA rules. Basic events are typically database state transitions like inserts, updates or deletions. More advanced systems allow the definition of complex events for example by means of event algebras. The conditions may refer to data in the database or data carried by the events. When a rule is triggered by a suitable event and the conditions are fulfilled then the action is carried out. A basic action is typically some operation within the database. Some systems allow the definition of complex events using an action algebra with operators like for example succession, non-deterministic choice, parallel flow and loops. There has been quite an extensive research for syntactic / semantic analysis of ECA-rules, particularly with regards to termination and confluence. However ECA-rules in active databases have the disadvantage of being limited to internal events and actions of the database.

8 Conclusions

ECA rules are the right approach to achieve a reactive behaviour for EMILI. Rules in general reflect human thinking and are therefore well suited to express procedures which are related to critical infrastructures. Furthermore, safety procedures are often specified in a rule based manner and are thus easy to translate to a more formal but also rule based formalism.

However, current ECA approaches suffer from substantial deficiencies which prevent them from being used in the context of EMILI. ECA rules as they are studied in the field of active databases are solely concerned with the modifications of tuples in a database and require some kind of rollback mechanism which is provided by the concept of transactions. However, such a rollback mechanism is not available if actions in the real world are considered, since they might not be fully reversible. Other, less mature concepts offer only support of atomic function calls which is not appropriate for the complex reactions. Furthermore, recent approaches to ECA rules offer only limited support for time respectively the timing of actions, if at all. Consequently they also provide no notion of states which massively depends on a notion of time. As we have learned from the use cases so far, such a simple support of actions is not sufficient for a successful application of ECA rules within EMILI. However, in order to decide which features need to be supported in the event and action language for EMILI further input from the still ongoing work on the use case specifications is required.

Actions have been approached on a number of fields and from different directions. However there is no comprehensive framework covering events, states and actions. The adoptions of the ECA approach in CEP languages come without clear semantics and an embedding formal system. Therefore problems like conflict resolution, confluence, ordering, termination and others have not been examined yet. The results on ECA rules in the active database research might help to fix these problems, however some results may not be transferable, as reactive behavior in CEP cannot be restricted to internal actions. Furthermore the ECA approach and its current variations are not expressive enough to specify extensive complex reactions to detected situations [48, pp. 7-9,11,12].

9 Outlook – A Declarative Event and Action Language for EMILI (DEAL)

The specification and evaluation of complex events and complex actions is among the main technical contributions of EMILI. The success of EMILI depends on a suitable specification language for complex events and complex actions and an efficient evaluation of this language. The main criterion for the quality of this language is how well it fits to the needs of the users, i.e. the use-case developers. This section covers general goals, assumptions on the working domains, and a first sketch of the language.

9.1 General Goals

9.1.1 Comfortable and intuitive specifications of (complex) events and (complex) actions

The specification of (complex) events and (complex) actions should be easy. Writing and particularly reading the specifications should neither require advanced programming skills nor much practice of the language. This eases the specification process itself but also the maintenance and the verification of the specifications. In addition to that a GUI editor, as described in deliverable D6.2, can be used to further assist users in writing and reading rules.

9.1.2 Clear Semantics of Specifications

As EMILI addresses safety applications correctness and verification of specifications are an important issue. A precise understanding of the meaning of a specification is therefore essential. Thus a clear semantics for event and action specifications has to be defined. The definition of the semantics should be done on two levels, i.e. both a declarative and an (equivalent) operational semantics is needed. The declarative semantics is close to the definition of the specification because it focuses on what is the result of the specifications. The operational semantics is close to the evaluation of the specification because it tells how a result is obtained from the specification. Using both types of semantics makes it possible to have an semantics which on the one hand is easy to understand for the use-case developers as they do not have to care about the evaluation process, but on the other hand ensures that the actual and the intended result of the evaluation coincide.

9.1.3 Integration of Event Streams, Static Data and Semantic Information from SOMAL

The specification of (complex) events and (complex) actions may rely on data which is not delivered on the event stream. Such data includes semantic information on the structure of the surveyed system (like a tunnel or power grid) which is provided by SOMAL as well as information on the current state of the system stored in a database.

9.1.4 Processing Multiple Event Streams

The evaluation engine should be able to process several event streams coming from different data sources. Events from different streams can be processed independently and/or be incorporated and processed together. One way for independent processing are networked evaluators. An incorporated processing allows the definition of complex events and complex actions which depend on events from different sources. It might be useful when the derived events and actions carry information about their provenance, i.e. about which selection of the incoming streams they are derived from. This would allow distinguishing between events which depend solely on real data and events which depend partly or totally on simulation. The latter is of paramount importance at every step of the process.

