View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Open Access LMU

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

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

D4.3
Dura – Concepts and Examples

Due date of deliverable: 31/12/2010
Actual submission date: 28/02/2011

Revision: Version 1.1

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)	Steffen Hausmann, Simon Brodt, François Bry
Contributor(s)	

Preface

The reactive event processing language, that is developed in the context of this project, has been called DEAL in previous documents. When we chose this name for our language it has not been used by other authors working in the same research area (complex event processing). However, in the meantime it appears in publications of other authors and because we have not used the name in publications yet we cannot claim that we were the first to use it.

In order to avoid ambiguities and name conflicts in future publications we decided to rename our language to Dura which stands for "Declarative uniform reactive event processing language". Therefore the title of this deliverable has been updated to "Dura – Concepts and Examples".

Index

1. Introduction

The following text combines an informal (none the less precise) introduction to Dura, a reactive event processing language tailored to reactive emergency management, and a formal language description that is intended for computer science experts. The text introduces the language Dura and its main concepts on the basis of practically relevant but simplified examples which are inspired by the three use cases [30, 34, 29, 11, 32].

Emergency management typically includes the assessment of situations and based thereon the execution of emergency procedures either as direct responses to an emergency or as counteractions to a yet emerging, or possibly emerging, emergency. These days, the knowledge about situation assessment and emergency procedures is written in manuals or is the (not necessarily conceptualized) expertise of emergency professionals.

Dura is a contribution towards a greater automation of emergency management. Therefore, Dura has been designed to provide means for the specification of this kind of emergency management related knowledge. The programming paradigm retained for Dura is that of declarative and reactive rules which enables an automatic situation assessment and even (semi)automatic reactions to (emerging) emergencies.

Dura combines event queries, queries to stateful objects and the specification of complex actions in a homogeneous and integrated fashion. All these three concepts (events, stateful objects, and actions) are highly desirable for modern emergency management systems in order to obtain a reactive and dynamic behavior [6, 5]. Complex event queries detect situations, stateful objects model states and properties of physical or virtual objects, and actions modify components of the infrastructure according to the detected situations and its state.

A core principle of Dura is a strict separation of query dimensions for both, complex event queries and complex actions. Approaches which mix several dimensions into single operators loose expressiveness [10]. While operators which are covering multiple dimensions are well suited for specific purposes, queries based on these operators are getting longish and complicated in more general cases. However, since emergency management often deals with very complicated circumstances, rather simple approaches which are interleaving several query dimensions are undesirable in this context.

The following description of Dura is divided into three main parts which are dealing with the three major concepts of Dura. The situation assessment is covered by means of complex event queries which are introduced in Section 3. Section 5 addresses states and stateful objects. And finally, the execution of actions and specification of emergency procedures is covered in Section 6. Moreover, section 4 covers the data cleansing capabilities of Dura and appendix A contains a formal description of the language.

2. Reactive Emergency Management in a Nutshell

Reactive emergency management integrates several components which need to interact in order to achieve a reactive behaviour which is highly desirable for a modern emergency management. Sensors spread all over the infrastructure collect data and monitor the condition of the infrastructure and its devices [30]. SCADA systems offer access to the information of entire subsystems and enable the remote operation of certain devices. Simulators predict the development of scenarios which requires a proper initialization of boundary conditions and the distribution of results to other components [31]. The event engine is evaluating Dura programs. It receives measurements from sensors and simulators in form of events, interprets the information, and derives an abstraction of the state of the infrastructure for further processing and presentation purposes. A graphical user interface supports operators by visualizing the state of the infrastructure. Moreover, it serves as an interface between the operators and the equipment of the infrastructure [33]. A trainings environment called SITE is used to enable the qualification and preparation of operators [33].

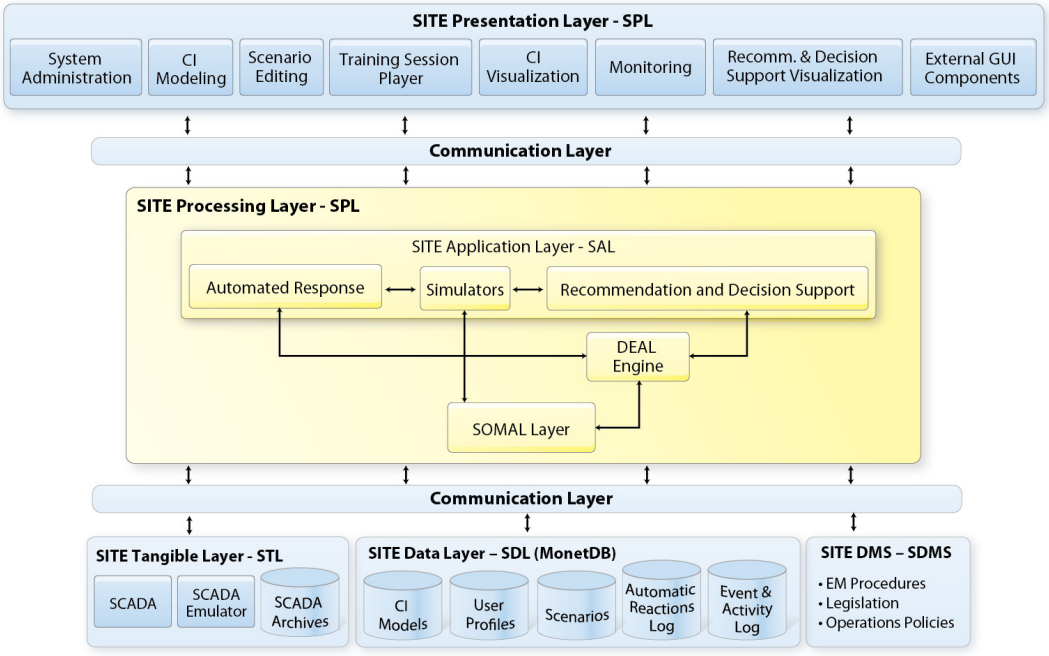

Figure 1: Overview of the architecture [35].

Most of the components cover only certain aspects required for reactive emergency management and cannot be used to operate a critical infrastructure on their own.[1] Therefore, it is crucial to provide a generic communication interface that distributes the information that is required by each component in an easy and efficient manner.

[1] SCADA systems are actually capable of operating critical infrastructures. However, even modern SCADA systems cover only certain aspects of the innovations of EMILI. Complex events and complex actions, integration of simulations, semantic information models, etc. are only partially supported.

[35] describes in detail how a communication layer suitable for the specific requirements of a reactive emergency management can be efficiently realized by a publish/subscribe architecture. Therefore, components connected to the communication layer publish which kind of messages they provides and subscribe to messages they are interested in. The communication layer then takes care that messages are distributed among connected components in the right way.

An alternative approach that stores events coming from a SCADA system inside a MonetDB database is described in [16]. Both approaches have certain advantages and are employed in parallel. Therefore it can be assumed that all kinds of events are distributed over the communication layer and are as wells stored in a MonetDB database.

The properties of the communication layer has a substantial impact on how components of the infrastructure interact. Messages are always sent from producers, such as sensors, to consumers, such as the SITE GUI. Querying and retrieval of information from other components is only indirectly possible. Note that components can be producers and consumers at the same time. SCADA systems, for instance, send sensor readings in form of event messages and are subscribed to commands from the GUI and the event processing system.

The event processing engine is a central element for situation assessment and reactive behaviour. It interacts with many components in order to achieve its task. Several components provide input for its computations. Sensors and SCADA systems describe the current conditions and incidents of the infrastructure in form of events. Furthermore, simulators provide events which are likely to happen in the future.

The engine evaluates Dura programs which describe how events and other informations are combined to a more abstract representation of incidents and the infrastructure's condition. Based on the input events that are stored in the MonetDB database and other semantic information, such as spatial relationships of sensors, the event engine derives new events, according to the Dura program. The derived events are then sent to other components that rely on these events (and are therefore subscribed to them).

Derived events are furthermore used to trigger actions that affect the actuators of the infrastructure. For executing actions, the event processing system issues an action request over the communication layer. Actuators which are capable of executing the command are subscribed to the according message and therefore are informed to execute the action. If information about the execution status of the command is available, it is sent back to the event engine via the communication layer.

In order to consider information coming from simulators, the event engine starts a simulation by sending an according action to the simulator. The simulator reads the required boundary conditions from the database and sends the result back to the event processing engine in form of events.

In certain conditions, the event engine needs input from the operator before drawing further conclusions or initiate actions. In order to get the desired feedback from the operator, the event engine initiates an action that is sent to the GUI and causes it to request the required information from the operator. The result is then sent back to the event engine in form of an

event.

Furthermore, the GUI provides a graphical overview of the infrastructures condition. Derived events (and events coming from sensors) that are related to the conditions of the infrastructure are thus used to adapt the graphical representation.

Note that this short overview is mainly considering interactions between the event processing engine and other components of the system. There are of cause further dependencies between components that have not been fully considered.

3. Events

The main task of emergency management is dealing with risk and risk avoidance [17]. Indeed, "in oder to provide proper emergency management, critical situations and risks have to be assessed" [30]. This includes the assessment of situations before and during an emergency and interventions of the emergency managers to either avoid a possible emergency or to get an emergency under control as quickly as possible.

This section deals with the assessment and interpretation of situations, that is, the collection of information coming from sensors and the subsequent derivation of higher level knowledge which represents the condition of the infrastructure in a more aggregated and abstract view.

The information coming from sensors is considered as a stream of events. In general an event is a message indicating that something of interest happens or is contemplated as happening (or not happening) [21]. This can be for instance a message of a smoke sensor that has detected smoke or the message of a temperature sensor which repeatedly communicates the current temperature at its location. Every events is associated with a time point or time interval which indicates when it actually occurred.[2] Having a notion of time is highly desirable for an event processing systems, since it enables to reason about temporal dependencies between the events of the event stream. Emergency management in general requires a notion of time and time-consciousness. It is therefore natural and requisite to consider events with a notion of time in emergency management.

3.1. Properties of Events

Event Messages Events are represented in Dura by XML-like semi-structured data of a restricted kind. Restricted means that the schema of each event type must provide bounds for the width and depth of valid event messages. Thus, recursive definitions, definitions using alternatives and unlimited repetitions of tags are not allowed in the schema of event definitions.

These restrictions of the schema are required to enable a simple and efficient mapping of event messages to flat tuples inside a database. Indeed, more generic event messages are not necessary for the implementation of the use cases and would significantly complicate the language and its processing.

[2] Time points can be seen as time intervals with the same lower and upper bound.

Not allowing structured event messages of an arbitrary form surely is no restriction for applications: In practice, there is a finite number of distinct types of events which can be specified as the application is designed.

Occurrence Time Every event is associated with its occurrence time. The occurrence time is represented in Dura by a closed and connected time interval $t = [b, e]$. For events which happen at a time point, the interval degenerates to a single time point $t = [b, b]$.

Events are denoted in the following e^t where e is the event message and $t = [b, e]$ its occurrence time.[3] Furthermore, b is called the beginning and e the end of the event. For convenience, given an occurrence time $t = [b, e]$, $begin(t)$ denotes b and $end(t)$ denotes e.

Further Times of Events Events may be associated with other times than the occurrence time. For instance, events which are coming from a simulator have an occurrence time which indicates when the data required for the simulation was actually available. It thus indicates how current the simulation events are.[4] Furthermore, a simulation event is associated with a semantic time which indicates when the event is deemed to occur according to the simulation. Likely, the semantic time will be after the occurrence time since the simulation is used to predict which events will occur in the future.

More times might be needed in some applications, even in some emergency management applications. Dura provides a framework for expressing such times. For the sake of simplicity and readability, this issue is not further addressed in this presentation of the language. So far, the three use cases [34, 29, 11] require only the two above mentioned times: time of occurrence in real life and time of occurrence according to a simulation. The necessary extension for accommodating more than these two times will be presented in more details if, in the future, the primary usage of Dura, the use cases, make it necessary.

