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

Project acronym: *EMILI*

Project full title: Emergency Management in Large Infrastructures

Grant agreement no.: 242438

D4.6 Modularization Mechanisms

Due date of deliverable: 31/12/2011
Actual submission date:
Revision: Version 1

Ludwig-Maximilians University Munich (LMU)

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

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

Index

1. Modular Programming

The notion of modular programming is a very common and well established software design technique which is included in many programming languages.

According to [?] “modularity in design refers to the splitting of a large software system into smaller connected modules. Modules are interconnected through their interfaces. The interconnection should be simple to avoid side effects and costly maintenance.”

The Linux documentation project defines modular programming as “a programming style that breaks down program functions into modules, each of which accomplishes one function and contains all the source code and variables needed to accomplish that function. Modular programming is a solution to the problem of very large programs that are difficult to debug and maintain. [?]”

Accordingly, the main benefits of modularization are that different developers can independently work on different modules, changes within a module have only a limited effect on other modules, and that the robustness and fault tolerance of programs is improved.

This deliverable discusses the advantages of modular programming for emergency management software in general and introduces a modular mechanism for the reactive event processing language Dura [?] which is developed in the context of the EMIL project.

1.1. Need for Modularization

Critical Infrastructures (CI) as they are regarded within the scope of the EMILI project are large and complex physical systems with a myriad of different components such as sensors and actuators. The components of CIs are often affiliated with each other to achieve certain tasks, such as the detection of fire which incorporates several sensors of different types [?].

The airport scenario, for instance, considers currently about 430 different actuators and 460 different sensors [?]. The Dura program of the airport use case currently focuses on certain aspects of emergency management in a public airport, but yet it already consists of 50 rules which will most likely be extended to around 80 rules [?]. The components are divided into 6 different systems which operate independently. However, in case of an emergency, the data from all sensors needs to be correlated in order to gain a more abstract representation of the infrastructures condition and of the current incidents. In addition, the different systems of the infrastructure, that are operating independently under normal conditions, need to closely cooperate in order to prevent harm of passengers, staff and the infrastructure itself during exceptional situations and emergencies.

The number of different component types and their complexity has direct impact on the size of the Dura program that is intended to support operators during exceptional and emergency situations. The more component types are regarded the more event types need to be considered and correlated by means of complex event queries. Likewise, the more complex the workflows within an infrastructure are, the higher the number of rules that are required to model the workflows in the event processing system becomes.

However, when a large program is not carefully designed, unintended flaws and inferences between different parts of the source program are likely. At the same time, as the size of a program increases it becomes harder to find the origin of errors and the more difficult it gets to eliminate errors without introducing further ones. Moreover, due to the missing structure of the code and potentially many dependencies between functions, large monolithic programs are harder to understand, debug, and maintain.

Modules facilitate the development of a system composed of several smaller segments instead of one big monolithic program. The concept is therefore perfectly suited to be adapted to the needs of emergency management and Dura.

Breaking up the program into logical segments that are separated by a clear interface mechanism entails several advantages. The code is split into small and semantically meaningful pieces which do not have (or have only litte) dependencies to other parts of the program. Therefore, each module can be more easily maintained and tested independently which facilitates the development of the program. It even makes it possible to develop multiple modules independently and by different persons. For instance, rules for the low level communication and control of external actuators can be developed by SCADA experts whereas high level rules which describe the behavior of complex workflows during emergencies can be developed by security experts.

Besides, modules with a clear interface mechanism can be adapted more easily and facilitate the reuse of code. As interfaces only specify what kind of events, stateful objects and actions are available and not how they are actually derived, the concrete implementation of rules can be easily exchange as long as the interfaces remain identical. At the same time, changes within a module have only a limited effect on other modules.

Modules also enable the encapsulation of auxiliary rules that provide intermediate results and should hence not interfere with other parts of the implementation. Thus, certain information can be hidden from the rest of the program if desired and unintended interference caused by other parts of the program are prevented.

1.2. Modularization in Rule Based Systems

In the area of classical logic programming there has been a considerable amount of research towards modular extensions for logic programming languages.

In the literature, several extensions for rule based languages, such as Prolog, have been proposed. In [**?**] information hiding and abstraction for Prolog is realized by means of structures and signatures which are used to control the visibility of terms and predicates. The approach is in the spirit of the Standard ML module system.

Gödel [**?**] is a rule based language which comes with a simple build in module system. In Gödel each module specifies explicitly which predicates are exported to, that is, are visible from, other modules. Furthermore, modules can import other modules by which means all visible predicates from the imported module become locally available. In this way, modules can be used to share functionality whereas auxiliary predicates can be hidden from other modules. Moreover, interferences between equally named predicates from different modules are avoided.

In [**?**] a very sophisticated and expressive modular framework for logic programming languages. The authors propose a set of operators on modules, namely, $\cup$, $\cap$ and $\star$ which can be use to assemble separate programs and their intensional and extensional knowledge into bigger ones. Based on these operators, which define an algebra on modules, the authors propose ways to cover *ordinary modules*, that is, support basic information hiding and import/export relationships, and exploit further and more fine grained extensions for modularization of logic programs called *logic modules*.

A more pragmatic approach for modular extensions of a logic based web query language which is based on the notion of stores is presented in [**?**]. The ideas behind stores is that they provide designated data areas of modules, namely a *private*, *in* and *out* part, and hence provide control over which data is visible to the according module and from other modules. Data that should be processed by a store is injected into its *in* part whereas derived data that can be imported by other modules is stored in its *out* part. Data that should be hidden to other modules resides in the *private* part of the according store.

It is worth to mention that work from the area of classical rule bases languages, in particular [**?**] and [**?**], focusses on programs that contain arbitrary (complex) computations which furthermore need to be aware of potentially conflicting intensional and extensional knowledge from different modules that are contributed from independent sources. As a consequence, these approaches provide or even rely on means that explicitly specify which kind of data is injected from one module to another. These means are crucial in the given area as it provides control to keep the data from certain modules separate in order to avoid potential inconsistencies. However, for emergency management applications these approaches seems unnecessary complex.

A comprehensive overview of modularity in logic programming can be found in [**?**].

Although there is a myriad of complex event processing engines and languages available, only few of them come with built in modularization mechanisms. One of them is Drools[1], a production rule based language which is tightly integrated with Java and thus bases its modularization on the packaging system from Java which means that modules in Drools are basically namespaces which serve as means to distinguish equally named events from different modules.

1.3. Good Practice in Object Oriented Languages

Not only emergency management application but also large projects in general often benefit from modularization mechanisms. This is reflected by the fact that modularization is a well established mean in the software engineering community.

Java and `C#` are examples for languages which provide a mature and well proven modularization mechanisms. In the following we will briefly introduce basic ideas and concepts of the module mechanism of Java which relies on the notion of packages.