9.1.5 Networked Evaluators

It should be possible to (statically) connect several instances of the evaluation engine in a network. These instances are called networked evaluators. The idea is to process data where it is generated. This idea also reflects the current settings in safety systems. There already exist systems for monitoring and controlling most parts of the infrastructure. However these systems have only a quite local and specific focus and therefore lack an global integrative view of the whole system.

A network of evaluators is useful for proceeding from a local, specialized view to a global, integrative view in several steps. For example: A first processor A could detect complex events in a part P1 of the system, a second processor B in another part P2, a third processor fed by the event streams generated by the nodes A and B could detect complex events referring to a part P containing (conceptually or topologically P1 and P2). All evaluators of such a network would be identical, yet run different complex event and complex action programs. To keep the complexity of implementation low, the network of evaluators is built statically, i.e. cannot be modified during run time. Thus an evaluator may treat input coming from another evaluator the same way as it treats input from a sensor, i.e. the other evaluator is just considered as “very intelligent sensor”.

The concept of networked evaluators opens the way for flexible applications of the evaluation engine. For instance, networked evaluators might help to integrate existing hardware configuration. To this end, a SCADA system can be integrated into the network as a common node. Instead of sending all basic events generated by the sensors of an infrastructure directly to a processing node of the network, the sensor readings are collected by a SCADA system and fed into the evaluator network in form of events. Since the evaluators of the network do not distinguish between different types of evaluator nodes the events generated by the SCADA system are considered as “basic” events by every evaluator of the network which receives them as input. The integration of several (SCADA) systems for different aspects or part of the infrastructure can be done exactly the same way.

Another application of networked evaluators is outlined in the following section about simula-

tions.

9.1.6 Simulations

For a modern and reasonable dynamic emergency management, EMILI needs support for simulations. To which extend simulations are required is currently discussed within the project and depends on the specific requirements of the use cases which need to be further developed.

However, using the idea of processing multiple event streams and the one of networked evaluators integration external simulators can be realized very easily. Consider the following illustration:

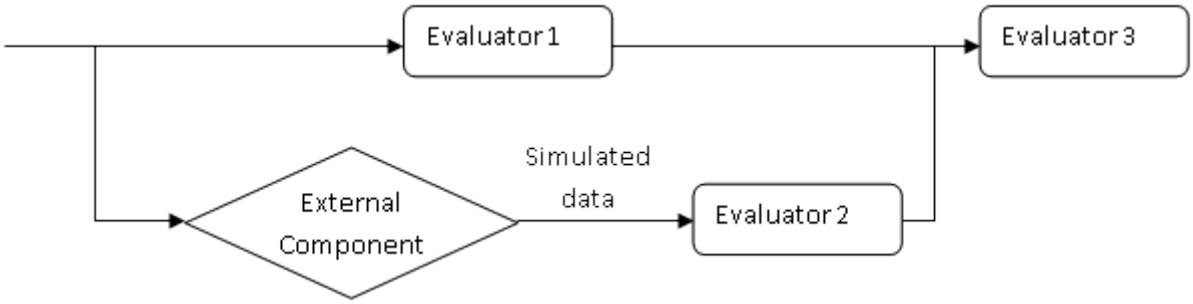

Figure 4: An external simulator integrated into the system as a networked evaluator

One of the networked evaluators in the diagram uses only real data from the sensors for its evaluation. A second evaluator additionally receives simulation data from an external component, which can be in turn based on real data provided by the event stream.

The second evaluator applies its set of queries to the output of the simulator. Its output is therefore marked as simulated data as well. Finally one could even add a third evaluator which works on the output the first and the second evaluator and is thus able to incorporate the conclusions from real and the simulated data. Such kind of evaluator enables the system to base its decision on the reaction to some situation, not only on the its knowledge of what is happening now but also on the predicted future development of the situation and even further it is able to take the consequences of the reaction into account.

9.1.7 Auditory Logging

There must be a way to store basic and complex events and even a log of the executed actions for later analysis or for documentation purposes. For example in case of serious accidents, responsibilities need to be identified. Furthermore a careful log allows learning from past events. Finally, normative and legal requirements need to be accounted for.

The (regular) evaluation should not depend on these logs. Therefore this logging component can be implemented as an independent component (as done in the SCADA system of PUPIN described in deliverable D3.1 annex a).

9.1.8 Time Model

In complex event processing there are two available time models: application time and system time. The primary model of EMILI should be the application time model. Experiences of different previous projects have shown that only this model is adequate for a faithful event and action processing system.