3.2. Complex Event Query Dimensions

In general, event queries access the payload of events, that is, the data events carry, detect patterns composed of multiple events, consider temporal and other relationships between events, accumulate events for negation and aggregation, and consider the provenance and spatial aspects of events.

Four of these dimensions have been described in [7], namely, *data extraction*, *event composition*, *temporal relationships*, and *accumulation*. The *provenance* of events and *spatial relationships* are two further aspects which are especially important for emergency management in large infrastructures.

For an easy-to-use, high level event query language the strict separation of query dimensions is desirable. A language which mixes multiple query dimensions in single operators may lead to

[3] The notation e^t is not part of Dura but of the meta-language used here for describing Dura.

[4] This is useful, for instance, to distinguish simulation events of several simulation runs with different initial conditions that are based on observations which are getting more and more accurate over time.

complicated queries that are hard to maintain, since simple semantic modifications of queries can result in large syntactical changes [6]. Although operators which are combining several query dimensions are well suited for very specific situations, a generic language requires more general operators which are independent of each other to gain high expressiveness.

In the following, the query dimensions are briefly introduced. They are described in more detail in Section 3.3, 3.6, 3.7, 3.8, and 3.9.

Data Extraction Events carry data, the so-called payload, which affects the further reactions of the system and the interpretation of events. Therefore, the data needs to be extracted from events and bound to variables in order to provide the data for further processing. The processing includes tests on the bound values, or comparisons of the data of multiple events.

Provenance Provenance of events in general is the explanation how an event has been derived, that is, the base events and rules which have (directly or indirectly) contributed to the derivation of an event. However, such a generic (and thus in terms of time and space efficiency exhausting) form of provenance is so far not required by the use cases [34, 29, 11]. For the use cases it is rather necessary to distinguish between events which have been derived from simulation events and events which are based on real-life data.

Since this form of provenance is application dependant, there is no specific mean to address the provenance of events in Dura. However, provenance is supported by including so-called provenance marks in the payload of events. The programmer is responsible for writing rules which are passing the marks from one event or action to another in the right way.

If it is required by the use cases then a more generic form of provenance, such as an easy access to the complete derivation history of events, could be included in the language in the future. But so far it seems to be sufficient to consider provenance as a special form of data extraction.

Spatial Relationships Spatial information that is carried by events can be used to check spatial relationships between events. Based on the spatial information, a confirmed alarm can be generated for example only if there are two alarm events that are emitted by sensors which are near each other.

Spatial relationships can be implemented by including for instance GPS coordinates or similar position information in the payload of events which is then interpreted by appropriate complex event queries.

If it is required by the use cases, a more generic form of spatial relations could be included in the language in the future. But for now spatial relations are considered as a special form of data extraction, because they are extremely application dependent. Note that, so far a generic approach for spatial relations is not required by the use cases [34, 29, 11].

Event Composition Complex events are derived by complex event queries which specify patterns of events. To express how events are combined in order to match a query, composition operators, such as conjunction and disjunction, are required. In Dura these operators are denoted `and` and `or`. Negation, also provided by Dura, is mentioned below under "Accumulation".

Due to the strict separation of query dimensions, no further composition operators are necessary. For example, a query for a sequence of events is expressed using the combination of multiple query dimensions, namely event composition and temporal relationships.

Temporal Relationships Event queries must be able to express temporal relationships between events such as "event *A* occurs before event *B*" or "events *A* and *B* occur within 2 minutes". In addition to that, further comparisons on the payload of events, such as a test if two alarms occurred in the same area, and spatial relations between objects are needed.

Often used temporal relations are *before, after, during, while, at* and *within*. Further relations are given in Section 3.7.

Accumulation Accumulation is required for several reasons. First of all, accumulation can be used for collecting events in a given time window and to combine the data of the collected events, for instance by computing the average of some of their values. Furthermore, accumulation is needed to express negation, that is, recognizing the absence of events, or to query the existence of events during a given time window.

3.3. Querying Simple Events

A pattern-based approach is used to query (simple or atomic) events and to extract data from their payload. That means that queries are specified in form of patterns which closely resemble the actually queried data. This gives rise to an intuitive and easy-to-use way of data extraction, since queries can be easily developed from the data which is actually queried.

Example Consider a sensor that emits temperature events which carry the id of the sensor, the measured temperature and the unit of the temperature measurement in the payload of the event. The following event message is an example of such an event which is emitted at time 11 and reports the temperature of 27°C from sensor `0x5`.

```
temp{ id{ "0x5" }, temperature{ 27 }, unit{ "centigrade" } }[11,11]
```

In order to extract values from these kind of temperature events, the pattern of the event's data is repeated and values that will be extracted are substituted by variables. Furthermore the keyword `event` and an event identifier are prefixed to the query in order to distinguish it from queries of stateful objects which are introduced in Section 5.

To extract for instance temperature values of sensor `0x5` which are given in `centigrade`, the variable `T` is used in the query pattern instead of the actual temperature value.

```
event e: temp{ id{ "0x5" },
               temperature{ var T }, unit{ "centigrade" } }
```

It is not necessary to repeat the complete data pattern for every query, values one is not interested in can be dropped from the pattern. Furthermore, event identifiers are not mandatory and can be skipped from the event query as well.

An incomplete query pattern is used in the following to query `temp` events and to extract the measured temperature, regardless of the sensor id or the unit of the measurement.

```
event: temp{ temperature{ var T } }
```

Event Identifier For now, event identifier seem to be redundant since they do not further restrict the event message and are not used in another part of the query. However, event identifiers will be used later for complex event queries which are querying multiple events. Furthermore, event identifiers can be included in the payload of derived events in order to provide provenance information in form of references to events.

3.4. Deductive Rules for Events

Complex event queries are deductive rules which detect new events based on the incoming simple (or atomic) events of the event stream(s). In general these rules have the form `DETECT < derived event> ON <event query> END`. The `derived event` part of a query is also referred to as rule head, the `event query` part as rule body.

When the `event query` finds a match on the events of the stream, the event specified in the rule head is derived and can be subsequently processed by other rules. Note that events which are contributing to a match of an event query are not consumed in any sense; they can contribute to an arbitrary number of event queries. It can even be assumed, that all events of the event stream are stored in the event engine forever.[5]

Deductive rules which query a single event can also be used to transform the representation of events. They can be used to harmonize the format of events which are carrying the same information in different representations. Having a harmonized representation of events is beneficial for the development and maintenance of further rules. Rules which are based on events of a certain type do not need to deal with the different representations of the same information. They can instead query the harmonized events which carry the information in a standardized way.

[5] Internally events are indeed deleted. However, only events which can no longer contribute to any complex event query, and are thus not relevant anymore, might be deleted.

Example Deductive rules are useful to unify the representations of events that are sent by sensors from different manufacturers. The first kind of sensors emits `measurement` events which state the temperature in degree Celsius whereas another kind of sensors emits `sensor-reading` events which state the temperature in degree Fahrenheit. The following rules are used to derive `temp` events of a unified form, representing the temperature consistently in degree Celsius.

```
DETECT
   temp{ id{ var Id }, temperature{ var C } }
ON
   event: measurement{ sensor{ var Id },
             temperature{ var C }, unit{ "centigrade" } }
END

DETECT
   temp{ id{ var Id }, temperature{ (var F - 32) * 5/9 } }
ON
   event: sensor-reading{ id{ var Id }, type{ "e3" },
             value-t3{ var F } }
END
```

Occurrence Time The occurrence time of derived events depends on the time of the events it was derived from. If the body of a rule contains only a single event, the occurrence time of the derived event coincides with the occurrence time of the event which matches the rule body. If the rule body contains multiple events, the occurrence time of the derived event is determined by the smallest time interval that covers the occurrence time of all events matched by the rule body. Thus, the occurrence time of a derived event is not the time of its detection.

Recursive Queries In order to enable an efficient evaluation of complex event queries, recursive definitions of rules are not permitted in Dura. Thus, recursive queries, such as `DETECT temp{ ... } ON event: temp{ ... } END`, cannot be specified in a Dura program.

This restriction is quite common in the field of complex event processing. It enables an efficient evaluation of complex event queries whereas the general expressiveness of queries is reduced. However, recursive queries can often be avoided in concrete applications. Indeed, complex event queries are designed to detect situations based on the occurrence of events and not to compute generic functions (which would actually require recursive rules). Moreover, none of the three use cases seems to require recursive deductive rules so far [34, 29, 11].

3.5. Hierarchies of Events

Deductive rules can also be used to define a hierarchy, or even a directed acyclic graph, for events. A hierarchy may express for instance that an event of type A is also of type B. This allows for programs that consist of few generic rules which react on abstract rules instead of multiple very specialized rules.

Example On a platform of a metro station the occurrence of smoke and an unusually high temperature is classified as an uncertain fire alarm. Moreover, an uncertain fire alarm as well as the usage of an SOS telephone is an uncertain alarm. Whenever an uncertain alarm occurs on a platform, the emergency operator is requested to verify or to revoke the alarm in order to trigger appropriate reactions.

Such a behaviour can be realized in two different ways. First of all, one could create different rules which request a confirmation from the emergency operator when smoke, high temperature or a SOS call is detected. These rules are highly redundant since they basically request the same confirmation message, only the bodies of the queries differ since the relationships between the events are not represented in the program.

A second and more generic approach uses the preceding rules to model the relationships between the events. Therefore, a single rule which requests a confirmation message in case an uncertain alarm occurs is sufficient, since, for instance, a smoke event automatically causes an uncertain alarm event.

```
DETECT
   uncertain-alarm{ area{ var A } }
ON
   event: uncertain-fire-alarm{ area{ var A } }
END

DETECT
   uncertain-fire-alarm{ id{ var Id }, area{ var A } }
ON
   event: smoke{ id{ var Id }, area{ var A } }
END

DETECT
   uncertain-fire-alarm{ id{ var Id }, area{ var A } }
ON
   event: high-temp{ id{ var Id }, area{ var A } }
END

DETECT
   uncertain-alarm{ area{ var A } }
ON
   event: sos-call{ area{ var A } }
END
```

3.6. Event Composition

So far, only rules with a single event query in their bodies have been considered. However, most incidents that are relevant for emergency management cannot be detected by considering only single events. The occurrence (or absence) of multiple events needs to be combined in order to represent a reasonable assessment of the situation.

Indeed, the integration of different emergency related systems is one of the major goals Dura

has been designed to contribute to. Current systems are insufficient for a reactive emergency management in many aspects. They are lacking capabilities of interacting with the physical world, have no notion of states do not support the timing of actions [5]. However, incidents of the past have shown that a lack of integration can lead to fatalities which could have been prevented by incorporating the information of different systems [12, 9]. Furthermore, a requirement of the use cases is the "integration of information from different subsystems, correlation of events and automatic response to relevant events, whenever feasible" [30].

In Dura, several event queries can be combined using either conjunction (`and`), disjunction (`or`) or negation (`not`, understood as expressing an absence). Both `and` and `or` can have two or more event queries as arguments and can be nested arbitrarily.

Example In the last example, the operator was requested to confirm every uncertain alarm. However, if smoke occurs in an area and the temperature in the same area is unusually high, the confirmation from the operator can be skipped and a certain alarm can be derived immediately because it is very unlikely that the two independent sensors are malfunctioning at the same time.

```
DETECT
   certain-alarm{ area{ var A } }
ON
   and{
      event i: high-temp{ area{ var A } },
      event j: smoke{ area{ var A } }
   }
END
```

Note that specifying the variable `var A` twice enforces that the are attributes of both events are equal. In addition, event identifiers, namely `i` and `j`, are added to the queries which can be used in combination with temporal relations to reference the occurrence time of the matched events.