[1] `http://http://www.jboss.org/drools`

In the literature, modules and packages are sometimes discriminated depending on whether further program elements can be added to a module after deployment [?]. Although this difference is important for verification and validation purposes, we will not further discriminate between both terms.

In Java, programs are organized in packages whereas packages are means to organize different classes into namespaces. Packages are defined by means of a hierarchical naming pattern that separates hierarchy levels by periods. In addition to the naming patter, the hierarchy of packages is also reflected in the file system where the actual files containing the rules are stored. Each package is stored in its own directory and subpackages are mapped to subdirectories. However, for small programs it may be desirable to omit the package structure. This is also feasible in Java and results in an unnamed package that includes all members of the given program.

Although the package hierarchy helps programmers to organize their code, there is (almost) no semantic relationship between the structure and the content of packages. However, packages can export their functionality by means of qualifier for the visibility of classes and functions. The visibility determines whether the members of a package can be accessed or used by other packages or whether they are hidden and thus not accessibly from outside the package. The restriction of access to some (axillary) parts of a program is commonly called encapsulation and is required to achieve a good separation of concern within a program.

There are two different ways how the provided functionality of other packages can be used within a package. First of all, one can reference the required member by its full qualified name that includes the name of its package. The second way is using the import declaration by which means the imported member can be referenced just by its name (omitting the package name).

2. Modularization Mechanisms for Dura

Dura is a rule based complex event processing language with integrated support of reactive rules and complex actions. The main purpose of Dura is the detection of abstract situations by means of complex event queries and the (semi-)automatic execution of appropriate complex actions. However, Dura is not designed for performing arbitrary computations as they are supported by other rule languages, like for instance Prolog. This is also reflected by the fact that, for the sake of efficiency and the lacking need of more expressive constructs from the use cases, only a less generic form of terms, namely structures, are supported in Dura.

Based on this background we decided to adapt a pragmatic modularization approach for Dura which resembles the work in [?] and other popular programming languages like Java. In this way we obtain an easy to use language extension with clear semantics that users are already familiar with. Although our approach seems to be straight forward, we obtain valuable insights in modularization mechanisms which origin from the presence of reactive rules and stateful objects in Dura.

There are arguably more expressive approaches, as for instance the work of Brigi et. al. [?] which could be adapted for Dura. However, the additional expressiveness of these approaches is motivated by needs that are not present in the case of emergency management. As already mentioned, the goal of Dura is not to compute arbitrary functions but rather to detect situations and react to them. Moreover, the higher expressiveness that is provided by these approaches leads to a higher complexity of the semantics of the according language which does not seems to be easily accessible for emergency management experts that have no expertise in the field of mathematical logic and model theoretical semantics of programming languages in general.

In section 2.4.8 we discuss further extensions of our approach that incorporate ideas from [?] and enable a looser coupling of modules.

2.1. Overview

Generally speaking, modules in Dura consist of a set of (declarative and reactive) rules. Each module defines its own namespace and thus name clashes between definitions from different sources are avoided. Modules can import the functionality from other modules and provide means for information hiding and abstraction. However, the presence of reactive rules and stateful objects requires special treatment which are further discussed in section 2.4.3 and 2.4.5.

The modularization mechanism of Dura relies on a schema for rules and stream definitions. The basic concepts for modules are furthermore adopted from the well proven and mature module mechanism of common high level languages, more precisely from the module mechanisms of Java.

Note that some of the concepts that are discussed in this section have already been briefly addressed by our preceding deliverable [?] in preparation of the adoptions that are prerequisite for a sound modularization mechanism. Therefore, our last deliverable was also intended to inform the project partners as early as possible of future adoptions and extensions of Dura that, back then, had not been covered in previous work on the language, so as to get the feedback necessary for the further development of the Dura language.

However, whereas [?] focusses on the compilation of programs, the focus of the present document lies on the modularization mechanism of Dura. In order to give a complete description of all required concepts it seems reasonable to recapitulate some of the already discussed ideas from another perspective.

2.2. Schemas as Interfaces

The notion of schema in Dura is a mean to separate the specification of properties of events, stateful objects, and actions from the actual definition of rules that describe how these events and stateful objects are actually derived and composite actions are actually defined.

Therefore, the schema can be regarded as some kind of interface mechanism which is desirable for modularization and in particular for encapsulation and information hiding. When certain events are exported by a module, that is, are visible from other modules, only the schema of the exported events becomes visible from outside the module, the respective rules that are deriving the events remain covered.

Moreover, a schema is desirable to model events and actions that are not derived by the event processing system itself but are instead either sent to the system from external sensors or which are provided to the system by external actuators. This information can then be used at compile time used to detect events that are queried but not derived by or sent to the event processing system.

In particular in the context of emergency management these properties are essential because programming errors, that in fact can be detected during compile time by means of a schema mechanism, can have severe consequences for passengers and the facilities in case of an emergency. This facet of Dura's modularization mechanisms is discussed in more detail in [**?**].

2.2.1. Types and Type Definitions

There are two different types in Dura, atomic types, such as **`int`** and **`string`**, and composite types that are composed of several atomic or composite types, such as `room-location` and `area` which are given in listing 1. Types are subsequently used in the schema of event definitions in order to provide information on how to interpret given literals and relations on these literals, as for instance equals and lower than.

Note that the type system of Dura intentionally remains straightforward. It does not consider overloading of used defined operators and similar concepts as the use cases currently do not require such functionality. Therefore user defined composite types, as they are introduced in the following, should only be considered as a mean to give names to commonly used composite types.

Dura comes with the following atomic types: **`boolean`**, **`int`**, **`long`**, **`float`**, **`double`**, **`string`**, **`identifier`**, **`duration`**, and **`timestamp`**. More elaborated composite types can be defined based on these atomic types as it is shown in listing 1. However, each type definition needs to have a unique name and although type definitions may in turn refer to composite types, type definitions must not be cyclic.

Listing 1: Definitions of the composite types `area` and `room-location`

```
TYPE area IS { id{int}, name{string} } END
TYPE room-location IS { room-number{int}, location{area} } END
```

Note that labels, such as `name` and `location`, are used to define a structure on the composite type. Therefore composite types in Dura are similar to data type structures (structs for short) which are known from other programming languages, like `C` or `C#`.

Constants can be defined in a manner that is similar to the definition of composite types. In this way, values that are used in multiple rules or even multiple times in a single rule need to be specified only once.

Listing 2: Definitions of the constant `temperature-limit`

```
CONST temperature-limit IS 42.0 END
```

2.2.2. Stream Schemas

Schemas and composite types are closely related, the main difference between both notions is, that the name of, for instance, an event is part of the event schema whereas the name that is assigned to a composite type definition is not part of the composite type itself.