With the application time model time-stamps of events are set during or shortly after the actual occurrence of an event. In our context this means that the time-stamps are set by the sensor or by a component which is very close to the sensor in the network. The goal is to keep the delay between event creation and setting its time-stamp as small as possible. The great advantage of this model is that the time-stamp reflects the real emergence of the event. Network conditions affect only the response time not the result, and changes in the network do not trigger changes in the queries. Thus the specifications of complex events and actions remain intuitive and are easy to handle for the developer. With the system time model events are time-stamped when arriving at the evaluation engine. The result is that the recognition of complex events depends on network conditions. So network conditions have to be considered explicitly in the specification of complex events and actions. Changes in the network or in its conditions might even imply adaptation of the queries. When network conditions are changing dynamically the result of complex event queries might become rather unpredictable for the developer. However, depending on the needs of the use cases, it might be reasonable to integrate both, or even multiple other, kinds of time models into the language.

9.1.9 Restrictive Data Schema

The incoming and the derived data have a compelling data schema which is known to the evaluation engine. The schema is restricted compared with DTDs or XML-Schema, since it allows only flat structures similar to relations of a database. However, based on the needs of the use cases, the support of more expressive schemas like one that disallows recursive definitions, i.e. nesting of elements to arbitrary depth is not possible, and forces a fixed order of the elements, can be integrated into the language. Considering the provided basic data types it is probably a good choice to start with those types immediately supported by MonetDB which forms the basis of the implementation of the CEP engine.

9.1.10 Events vs. Actions

Events and actions are two concepts which should be kept separated. Events are observations of state transitions, i.e. events observe changes in the system. In contrast actions are operations which cause changes in the physical world which in turn might result in a state transition. However, every action causes an event which confirms the execution of an action and which can be queried by appropriate rules.

Interpreting events as changes in the state of the system is very natural. However the incoming

data streams will often contain elements which do not represent changes. For example the stream might contain repeating messages of the same sensor with exactly (or almost) the same value. Therefore the language should offer means to filter such rather insignificant events. States or rather events caused by state changes are one possibility to achieve such an behavior.

Furthermore one can distinguish two different types of actions: authorized and non-authorized. An authorized action is an action which is explicitly approved by the operator of the infrastructure. Non-authorized actions have not been approved by the operator and are therefore *not* executed by the system. Some actions may be authorized by default, meaning that they do not need an interactive authorization. It might be useful to keep a log of the non-authorized actions. Thereby one could observe which actions are frequently rejected which might indicate a need for an improvement of the system or a need for a better training of the personnel. The language should make this possible. Thus rejected options need to be recorded, together with the reasons for the choice.

9.1.11 Properties of Events

Finite Duration Complex events generally have a (non-zero) duration, i.e. their time-stamps are not a single time-point, but represent a connected time-interval. This interval is given by two time-points expressing the beginning and the end of the event respectively. The time-stamps are set accordingly to the application time model. The duration of every event needs to be finite, that is, events with a possibly infinite duration should not be expressible in the language. This assumption is necessary for a proper implementation of the evaluators (at least when exact answers are needed): In presence of events with possibly infinite durations, evaluators would have to keep data forever (or drop potentially needed events when memory is running out). The deletion of events at this point is only relevant for the evaluation system. It does not imply that the presentations of the events at the SITE level are removed, too.

Carried Data An event may carry different forms of data. There is

Payload in form of key value pairs. See the use-case descriptions in Deliverable D3.1 for the concrete form of the payload.

Fuzzy values which come from sensors or could represent the probability/trust of some data. Fuzzy values can be part of the structured data.

Event-Identity which is needed to express causality, i.e. which event or which set of events has caused a complex event.

Time-stamps that, as mentioned above, specify connected time-intervals. A time-stamp according to the Application-Time model is obligatory. Additional time-stamps according to the system time model could be added as needed.

Provenance. A (complex) event must carry some information on its provenance, i.e. its dependencies on other events (particularly from the incoming data-streams). This feature might

be useful for distinguishing simulation data from real data and for expressing causality between events and/or actions.

Context. It might be useful when an event carries information about which context or state, or even which rule, it was created in or from.