Example Event composition also allows for a more compact and simple representation of several rules which are compound of the same sub-queries and have the same head. For instance, the four rules of the example in Section 3.4 can be rewritten to the following two equivalent, even so more compact, rules.

```
DETECT
   uncertain-alarm{ area{ var A } }
ON
   or{
      event: uncertain-fire-alarm{ area{ var A } },
      event: sos-call{ area{ var A } }
   }
END

DETECT
   uncertain-fire-alarm{ id{ var Id }, area{ var A } }
ON
   or{
      event: smoke{ id{ var Id }, area{ var A } },
      event: high-temp{ id{ var Id }, area{ var A } }
   }
END
```

3.7. Temporal (and other) Relationships

Temporal relationships play an important role in queries involving multiple events. They specify temporal dependencies between occurrence times of events, for instance a query only matches if a certain event occurs after some other event. Besides temporal relationships, constraints on the extracted data can be specified, such as the value of a variable binding must be greater than a certain value.

Temporal relations are used to constrain the occurrence time of events. They are specified in the `where` part of a query, that can be attached at the end of any conjunction and disjunction. There are two different types of temporal relations: qualitative relations which restrict the order and quantitative relations which restrict the duration of events.

Temporal Relations for Time Intervals Allen's thirteen relations [2] are a well-established convenient mean to express qualitative temporal relations between time intervals. The thirteen relations include *before, meets, overlaps, starts, during, finishes*, the corresponding inverses and *equals*.

In Dura, these relations can be used to compare time intervals of events, such as the occurrence time of events, which are given by the respective event identifiers. For instance, given two event identifiers `i` and `j`, the expression `i before j` guarantees that the occurrence time of the event `i` has ended before the occurrence time of the event `j` has started, that is $end(i) < begin(j)$. The precise semantics of all relations are given in [10, p. 179].

In addition to Allen's thirteen relations, Dura supports two more quantitative temporal relations, namely `while` and `at`. The expression `i while j` guarantees that the occurrence time of `i` is contained in that of `j`, that is, $begin(j) \leq begin(i)$ and $end(i) \leq end(j)$.[6] The relation `at`

[6]Note that `contains` is defined similarly but requires strict containment of the time intervals.

takes an interval and a time point as parameters and checks whether the time point lies in the time interval, that is, `i at t` guarantees that $begin(i) \leq t$ and $t \leq end(i)$.

Furthermore, there are two quantitative or metric temporal relations, namely `within` and `apart` . Given event identifiers i_k and a duration d, the expression $\{i_1, \ldots, i_n\}$ `within` d guarantees that $min\{begin(i_1), \ldots, begin(i_n)\} - max\{end(i_1), \ldots, end(i_n)\} \leq d$ holds for the occurrence times of the according events. The relation `apart` is defined similarly with $\geq$ substituted for $\leq$.

Temporal Relations for Time Points Temporal relations between time points can be specified using expressions of the form $t_1 - t_2 \leq d$ and $t_1 - t_2 < d$, where d is a duration and t_1 and t_2 are time points, such as `begin(i)` and `end(i)` for an event identifier `i`.

Note that the given qualitative and quantitative temporal relations for time intervals can be easily transformed to expressions comparing the beginning and ending of time intervals. However, the transformation of one relation comparing time intervals may result in several expressions comparing time points. For instance, `i during j` becomes `begin(j)< begin(i)` and `end(i)< begin(j)`.

For convenience, the programmer can choose between both kinds of relations: temporal relations for time intervals or time points. Depending on the temporal constraints, one kind of temporal relations can be better suited than the other. Thus, having the choice between both kinds of relations is desirable.

Relations on Event Data Beside temporal relations, there are further relations which can be used in combination with variables. `=` and `!=` test whether the values of two variables are (un)equal. If the variables are bound to numbers, the additional relations `<`, `<=`, `>`, and `>=` can be used to compare the bounded values.

Example The following example is a generalization of the first example of Section 3.6. Instead of deriving a `certain-alarm` event on the occurrence of specific sensor signals, a certain alarm is derived whenever at least two uncertain alarms occur in the same area within 2 minutes.

Two conditions, one temporal condition and one condition on the data of the events, are added to the `where` part of the query. The first condition restricts the time interval in which both events can to occur to at most 2 minutes. The second one requires the ids of both events to be different.

```
DETECT
   certain-alarm{ area{ var A } }
ON
   and{
      event i: uncertain-alarm{ area{ var A } },
      event j: uncertain-alarm{ area{ var A } }
   } where { {i,j} within 2 min, event i != event j }
END
```

Note that, in this example, there are always two certain alarms derived, if two uncertain alarms occur. Because both queries in the body of the rule are equal, they are matching the same events. Therefore, each of the two `uncertain-alarm` events can be matched by either the first or the second query and thus two certain alarms are derived by the rule.

If this behaviour is undesirable, other means that are introduced in Section 5.5 can be used to prevent the derivation of multiple uncertain alarms. However, these means require a somewhat different modeling of the systems.

Time Interval Operators If desired, new time intervals can be computed based on the occurrence time of events by means of the following functions. These time intervals can then be used in combination with Allen's thirteen relations to (indirectly) specify time constraints between events. Below, e is an event identifier, d a duration and t the resulting time interval.

- `extend(e, d)`: extends the time interval of `e` at its end by the duration `d`, i.e., $begin(t) = begin(e)$ and $end(t) = end(e) + d$.
- `shift-forward(e, d)`: the time interval of `e` is shifted forward in time by the duration `d`, i.e. $begin(t) = begin(e) + d$ and $end(t) = end(e) + d$.
- `from-start(e, d)`: extends the time interval of `e` from its start by the duration `d`, i.e. $begin(t) = begin(e), end(t) = begin(e) + d$.

Further functions include `shorten`, `extend-begin`, `shorten-begin`, `shift-backward`, `from-end`, `from-end-backward` and `from-start-backward`. Their precise semantics are given in [10, p. 105]. Since event identifiers are used to refer to time intervals of events, the given functions can be specified wherever an event identifier can be specified. In particular, nesting of several functions is possible.

Note that these functions are not required if temporal relationships between events are specified using expressions of the form $t_1 - t_2 \leq d$. The extension and shifting of time intervals can be easily incorporated into these kind of expressions by adapting the value of d.

3.8. Event Accumulation

Time interval operators are often used in combination with event accumulation to enable non-monotonic queries, such as negation and aggregation. Non-monotonic queries are queries for which it cannot be guaranteed that they still match the stream of events when new events occur.

For example, the query `not event: e{}` only matches the event stream as long as no `e` event occurs.

In common databases, non-monotonic queries can be easily evaluated since the complete dataset is stored in the database. However, in the field of complex event processing, events arrive over time from a conceptually infinite stream of events which makes the evaluation of non-monotonic queries with respect to the whole stream impossible. For instance, for evaluating a negated query, one would have to wait until all events have arrived in the system, that is, possibly forever.

Therefore, means are provided to collect events in a finite time window and evaluate queries on the collected events of the given time window.

Negation Negation is expressed as a prefix operator `not`. The time during which the negated event query should not match needs to be constrained to a bounded time window. This can be accomplished by using temporal relations in combination with a relative time interval.

Example Temperature sensors in a metro station report the current temperature in regular intervals, for instance every 10 seconds. Therefore, one can conclude that a sensor is broken if 12 seconds after the last measurement no further temperature event is received from the sensor.

```
DETECT
   sensor-broken{ id{ var Id } }
ON
   and{
      event i: temp{ id{ var Id } },
      not event j: temp{ id{ var Id } }
   } where { j during extend(i, 12 sec) }
END
```

The condition `j during extend(i, 12 sec)` ensures that only temperature events match the non negated part of the second query which occur at most 12 seconds after the temperature event matched by the first query. And since the second query is negated, the complete event query matches only if no such temperature event occurs.

The evaluation of the query is feasible since the negated part of the query is limited to a finite time window. If a temperature event occurs, further temperature events need to be regarded for the next 12 seconds only in order to answer the query. Without any time constraints on the negated part of the query one would need to wait forever to give a correct answer to the query which is certainly impossible.

Aggregation Aggregation is used to combine the values from several events which have been collected within a certain time interval. Collection and aggregation are expressed in different parts of the query. The collection of events and the extraction of data is done in the

body of a query whereas the aggregation is specified in the query head where derived events are constructed.

In order to group together several values which have been extracted from events `all` and `group-by` are used. `group-by` specifies according to which attributes events are separated into groups and `all` collects all values of a group in order to pass them to one of the provided aggregation functions `sum`, `min`, `max`, and `mean`.

Example Since sensors suffer from inaccuracies and measurement errors [6] it is necessary to perform data cleansing in order to obtain reliable measurements and to avoid false alarms.

So far we got the feedback from the use case partners that very simple filters are sufficient for the implementation of the use cases. Therefore data cleansing of the sensor signals can be either done by the sensors itself, the SCADA system or, if the latter two options are not available, by appropriate complex event queries.

The following rule uses event aggregation and grouping to derive the average temperatures of all areas within the last five minutes whenever a temperature event is detected.

```
DETECT
   avg-temp{ area{ var A },
      temperature{ avg(var C) } } group-by {event i, var A}
ON
   and{
      event i: temp{ },
      event j: temp{ area{ var A }, temperature{ var C } }
   } where { j during from-begin-backward(i, 5 min) }
END
```

Existential Queries In general, event queries of a rule may match multiple events of the event stream and thus the rule may derive multiple events as well. However, sometimes one is only interested whether there is at least one event that matches a query instead of every event that matches it. To this end queries can be existentially quantified by adding the keyword `exists` in front of them.

Example For instance, a high temperature should only be detected if the temperature exceeds 80°C and the average temperature has exceeded 50°C in the sensor's area within the last 30 seconds.

```
DETECT
   high-temp{ id{ var Id }, area{ var A } }
ON
   and{
      event i: temp{
         id{ var Id },
         area{ var A },
         temperature{ var T1 } },
      exists event j: avg-temp{
         area{ var A },
         temperature{ var T2 } }
   } where { j during from-begin-backward(i, 30 sec),
             var T1 >= 80, var T2 >= 50 }
END
```

If the existential quantification is dropped from the second query, a `high-temp` event is derived every time the average temperature has exceeded its limit. However, the query is intended to check whether the average temperature was too high at least once and thus the existential quantification is needed to derive only a single `high-temp` event.

3.9. Provenance

One of the major innovations of the EMILI project is the use of fast-computed simulation to improve the management of emergencies in large infrastructures. Simulators are used to predict the development of a certain scenario, such as fire in a station, and to enable appropriate reactions ahead of time.

When a simulation is triggered, for instance by the detection of fire, the results of the simulation are sent to the event processing system in form of events. However, if both kinds of events, simulation as well as real-life events, are present in the event processing system, one needs to make sure that events of different types can be distinguished, since they need to be treated differently.

To this end, the provenance information of an event, that is, whether it is the output of a simulator or a real-world sensor, can be included in the payload of the event in the form of provenance marks. Events coming from sensors are marked with `real-life` whereas events coming from a simulator are marked with `simulation`.

Example In order to properly deal with the provenance information of events, the queries of Section 3.6 need to be adapted. The provenance mark of events is just passed over from the base to the derived event.

```
DETECT
   uncertain-fire-alarm{
      id{ var Id }, area{ var A }, provenance{ var P } }
ON
   or{
      event: smoke{ id{ var Id },
         area{ var A }, provenance{ var P } },
      event: high-temp{ id{ var Id },
         area{ var A }, provenance{ var P } }
   }
END
```

Example In the example of Section 3.7, a certain alarm is derived when two uncertain alarms occur within a certain time window. But since the queries does not distinguish between simulated and real-life events, a certain alarm can be derived by simulated smoke events which is highly undesirable; emergency operators can be easily confused and distracted by such events.

Thus, the rule needs to be adapted to derive only certain fire alarms if they are based on real-life events. For this purpose, the query is extended in order to consider the provenance information of the events.

```
DETECT
   certain-alarm{ area{ var A }, provenance{ "real-life" } }
ON
   and{
      event i: uncertain-alarm{
         area{ var A }, provenance{ "real-life" } },
      event j: uncertain-alarm{
         area{ var A }, provenance{ "real-life" } }
   } where { {i,j} within 2 min, event i != event j }
END
```

In the preceding two examples the provenance mark of an event is the same for the base events of a query and the derived event. Either they are both real-life events or simulation events. However, this does not always hold as the next example shows.