Listing 3 contains a schema for `temp` events. The name of the event is also referred to as the event type and labels and types that are specified inside the schema are referred to as the attributes of the event. In the given example, the event has the type `temp` and its attributes are `sensor`, `value`, and `area`.

Listing 3: A sample schema for `temp` events

```
temp{
  sensor{int},
  value{float},
  area{ id{int}, name{string} }
}
```

The schema contains all information on the properties of events, stateful objects and actions. In particular it specifies the shape of the payload, that is, which type of values are contained in the payload and how these values are structured.

2.2.3. Constraints for Types and Schemas

Constraints can be added to types and schemas to provide additional information on their attributes. They are particularly useful to specify further information on temporal relationships between different attributes which are required for the evaluation of queries. However, constraints can also be used for query optimization either during compile time or during runtime.

Listing 4: Definition of the build-in type `time-interval`

```
TYPE time-interval IS
  { begin{timestamp}, end{timestamp} } where {begin <= end} END
```

Constraints are specified in an additional **`where`** part to a type definition or schema. Thereby, atomic types that should be constrained are referenced by the path of labels that leads to the corresponding type. In listing 4, the constraints of the type **`time-interval`** specify that the timestamp of the attribute `begin` is always lower or equal to the timestamp of the attribute `end`.

Likewise, constraints can be added to a schema by specifying the constraints for the schema in a subsequent `where` part. Thereby constraints of types that are used in schema definitions are, in a slightly modified form, also added to the constraints of the schema. If, for instance, the type `time-interval` is used in a schema, the constraints of the attributes `begin` and `end` are added to the schema as well. However, in doing so, the path of the constraints needs to be adapted internally in order to match the path of the attribute that specified the `time-interval` type.

2.3. Stream Definitions

Stream definitions, that is, definitions of events, stateful objects, and actions, intent to combine related schema specifications and rules. Event definitions, for instance, consist of the event schema which specifies the properties of the according event and a (potentially empty) set of rules which specify how the according event is actually derived from other events.

All rules deriving the same event type, type of stateful object or specify the same type of complex action need to be grouped together in a common definition. Moreover, the schema of the definition needs to correspond to the names of derived events, stateful objects, or actions of the given rules and there must not be multiple definitions for the same name. Note that the way in which these definitions restrict programmers to organize their rules in a program is quite common in modern high-level languages. In Java, the attributes of an object and functions which interact with these properties need to be specified in the same class definition.

The introduction of definitions and the associated restrictions is convenient as all essential properties of a certain stream are collected in a single part of the program and cannot be scattered arbitrarily among a (potentially huge) program or, even worse, among several files. On the downside, one has to scan the complete program in order to get the overview of all available definitions. However, with the emergence of techniques that enable the creation of sophisticated editors, such an overview can be easily generated on the fly and presented to developers while they are editing a program.

According to section 2.2.3, the schema of definitions can contain additional constraints that need to hold for any event, stateful object or action of the corresponding definition. Hence, specifying additional constraints in the schema introduces further stability to the properties of the defined elements. Recall that any rule which is added to the definition needs to correspond to the schema, in particular the properties of derived events, stateful objects and actions need to fulfill the specified constraints. Therefore, from a more abstract perspective, definitions can also be considered as interfaces which specify the properties of events, stateful objects and actions whereas the concrete specification of rules is hidden inside each definition.

2.3.1. Structure of Definitions

The generic structure of definitions in Dura is given in listing 5. The `WITH` part of each definition can be omitted which renders the corresponding type atomic. In case of events and actions this implies that the definitions specify external events and actions which needs to be treated separately and what is further discussed in section 2.4.7.

Listing 5: The generic structure of definitions

```
EVENT
  event schema
WITH
  declarative rules (event queries)
END

STATEFUL OBJECT
  stateful object schema
WITH
  declarative rules (stateful object queries)
END

ACTION
  action schema
WITH
  (a single) complex action rule
END
```

Note that although event and stateful object definitions may contain several rules in their with part, action definitions can at most contain one complex action rule. This limitation is desirable because otherwise the execution of a single action with multiple rules in the corresponding definition actually triggers multiple actions. Therefore, the execution of a single action may result in multiple `succeeded` events or, even more odd, it may result in one `succeeded` *and* one `failed` event.

Definition Type	Implicit Attributes
event	`id{`**`identifier`**`}`, `reception-time{`**`time-interval`**`}`
stateful object	`id{`**`identifier`**`}`, `valid-time{`**`time-interval`**`}`
action	`id{`**`identifier`**`}`, `initiation-time{`**`time-interval`**`}`

Table 1: Implicit attributes of definitions

The schema of each definition is implicitly extended by default attributes that are specific to the corresponding type of the definition. Table 1 contains the implicit attributes of each definition type. Note that although implicit attributes are part of the schema, they may only be queried in the body of rules but need to be omitted in the head of rules. As a consequence, the values for implicit attributes of events, stateful objects and actions are always determined by the event processing system.

2.3.2. Implicit Event and Action Definitions

Beside the events and actions that are explicitly specified in a program there are further implicit definitions which are caused by the specification of stateful object and action definitions. Recall that the execution of actions entails events which indicate the outcome of the action. As a result, definitions of the entailed events are implicitly added to the program. Likewise, the modification of stateful objects is triggered by special actions and is indicated by means of events whose definitions are also implicitly added to the program.

The implicit definitions do not need to be given by programmers as they are automatically generated during the compilation process. However, one needs to be aware of their form, more specifically, of their schema, in order to used them properly in event queries and reactive rules.[2]

The name of implicitly defined events and actions is composed of the name of the object that causes the implicit definition and an additional identifier, such as `succeeded`, which are separated by an additional dollar sign. Consequently, the dollar sign occurs in names of events and actions that are used in event queries, reactive rules and complex action rules. However, the dollar sign may only occur in the context of implicit definitions and cannot be contained in the name of arbitrary events, actions or even stateful object.

Listing 6: A generic action definition

```
ACTION
  name{ attributes }
WITH
  ...
END
```

Listing 6 contains an abstract action definition. The concrete name and attributes for the action are not of further interest for the following examples. This definition implicitly defines the three events whose event definitions are explicitly given in listing 7. For the sake of simplicity, the implicit attributes are omitted from this and the following definition. However, an exhaustive example on stream definitions containing all attributes of the according definitions is given in appendix A.

Whenever the action *name* is executed, a corresponding *name*`$initiated` event is derived by the event processing system. The values of parameters that have been passed to the action recur in the derived event. The same holds for the corresponding `succeeded` and `failed` event. Note that hence `initiated`, `succeeded` and `failed` events which are caused by the same instance of an action can be identified because the `id` of the common action is contained in the `payload` of the respective events.

Similar to action definitions, definitions of stateful objects implicitly specify further definitions. But whereas action definitions just implicitly specify event definitions, stateful objects defini-

[2]Section 2.4.6 contains examples for which it is indeed desirable to explicitly specify usually implicit definitions.