9.1.12 Non-Consuming Evaluation Strategy

There are two different strategies for the evaluation of event streams which particularly concern the way how memory is freed from events which are not needed anymore: The consuming and the non-consuming approach. The consuming approach is easy to implement, as events are deleted immediately after being processed once. This approach does not need an involved garbage collection. However it can be hard for the developer to predict the actual effect of some specification. This contradicts the wish for an clear semantics (c.f. section 9.1.2). The non-consuming approach chooses just the opposite way. In this approach an event is only deleted when there is no rule left which still wants to use that event. This requires a sophisticated form of garbage collection and therefore the non-consuming approach is harder to implement. Correct garbage collection depends on the used time-model (c.f. section 9.1.8). In [19] the foundation of such an garbage collection needed for the non-consuming approach has been made.

The non-consuming approach has several advantages for the developers: The effect of some specification is foreseeable as it does not depend on runtime conditions. Particularly a declarative semantics for specifications can be defined. Therefore the non-consuming approach is easy to understand and to use for developers.

9.1.13 Filters

There are different kinds of filters required for EMILI. The incoming data will often carry a significant amount of data which does not contribute to the detection of (complex) events. For instance the stream might contain repeating messages of the same sensor with exactly (or almost) the same value. Another example are sensors which (currently) deliver values in an uncritical range. A third example might be sensors delivering impossible or noisy data (like a temperature rising faster than the laws of physics allow).

Filters are a very sensitive part of the system. On the one hand filters may significantly increase the performance of the system and ease the definition of rule and action specifications. On the other hand filters are also able to corrupt the correctness of the overlying specifications. Even worse, the kind of implementation of filters has to be chosen carefully as it may turn otherwise that this apparent optimization is in fact counterproductive. Therefore filters should not be defined by the user but rather be automatically generated by the system. This can be achieved in a similar manner as the automatically detection of events that are not used by any rule of the enginge anomore as it was described in the last section. For instance, if every rule queries only temperature events with a temperature higher than 40°C, all temperature events with a lower temperature can be automatically filtered by the system.

For the reliable detection of impossible sensor data and the cleansing of noisy sensor data physical models are indispensable. Deliverable D4.1 contains an introduction to data cleansing within the scope of EMILI and explains why physical models are required for this task and how they can be used for data cleansing.

9.2 Language Concepts and Design

9.2.1 Complex Event Queries

Complex Event queries have to cover 4 dimensions which have been worked out in [19] and are listed below. It has shown that these dimensions are better treated independently by the CEP language. Indeed, operators of the language which mix the dimensions often lead to an incomplete covering of the dimensions [19].

1. Extracting and Using Data in Events

   Events are carrying data which has to be accessed for example for joining events in complex event queries and for the creation of a new (complex) events or actions out of existing ones.

   As events and particularly complex events probably carry structured data most of the time, the first task of the language is to make the data accessible. The support of a pattern-based extraction of the data will significantly raise the comfort of the language. The "Languages Design" part includes an example for pattern-based data extraction.

   The second task, are relations on the data extracted from the events. They are need for selecting particular events and for joining events in complex event queries. Most of the time, these relations will be simple comparisons or arithmetic operations. Such operations will be supported by default. Further relations could be added by need. Additional user-defined relations might be possible.

2. Detect patterns composed of multiple events - Complex Events

   Complex events are created as result of complex event queries. Complex event queries ask for a pattern of events, i.e. a number of events which are somehow related in matters of time and content. They are obviously essential for recognizing complex situations out of the basic events arriving on the stream.

3. Temporal Relations

   Temporal relations are needed to express how events are related in time. For example one could want to express that two events have to occur within a time window of at most 1 hour, or that event B has to come after event A but not more than 5 minutes.

   Temporal relations have to be bounded, i.e. there is always some finite time window in which the relation can be evaluated. For example one cannot just say that event B has to come after event A, but one has to restrict the after to a finite time window of 5 minutes. This is necessary to avoid infinite waiting for some second event which causes the first

event to be stored forever. When using the concept of workflow diagrams as described later, however, an explicit definition of the time window might become unnecessary as the system is able to derive it from the diagram. So writing just “event B after event A” might be allowed in this case, though.

The language will provide some basic predefined temporal relations. Furthermore it will support user-defined temporal relations based on the basic ones. These definitions will be non-recursive. This restriction prevents unbounded temporal relations.

4. Accumulation, Aggregation and Negation

   Accumulation of events is needed for several reasons. First for just collecting events which carry similar data. For example one might want to have a single event which unites the data of all sensors located in one area instead of one event per sensor. Furthermore accumulation is the basis for aggregation and negation.

   Accumulations on event streams are only possible over finite time windows. The end of the finite time window has to be known at the beginning of the accumulation, as collected events may have to be stored forever else.

   Consequently aggregation and negation are also limited to finite time windows. The language will support a number of standard aggregation functions. Again some possibility for user-defined aggregation functions might be useful here. This merely depends on the wishes of the developers.