Example When fire is detected in a metro station, a simulation is started in order to get a first impression of the likely development of the scenario. If the outcome of the simulation suggests that there will be smoke in a certain area, the emergency operator needs to be informed in form of a `smoke-threat` event in order to involve this information in the selection of the escape paths.

Note that although the smoke event is a simulation event, the derived `smoke-threat` event is marked as a real-life event, since the threat of persons in the area is indeed real.

```
DETECT
   smoke-threat{ area{ var A }, provenance{ "real-life" } }
ON
   and{
      event: certain-alarm{ },
      event: smoke{ area{ var A }, provenance{ "simulation" } }
   }
END
```

4. Data Cleansing

In deliverable D4.1 [6] we identified three orthogonal dimensions of data cleansing, namely *noise reduction and smoothing*, *filtering*, and *generation*. These dimensions of data cleansing are mostly independent of each other and cover certain aspects of the general term data cleansing. Noise reduction and smoothing encompasses methods that deal with stochastic noise and noise caused by interferences from other activities in the infrastructure. Filtering identifies and removes faulty data that occurred cause of sensor failures and human errors. Last but not least, generation tries to fill gaps in the data due to missing sensors or transmission errors.

Because all three dimensions are orthogonal, they require different approaches and methods to be adequately covered. Moreover, the approaches required for each single dimension as well as the demands to the dimensions differ from use case to use case depending on, for instance, the capabilities of the underlying SCADA systems.

4.1. Data Cleansing with Dura

Dura comes with several different capabilities to express data cleansing withing the event processing system. The available methods can be classified according to the three dimensions of data cleansing.

Noise Reduction and Smoothing Noise reduction and smoothing is by far the most basic kind of data cleansing, as it is often carried out by the SCADA system or even by the sensors itself. However, Dura makes it as well possible to express noise reduction and smoothing filters by means of declarative rules.

The following two rules implement a moving average smoothing filter. Whenever a new temperature measurement is detected, the moving average of temperature measurements in the same area and from the same sensor that occurred in the last two minutes is computed.

```
DETECT
   avg-sensor-temp{ id{var Id},
           temperature{avg(var C)} } group-by {event i}
ON
   and{
      event i: temp{ id{var Id} },
      event j: temp{ id{var Id}, temperature{var C} }
   } where { j during from-begin-backward(i, 2 min) }
END

DETECT
   avg-area-temp{ area{var A},
           temperature{avg(var C)} } group-by {event i}
ON
   and{
      event i: temp{ area{var A} },
      event j: temp{ area{var A}, temperature{var C} }
   } where { j during from-begin-backward(i, 2 min) }
END
```

Filtering Faulty data due to sensor failures or human errors can often be recognized by cross-checking information from different sources. This can be realized, for instance, by correlating the information of different sensor types to detect fire, instead of relying on a single sensor or sensors of the same type which have the same (physically induced) deficiencies.

```
DETECT
   fire{ area{var A} }
ON
   and{
      event i: smoke{ area{var A} },
      event j: temp{ area{var A}, temperature{var T} }
   } where { {i,j} within 2 min, var T >= 50 }
END
```

Another way of filtering sensor data is to incorporate expert knowledge or information from physical models to identify impossible or rather very unlikely sensor readings. For example, temperature values above 200 degree are very unlikely in a metro station under normal conditions. Therefore one can drop temperature readings above a certain threshold if the average temperature that has been reported by the sensor within the last two minutes is significantly lower.

```
DETECT
   filtered-temp{ id{var Id}, temperature{var T} }
ON
   and{
      event i: temp{ area{var A}, temperature{var T} },
      event j: avg-sensor-temp{
                  area{var A}, temperature{var Avg} }
   } where { i finishes j,
             or{var T <= 200, var T <= var Avg*5} }
END
```

Generation The physical models that are currently elaborated for the use cases can be used to fill gaps caused by missing or lost sensor data. To this end, missing values are substituted by values coming from a simulator.

The following rule uses simulated values to fill potential gaps of temperature sensor readings. Every 10 seconds, a sensor sends its current measurement to the event processing system. Thus, whenever a sensor reading is missing, that is, 12 seconds after the last temperature event there is still no new measurement, the simulated temperature value of the sensor is considered as if it came from the actual sensor.[7]

```
DETECT
   filtered-temp{ id{var Id}, temperature{var T} }
ON
   and{
      event i: temp{ id{var Id}, provenance{"real-life"} },
      event j: temp{ id{var Id},
         temperature{var T}, provenance{"simulation"} },
      not event k: temp{ id{var Id}, provenance{"real-life"} }
   } where { {j,k} within extend(shift-forward(i, 8 sec), 4 sec) }
END
```

4.2. Advanced Data Cleansing Methods

Dura has some basic yet powerful means to implement data cleansing methods for each of the three dimensions of data cleansing. Particularly being able to write generic rules that express expert knowledge and takes properties of the infrastructure into account is a powerful tool for implementing data cleansing with Dura.

In the literature more advanced and very sophisticated approaches for data cleansing are described [28, 18, 23, 13, 14, 22, 20, 1]. Especially for the first dimension, noise reduction and smoothing, there exists a myriad of different approaches. However, we got the feedback from our use case partners, that the way how data cleansing is realized in Dura is sufficient for the implementation of their use cases because most of the required data cleansing functionality is

[7] Note that this rule can be rewritten so that simulated values are only computed if required.

already provided by their SCADA systems. Missing functionality can be implemented with more or less elaborated rules in the fashion of the rules that were given in the last section.

Therefore, no powerful library for sensor data cleansing is incorporated into the language. The current capabilities of Dura are sufficient and more powerful approaches are not required for the implementation of the use cases, and as data cleansing is highly use case dependent providing a library for data cleansing which is not backed by concrete use case requirements makes little sense.

5. States

States are a desirable mean for reactive behaviour in emergency management. The reaction to a detected event might differ depending on the current state of the infrastructure.

For instance, if a bathroom light breaks in metro station an appropriate event is automatically sent to the system. Under normal operation conditions this information is presented to the operator who informs the facility manager responsible for the station. However, if there is fire in the station, there are way more important tasks to be taken care of. Every information that is not directly related to the emergency is a potential distraction for the operator. Thus, information about broken bathroom lights are not presented to the operator during an emergency.

This example illustrates how the state of a station has an impact on the behavior of the system, i.e., the information of the broken bathroom light is only displayed to the operator if the station is in the state “normal operation mode”.

5.1. States vs. Stateful Objects

Stateful Objects The notion of *stateful objects* is a generic concept which is well suited to model physical or abstract objects which are changing their state over time, as for example a platform of a metro station where people are continuously entering or leaving, trains are arriving, etc., or an escalators that can be running upwards, downwards or being stopped. Furthermore, stateful objects can be used to store information for arbitrary time, in contrast to events which are inherently volatile.

A *stateful object* is represented by terms of semi-structured data which are associated with a valid time. The valid time describes when the data that is associated with a stateful object is (or was) actually valid. The notion of valid time is often used in temporal databases [19, 24] and enables to query not only the current content of a database but also its history.

Example For a certain scenario it might be necessary to keep track of persons on the platform and the safety condition of the platform. Initially the platform is empty and its safety condition is `safe`. Subsequently, sensors report that there are 11 and, some moments later, 23 persons on the platform. Finally, fire is detected on the platform and thus the safety conditions of the platform are set to `unsafe`.

This scenario is represented by the following set of data terms which are characteristics of the same stateful object at different times. In the following $\texttt{o}^{[t_1,t_2[}$ denotes a data term o which is valid from time t_1 to t_2. The time point uc reads *until changed* and has the meaning that this data term is representing the properties of the respective stateful object until it is updated or terminated.

```
platform{ id{ "a" }, person-count{ 0 }, safety{ "safe" } }[0,5[
platform{ id{ "a" }, person-count{ 11 }, safety{ "safe" } }[5,13[
platform{ id{ "a" }, person-count{ 23 }, safety{ "safe" } }[13,31[
platform{ id{ "a" }, person-count{ 23 }, safety{ "unsafe" } }[31,uc[
```

To deal with changing attributes of stateful objects, queries can contain temporal relations in order to get the characteristics of a stateful object during a certain time interval or at a certain time point. For instance, one can query the attribute `person-count` of the stateful object `platform-a` which has been observed at time 7. Given the preceding representation of the stateful object, the answer to the query is 11.

States The notion of stateful objects can also be used to model *states*. States are a concept of the semantic level which are mainly used to represent situations, such as, the operation mode of a station or the execution state of a workflow. They are an abstraction of the observed events.

States can be used to enable (or disable) certain rules of the program depending on the current state of the system. Furthermore, states can be used to group together semantically related rules and thus provide a mean for the structuring of a program.

Example In the example given in the motivation of this section, the metro station could be either in the state `normal` or `emergency`.[8] Thus, the succeeding set of data terms represents that the state of the station is `normal` from time point 0 to 31 and then the state changes to `emergency`.

```
operation-mode{ "normal" }[0,31[
operation-mode{ "emergency" }[31,uc[
```

5.2. Properties of Stateful Objects

Although stateful objects and events look quite similar at the first sight, they do have quite different properties. While events can be regarded as messages which *inform about changes* of a physical or abstract object's properties, stateful object are rather means to *keep track* of a physical or abstract object's properties.

[8] Depending on the requirements of the scenario, one could easily introduce further states.

Payload Similar to events, stateful objects carry XML-like semi-structured data of a restricted kind, the so-called payload. It needs to comply to the same restrictions as the payload of events does, that is, recursive definitions, definitions using alternatives, and unlimited repetitions of tags are not allowed.

But in contrast to events, the payload of stateful objects can be modified. Therefore, stateful objects are well suited to keep track of the properties of physical or abstract objects which are changing over time.

Valid Time Because the properties of physical and abstract objects change over time, the properties of the respective stateful object needs to change over time as well. Therefore, a stateful object is represented by a set of data terms that are associated with a right open time interval $[b, e[$, the so-called valid time.[9] The valid time indicates when each data term is actually reflecting the properties of the stateful object. Usually one of the terms has a valid time with an unknown ending, i.e., the ending is set to uc.

Modifying Stateful Objects When a stateful object is updated, the set of data terms that is representing the stateful object is updated respectively. Therefore, the ending of the currently valid data term is changed from uc to now and a new data term with the updated values is added to the respective stateful object's set of data terms.

Example In the preceding examples, the state of the metro station is `normal` from time point 0 to 31 and is then updated to `emergency`. However, initially only one data term with the associated valid time $[0, uc[$ is representing the stateful object, since it is unknown until time 31 that the state will be updated to `emergency`.

```
operation-mode{ "normal" }[0,uc[
```

Eventually, at time 31 fire is detected at one of the station's platforms and subsequently the state of the station is updated. Therefore, the ending of the currently valid data term `operation-mode{ "normal" }`$^{[0,uc[}$ is set to 31 and a new data term with the updated values is added to the set of data terms which is representing the stateful object of the station's state.

```
operation-mode{ "normal" }[0,31[
operation-mode{ "emergency" }[31,uc[
```

Detection of Stateful Objects Events can be detected only when they have occurred, i.e., when their ending is earlier or equal to now. In contrast, stateful objects can be "detected" as soon as they have been created. Thus, states can be queried before they have ended, furthermore they can even be queried before it is actually known when they will end.

[9] Note that closed intervals would be sufficient if a discrete time model was used. However, considering a continuous time model makes neither the usage of the language nor the evaluation of queries more complicated.

Similar to events, temporal constraints and conditions on the data of stateful objects can be added to the `where` part of queries in order to restrict the valid time and the content of the matching stateful objects.

5.3. Querying States and Stateful Objects

Event queries and queries of stateful objects can be easily combined into a single query, as opposed to event-condition-action (ECA) rules [27] which strictly separate queries of event and static data.[10] Combining both kinds of queries allows for much more sophisticated and flexible queries (including queries with the semantics of ECA rules).

Event queries are denoted by the keyword `event` whereas queries of stateful objects are denoted by `state`. Both kinds of queries can be combined by the composition operators `and`, `or` and `not` (understood as expressing an absence and discussed below). Additionally, further temporal constraints on events and states can be added to the `where` part of any query.