Listing 7: Events that are implicitely defined by the definition of listing 6

```
EVENT
  name$initiated{ payload{ attributes } }
END

EVENT
  name$succeeded{ payload{ attributes } }
END

EVENT
  name$failed{ payload{ attributes } }
END
```

tions implicitly specify action and event definition because they can be modified by means of actions and updates of stateful objects entail events.

Listing 8: A generic stateful object definition

```
STATEFUL OBJECT
  name{ attributes }
WITH
  ...
END
```

The implicit definitions that are derived from the stateful object definition of listing 8 are explicitly given in listing 9. Again, for the sake of simplicity, the implicit attributes are omitted from this definition.

Note that although *attributes* contains implicit attributes such as the `id` of the stateful object, only attributes that are explicitly specified in the schema can be specified in the *name*`$create` and *name*`$update` action. In addition, all explicit attributes need to be specified completely. In particular this means for `update` actions that attributes which should not be updated need to be specified as well.

Internally, the execution of an `update` action is realized by means of a `terminate` and `create` action. Therefore, if an `update` action is executed which leaves all attributes of the stateful object equal, that is, it actually does not modify the explicit attributes of the abject, the `update` action is nevertheless executed successfully. Moreover, the execution results in an `updated` event even though the attributes that have been explicitly specified in the schema of the stateful object have not been changed. However, the implicit attributes `valid-time` and `id` are indeed modified.

Listing 9: Implicit definitions that are derived from the definition of listing 8

```
ACTION
  name$create{ values{ explicit attributes } }
END

ACTION
  name$terminate{ query{ id{identifier} } }
END

ACTION
  name$update{ query{ id{identifier} }, values{ explicit attributes } }
END

EVENT
  name$created{ payload{ attributes } }
END

EVENT
  name$terminated{ payload{ attributes } }
END

EVENT
  name$updated{ payload{ attributes } }
END
```

2.3.3. Specifying Success and Failure in Atomic Action Definitions

Receiving feedback about the progress of actions, that is, whether an action has been executed successfully or has failed, is crucial for many applications. For instance, some actions may only be executed if a preceding action has been executed successfully, or there may be a fallback option which should be executed if a certain action fails.

In Dura all information is inherently communicated by means of events.[3] Sensors readings and input from the operator is sent to the event processing system in form of events, updates of stateful actions and the initiation of actions is indicated by means of evens, simulation results are fed into the event processing system in form of events, etc. Consequently, actuators need to report the success or failure of an action execution by means of events as well.

Listing 10 shows on a conceptual level how events that indicate the success or failure of an action execution can be connected with an atomic action definition. Hence, whenever the specified event queries match, the corresponding *name*`$succeeded` and *name*`$failed` events are automatically derived by the event processing system. Opposed to the given generic event queries, these events have a special semantic that plays an important role for the definition of composite actions.

[3]If they are also regarded as information, actions are exceptions to this statement.

Listing 10: Specifying success and failure of atomic actions

```
ACTION
  name{ attributes }
  succeeds on{ event query }
  fails on{ event query }
END
```

Dura leaves it to the programmer to ensure that conditions for success and failure of actions do not overlap. Although an automatic recognition of possible overlaps is certainly worth to be investigated, we decided not to formally investigate issues before they have been sufficiently experienced in practice.

A more concrete and elaborated example of how to specify the success and failure of external actions is given in listing 11. The action `assign-warden` is considered successful if the execution of the action is confirmed by the corresponding warden with a matching `confirmation-message` event within 20 seconds after the initiation of the action.[4] Otherwise the actions is considered as failed.

Note that the (unique) id of action instances is contained in the payload of the according *name*`$initiated` and *name*`$succeeded` events. Indeed, the `id` in the `payload` of *name*`$initiated` and *name*`$succeeded` events that are referring to the same action instance is the same. The `id` is subsequently used in the **`succeeds on`** and **`fails on`** part of the action definition in listing 11 in order to prevent the confusion of multiple instances of the same action that are simultaneously executed.

Listing 11: Concrete example for the specification of success and failure

```
ACTION
  assign-warden{ area{string}, actor{string} }
  succeeds on{
    and{
      event e: action$initiated{ payload{ id{var Id} } },
      event f: confirmation-message{ ref{var Id} }
    } where { {e,f} within 20 sec }
  }
  fails on{
    and{
      event e: action$initiated{ payload{ id{var Id} } },
      not { event f: action$succeeded{ payload{ id{var Id} } } }
    } where { {e,f} within 20 sec }
  }
END
```

[4]Mind the difference between the initiation of actions inside the event processing system and their actual execution by the actuator.

Note that in this example the name of the action is referenced by the keyword `action` instead of its actual name `assign-warden`. Although it does not make any difference for atomic action definitions, for the success and failure of composite actions it is crucial to properly reference actions with the according keyword.

Although it is crucial to know whether an (external) action has been executed successfully by an actuator, in practice there are actuators that simply cannot provide the desired feedback. When a report on the outcome of an action is not directly provided by an actuator and yet it is indispensable for a certain task one need to measure the status of the action indirectly by means of sensor readings that are related to the goal of the action. For instance, if a ventilator does not give any feedback on its current operation status and there is an anemometer close by which measures the current wind speed one can use this information to draw conclusions whether the ventilator is actually operating or not. However, as arbitrary queries can be specified in the **`succeeds on`** and **`fails on`** part of action definitions this scenario can be easily covered with Dura as well.

2.4. Modules

The Modularization mechanism of Dura is built on the notion of definitions and schemas that are discussed in section 2.2 and 2.3. These notions are combined with ideas from [**?**] and Java which are adapted for our requirements.

A simple notion of program modules is retained so as to keep the language simple and therefore easy to use. Our approach towards modularization mechanisms for Dura has the advantage that programmers which are already familiar with the modularization mechanisms of common languages are also familiar with the basic ideas of Dura's modularization mechanisms. Moreover, there is no need to reinvent the wheel for Dura because the available concepts can be easily adapted to our needs and yet they provide a substantial improvement to programs as it is discussed in section 1.1.

2.4.1. Basic Concepts

Modules are sets of rules which can be distributed among several files in the filesystem and which are associated with a unique namespace. Modules can be furthermore organized in a hierarchical structure that may or may not be reflected in the way the different program parts are stored on the file system.

The content of different modules is split among separate files each beginning with the declaration of the module name according to the usual naming pattern which separates different hierarchy levels by periods. Figure 1 contains an excerpt of the rule structure from the airport use case.