9.2.2 Complex Actions

The language will allow the specification of reactions to the detected events which are proposed to a human or are executed at once. The actions are able to use the data extracted from the triggering events. Basic actions may be “sound the alarm”, “activate the sprinkler system”, “increase the ventilation”, “close the entrance” and so on. The definition of basic actions will need an interface to components external to the evaluation engine, i.e. to controllers for the alarm bell, the sprinkler system and so on.

Complex actions are used for combining basic actions to an extensive reaction to the current situation. Among others complex actions are essential for traffic infrastructures: A simple smoke extraction in a tunnel for instance requires opening of fire dampers, activation of smoke-extraction fans, activation of jet fans for mastering the longitudinal velocity, etc.

Section 3 contains a detailed introduction to complex actions as they are desired for DEAL.

9.2.3 Database Queries to Static Data

Specifications of (complex) events and (complex) actions may contain queries to databases of static data, i.e. to data not delivered on the data streams. For example messages from a sensor probably contain an identifier of the sensor but not its exact (static) location. Using the identifier

of the sensor this information can easily by joined from static data. A similar pattern syntax for accessing static data as for accessing data in events is desirable.

9.2.4 Structuring Features

The language will support classical structuring features like modules, blocks / groups of rules, local definitions and names for complex events and actions. Furthermore the language has a specific feature called "contexts" which also have a structuring effect but significantly go beyond simple structuring. "Contexts" are addressed in the next paragraph.

9.2.5 Workflow Diagrams and Specifying Contexts

Many events and actions can only occur in specific states of the system. Specifying states in some kind of workflow diagram accessible to the evaluation engine may have several advantages. First of all it is probably much more efficient to evaluate only those rules which really have a chance to match in the current state of the system than always evaluating all rules. Furthermore formalizing the specific properties of a state may simplify the definitions of the rules for this state as the state specific information has not to be repeated in each query, but can be added to the query by the evaluation engine.

Contexts are an abstraction of states. A context contains a set of states and formalizes the common properties of all these states. Contexts can be seen as macro state of the system. Put the other way round a state is an atomic context of the system. From this perspective the last two sentences of the preceding paragraph mean that it is easier to write context dependent rules, i.e. rules which implicitly use the context information, than writing general rules which have to carry the context information explicitly each time.

Contexts can form a complete hierarchy reaching from a very general, perhaps global, system context over more and more specific contexts to the most specific form of contexts namely states.

9.2.6 Language Concept Patterns for accessing data in events

Patterns are a very intuitive way to access structured data. As it was already discussed, the structure of events is limited to flat relations. However, a pattern base approach can be used for querying this kind of data as well and the query mechanism remains unmodified if the structure of events is modified towards a more expressive schema based on the needs of the use cases.

Consider the example below. A sensor station in a tunnel sends a message which contains several different values of different categories. Such a message could look like this:

XML data term:

```
<tunnel_observation>
  <sensorID>1234</sensorID>
  <climate> <temp>10</temp> <humidity>20</humidity> </climate>
  <traffic> <numCars>50</numCars> <avgSpeed>30.2</avgSpeed> </traffic>
</tunnel_observation>
```

If one wants to write a query which matches exactly those `tunnel_observation` messages from sensor 1234 this could be done like this.

XML pattern term:

```
<tunnel_observation>
  <sensorID>1234</sensorID>
  <climate> <temp>var T</temp> <humidity>var H</humidity> </climate>
  <traffic> <numCars>var C</numCars> <avgSpeed>var S</avgSpeed> </traffic>
</tunnel_observation>
```

The above pattern matches only `tunnel_observation` which have `sensorID` 1234 and therefore come from this sensor. “var T”, “var H” , “var C” and “var S” denote variables which are bound to the values contained in the message. The same query in another syntax which is perhaps easier to read and write: Complete pattern term:

```
tunnel_observation[
  sensorID[ 1234 ],
  climate[ temp[ var T ], humidity[ var H] ],
  traffic[ numCars[ var C ], avgSpeed[ var S ] ]
]
```

The above query is quite comfortable when one is interested in all of the data contained in the message. However one will probably need only parts of the data most of the time. In this case partial patterns are very useful: Incomplete query term:

```
tunnel_observation[[ sensorID[ 1234 ],
        desc temp[ var T ]
]]
```

The above pattern returns only the temperature. The double brackets denote that the pattern is incomplete, i.e. that the `tunnel_observation` may contain further entries. `desc` tells that there is a “temp” entry somewhere in the `tunnel_observation`. This avoids specifying all intermediary entries (which of course would not be a problem in this particular example).