Example Reconsider the example in the motivation of this section. When a bathroom light breaks, a `maintenance-required` event is only derived if the station is in the state `normal`. Otherwise, no further events are derived and consequently the information is not presented to the operator. This behavior is implemented by the following rule.

```
DETECT
   maintenance-required{
      type{ "broken light" }, area{ var A } }
ON
   and{
      event e: light-broken{ id{ var I }, area{ var A } },
      state s: operation-mode{ "normal" }
   } where { s at end(e) }
END
```

Note that in this example the valid time of the stateful object is constrained in the `where` part of the query. Any matching stateful object is constrained to be valid at the end of the `light-broken` event.[11] If the temporal constraint is dropped, a `maintenance-required` event will be derived, every time the metro station has been or even will be in the state `normal`.

Identifiers of Stateful Objects Identifiers of stateful objects are similar to event identifiers. They are used in the `where` part of a query to establish relationships between events and other stateful objects. Furthermore, they can be included in the payload of derived events and stateful objects in order to provide provenance information.

[10] Static data, as queried by the condition part or Event-Condition-Action rules, is represented in Dura by stateful objects which are always valid and never modified.

[11] Note that this semantic is equivalent to the semantic of the respective ECA rule.

Temporal Relations The temporal relations and interval operations which have been given in Section 3.7 are reused for constraining the valid time of stateful objects. In this context, event identifiers and identifiers of stateful objects can both be regarded as references to time intervals, that is, the occurrence time of events and the valid time of stateful objects, respectively.

Occurrence Time The occurrence time of the detected event is independent of the valid time of the queried stateful objects. The occurrence time of an event describes when the base events which lead to the detection of the event have been observed. In contrast, queries of stateful object are used to test whether certain conditions of physical or abstract objects are satisfied. Therefore the valid time is not considered for the determination of event occurrence times.

5.4. Modifying Stateful Objects

Stateful objects can be created, terminated and (indirectly) updated by means of the system operators `create-object`, `terminate-object` and `update-object`.

The actions `create-object` and `terminate-object` are used to create new or terminate existing stateful objects. Therefore, they require as an argument either the payload and type of the stateful object that is to be created or a reference to the stateful object that should be terminated.

To actually trigger these actions, reactive rules of the form `ON <event query> DO <action> END` are used. One main difference between reactive rules and rules which have been discussed so far is that reactive rules have side effects, as for instance in the following examples, they modify stateful objects. Reactive rules are discussed in more detail in Section 6.

Example In the metro use case only a single station and the tunnels directly connected to this station are regarded [29, p. 64]. Thus, only objects which are located in the physical boundaries of the use case are actually considered. This restricted view also applies for trains which are therefore suddenly appearing at the end of a tunnel, heading to a platform of the station, and then disappear at the other end of the tunnel. What happens to trains that are not inside the boundaries of the use case is not known to the system and not further regarded. Consequently, a stateful object `train` which is representing an actual train, such as “train 23”, is created when it enters the boundaries of the use case, that is, appears in the tunnel, and is terminated again when it leaves the tunnel.

```
ON
   event: train-enter{ id{ var T }, position{ var P } }
DO
   action: create-object{
      object{ train{ id { var T }, position{ var P } } } }
END

ON
   and{
      event e: train-leave{ id{ var T } },
      state s: train{ id{ var T } }
   } where { s at end(e) }
DO
   action: terminate-object{ ref{ state s } }
END
```

The second of these queries can be written in a more compact manner by passing a query for stateful objects rather than its reference to the `terminate-object` action. Thereby all stateful object which are matching the given query are terminated. The same abbreviation can be used for the `update-object` action.

```
ON
   event e: train-leave{ id{ var T } }
DO
   action: terminate-object{ query{ train{ id{ var T } } } }
END
```

Furthermore, the payload of stateful objects can be updated by means of the `update-object` action. It takes a reference to the stateful object that is to be modified and a set of values that should be updated.

Example The location of trains is observed by sensors which are spread across the entire metro system. Whenever a train passes a sensor the id and position of the train is reported to the system in form of events. These event are then used to update the stateful object of the respective train.

```
ON
   and{
      event e: train-passed{
         train-id{ var T }, location{ var P } },
      state s: train{ id{ var T } }
   } where { s at end(e) }
DO
   action: update-object{
      ref{ state s }, values{ position{ var P } } }
END
```

5.5. Querying State Changes

When stateful objects are modified, events that inform about the update are derived by the event engine. These events can be queried in other rules in order to react to the modified conditions.

There are three different event types reporting about the modification of stateful objects: `object-created`, `object-terminated`, and `object-updated`. The former two event types carry the reference, the name, and the values of the stateful object that was created or terminated in their payload. The latter event type carries the reference of the modified stateful object and its values before and after the update.

Example In the last section stateful object was created (and subsequently terminated) for each train that appeared in a part of the metro system that is modeled by the use case. Thus, one can derive that if a stateful object for a `train` is created, the train is approaching a station.[12]

```
DETECT
   train-approaching{ id{ var Id } }
ON
   and{
      event e: object-created{ ref{ var T } },
      state s: train{ id{ var Id } }
   } where { s at end(e), state s = var T }
END
```

Because the `object-created` event carries the data of the created stateful object in addition to the reference to the stateful object in its payload, the query can also be written in the following way. However, depending on further aspects of the query one of the two variants can be more appropriate or even required.

```
DETECT
   train-approaching{ id{ var Id } }
ON
   event: object-created{ object{ train { var T } } }
END
```

Example Events that are caused by the creation of stateful objects can be further used to implement (simple) filters. For instance, fire on a platform will be detected by a large amount of sensors. Therefore a high number of events are derived which are all reporting about the same incident. Nevertheless, the evacuation of the platform should only be initiated once. However, the operation mode of the station is set only once to emergency. Therefore, the event reporting about the modification of the operation mode to emergency is queried and thus the evacuation of the station is triggered only once.

[12] Of course, this can also be derived directly from the `train-enter` enter event that triggers the creation of the stateful object.

```
ON
   object-updated{ new{ operation-mode{ "emergency" } } }
DO
   action: evacuate-station{ }
END
```

Note that actions, such as `evacuate-station`, are addressed in detail in Section 6. For now it suffices to know that on the occurrence of the `object-updated` event, a yet to be defined action called `evacuate-station` is executed.

5.6. States for Modularization

Rules which are relying on the same state and are thus semantically related can be grouped together by the `WHILE <state query> LET <rules> END` statement. Whenever the state query matches the currently valid stateful objects, the embodied rules of the while statement are activated. If the state subsequently changes and thus the state query does not match anymore, the rules are deactivated again.

While statement can be arbitrarily nested which enables the modularization and structuring of rules not only by states but also by sub-states.

Example The rule given in Section 5.3 derives a maintenance request for a broken bathroom light only if the station is in normal operation mode. It can be rewritten using a while statement. To this end the query of the stateful object is moved to the `WHILE` part of the statement.

```
WHILE
   state: operation-mode{ "normal" }
LET
   DETECT
      maintenance-required{
         type{ "broken light" }, area{ var A } }
   ON
     event: light-broken{ id{ var I }, area{ var A } }
   END
END
```

Note that the two versions of the query are indeed equivalent. The temporal conditions on the valid time in the `where` part of the original query are implicitly represented by the while statement.

However, the version of the query with the while statement is preferable to the first version. Further rules can be easily added to the while statement whereas the query of the stateful object does not need to be repeated in the queries that are added. In general, redundant queries which are used in several rules should be avoided, since redundancies in the rules can easily lead to copy-and-paste errors and inconsistencies when single rules are adapted.

5.7. Reasoning on Stateful Objects

Similar to reasoning on events, reasoning on stateful objects is a declarative way of deriving new stateful objects based on a set of currently valid stateful objects. In Dura these rules have the form `DERIVE <stateful object> FROM <state query> END`.

These kind of rules can be considered as database-like views on stateful objects. When the `state query` matches the currently valid stateful objects, the `stateful object` specified in the head of the query is derived. If the stateful objects matched by the rule's body change over time and thus the state query does not match any more, the derived `stateful object` is automatically terminated again.

Valid Time The valid time of the derived stateful object is determined by the valid time of all stateful objects it is based on. More precisely, the derived valid time is an interval ranging from the maximum starting to the minimum ending of all stateful objects matched by the query's body. If the valid times of all to a query contributing stateful objects are overlapping, the derived valid time is equal to the intersection of all those valid times.

Example In the metro use case there are three different kinds of operation modes, namely `normal`, `exceptional`, and `emergency` [29, p. 42]. However, for some scenarios one would like to have a finer granularity of states. For instance, when fire breaks out or a station is flooded, the operation mode of the station is in both cases emergency. Depending on the more specific state, that is fire or flooded, there might be different emergency procedures required. The evacuation of the station might for instance differ for fire and a flooding, whereas trains heading to the station are redirected in both cases.

To realize the described dependencies of states, a deductive rule is used to express that the operation mode `fire` implies the operation mode `emergency`. Therefore, rules or emergency procedures which are applied in any emergency situation can query the operation mode emergency, whereas rules which are specific for a fire can query the operation mode fire.

```
DERIVE
   emergency{ }
FROM
   state: fire{ }
END
```

Recursive Rules For the same reasons as for events, recursive definitions of deductive rules on stateful objects are not permitted in Dura. Therefore, recursive rules such as `DERIVE operation-mode{ "emergency" } FROM operation-mode{ "fire" } END` cannot be specified in a Dura program.

However, the same effect of such a rule is achieved by the following two non-recursive rules in combination with the rule from above.

```
DERIVE
   operation-mode{ "emergency" }
FROM
   state: emergency{ }
END

DERIVE
   operation-mode{ "fire" }
FROM
   state: fire{ }
END
```

5.8. Accumulation of Stateful Objects

During query evaluation it might occur that multiple stateful objects are representing the same physical object. This can be desirable, for instance, if several sensors are estimating the number of persons in an area and one wants to keep track of all measurements by storing each of them in a stateful object. However, having multiple stateful objects can also be undesirable, for instance if the same stateful object is derived multiple times by reasoning on states, just because of different justifications.

Accumulation of stateful objects is intended for merging several stateful objects into a single stateful object. It works similar to event accumulation. Stateful objects are queried, grouped together and finally an aggregation function is applied to select a single representative aggregate.

Thus, by choosing an appropriate aggregation function, accumulation can either be used to derive a more abstract representation of currently valid stateful objects or to derive an unambiguous representation of the currently valid stateful objects.

Example The following rule computes the average number of persons based on the currently valid stateful objects of all areas.

```
DERIVE
   aggregated-area{ person-count{ avg(all var P) } }
FROM
   state s: area{ person-count{ var P } }
END
```

Grouping of stateful objects is only possible for queries which guarantee that the number of data terms remains finite in each group. This can basically be achieved by specifying a finite time window in which grouping is carried out. To this end, the matching stateful objects, more precisely, the matching data terms of stateful objects, are implicitly separated into groups based on their valid time. The groups are determined in such a way that for the duration of each group no matching stateful object is updated, created or terminated.

For instance, the preceding data terms (abbreviated by o_1 and o_2) of two stateful objects are

separated into three groups. The first group contains o_1 and has the duration $[0, 2[$, since at time 2 a stateful object, represented by the data term o_2, is created. The second group contains o_1 and o_2 and last from time 2, which is the ending of the first group, up till time 7, which is the time point when the stateful object represented by o_1 is terminated. The last group has the duration $[7, 11[$ and contains o_2.

```
o1: area{ id{ "2x3" }, person-count{ 7 } }[0,7[
o2: area{ id{ "7a" }, person-count{ 13 } }[2,11[
```

During aggregation, the number of persons of each group's entries are collected and their average is computed. Finally, the resulting stateful objects, which is represented by the following data terms, is derived.[13]

```
aggregated-area{ person-count{ 7 } }[0,2[
aggregated-area{ person-count{ 10 } }[2,7[
aggregated-area{ person-count{ 13 } }[7,11[
```

Note that the three data terms are representing the same stateful object. Moreover, the valid times of the data terms are pairwise distinct. Thus, aggregation of stateful objects merges several stateful object into a single stateful object, nevertheless the number of data terms may increase during aggregation.