According to figure 1, files associated with the module `airport.alarms.detection` need to begin with the declaration `MODULE airport.alarms.detection`. A convenient, but not

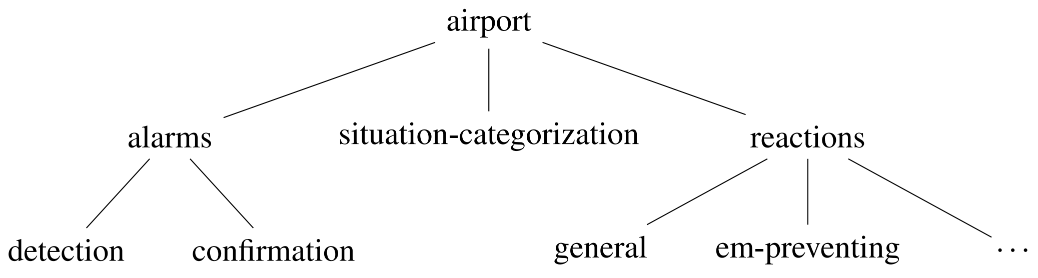

Figure 1: Structure of rules from the airport use case

mandatory, convention is that all files of the module are stored in the file system under `airport/alarms/detection`.[5]

Within each module, definitions of events, stateful objects and actions need to be unique, that is, there must not be two definitions for the same type with the same name regardless of whether their attributes differ or not. However, as the module name is also part of the according name of the definition, there may indeed be two definitions with the same basic name as long as they are associated with different modules.

2.4.2. Modifiers for the Visibility of Definitions

Modifiers specify the scope or the visibility of definitions. They specify which definitions can be used by rules of other modules and which definitions remain hidden from outside. Hence, modifiers are crucial to realize information hiding and encapsulation as it is desirable for high level languages.

Similar to Java there are three different modifiers for the visibility of definitions in Dura, namely **`public`**, **`private`**, and the default scope that is implicitly applied if no visibility is explicitly specified.[6] The meaning of these modifiers is summarized in table 2 where + denotes visible and - denotes not visible.

Modifier	File	Module	World
`public`	+	+	+
no modifier	+	+	-
`private`	+	-	-

Table 2: Impact of modifiers on the visibility of definitions

Public definitions can be imported and used by rules of any module of the complete program whereas private definitions can only be use by rules of definitions that are stored in the same file

[5]On Windows based operating systems \ needs to be substituted for /.
[6]Note that Java's `protected` is omitted as there is no inheritance in Dura.

as the private definition itself. Definitions with the default scope are visible to rules of the same module, whereby the rules do not need to be contained in the same file.

Note the different semantics of the term visible for event and stateful object definitions on the one hand and action definitions on the other hand. If an event or stateful object definition is visible within a file this means that it can be queried in the body of a rule. By contrast, if an action definition is visible within a file it means that the action can be used in a reactive rule, that is, it can be actually executed.

The scope is specified in front of the according definition. Listing 12 provides the definition of `unconfirmed-alarm` events that is referred to in the following examples. Note that the definition of `unconfirmed-alarm` is visible from rules of arbitrary modules whereas the stateful object definition of `area-state` is only visible by rules that are associated with the `airport.alarms.detection` module.

Listing 12: Event definition for `unconfirmed-alarm` events

```
MODULE airport.alarms.detection

public
EVENT
  unconfirmed-alarm{ area{string} }
WITH
  DETECT
    unconfirmed-alarm{ area{var A} }
  ON
    event e: area-state$updated{
      payload{ area{var A}, mode{"alert1"} }
    }
  END

  ...
END

STATEFUL OBJECT
  area-state{ area{string}, mode{string} }
END

...
```

Opposed to the way visibility of classes is handled in Java, in Dura one file may contain several public declarations and there is no obligatory naming scheme for filenames. This deviation from Java's module system is mainly caused by the lack of corresponding notion of classes which combines properties of objects and functions on these objects in a single class definition. In Dura, related events, stateful objects and actions are coupled loosely whereby their visibility needs to be specified independently and thus it is reasonable that one file may contain several public definitions.

2.4.3. Loading Modules

The definitions and functionality of the reactive rules of a module can be used by other modules if desired. However, before any definition from a module can be used, the entire rules of the according module need to be loaded, at least conceptually, into the set of rules that is evaluated during runtime. Afterwards, all definitions that are visible from the loading module can be used as if they have been defined locally.

At this point, it is important to understand that by loading a module, all its rules, in particular its reactive rules, are added to the set of rules that is evaluated during runtime. In a certain sense, by loading a module all its reactive rules are activated.

For instance, prior to the usage of `unconfirmed-alarm` from listing 12, the module `airport.alarms.detection` needs to be loaded by means of the `LOAD airport.alarms.detection` command. As the `unconfirmed-alarm` definition is visible from arbitrary modules the corresponding events can subsequently be used in event queries by referencing them with their full name, including the name of the modules they are defined in.

Listing 13: Example of the load mechanism in Dura

```
MODULE airport.alarms

LOAD airport.alarms.detection
LOAD airport.alarms.confirmation

...
```

Note that, submodules are only loaded if it is explicitly specified by their parent module. For instance, in listing 13, it is explicitly specified that all submodules of `airport.alarms` should also be loaded. However, without the two `LOAD` statements they just would not have been loaded.

When a module is loaded, all modules that are in turn loaded by this module are indirectly loaded as well. Therefore, when the module `airport.alarms` from listing 13 is loaded, the two submodules that are explicitly specified are indirectly loaded as well. However, by default, only definitions that are located in the module that is actually loaded can be be used. Therefore, definition from modules that are indirectly loaded cannot be used in the loading module. Thus, when just the module `alarms` is loaded, `unconfirmed-alarms` cannot be used in queries although the according definition has been indirectly loaded.

This behavior seems reasonable because it makes it possible to hide auxiliary definitions that are just loaded by a certain module for internal purposes. However, this default behavior can be adapted by explicitly specifying which definitions from indirectly loaded modules should be also automatically exported to other modules.

In the preceding example it is desirable that `confirmed-alarms` that are defined in the module `airport.alarms.confirmation` are automatically available when the module `airport.`

alarms is loaded, whereas unconfirmed-alarms should remain hidden in the first place.[7]

This behavior is realized by means of the EXPORT statement as it is demonstrated in listing 14. Note that EXPORT can either be followed by the full name of a definition, which exports only the given definition, or it can be followed by the name of a module which exports all public definition of the given module.

Listing 14: Example of the export mechanism in Dura

```
MODULE airport.alarms

LOAD airport.alarms.detection
LOAD airport.alarms.confirmation

EXPORT airport.alarms.confirmation.confirmed-alarm

...
```

Exporting definitions is a powerful mean to realize versatile submodule structures that automatically provide certain definitions whereas auxiliary definitions can be easily hidden.

2.4.4. Importing Definitions

Importing definitions is basically just a shorthand that makes it possible to reference a definition by its basic name, omitting the name of the according module. However, only definitions that can anyhow be used in a module, that is, definitions that have been loaded as it has been described in the preceding section, can be imported by a module. Hence, importing definitions is just a mean to abbreviate names of definitions that are anyhow available in a module.

