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

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

D4.5 Implementation

Due date of deliverable: 30/06/2011
Actual submission date: 31/08/2011

Revision: Version 1

Ludwig-Maximilians University Munich (LMU)

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

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

Index

Preface

This document is a report about the prototype implementation if the complex event processing language Dura and its evaluation engine Event-Mill. It complements the implementation code and the documentation files which were uploaded to the project server (`https://bscw.iais.fraunhofer.de/bscw/bscw.cgi/d238038/event-mill-and-examples.zip`). The implementation code forms the main part of Deliverable D4.5. The report is focused on information that the other work packages WP2, WP3, WP5, WP6 need to use and integrate the prototype in their systems.

First the report describes the interfaces for providing a Dura program and for compiling and executing that program (Section 1-3). Moreover the io-interfaces for providing the initial data of states and the stream data of events are explained (Section 4). Both interfaces are illustrated by two simple examples (Section 5-7).

Second the report provides an extended description of the Dura type and schema system (Section 8) and introduces Dura_C a subset of the Dura language that omits syntactic sugar and is used as intermediate step in bootstrapping the compilation of the full Dura language (Section 9).

Note that the report does not intend to provide a detailed description of the internal compilation and execution process, but concentrates on the external effects, i.e. those steps that are relevant for the users of Dura and the Event-Mill engine.

Part I.
The Event-Mill Engine

1. Setup & Configuration

1.1. Needed Software

- MonetDB SQL database (`http://www.monetdb.org/Downloads`)
- Java 1.6 or higher[1] (`http://www.oracle.com/technetwork/java/javase/downloads/index.html`)
- event-mill.jar

1.2. Getting started

1. Install Java 1.6
2. Install MonetDB SQL Server
3. Adopt the contained emili.configuration.xml file or delete the emili.configuration.xml file, execute event-mill.jar and insert the required config data. For help to the config dialog see the description of the config command.

   The contained config file should cooperate by default with a local out of the box installation of MonetDB when the DB-server is started manually before starting the Event-Mill engine. If the Event-Mill engine needs to start the DB-server by itself the start-up command for the DB-server is likely to need some adoption.
4. Execute event-mill.jar if it is not already running because of the configuration. It is recommended to start the MonetDB server manually before executing event-mill.jar, however if the DB-server start-up command is correctly configurated this should not be required.

2. The Main Dialog – Compiling and Initialising a Dura Program

2.1. The `config` command

Syntax: `config`

Enters the configuaration dialog. Mostly intended for a first configuration. For configuartion changes editing the config file is probably more convenient.

The config dialogs asks for the follwoing information:

[1] Currently it is recommended not to use Java 1.7 but to wait at least for its first update.

- **url:** The JDBC url of the (MonetDB) database server
- **user:** Username for the database
- **password:** Password for the database
- **database server start command:** Operating system command for starting the (MonetDB) database server
- **schema name for meta tables of engine:** The name of the schema where the meta data of the engine should be stored. The schema may not exist, yet.

2.2. The `compile` command

Syntax: `compile NAMESPACE.PROGRAM_NAME, DURA_FILE.dura`

Compiles the Dura program specified in `DURA_FILE` and stores the compiled program specification under name `NAMESPACE.PROGRAM_NAME`. Fails if there already exists a program specification with name `NAMESPACE.PROGRAM_NAME`.

Syntax: `compile NAMESPACE.PROGRAM_NAME, DURA_FILE.dura, SQL_SPEC_File.xml`

Compiles the Dura program specified in `DURA_FILE` and stores the compiled program specification together with the specified name `NAMESPACE.PROGRAM_NAME` in the `SQL_SPEC_File.xml` file.

2.3. The `load` command

Syntax: `load spec SQL_SPEC_FILE.xml`

Loads the compiled form of a program specification contained in the file `SQL_SPEC_FILE.xml` and stores the program specification in the database using the name specified in the file. Fails if there already exists a program specification with the same name.

2.4. The `list specs` command

Syntax: `list specs`

Lists the names of all available program specifications.

2.5. The `remove spec` command

Syntax: `remove spec NAMESPACE.PROGRAM_NAME`

Removes the program specification with name `NAMESPACE.PROGRAM_NAME` if such exists.

2.6. The `clear specs` command

Syntax: `clear specs`

Removes all existing program specifications.

2.7. The `init` command

Syntax: `init NAMESPACE.PROGRAM_NAME:INSTANCE_NAME, META_SCHEMA, INPUT_SCHEMA, WORKING_SCHEMA, OUTPUT_SCHEMA, LOG_SCHEMA`

Initialize the program instance named `NAMESPACE.PROGRAM_NAME:INSTANCE_NAME` at the database location given by `META_SCHEMA`, `INPUT_SCHEMA`, `WORKING_SCHEMA`, `OUTPUT_SCHEMA` and `LOG_SCHEMA` . Here `META_SCHEMA`, `INPUT_SCHEMA`, `WORKING_SCHEMA`, `OUTPUT_SCHEMA` and `LOG_SCHEMA` denote different not yet existing schema which are used for storing the meta data of the program, the input buffers (tables) for the incoming events, the buffers for the internal query evaluation, the buffers (tables) for providing the derived events and actions and the tables for writing a log. During initialization all necessary schema and tables are created and the meta data of the program is set to its initial state.

2.8. The `list progs` command

Syntax: `list progs`

Lists the name of all instantiated program instances.

2.9. The `run` command

Syntax: `run NAMESPACE.PROGRAM_NAME:INSTANCE_NAME`

Loads the program instance named `NAMESPACE.PROGRAM_NAME:INSTANCE_NAME` and switches to the execution control for that program instance.

See the Section 3 for execution control commands.