Example In the example of Section 5.7 a deductive rule is used to express that `fire` implies `emergency`. Consequently, when fire is detected, two stateful objects representing the operation mode of the station are derived, namely `operation-mode{ "fire" }` and `operation-mode{ "emergency" }`.

Although it is desirable to derive two stateful objects in order to facilitate the development of generic rules, operators can be easily distracted by a large number of operation modes which are valid at the same time. Therefore, only one representative, and thus unambiguous, operation mode should be selected and presented to the emergency manager.

For instance, consider the following set of data term of the stateful object `operation-mode`.

```
operation-mode{ "emergency" }[5,17[
operation-mode{ "exceptional" }[5,15[
operation-mode{ "exceptional" }[7,17[
```

The subsequent rule combines all those terms into a single stateful object of the type `unique-operation-mode` that has a unique value for every point in time. To this end, the operation mode which deserves the most attention, in this example the operation mode `emergency`, is selected by the aggregation function `max`.[14]

[13] Note that this example only demonstrates what the overall result of the query is. Indeed, intermediate results are derived in a stepwise manner while the stateful objects are changing over time.

[14] Note that for the selection of the most relevant operation mode a total ordering of the (finitely many) operation modes is required.

```
DERIVE
   unique-operation-mode{ max( all var M ) }
FROM
   state: operation-mode{ var M }
END
```

Note that, although the operation mode of the station is continuously set to `emergency` from time 5 to 17, the derived stateful object `unique-operation-mode` is represented by three data terms with associated valid time intervals. The creation of three data term instead of one is caused by the implicit separation of the data terms into finite groups.

However, for each point in time at most one data term of the stateful object `unique-operation-mode` is valid. Thus, the derived stateful object `unique-operation-mode` is indeed unambiguous whereas the stateful object `operation-mode` is not.

```
unique-operation-mode{ "emergency" }[5,7[
unique-operation-mode{ "emergency" }[7,15[
unique-operation-mode{ "emergency" }[15,17[
```

If required, further means to join the three valid time intervals into a single time interval can be investigated in the future. However, if only temporal relation that constrain stateful objects to be valid at certain time points, such as `s at end(e)`, are used in a program, having three associated time intervals or one connected time interval does not make any difference. Indeed, so far only queries constraining time points of stateful objects are required in the use cases.

6. Actions

Actions enable reactive behaviour which is highly desirable for modern emergency management. Based on the occurrence of events and the conditions of stateful objects, actions can be triggered to control the equipment of the infrastructure, change properties of stateful objects, and to cause state changes.

Similar to events, there are two types of actions: atomic and complex actions. Atomic actions are simple commands which cannot be further decomposed and are thus directly executed either inside the event engine or externally by the equipment of the infrastructure. In contrast, complex actions are composed of several (atomic or complex) actions which may be further constrained, for instance, on their execution order or execution time.

Actions are triggered by reactive rules. In contrast to declarative rules on events and stateful objects, which do not have any side effects and can be seen as views known from database systems, reactive rules effect “the outside world” or the internal state of the system by modifying stateful objects, that is, they are not declarative.

6.1. Properties of Actions

Payload Actions have a payload which can be set on their initiation. In this way it is possible to pass different parameters to actions which influences their execution. For instance, to turn on a specific light on a platform the id of the light can be included in the payload of the action.

Occurrence Time The occurrence time of an action is the smallest interval which includes the beginning of the action, i.e., when it was actually executed, and its ending, i.e., when the action finally succeeded. In the following, actions will be denoted a^t where a is the action itself and t its occurrence time.

Internal vs. External Actions Actions can be classified into two different categories: internal and external actions. Internal actions are directly executed by the event engine, such as the modification of stateful objects. In contrast, external actions are intended to influence the equipment of the infrastructure and thus they cannot be executed by the event processing system directly. Therefore, each device that should be able to execute an external action needs to be connected to the event processing system by an adaptor and needs to be subscribed to all kinds of actions it can execute. Thereby, the corresponding actions of a device are initiated inside the event engine and subsequently sent to the device, where they are finally executed.

6.2. Dimensions of Complex Actions

For each of the complex event query dimensions there is a corresponding dimension of complex actions. A complex action has several parameters, is itself composed of sub-actions and specifies time constraints on the execution of its sub-actions. Furthermore, the cause of the execution of an action is described by its provenance.

Data Injection Data injection in actions corresponds to data extraction in event queries. Actions have parameters which have an impact on their execution. Instead of extracting data, as in case of event queries, values are passed to actions in order to set the value of the parameters. A parameter of an action can for instance specify the updated value of a stateful object or the intensity of the air flow generated by a ventilator of the infrastructure.

Provenance The provenance of an action explains why an action has been executed. The execution of an action is triggered either directly by an event or by another action. Thus there is always an event which (indirectly) triggers the execution of an action. Therefore, the provenance of an action may not only comprises the actions which have (directly or indirectly) triggered the execution of an action, but also the provenance of the events which caused its execution.

Action Composition An action can be composed of several (basic) actions. This allows to split complicated actions into several simple actions. Furthermore, an action can be easily reused among different rules.

Temporal Relationships Temporal relationships are required to specify temporal dependencies between several actions. For instance, one might specify that action *A* is to be executed after action *B*. Furthermore, time windows are used to declare how long the execution of a sequence of actions should require at most. If the given time is exceeded, the execution of the composed action is considered as failed.

Accumulation In contrast to events where accumulation groups together several events, accumulation of actions is required to specify the execution of an arbitrary number of actions. In contrast to the composition of actions, the number of actions may not be known in advance and depends on the parameters of an action. An action which requires accumulation is, for instance, "turn on the lights in all areas of the evacuation route *X*".

6.3. Atomic Actions

Atomic actions are similar to base events of event queries. They are (simple) commands which cannot be further decomposed and can be directly performed either externally by the equipment of the infrastructure, such as "turn on the light with the id 2x3", or internally by the event engine, such as "create a new stateful object".

Example In case of a fire alarm on a metro station's platform, people should be able to quickly leave the platform. To this end, there are emergency lights pointing to the emergency exits of each station. Each of the lights is connected to the SCADA system of the metro station and thus can be remotely turned on and off.

The SCADA system is in turn connected to the event processing system by means of an appropriate adaptor. It is further subscribed to `turn-on-light` actions and therefore, if a `turn-on-light` action is initiated in the event processing system, the action is sent to the SCADA where it is actually executed.

For instance, the initiation of the following action will cause the SCADA system to turn on the emergency light with the id `0x23e`.

```
action: turn-on-light{ id{ "0x23e" } }
```

6.4. Reactive Rules (Event-Condition-Action Rules)

Reactive rules integrate declarative queries for events and stateful objects and the execution of imperative actions. In Dura, reactive rules are of the form `ON <query> DO <action> END`.

Whenever the `query` matches the specified events and stateful objects, the execution of the `action` is triggered. Thus, reactive rules are the connection between declarative rules and non-declarative actions.

Event-Condition-Action (ECA) rules have been extensively studied in the field of active databases [8, 36, 26]. However, the available systems are not convenient for out purposes. They are lacking capabilities of interacting with the physical world, have a limited notion of states, and do not support the timing of actions [5, pp. 26-29].

In contrast, reactive rules in Dura offer a homogeneous and time aware approach for querying complex events, static and dynamic objects and the execution of internal and external complex actions. Therefore, they are a generalization of ECA rules which are only considering basic events, static relations and the execution of internal actions without any notion of time.

Example Emergency lights are represented in this example by stateful objects with an id, the area and further dynamic information, such as the current state of the light (on or off).

Whenever a confirmed alarm is detected in an area, all emergency lights of this area are switched on. To this end, a query of stateful objects is used to obtain all emergency light ids of the particular area. Then a reactive rule is used to trigger a `turn-on-light` action for each light which is subsequently sent to the SCADA system where it is finally executed.

Depending on further aspects, such as the simulated smoke propagation on escape paths, one could even turn on certain emergency lights only. However, for the sake of simplicity, this is not considered in the example.

```
ON
   and{
      event e: confirmed-alarm{ area{ var A } },
      state s: emergency-light{ area{ var A }, id{ var Id } }
   } where { s at end(e) }
DO
   action a: turn-on-light{ id{ var Id } }
END
```

6.5. Events Entailed by Actions

When an (internal or external) action is executed, it cannot be known in advance how long it will take to execute the action or whether it will be executed successfully or not. Therefore, whenever the execution of an action is triggered by a reactive rule, several events which provide further information on the progress of the action are generated.

There are three event types which are caused by the execution of an action, namely `action-initiated`, `action-succeeded`, and `action-failed`. Events of the former type occur whenever the execution of an action begins, whereas only one event of the latter two types occurs, depending on whether the execution of the action was successful or not.

These events take a special role during the execution of actions, because actions cannot be queried in Dura. Actions are just initiated by reactive rules which are in turn triggered by events. However, because it is necessary to observe whether an initiated action has been executed successfully or not, the progress of an action is indicated by the occurrence (or absence) of the respective events that are described above. These events can be in turn queried in the body of (reactive) rules and initiate the execution of actions, such as the initiation of a compensating action on the occurrence of an `action-failed` event.

All three event types carry additional information about the name, action reference and payload of the executed action. They can be queried by further rules, for instance to execute additional actions if the execution of a certain action fails.

Example Reconsider the last example, when a `confirmed-alarm{ area{ 42 } }`$^{[3,11]}$ occurs and at the same time there is a stateful object `emergency-light{ area{ 42 }, id{ "0x23e" }, state{ "off" } }`$^{[0,uc[}$ the conditions of the query are fulfilled and the `turn-on-light` action is executed with `0x23e` substituted for `var Id`. When the execution of the action begins, the following event is derived by the event engine.

```
action-initiated{
   ref{ 352f }, action{ turn-on-light{ id{ "0x23e" } } } }[11,11]
```

When the light is finally reported to be turned on by the SCADA system which actually carried out the action, an `action-succeeded` event follows. However, if the light could not be turned on for some reasons, a similar `action-failed` event is generated instead.

```
action-succeeded{
   ref{ 352f }, action{ turn-on-light{ id{ "0x23e" } } } }[12,12]
```

Events Entailed by External Action For the generation of `action-succeeded` and `action-failed` events, the device, or more precisely the adaptor connected to the device, needs to propagate the information whether the action has been executed successfully or not back to the event processing engine.

However, this information is is not always available because some devices are not capable of observing the progress of an executed action. Therefore, the execution status of the action can only be determined by sensors that verify the execution indirectly by measuring parameters which are affected by the execution of the action.

This problem can be addressed by explicitly specifying when an action is regarded as being executed successfully. This is done by including an event query in an action specification: `DO <action> SUCCEEDES ON <query> END`. The `query` needs to be timely bounded but does not have any further restrictions. If it matches the events on the event stream an appropriate `action-succeeded` event is derived. Alternatively, an `action-failed` event is derived if the

`query` does not match the events on the stream during the given time window.[15]

Example Basically all safety relevant actuators of the infrastructure are equipped with sensors that monitor their state. For instance, fire dampers are connected to a SCADA system and can be opened and closed by sending action requests to the SCADA. Contact sensor monitor the actual position of the fire damper. When an action is triggered, these sensors give information on whether the action has been executed successfully. Thus, events coming from the contact sensors can be used to derive `action-succeeded` and (if they are absent within a given time frame) `action-failed` events.

```
ON
   ...
DO
   action a: open-damper{ id{ var Id } }
SUCCEEDES ON
   event e: damper-position{ id{ var Id }, position{ "open" } }
   where { {begin(a), e} within 30 sec }
END
```

Events Entailed by State Changes Events that are caused by changes of stateful objects (cf. Section 5.5) are basically just `action-succeeded` events. The modification of a stateful object is triggered by a reactive rule, such as `ON ... DO action: update-object{ ... } END`. When the update action succeeds a `action-succeeded` event occurs and because the stateful object has been modified an `object-modified` event is derived as well. However, both events are basically giving the same information, they just differ in their representation.