If the import statement is omitted from listing 15, the unconfirmed-alarm events are still available, however they then need to be referred by their full name, namely airport.alarms .detection.unconfirmed-alarm. Depending on the concrete situation one or another way may be more convenient.

2.4.5. Imports and Reactive Rules

In contrast to other approaches, we actually need to distinguish between loading modules and importing definitions. This is mandatory as modules may internally rely on reactive rules that alter stateful objects to provide their functionality. For instance, unconfirmed-alarms from listing 12 are only derived if the according area is changes its state in a certain way. In turn, the state of the area is adapted by a reactive rule that is contained in the airport.alarms .detection module. Therefore, if just the definition unconfirmed-alarm is imported, the

[7] They can still be made available by manually loading the airport.alarms.detection module.

Listing 15: Example of the import mechanism in Dura

```
MODULE airport.reactions.general

LOAD airport.alarms.detection
IMPORT airport.alarms.detection.unconfirmed-alarm

ON
  and{
    event e: unconfirmed-alarm{ area{var A} },
    state s: responsible-warden{ area{var A}, actor{var W} }
  } where { state s valid-at end(event e) }
DO
  action a: assign-warden{ area{var A},  actor{var W} }
END

ACTION
  assign-warden{ area{string} }
  succeeds on{ ... }
  fails on{ ... }
END
```

adaptation of the `area`'s state by means of a reactive rule is missing and therefore the imported definition does not provide the intended semantic.

According to this examples, the semantics of programs that are contained in modules differs depending on whether the reactive rules of the according module are considered during runtime or not. As it is natural that during the development of a certain module all its reactive rules are considered as being evaluated, it is mandatory that this still holds if the functionality of the module is imported by another program as well. Therefore, it is also mandatory in Dura to maintain this behavior when the functionality of a module is imported into a program and hence modules, more precisely all rules of a module, need to be loaded before its functionality can be imported.

In purely declarative approaches without reactive rules it is indeed sufficient having only the `IMPORT` statement included in a language. However, neither do reactive rule have a name so that they can be addressed properly nor are they declarative, that is, they have side effects which can directly or indirectly affect the behavior of other rules. This is a major difference to other approaches that are described in the literature.

2.4.6. Visibility of Implicit Event and Action Definitions

The visibility that is set for implicit definitions depends on the type of the definition and the visibility of the corresponding explicit stateful object or action definition. More precisely, the visibility of implicit event definitions is inherited from the according explicit definition, whereas the visibility of implicit action definitions is set to the default visibility. This behavior has the effect that events which are related to a public stateful object are also visible within the whole program whereas updates of the object remain in control of the module that contains the definition.

Consider, for instance, the stateful object `operation-mode` which has been used in many examples of prior documents [**?**, **?**, **?**]. As this particular stateful object represents a variable that plays an important role for many details of both the airport and the metro use case, it needs to be visible for all rules no matter what module they are contained in. Consequently, the corresponding stateful object is declared public. Hence, values of the stateful object can be queried by rules of the entire program and the according implicit events, which inform about changes of the stateful object, are also visible in all modules of the program. Moreover, the stateful object can only be updated by rules that belong to the `airport.situation-categorization` module.

To further restrict which rules can update the stateful object `operation-mode`, the scope of the according `operation-mode$update` action needs to be adapted. To this end, the usually implicit definition needs to be explicitly added to the program whereby the desired scope can be explicitly specified as well.[8] However, there is no need to specify the attributes of the `operation-mode$update` action as they are already determined by the name of the action and the definition of the stateful object `operation-mode`. Consequently, they do not need to be specified in the according definition.

Listing 16: Definitions related to the stateful object `operation-mode`

```
MODULE airport.situation-categorization

public
STATEFUL OBJECT
  operation-mode{ mode{string} }
END

private
ACTION
  operation-mode$update{ }
END
```

In general, this approach can be used with all implicitly defined events and actions. An explicit definition for usually implicit event and action definition is added to the program which just contains the name of the corresponding event or action without any further attributes. The visibility of the (now explicit) definition can then be adapted by adding the desired modifiers. Listing 17 contains explicit definitions for the usually implicitly defined events of a generic actions called *name*. The form of explicit definitions for usually implicitly defined actions resembles the form of the given event definitions.

[8]In order to completely prevent any modifications the visibility of the actions `operation-mode$create` and `operation-mode$terminate` needs to be adapted as well.

Listing 17: Explicit event definitions for usually implicitly defined events

```
EVENT
  name$initiated{ }
END

EVENT
  name$succeeded{ }
END

EVENT
  name$failed{ }
END
```

Note that listing 7 and 9 contain similar definitions which are only intended to illustrate the form, in particularly the attributes, of the implicitly defined events and actions. However, the definitions that were given in these two listings cannot be used in sound Dura programs.

2.4.7. Controlling Information Exchange with External Components

Modifiers as they are discussed in the preceding sections control the visibility of definitions within the event processing system. However, neither do they consider whether external events are visible to the event processing system nor whether events and action are visible to external components.

To specify the flow of events and actions between external components and the event processing system two further modifiers are introduced, namely `input` and `output`. They can be used in combination with the three modifiers that have been introduced in section 2.4.2. The `input` modifier can be used in conjunction with event definitions and specifies that events of the corresponding types are sent to the event processing system by external components such as sensors. In contrast, `output` in front of an event or action definition specifies that the event processing system needs to provide the according events and actions to external components. Because stateful objects cannot be queried directly by external components, only the `input` modifier can be uses in combination with stateful objects in order to be able to provide initial states.

Note that most atomic events and action definitions need to be specified as being either of type `input` or `output`. Most likely, only atomic events and actions that are provided by the event processing system, that is, events and actions that are implicitly defined by event and stateful object definitions, will be specified without the two modifiers. Moreover, normally implicitly defined events and stateful objects can only be declared with either no or just the `output` modifier and actions cannot be associated with the `input` modifier.

A more detailed descriptions of how the concrete communication between external components and the event processing system is realized and which impact both modifiers have inside the event processing system can be found in [?], [?] and [?].

Listing 18: Making updates of `operation-mode` visible for external components

```
public output
EVENT
  operation-mode$updated{ }
END
```

This mechanism can also be used to forward notifications of stateful object updates to external components. For instance, it seems desirable that the SITE Gui is automatically notified if the operation mode changes so that it can inform the operator accordingly. This behavior can be achieved by adapting the visibility of the `operation-mode$update` from listing 17 as it is demonstrated in listing 18.

2.4.8. Extensions for a looser coupling of Modules

The mechanism that is described in the preceding section can be generalized towards a more versatile information exchange between different modules in a manner that is similar to the approach that is proposed in [?]. Currently, all rules that derive a certain event need to be specified in the same definition and thus in the same file. If a module wants to use definitions from other modules it needs to know these modules upfront in order to be able to load them. It cannot just declare that it is interested in events of a certain type.