When the data schemes are restricted and known then incomplete patterns can be realized without negative effect on the performance of the evaluation engine.

9.2.7 Using rules to derive higher level knowledge

When doing complex event and action processing one frequently wants to express things like

"When the smoke and the temperature sensor in area A are both activated within 20 seconds then there is a fire in area A"

in the case of complex events or

"When there is a fire in Area A then alarm the fire department"

in the case of actions.

Rules are a natural way to do so. The corresponding rules of the above example looks like this:

```
highTemp(A) AND smoke(A) => fire(A)

fire(A) -> alarmFireDepartment(A)
```

Furthermore rules allow using derived events (and actions) the same way as basic events. This feature is called rule-chaining. In the above example fire is probably classified as emergency. Thus one might want to derive not only a specific fire event but also a more general emergency event.

```
fire(A) => emergency(A)
```

The preceding rule achieves such a behavior by expressing

"When there is a fire, then there is an emergency."

Rule-chaining allows proceeding from basic over more complex to very complex events in several steps. In this way each of the rules is kept rather simple which may significantly ease the specification and the maintenance of the rules.

The same effect of generating an emergency event could have been realized by duplicating the primal rule:

```
highTemp(A) AND smoke(A) => emergency(A)
```

But this formalization has the drawback that besides being less readable, it makes it necessary to change two rules, if the conditions for the detection of a fire are adapted which can be easily forgotten or overseen.

9.2.8 XChange^{EQ} – the basis of DEAL

XChange^{EQ} is a rule based event query language which has been introduced in [19]. The language uses pattern matching as it was described in the previous section to detect events of an event stream. According to the needs of the use cases, the language will be adapted and extended, but its main principles will remain the same. However, to give a first impression of the language, consider the following rules based on the previous example:

"When the smoke and the temperature sensor in area A are both activated within 20 seconds then there is a fire in area A."

The according XChange^{EQ} rule looks like:

```
DETECT
  fire{ var A }
ON
  and{
    event s: smoke{{ area{{ var A }} }},
    event t: highTemp{{ area{{ var A }} }}
  } where { {s,t} within 20 seconds }
END
```

The **ON** part of the query matches on smoke and highTemp events of the event stream. The variable `var A` ensures that both events contain the same area node, i.e. are located in the same area. The where clause specifies that both events must occur within a time interval of 20 seconds. If these conditions are satisfied, a fire event, which contains the area of the sensors, is generated by the system.

Note that if multiple events are composed in a single query, it is mandatory to specify a time window which constraints how far in time both events can be apart.

For the following example let us assume that a sensor emits a measured value every 10 seconds. This information can be used to detect faulty sensors:

"If 12 seconds after a temperature event, no new temperature event is received from the same sensor, the sensor is broken."

```
DETECT
  failure{ var S }
ON
  and{
    event t : temp{{ sensor_id{{ var S }} }},
    event r : timer:from-end [ event t, 12 sec ],
    while r : not temp {{ sensor_id{{ var S }} }}
  }
END
```

Basically, this rule constructs a time interval during which no other temperature event of the same sensor should occur. This is achieved by creating a relative timer event r which extends the duration of the event t by twelve seconds and the additional condition that, while the timer event r lasts, not temperature event of the same sensor should occur in the event stream.

The last example demonstrates how XChange^{EQ} can be used to aggregate multiple events within a certain time period. This might help to smooth the measurements of a sensor.

"If a temperature event occurs, calculate the average temperature measured by this sensor in the last minute."

```
DETECT
  avg_temp{ var S, value{ avg(all var T) } }
ON
  and {
    event t : temp{{ sensor_id {{ var S }} }},
    event i : timer:from-start-backward[ event t, 1 min ],
    while i : collect temp{{ sensor_id{{ var S }}, value{{ var T }} }}
  }
END
```

This rule also uses a relative time event to specify a time interval during which all temperature events of the same sensor should be gathered. Note that the template matching the temperature event, which is specified after the collect statement, contains a new variable `var T`. Since multiple temperature events may be gathered by the statement, the variable `var T` may be bound to different temperature values and therefore the **DETECT** part of the query uses the all keyword to specify that all these values should be concerned during the computation of the average.

References

[1] J. J. Alferes, A. Brogi, J. A. Leite, and L. M. Pereira. Evolving logic programs. In *JELIA*, pages 50–61, 2002.

[2] J. C. M. Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, 335(2-3):131–146, 2005.