6.6. Action Composition

In analogy to events, actions can be composed of multiple actions which are connected by either a conjunction (`and`) or a disjunction (`or`). Both operators can have two or more actions as arguments and can be arbitrarily nested.

Note that conjunctions and disjunctions only effect how the composition of actions is regarded as successful, they do not affect which actions are executed. If there are no further constraints, all specified actions are always executed in parallel for both kinds of operators. However, a conjunction is only regarded successful if all specified actions are executed successfully, whereas disjunctions are regarded successful if at least one specified action is executed successfully.

Note that it does not make sense to provide a negation for the composition of actions: Actions not to perform are simply not mentioned.

[15] Note that because of the implicit negation of the `query` it needs to be timely bounded.

Conjunction Conjunctions are used to specify that several actions should be executed. If no further constraints are present, the specified actions are executed in parallel. When all actions specified in a conjunction are executed successfully, the execution of the conjunction is regarded successfully. However, if a single action fails, the execution of the conjunction fails as well.

Disjunction Disjunctions are quite similar to conjunctions. All specified actions of a disjunction are executed in parallel. In contrast to conjunctions, it suffices that at least one of the specified actions succeeds to render the execution of the complete disjunction successful.

Example If there are first signs that there might be fire in an area of a metro station, that is, an `uncertain-fire-alarm` occurs, it is desirable to make certain arrangements in order to be well prepared if the alarm is confirmed. These preparations can include for instance opening the fire dampers of the area (if there are any), activate the lighting and the adaptation of the ventilation regime.

```
ON
   event: uncertain-fire-alarm{ area{ var A } }
DO
   and{
      action a1: open-fire-dampers{ area{ var A } },
      action a2: turn-on-lights{ area{ var A } },
      action a3: fresh-air-supply{
                     area{ var A }, intensity{ "max" } }
   }
END
```

6.7. Temporal Relations

So far, only actions which are executed in parallel have been regarded. However, for a reactive emergency management it is highly desirable to influence the execution order of actions and to impose time constraints between several actions.

The same temporal relations introduced in Section 3.7 can be used to specify such relations between actions. Similar to event queries and queries of stateful object, they can be included in the `where` part of a composition operator.

For instance, the relation `a1 before a2` ensures that the action `a2` is only executed after the action `a1` has terminated successfully. Relations such as `within` can further be used to specify time constraints between actions which have an effect on whether and complex action is regarded as successful or not.

In general, relations which imply an order on the execution of actions, such as `before` and `after`, have an impact when the specified actions are actually executed. In contrast, relations which verify time constraints on the duration of actions, such as `within` and `apart`, do not

influence the execution of actions but rather influence when a composed action is successful. Therefore, expressions that are only verifying time constraints are shorthands for respective event queries in the `SUCCEEDES ON` part of an action specification.

Note that binary relations can be used to specify expressions such as `a before b before c`. They are considered syntactic sugar for expression without the nesting of relations. In the given example, the expression translates to `a before b, b before c`.

Example In the last example, the fire dampers are opened and simultaneously the ventilation regime is adapted. However, for some areas of the metro station it might be necessary to open the fire dampers first and only after they have been opened successfully the ventilation regime is adapted.

```
ON
   event: uncertain-fire-alarm{ area{ var A } }
DO
   and{
      action a1: open-fire-dampers{ area{ var A } },
      action a2: turn-on-lights{ area{ var A } },
      action a3: fresh-air-supply{
                    area{ var A }, intensity{ "max" } }
   } where { a3 after a1 }
END
```

It seems to be desirable to have a mean that allows to directly specify sequences of actions. This could be achieved for instance by introducing composition operators with square brackets. Such an operator would be interpreted as syntactic sugar for the same operator with curly brackets and an appropriate `where` part that realizes the sequential execution of the actions in the same order as they are specified in the rule.

6.8. Complex Action Specification

Complex actions can not only be specified in reactive rules, but can also be named in order to share the same complex actions among several rules. In Dura this is done with the `FOR <name > DO <complex action> END` statement. If the action `name` is initiated by a reactive rule, the `complex action` given in the body of the complex action specification is executed.

Therefore, complex actions can be easily shared among several rules which is highly desirable in the context of emergency management since rules need to be robust and easy to maintain. If there is a high redundancy among the rules, errors during the development and maintenance of rules are likely, such as copy-and-past errors and inconsistencies if only some of the redundant rules are modified.

Example In the last example, preventive actions are executed when an uncertain fire alarm is detected in a certain area. However, the same actions can be executed when a burning train

is approaching a station. But instead of copying the actions to a second rule which is detecting that a train on fire is approaching a station, a complex action specification is used to give the preventive actions a name. The complex action is then used in the rules that are dealing with uncertain fire alarms and trains on fire approaching a station.

```
FOR
   preventive-fire-suppression{ area{ var A } }
DO
   and{
      action a1: open-fire-dampers{ area{ var A } },
      action a2: turn-on-lights{ area{ var A } },
      action a3: fresh-air-supply{
                    area{ var A }, intensity{ "max" } }
   }
END
```

```
ON
   event: uncertain-fire-alarm{ area{ var A } }
DO
   action: preventive-fire-suppression{ area{ var A } }
END
```

```
ON
   and{
      event e: certain-fire-alarm{ area{ var A } },
      state s: train{ id{ var A }, destination{ var S } }
   } where { s at end(e) }
DO
   action: preventive-fire-suppression{ area{ var S } }
END
```

6.9. Conditional Actions

Actions can be associated with additional conditions that need to be fulfilled before an action is executed. So-called conditional actions are especially useful for the specification of alternatives or some kind of "compensating" action that is executed if, for instance, a previously initiated action fails.

Conditional actions have the form `IF <query> THEN <action> ELSE <action> END` whereas the else part of the statement is optional. The `query` needs to be timely bounded in order to make the evaluation of the implicit negation in the else part feasible. Therefore, event identifiers and identifiers of stateful objects which are defined outside the statement, that is in the reactive rule it is contained in, can be used in the query of the conditional action.

Example In the preceding examples, preventive fire suppression always includes the opening of fire dampers and the adaption of the ventilation regime. However, in case of an uncertain

fire alarm it can be sufficient to open the fire dampers and only if the fire dampers cannot be opened within a certain time frame, the ventilation regime is adapted instead.

```
ON
   event e: uncertain-fire-alarm{ area{ var A } }
DO
   and{
      action a: open-fire-dampers{ area{ var A } },

      IF not event f: action-succeeded{ ref{ action a } }
         where { {e,f} within 30 sec }
      THEN
         action: fresh-air-supply{
                     area{ var A }, intensity{ "max" } }
      END
   }
END
```

Conditional actions can furthermore be used to implement some sort of for each statement. The query part of the statement can potentially match several events and stateful objects and thus multiple actions can be carried out. Consequently, an action is initiated for each matching stateful object.

Example Throughout the last examples, the complex action `open-fire-dampers` is used to open all fire dampers of a certain area. This complex action can be realized by means of a conditional action which executes the atomic action `open-fire-damper` for every fire damper in an area.

```
FOR
   action a: open-fire-dampers{ area{ var A } }
DO
   IF state s: fire-damper{ id{ var Id }, area{ var A } }
      where { s at begin(a) }
   THEN
      action: open-fire-damper{ id{ var Id } }
   END
END
```

Note that the execution of the if statement is only successful if all actions that are initiated by the statement are executed successfully.

7. Related Work and Conclusion

7.1. Related Work

A major achievement of Dura is the integration of complex event queries, stateful objects and reactive rules in a homogeneous fashion.

Prior and well established approaches like production rules and ECA rules, which have been extensively studied in active databases, have significantly influenced the design of Dura. However, the specific requirements of emergency management require an adoption and generalization of existing approaches.

The separation of event queries and queries of static data is too restrictive for our purposes. There are extensions which integrate both kinds of queries, such as EA rules [15] and $(EC)^*A$ rules [4]. However, conditions in ECA rules only refer to the recent content of the database. Therefore, they do not provide any notion of states, because states are constantly changing over time and thus require a special treatment of temporal aspects.

Furthermore, time and timing of actions is not considered by current approaches. Due to their origin in database most of them consider only internal actions, such as updates of the database, and external function calls.

Process calculi [3, 25] are used to extend action specifications of ECA rules towards complex action definitions capable of specifying sequences of actions and influencing the execution order of actions. However these approached often rely on some kind of backtracking mechanism or notion of transaction which is inherently not available in real-life applications. Moreover, feedback of the success of actions might only be available in an indirect manner and thus needs special treatment.

A more comprehensive survey on state-of-the-art of complex event processing and event condition action rules can be found in [6, 5].

7.2. Conclusion

This document introduces a language sketch of Dura based on the information collected from the use case designers and the descriptions of the use cases. The examples and concepts introduced in this text form the basis for implementing reactive rules for the use cases.

So far, SOMAL has not been considered in designing Dura. Indeed, it seems to be more related to implementing use cases than to designing Dura. Queries of a (possibly dynamic) ontology can be realized in Dura by expressing the ontology as stateful objects.

The design of Dura has been carefully adjusted to the requirements of the use cases [30, 34, 29, 11] and the findings of previous research activities [6, 5, 10]. It should therefore be well suited for emergency management in large infrastructures. Nonetheless there might be yet still unknown requirements from the use cases that are not sufficiently supported by the language and thus adaptations of the language might be necessary in the future.

The language design integrates complex events, states and stateful objects, and complex actions in a homogeneous and time aware fashion which distinguishes Dura from existing approaches. However, all three concepts are highly desirable for a reactive management of emergencies in large infrastructures [6].

A core issue is however an efficient implementation of the Dura language making a fast complex event processing possible. Indeed, the vision of a novel form of reactive emergency

management requires a very fast detection of possibly very complicated complex events.

Currently, an efficient run time system for Dura is developed which is based on the database system MonetDB. Our efforts in both implementing Dura using this run time system and in conceiving and implementing this run time system will be presented in the next deliverable.