Our approach can be generalized so that there may be indeed several definitions for the same type that are distributed among several files. Therefore, events of the same type can be derived by several modules and the **`input`** and **`output`** qualifier control how events flow between several modules in addition to the flow of events between external components and the event processing system.

Therefore, modules are coupled more loosely and integration of modules from independent sources is facilitated. However, in the context of this project modules are considered as a mean to structure large programs rather than being a mean to integrate different modules from independent sources. Moreover, it is crucial in the area of emergency management that all involved components are well known and understood. Therefore it does not seems to be desirable to provide further means that ease the integration of arbitrary modules that are not well known to the programmer.

2.4.9. Specifying Reactive Rules in Modules

Reactive rules have a special role in the module system of Dura. Whereas declarative rules derive new events or stateful objects and complex action rules define new actions, reactive rules only trigger actions on the occurrence of certain events. They just make the transition from the declarative world of events and stateful objects to the imperative world of complex actions but do not provide new definitions of any kind. As a consequence, reactive rules do not need to be specified within any definition.[9] They are just specified where common definitions are given in a file.

As it is described in section 2.4.3, reactive rules have a special status in modules. When a module is loaded, all its contained reactive rules are added to the set of rules that is evaluated during runtime. Accordingly, modifiers for the visibility of definitions cannot be used in combination with reactive rules.

Listing 15 in section 2.4.1 already provided an example for the usage of reactive rules in combination with definitions.

2.4.10. Further Means for Structuring Rules

The `WHILE` statement in Dura is another mean for the structuring of rules in a program. It is orthogonal to definitions in the sense that definitions group together rules that derive the same type of events and stateful objects or that define the same type of actions. In contrast, `WHILE` groups rules according to a stateful object, that is, it groups rules that should be applied in the same situation or context.

Consequently, the integration of the statement into the module system of Dura is somewhat cumbersome. `WHILE` statements activate and deactivate the rules they contain and hence, during runtime, they can be regarded as a set of common rules. Therefore `WHILE` statements need to be included in definitions and all rules that are contained in the statement need to match the schema of the corresponding definition. As a result, `WHILE` statements can only contain rules that derive the same type of events and stateful objects or define the same type of actions.

Reactive rule that are specified in a `WHILE` statement are treated in a special way. For the same reasons as they have been discussed in the preceding section, `WHILE` statements that only contain reactive rules do not need to be included in any definition.

[9]Note that reactive rules may query several events and execute several actions which makes a unique attribution to a single event or action definition rather unintuitive.

A. Example of Implicit Definitions Caused by an Explicit Definition

This section contains an exhaustive listing of all implicit event and action definitions that are related to the definition of the stateful object `operation-mode` that is given in listing 19. In the subsequent listings 20, 22, 21, and 23 all attributes, in particularly the implicit ones, of the according stream definitions are contained.

Listing 19 contains a definition of the stateful object `operation-mode` like a programmer would specify it in a Dura program. The definition contains only explicit attributes that actually carry the values the programmer is interested in.

Listing 19: User defined stateful object `operation-mode`

```
STATEFUL OBJECT
  operation-mode{ mode{string} }
END
```

Internally, the definition from listing 19 is augmented with implicit attributes, like, for instance, attributes which specify at at which time the stateful object is valid as it is shown in listing 20. Usually these attributes can be queried in a program but their value cannot be determined explicitly by the programmer.

Listing 20: Definition of `operation-mode` inclusive its implicit attributes

```
STATEFUL OBJECT
  operation-mode{
    id{identifier},
    valid-time{time-interval},
    mode{string}
  }
END
```

Be aware that in the following the values of *name*`.payload.id` in listing 21 are referring to the instance of a stateful object, whereas the values of *name*`.payload.id` in listing 23 are referring to the instance of an action. Accordingly, the values in the `payload` attribute of, for instance, the `operation-mode$created` event are referring to the instances of the stateful object `operation-mode`. In contrast, the values of the `payload` attribute of, for instance, the `operation-mode$terminate$initiated` event are referring to instances of the action `operation-mode$terminate`.

Listing 21: Implicit event definitions caused by listing 20

```
EVENT
  operation-mode$created/terminated/updated{
    id{identifier},
    reception-time{time-interval},

    payload{
      id{identifier},
      valid-time{ begin{timestamp} },
      mode{string}
    }
  }
END
```

Listing 22: Implicit action definitions caused by listing 20

```
ACTION
  operation-mode$create{
    values{ mode{string} }
  }
END

ACTION
  operation-mode$terminate{
    query{ id{identifier} }
  }
END

ACTION
  operation-mode$update{
    query{ id{identifier} },
    values{ mode{string} }
  }
END
```

Listing 23: Implicit event definitions caused by listing 22

```
EVENT
  operation-mode$create$initiated/succeeded/failed{
    id{identifier},
    reception-time{time-interval},

    payload{
      id{identifier},
      initiation-time{ begin{timestamp} },
      values{ mode{string} }
    }
  }
END

EVENT
  operation-mode$terminate$initiated/succeeded/failed{
    id{identifier},
    reception-time{time-interval},

    payload{
      id{identifier},
      initiation-time{ begin{timestamp} },
      query{ id{identifier} }
    }
  }
END

EVENT
  operation-mode$update$initiated/succeeded/failed{
    id{identifier},
    reception-time{time-interval},

    payload{
      id{identifier},
      initiation-time{ begin{timestamp} },
      query{ id{identifier} },
      values{ mode{string} }
    }
  }
END
```

B. Dura Grammar

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

preamble ::= description pckg imprt (constDefinition | typeDefinition)* ;

pckg ::= ('MODULE' path)? ;

imprt ::= (('IMPORT'|'EXPORT') path)* ;

description ::= ('DESCRIPTION' STRING)? ;

/**
 * constraints:
 *   all schemaDefinition labels are pairwise distinct
 *   type name ID is unique
 *   no cyclic definitions
 */
typeDefinition ::= 'TYPE' ID 'IS' basicType typeSupplement '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 need to comply with schmaDenfinition
 */
eventDefinition ::= modifiers 'EVENT' schemaDefinition schemaSupplement
                       ('WITH' eventSpecification*)? 'END' ;

/** constraint: stateful object definition schemaDefinition is unique */
stateDefinition ::= modifiers 'STATEFUL OBJECT'
                       schemaDefinition schemaSupplement 'END' ;

/** constraint: action definition schemaDefinition is unique */
actionDefinition ::= modifiers 'ACTION' schemaDefinition schemaSupplement
                        ((succeedsOn failsOn?) | (failsOn succeedsOn?))?
                        ('WITH' actionSpecification)? 'END' ;
```