[3] J. C. M. Baeten and W. P. Weijland. Process algebra. *Cambridge Tracts in Theoretical Computer Science*, 18, 1990.

[4] E. Behrends, O. Fritzen, W. May, and F. Schenk. Combining eca rules with process algebras for the semantic web. In *In Proceedings of Second International Conference on Rules and Rule Markup Languages for the Semantic Web*, pages 29–38. IEEE Press, 2006.

[5] E. Behrends, O. Fritzen, W. May, and F. Schenk. Event algebras and process algebras in eca rules. In *Fundamenta Informaticae 82*, pages 237–263. IOS Press, 2008.

[6] E. Behrends, O. Fritzen, W. May, and D. Schubert. An ECA engine for deploying heterogeneous component languages in the Semantic Web. In *Proc. Int. Workshop Reactivity on the Web*, volume 4254 of *LNCS*, pages 887–898, 2006.

[7] B. Berstel, P. Bonnard, F. Bry, M. Eckert, and P.-L. Pătrânjan. Reactive rules on the Web. In *Reasoning Web, Int. Summer School*, volume 4636 of *LNCS*, pages 183–239. Springer, 2007.

[8] A. J. Bonner and M. Kifer. Transaction logic programming. In *ICLP*, pages 257–279, 1993.

[9] A. J. Bonner and M. Kifer. Transaction logic programming (or, a logic of procedural and declarative knowledge). Technical report, University of Toronto, 1995.

[10] F. Bry and M. Eckert. Twelve theses on reactive rules for the Web (invited paper). In *Proc. Int. Workshop Reactivity on the Web*, volume 4254 of *LNCS*, pages 842–854. Springer, 2006.

[11] F. Bry and M. Eckert. Rule-Based Composite Event Queries: The Language XChange^{EQ} and its Semantics. In *Proc. Int. Conf. on Web Reasoning and Rule Systems*, volume 4524 of *LNCS*, pages 16–30. Springer, 2007.

[12] F. Bry, M. Eckert, and P.-L. Pătrânjan. Reactivity on the Web: Paradigms and applications of the language XChange. *J. of Web Engineering*, 5(1):3–24, 2006.

[13] F. Bry, M. Eckert, and P.-L. Pătrânjan. XChange: Rule-based reactivity for the Web. In M. Lytras, editor, *Semantic Web Fact Book*. AIS SIGSEMIS, 2006.

[14] F. Bry, M. Eckert, P.-L. Pătrânjan, and I. Romanenko. Realizing business processes with eca rules: Benefits, challenges, limits. In *Proceedings of 4th Workshop on Principles and Practice of Semantic Web Reasoning, Budva, Montenegro (10th–11th June 2006)*, LNCS, 2006.

[15] F. Bry, M. Eckert, P.-L. Pătrânjan, and I. Romanenko. Realizing business processes with ECA rules: Benefits, challenges, limits. In *Proc. Int. Workshop on Principles and Practice of Semantic Web*, volume 4187 of *LNCS*, pages 48–62. Springer, 2006.

[16] S. Ceri, R. Cochrane, and J. Widom. Practical applications of triggers and constraints: Success and lingering issues (10-year award). In A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 254–262. Morgan Kaufmann, 2000.

[17] P. Doherty, J. Gustafsson, L. Karlsson, and J. Kvarnström. Tal: Temporal action logics language specification and tutorial. *Electron. Trans. Artif. Intell.*, 2:273–306, 1998.

[18] M. Eckert. Reactivity on the Web: Event Queries and Composite Event Detection in XChange. Master's thesis (Diplomarbeit), Institute for Informatics, University of Munich, 2005.

[19] M. Eckert. *Complex Event Processing with XChange^{EQ}: Language Design, Formal Semantics and Incremental Evaluation for Querying Events*. PhD thesis, Institute for Informatics, University of Munich, 2008.

[20] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A logic programming approach to knowledge-state planning: Semantics and complexity. *ACM Trans. Comput. Log.*, 5(2):206–263, 2004.

[21] T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. A framework for declarative update specifications in logic programs. In *IJCAI'01: Proceedings of the 17th international joint conference on Artificial intelligence*, pages 649–654, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[22] C. Forgy. OPS5 user's manual. Technical Report CMU-CS-81-135, Carnegie Mellon University, 1981.

[23] C. Forgy and J. P. McDermott. OPS, a domain-independent production system language. In *Proc. Int. Joint Conf. on Artificial Intelligence*, pages 933–939. William Kaufmann, 1977.

[24] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[25] M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17:301–322, 1993.