A. Dura EBNF Grammar

The constraints mentioned below in the comments are intended as part of the specification of the semantic analysis.

```
program ::= preamble ( eventDefinition | stateDefinition
                     | actionDefinition | reactiveRule)+

preamble ::= (typeDefinition | constDefinition)*

/**
 * constraints:
 *   all schemaDefinition labels are pairwise distinct
 *   type name ID is unique
 *   no cyclic definitions
 */
typeDefinition ::= 'TYPE' ID 'IS' basicType 'END'
                 | 'TYPE' ID 'IS' schemaDefinition typeSupplement? 'END'
                 | 'TYPE' ID 'IS' '{' schemaDefinition (',' schemaDefinition)* '}'
                      typeSupplement? 'END'

/** constraint: const name ID is unique */
constDefinition ::= 'CONST' ID 'IS' constTerm 'END'

/**
 * constraints:
 *   event definition schemaDefinition needs to be unique
 *   schema of derived events needs to comply with schmaDenfinition
 */
eventDefinition ::= 'EVENT' schemaDefinition typeSupplement?
                       ('WITH' (eventSpecification | whileStatement)*)? 'END'

/** constraints:
 *    stateful object definition schemaDefinition is unique
 *    schema of derived stateful objects needs to comply with schmaDenfinition
 */
stateDefinition ::= 'STATEFUL OBJECT' schemaDefinition
                       ('WITH' stateSpecification*)? 'END'

/** constraints:
 *    action definition schemaDefinition is unique
 *    schema of complex actions needs to comply with schmaDenfinition
 */
actionDefinition ::= 'ACTION' schemaDefinition ('WITH' actionSpecification*)? 'END'

eventSpecification ::= 'DETECT' constructTerm ('group by'
```

```
                          '{' binding (',' binding)* '}')? 'ON' eventQuery 'END'

stateSpecification ::= 'DERIVE' constructTerm 'FROM' stateQuery 'END'

actionSpecification ::= 'FOR' term 'DO' action ('SUCCEEDES ON' eventQuery)? 'END'
                      | 'FOR' 'action' ID? ':' term 'DO' action
                           ('SUCCEEDES ON' eventQuery)? 'END'

reactiveRule ::= 'ON' eventQuery 'DO' action ('SUCCEEDES ON' eventQuery)? 'END'

whileStatement ::= 'WHILE' stateQuery 'LET'
                      (eventSpecification | whileStatement)* 'END'

/** constaint: at least one positive event query in every disjunct of the dnf */
eventQuery ::= 'and' '{' eventQuery (',' eventQuery)* '}' querySupplement?
             | 'or' '{' eventQuery (',' eventQuery)* '}' querySupplement?
             | 'exists' '{' eventQuery '}' querySupplement?
             | 'exists' eventQuery
             | 'not' '{' eventQuery '}' querySupplement?
             | 'not' eventQuery
             | 'event' ID? ':' eventQuery
             | ifStatement
             | atomicEventQuery

atomicEventQuery ::= 'event' ID? ':' term querySupplement?
                   | 'state' ID? ':' term querySupplement?

stateQuery ::= 'and' '{' flatStateQuery (',' flatStateQuery)* '}' querySupplement?
             | 'or' '{' flatStateQuery (',' flatStateQuery)* '}' querySupplement?
             | flatStateQuery (',' flatStateQuery)*

flatStateQuery ::= 'not' atomicStateQuery
                 | atomicStateQuery

atomicStateQuery ::= 'state' ID? ':' term querySupplement?

action ::= 'concurrent' '{' action (',' action)* '}' actionSupplement?
         | 'and' '{' action (',' action)* '}' actionSupplement?
         | 'or' '{' action (',' action)* '}' actionSupplement?
         | 'action' ID? ':' action
         | ifStatement
         | atomicAction

ifStatement ::= 'IF' eventQuery 'THEN' action ('ELSE' action)? 'END' actionSupplement?

/** constraint: action term is actually defined */
```

```
atomicAction ::= 'action' ID? ':' term actionSupplement?

querySupplement ::= (where | let | grouping)+

actionSupplement ::= where+

/** constraint: no variables occur in mathFormula */
typeSupplement ::= 'where' '{' mathFormula (',' mathFormula)* '}'

let ::= 'let' '{' unification (',' unification)* '}'

unification ::= dataVariable '=' expr

grouping ::= 'group by' '{' binding (',' binding)* '}'
           | 'group by' '{' binding (',' binding)* '}'
                'aggregate' '{' aggregation (',' aggregation)* '}'

binding ::= dataVariable
          | identifier
          | untypedIdentifier

aggregation ::= dataVariable '=' aggregationOp '(' dataVariable ')'

where ::= 'where' 'and' '{' conditions (',' conditions)* '}'
        | 'where' 'or' '{' conditions (',' conditions)* '}'
        | 'where' '{' conditions (',' conditions)* '}'

conditions ::= 'and' '{' conditions (',' conditions)* '}'
             | 'or' '{' conditions (',' conditions)* '}'
             | condition

condition ::= intervalFormula
            | mathFormula

/** constraints:
  *   either both or none of the expressions is of type duration
  */
mathFormula ::= expr arithmeticRelation expr

intervalFormula ::= '{' timeInterval ',' timeInterval
                        (                    '}' 'apart-by' duration
                        | (',' timeInterval)* '}' 'within' duration
                        )
                  | timeInterval (intervalRelation timeInterval | 'at' timePoint)

/** constraint: variable is of type timeinterval */
```

```
timeInterval ::= dataVariable
               | identifier
               | untypedIdentifier
               | relativeTimerOp '(' timeInterval ',' duration ')'

/** constraint: variable is of type timepoint */
timePoint ::= dataVariable
            | intervalOp '(' timeInterval ')'

/** constraint: all timeunits are are pairwise distinct */
duration ::= time+

time ::= NUMBER timeUnit

/**
 * constraints:
 *   all term labels are pairwise distinct
 *   no cyclic definitions
 */
term ::= label '{' '}'
       | label '{' termLeaf '}'
       | label '{' term (',' term)* '}'

/** constraints:
 *    variables are unified with basic types
 *    constants are unified with basic types
 *    identifiers are unified with identifier types
 */
termLeaf ::= dataVariable
           | constant
           | identifier
           | duration
           | STRING
           | NUMBER

/**
 * constraints:
 *   all constTerm labels are pairwise distinct
 *   no cyclic definitions
 */
constTerm ::= STRING
            | NUMBER
            | constant
            | duration
            | label '{' '}'
            | label '{' constTerm (',' constTerm)* '}'
```

```
/**
 * constraints:
 *   all schemaDefinition labels are pairwise distinct
 *   no cyclic definitions
 */
schemaDefinition ::= label '{' basicType '}'
                   | label '{' compositeType '}'
                   | label '{' schemaDefinition (',' schemaDefinition)* '}'

expr ::= mathExpr
       | identifier

mathExpr ::= (multExpr) (('+' | '-') multExpr)*

multExpr ::= (powExpr) (('*' | '/') powExpr)*

powExpr ::= (atom) ('^' atom)*

/**
 * constraints:
 *   constants used in aritmetic expressions need to be a number
 */
atom ::= '(' expr ')'
       | intervalOp '(' timeInterval ')'
       | aggregationOp '(' dataVariable ')'
       | dataVariable
       | constant
       | path
       | (NUMBER) (timeUnit time*)?
       | STRING

constructTerm ::= label '{' '}'
                | label '{' expr '}'
                | label '{' constructTerm (',' constructTerm)* '}'

path ::= label ('.' label)*

label ::= aggregationOp
        | intervalOp
        | ID

basicType ::= ( 'string' | 'int' | 'long' | 'double' | 'float'
              | 'boolean' | 'identifier' | 'timestamp' | 'duration')

/** constraint: composite type ID is actually defined */
```

```
compositeType ::= ID

identifier ::= ('event' | 'state' | 'action') ID

untypedIdentifier ::= ID

dataVariable ::= 'var' ID

/** constraint: constant ID is actually defined */
constant ::= 'const' ID

intervalRelation ::= ( 'before' | 'contains' | 'overlaps' | 'after' | 'during'
                     | 'overlapped-by' | 'starts' | 'finishes' | 'meets'
                     | 'started-by' | 'finished-by' | 'met-by' | 'equals'
                     | 'while' | 'valid-at' | 'valid-during')

relativeTimerOp ::= ('extend' | 'shorten' | 'extend-begin' | 'shorten-begin'
                    | 'shift-forward' | 'shift-backward' | 'from-end'
                    | 'from-end-backward' | 'from-start' | 'from-start-backward'
                    | 'from-begin' | 'from-begin-backward')

intervalOp ::= ('begin' | 'end')

aggregationOp ::= ('max' | 'min' | 'mean' | 'avg' | 'count')

timeUnit ::= ('day' | 'days' | 'hour' | 'hours' | 'min' | 'sec' | 'ms')

arithmeticRelation ::= ('<' | '<=' | '=' | '!=' | '>' | '>=')

ID ::= ('a'..'z' | 'A'..'Z') ('a'..'z' | 'A'..'Z' | '0'..'9' | '-' | '_')*

NUMBER ::= ('0'..'9')+ '.' ('0'..'9')* Exponent?
         | ('0'..'9')+ Exponent?

Exponent ::= ('e' | 'E') ('+' | '-')? ('0'..'9')+

STRING ::= '"' (EscapeSequence | ~('\\' | '"'))* '"'

EscapeSequence ::= '\\' ('b' | 't' | 'n' | 'f' | 'r' | '\"' | '\'' | '\\')

COMMENT ::= '//' ~('\n' | '\r')* '\r'? '\n'
          | '/*' (.)* '*/'

WS ::= (' ' | '\t' | '\r' | '\n')
```

References

[1] Data cleaning material collection. `http://paul.rutgers.edu/~weiz/readinglist.html`.

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

[3] 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.

[4] H. Behrends. An operational semantics for the activity description language adl. Technical report, Universität Oldenburg, 1994.

[5] S. Brodt, S. Hausmann, and F. Bry. Deliverable D4.2: Reactive rules for emergency management, 2010.

[6] S. Brodt, S. Hausmann, F. Bry, O. Poppe, and M. Eckert. Deliverable D4.1: A survey on IT-techniques for a dynamic emergency management in large infrastructures, 2010.

[7] F. Bry and M. Eckert. Rule-based composite event queries: the language XChange^{EQ} and its semantics. In *Proceedings of the 1st international conference on Web reasoning and rule systems*, RR'07, pages 16–30, Berlin, Heidelberg, 2007. Springer-Verlag.

[8] S. Ceri, R. Cochrane, and J. Widom. Practical applications of triggers and constraints: Success and lingering issues. 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.

[9] Disaster report: Daegu subway. `http://eng.nema.go.kr/sub/cms3/3_4.asp`.

[10] M. Eckert. *Complex Event Processing with XChangeEQ: Language Design, Formal Semantics and Incremental Evaluation for Querying Events*. PhD thesis, Institute for Informatics, University of Munich, 2008.

[11] J. L. M. Español. Deliverable D3.1 annexe c: Specific report for use case III, power networks, 2010.

[12] Fire investigation summary Düsseldorf. `http://www.nfpa.org/assets/files/pdf/dusseldorf.pdf`.

[13] H. Galhardas. *Data Cleaning: Model, Language and Algoritmes*. PhD thesis, University of Versailles, 2001.

[14] H. Galhardas, D. Florescu, and D. Shasha. Declarative data cleaning: Language, model, and algorithms. In *In VLDB*, pages 371–380, 2001.

[15] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Event specification in an active object-oriented database. In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, SIGMOD '92, pages 81–90, New York, NY, USA, 1992. ACM.

[16] F. Groffen. Deliverable D5.2: Library for sensor interaction, 2010.

[17] G. Haddow, J. Bullock, and D. P. Coppola. *Introduction to Emergency Management*. Butterworth-Heinemann, 2008.

[18] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kauffman, 2001.

[19] C. S. Jensen and R. T. Snodgrass. Temporal data management. *IEEE Trans. on Knowl. and Data Eng.*, 11:36–44, January 1999.

[20] E. M. Knorr. *Outliers and Data Mining: Finding Exceptions in Data*. PhD thesis, University of British Columbia, 2002.

[21] D. Luckham and R. Schulte. Event processing glossary. `http://www.complexevents.com/?p=361`, July 2008.

[22] A. E. Monge. *Adaptive detection of approximately duplicate database records and the database integration approach to information discovery*. PhD thesis, University of California, San Diego, 1997.

[23] H. Müller and J.-C. Freytag. Problems, methods, and challenges in comprehensive data cleansing. *HUB-IB-164, Humboldt University Berlin*, 2003.

[24] G. Ozsoyoglu and R. T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Trans. on Knowl. and Data Eng.*, 7:513–532, August 1995.

[25] 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.

[26] 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.

[27] N. W. Paton and O. Díaz. Active database systems. *ACM Comput. Surv.*, 31:63–103, March 1999.

[28] Rahm and Do. Data cleaning: Problems and current approaches. *IEEE Bulletin 23(4)*, 2000.

[29] N. Seifert and M. Bettelini. Deliverable D3.1 annexe b: Specific report for use case II, public transport, 2010.

[30] N. Seifert and M. Bettelini. Deliverable D3.1: Use cases requirements analysis and specification (main report), 2010.

[31] N. Seifert, M. Bettelini, and S. Rigert. Deliverable D3.2 annexe d: Simulation methodology, 2011.

[32] N. Seifert, M. Bettelini, and S. Rigert. Deliverable D3.2: Concrete use case models, 2011.

[33] A. Usov. Deliverable D6.2: User interface requirements for site, 2010.

[34] S. Vraneš, V. Mijović, N. Tomašević, G. Konečni, V. Janev, and L. Kraus. Deliverable

D3.1 annexe a: Specific report for use case I, airport, 2010.

[35] S. Vraneš, M. Stanojević, V. Janev, N. Tomašević, and V. Mijović. Deliverable D6.3 design of the first prototype of the integrated emili-site system, 2011.

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