```
actionSpecification ::= 'FOR' 'action' ID? ':' term 'DO' actionComposition 'END'

actionComposition ::= 'concurrent' '{' flatAction (',' flatAction)* '}'
                         ( (executionConstraints ( (succeedsOn failsOn?)
                                                 | (failsOn succeedsOn?))? )
                         | (succeedsOn ( (failsOn executionConstraints?)
                                       | (executionConstraints failsOn?))? )
                         | (failsOn ( (succeedsOn executionConstraints?)
                                    | (executionConstraints succeedsOn?))? )
                         )?
                    | flatAction
                    ;

executionConstraints ::= 'where' '{' atomicConditionFormula
                            (',' atomicConditionFormula)* '}' ;

succeedsOn ::= 'succeeds on' '{' eventQuery '}' ;

failsOn ::= 'fails on' '{' eventQuery '}' ;

flatAction ::= 'action' ID? ':' constructTerm
             | 'action' ID? ':' actionComposition
             ;

/** constraint: query is range restricted with respect to term */
eventSpecification ::= 'DETECT' constructTerm
                         ('group by' '{' binding (',' binding)* '}')?
                         'ON' eventQuery 'END' ;

/** constraint: action is range restricted with respect to query */
reactiveRule ::= 'ON' eventQuery 'DO' limitedActionComposition 'END' ;

limitedActionComposition ::= 'concurrent' '{' flatAction (',' flatAction)* '}'
                                  executionConstraints?
                           | flatAction
                           ;

/** constaint: at least one event query in every disjunct */
eventQuery ::= 'and' '{' flatEventQuery (',' flatEventQuery)* '}' querySupplement
             | 'or' '{' flatEventQuery (',' flatEventQuery)* '}' querySupplement
             | flatEventQuery
             ;

flatEventQuery ::= 'not' '{' atomicEventQuery '}' querySupplement
                 | 'not' atomicEventQuery
                 | 'exists' '{' atomicEventQuery '}' querySupplement
                 | 'exists' atomicEventQuery
                 | atomicEventQuery
                 ;

complexSubquery ::= 'and' '{' flatEventQuery (',' flatEventQuery)* '}' querySupplement
                  | 'or' '{' flatEventQuery (',' flatEventQuery)* '}' querySupplement
                  ;

atomicEventQuery ::= 'event' ID ':' ('action' ID '$') qualifiedTerm querySupplement
                   | 'event' ID ':' qualifiedTerm querySupplement
                   | 'state' ID? ':' qualifiedTerm querySupplement
                   | 'event' ID? ':' complexSubquery
                   ;

action ::= 'concurrent' '{' atomicAction (',' atomicAction)* '}'
         | atomicAction
         ;

/** constraint: action term is actually defined */
atomicAction ::= 'action' ID? ':' constructTerm ;

querySupplement ::= (where | let | grouping)* ;

actionSupplement ::= where* ;

typeSupplement ::= tupleConstraints? ;

schemaSupplement ::= tupleConstraints? relationConstraints? ;

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

unification ::= typedDataVariable '=' expr ;

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

binding ::= dataVariable
          | identifier
          ;

aggregation ::= typedDataVariable '=' aggregationOp '(' dataVariable ')' ;
```

```
/** constraint: no variables occur in mathFormula */
tupleConstraints ::= 'where' '{' atomicMathFormula (',' atomicMathFormula)* '}' ;

relationConstraints ::= 'constraints' '{' atomicRelationConstraint
                           (',' atomicRelationConstraint)* '}' ;

atomicRelationConstraint ::= ('bound' | 'primary-key') '(' path ')'
                          | ('foreign-key') '(' path ',' path ')'
                          ;

where ::= 'where' 'and' '{' conditionFormula (',' conditionFormula)* '}'
        | 'where' 'or' '{' conditionFormula (',' conditionFormula)* '}'
        | 'where' 'not' '{' conditionFormula (',' conditionFormula)* '}'
        | 'where' '{' conditionFormula (',' conditionFormula)* '}'
        ;

conditionFormula ::= 'and' '{' conditionFormula (',' conditionFormula)* '}'
                   | 'or' '{' conditionFormula (',' conditionFormula)* '}'
                   | 'not' '{' conditionFormula (',' conditionFormula)* '}'
                   | atomicConditionFormula
                   ;

atomicConditionFormula ::= atomicIntervalFormula
                         | atomicMathFormula
                         ;

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

atomicIntervalFormula ::= '{' timeInterval ',' timeInterval
                             (                    '}' 'apart-by' duration
                             | (',' timeInterval)* '}' 'within' duration )
                        | timeInterval ( intervalRelation timeInterval
                                       | timepointRelation timePoint )
                        ;

/** constraint: variable is of type timeinterval */
timeInterval ::= dataVariable
               | identifier
               | 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)* '}'
       ;

qualfiedTerm ::= path '{' '}'
               | path '{' termLeaf '}'
               | path '{' term (',' term)* '}'
               ;

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

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

internalSchemaDefinition ::= internalLabel '{' '}'
                           | internalLabel '{' basicType '}'
                           | internalLabel '{' compositeType '}'
                           | internalLabel '{' internalSchemaDefinition
                                (',' internalSchemaDefinition)* '}'
                           ;

internalLabel ::= aggregationOp
                | intervalOp
                | ID
                | ID '$' ('initiated' | 'failed' | 'succeeded')
                | INTERNAL_ID
                | INTERNAL_ID '$' ('initiated' | 'failed' | 'succeeded')
                ;

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

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

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

modifiers ::= modifier* ;

modifier ::= 'input'
           | 'output'
           | 'log'
           | 'public'
           | 'private'
           ;

expr ::= mathExpr
       | identifier
       ;

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

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

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

/**
 * constraints:
 *   constants used in aritmetic expressions need to be a number
 */
atom ::= '(' mathExpr ')'
       | ('least' | 'greatest') '(' mathExpr (',' mathExpr)* ')'
       | intervalOp '(' timeInterval ')'
       | clockFunction
       | dataVariable
       | constant
       | '-' atom
       | aggregationOp '(' dataVariable ')'
       | path
       | (NUMBER) (timeUnit time*)?
       | STRING
       ;

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

clockFunction ::= 'system' '.' ('now' | 'tuple_id_seq') '(' ')' ;

label ::= aggregationOp
        | intervalOp
        | ID
        | ID '$' ('initiated' | 'failed' | 'succeeded')
        | ('action' | 'event' | 'state' | 'succeeded' | 'failed' | 'initiated')
        ;

type ::= basicType
       | compositeType
       ;
```

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

/** constraint: composite type ID is actually defined */
compositeType ::= ID
                | path
                ;

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

dataVariable ::= 'var' ID ;

typedDataVariable ::= 'var' type ID ;

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

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

timepointRelation ::= 'valid-at' ;

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')  ;
```