[26] E. Giunchiglia, G. N. Kartha, and V. Lifschitz. Representing action: Indeterminacy and ramifications. *ARTIFICIAL INTELLIGENCE*, 95, 1997.

[27] E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, H. Turner, and J. Lee. Nonmonotonic causal theories. *Artificial Intelligence*, 153:2004, 2004.

[28] E. Giunchiglia and V. Lifschitz. An action language based on causal explanation: Preliminary report. In *In Proc. AAAI-98*, pages 623–630. AAAI Press, 1998.

[29] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[30] ILOG. ILOG JRules. `http://www.ilog.com/products/jrules`.

[31] JBoss.org. Drools. `http://www.jboss.org/drools`.

[32] G. Knolmayer, R. Endl, and M. Pfahrer. Modeling processes and workflows by business rules. In *Business Process Management, Models, Techniques, and Empirical Studies*, volume 1806 of *LNCS*, pages 16–29. Springer, 2000.

[33] A. Kozlenkov and A. Paschke. Prova rule language. `http://www.prova.ws/`.

[34] A. Kozlenkov, R. Penaloza, V. Nigam, L. Royer, G. Dawelbait, and M. Schroeder. Prova: Rule-based Java scripting for distributed web applications: A case study in bioinformatics. In *Current Trends in Database Technology (EDBT)*, volume 4254 of *LNCS*, pages 899–908. Springer, 2006.

[35] K. G. Kulkarni, N. M. Mattos, and R. Cochrane. Active database features in sql3. In *Active Rules in Database Systems*, pages 197–219. Springer Verlag, 1999.

[36] J. A. Leite. *Evolving knowledge bases: specification and semantics*. IOS Press, 2003.

[37] J. A. Leite, J. J. Alferes, and L. M. Pereira. On the use of multi-dimensional dynamic logic programming to represent societal agents' viewpoints. In *EPIA*, pages 276–289, 2001.

[38] R. Milner. Calculi for synchrony and asynchrony. *Theor. Comput. Sci.*, 25:267–310, 1983.

[39] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[40] MS Analog Software. ruleCore(R) Complex Event Processing (CEP) Server. `http://www.rulecore.com`.

[41] A. Paschke. Eca-lp / eca-ruleml: A homogeneous event-condition-action logic programming language. *CoRR*, abs/cs/0609143, 2006.

[42] A. Paschke and H. Boley. Rules capturing events and reactivity. In *Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches*, pages 215–252. IGI Global, 2009.

[43] A. Paschke, A. Kozlenkov, and H. Boley. A homogenous reaction rule language for Complex Event Processing. In *In Proc. 2nd Int. Workshop on Event Drive Architecture and Event Processing Systems*, 2007.

[44] A. Paschke, A. Kozlenkov, H. Boley, S. Tabet, M. Kifer, and M. Dean. Reaction RuleML. `http://ibis.in.tum.de/research/ReactionRuleML/`, 2007.

[45] N. W. Paton, J. Campin, A. A. A. Fernandes, and M. H. Williams. Formal specification of active database functionality: A survey. In *Rules in Database Systems*, pages 21–37, 1995.

[46] P.-L. Pătrânjan. *The Language XChange: A Declarative Approach to Reactivity on the Web*. PhD thesis, Institute for Informatics, University of Munich, 2005.

[47] Sandia National Laboratories. Jess, the rule engine for the Java(TM) platform. `http://herzberg.ca.sandia.gov/`.

[48] K.-U. Schmidt, D. Anicic, and R. Stühmer. Event-driven reactivity: A survey and requirements analysis. In *SBPM2008: 3rd Int. Workshop on Semantic Business Process Management in Conjunction with the 5th European Semantic Web Conf. (ESWC'08)*. CEUR Workshop Proceedings, 2008.

[49] M. Seiriö and M. Berndtsson. Design and implementation of an ECA rule markup language. In *Proc. Int. Conf. on Rules and Rule Markup Languages for the Semantic Web*, volume 3791 of *LNCS*, pages 98–112. Springer, 2005.

[50] T. C. Son and C. Baral. Formalizing sensing actions a transition function based approach. *Artif. Intell.*, 125(1-2):19–91, 2001.

[51] M. Stonebraker and G. Kemnitz. The postgres next generation database management system. *Commun. ACM*, 34(10):78–92, 1991.

[52] J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996.

[53] J. Widom and S. J. Finkelstein. Set-oriented production rules in relational database systems. In *SIGMOD Conference*, pages 259–270, 1990.

[54] D. Zimmer and R. Unland. On the semantics of complex events in active database management systems. *Data Engineering, International Conference on*, 0:392, 1999.