2.10. The `remove prog` command

Syntax: `remove prog NAMESPACE.PROGRAM_NAME:INSTANCE_NAME`

Removes the program instance named `NAMESPACE.PROGRAM_NAME:INSTANCE_NAME` if such exists. Also deletes all schemas and therefore data used by the program instance.

2.11. The `clear progs` command

Syntax: `clear progs`

Removes all existing program instances. Also deletes all schemas and therefore data used by these program instances.

2.12. The `clear` command

Syntax: `clear`

Removes all existing program instances and specifications. Same effect as `clear progs` followed by `clear specs`.

2.13. The `uninstall` command

Syntax: `uninstall`

Deletes the schema used for the meta data of the Event-Mill engine but does not remove the data of existing program instances. When carried out after the clear progs command, no data related to Event-Mill remains in the database.

2.14. The `quit` command

Syntax: `quit`

Exits the Event-Mill engine.

3. The Control Dialog – Running a Dura Program

3.1. The `init` command

Syntax: `init`

Performs last initialization steps for running the program Particularly copies the initial static and stateful data from the input buffers (tables) into the working buffers (tables).

3.2. The `start` command

Syntax: `start`

Starts/Resumes the query execution for the current program.

3.3. The `stop` command

Syntax: `stop`

Stops the query execution for the current program. Resume with start.

3.4. The `quit` command

Syntax: `quit`

Stops the query execution for the current program if it is running. Returns to the administration command level.

Note: Using the run command for the same program instance, the program execution can be resumed at the point it had been stopped.

4. Delivering and Retrieving Data

4.1. Dura Type Names

A simple type name in Dura starts with any alphabetic or an underscore character followed by an arbitrary number of alphabetic, digit, underscore or dash characters.

Dura provides packages as mean for structuring a program. A simple package name starts with any alphabetic or an underscore character followed by an arbitrary number of alphabetic, digit, underscore or dash characters and ends with a dot. A general package name is an arbitrary long sequence of simple package names.

A general type name in Dura consists of a (potentially empty) package name followed by a simple type name.

4.2. Mapping to Table Names

As SQL does not allow dots and dashes to be part of table names the following encoding for type names is chosen when creating the corresponding database tables: Each underscore is replaced by a double underscore, each dot is replaced by "_o_" and each dash is replaced by "_d_". This encoding is unambiguous.

The table mapping for some type can be obtained from the meta schema (see below) of a program with the following query:

```
SELECT "table" FROM META_SCHEMA."buffers"
WHERE
   "prog_package"='PACKAGE_PART_OF␣TYPE_NAME' AND
   "simple_name"='SIMPLE_NAME_PART_OF␣TYPE_NAME' AND
   "type" = 'INPUT'
```

;

4.3. Dura Attribute Names

A simple attribute name in Dura starts with any alphabetic or an underscore character followed by an arbitrary number of alphabetic, digit, underscore or dash characters.

A general attribute name consists of an arbitrary number of simple attribute names which are separated by a dot. The general attribute names are introduced to provide so-called “complex attributes” as mean for structuring the data of an event, state or action. Basically a complex attribute consists of those attributes where the sequence of simple attribute names forming the name of the complex attribute is a prefix of the sequence of simple attribute of the attributes.

4.4. Mapping to SQL Attribute Names

As SQL does not allow dots and dashes to be part of attribute names the following encoding for type names is chosen when creating the corresponding attribute in a database tables: Each underscore is replaced by a double underscore, each dot is replaced by "_o_" and each dash is replaced by "_d_". This encoding is unambiguous.

4.5. Mapping of Attribute Types

All Dura types are mapped to flat tuples where the structure is somehow encoded into the (general) attribute names (See also 8.2). The mapping of the basic Dura types to SQL types is shown in Table 1.

Dura	MonetDB
TIMESTAMP	BIGINT
DURATION	BIGINT
BOOLEAN	BOOLEAN
INT	INT
LONG	BIGINT
FLOAT	REAL
DOUBLE	DOUBLE
STRING	CLOB
ID	BIGINT

Table 1: Mapping of basic Dura types to MonetDB SQL types

4.6. Schema

The Event-Mill engine uses five different schema in the database for each initialized program:

- The meta schema for storing the meta data of a program
- The input schema for the tables buffering the incoming events
- The working schema for the buffers used by the internal query evaluation
- The output schema for the tables providing the derived events and actions
- The log schema for tables that contain logs for certain events, states and actions

If a type is an input type, then for this type there is a table in the input schema and in the working schema. If a type is an output type, then for this type there is a table in the output schema and in the working schema. External components write to tables in the input schema and read from tables in the output schema.

4.7. Input

4.7.1. Delivering Events

The incoming events of a certain type have to be inserted into the corresponding table within the input schema. The payload of the event is stored in the attributes of the inserted tuple.

The initial states of a certain type have to be inserted into the corresponding table within the input schema. The state data is stored in the attributes of the inserted tuple.

4.7.2. Initialize Static Data

Static data is treated as states. Thus static data is provided as initial states of a certain (state) type which never change.

The "_o_id" attribute of the inserted tuple has a special purpose and MUST NOT be set by the payload of the event. Instead it is required to obtain the default value specified in the table definition.

Note: "_o_id" represents the ".id" attribute on a higher level. Attributes starting with "_o_" should never be set explicitly.

4.8. Output

4.8.1. Retrieving Events and Actions

The outcoming events of a certain type are stored in the corresponding table within the output schema.

The events/tuples can be read incrementally using their "_o_id" attribute and the first output min-time value of the query performing the copying from the internal buffer/table for the event type to the corresponding table in the output schema. This means:

1. Obtain value of first output min-time

```
SELECT "output_min_time_value_1" FROM META_SCHEMA."queries"
   WHERE "type" = 'OUTPUT' AND "output" = TYPE_NAME ;
```

2. Copy all tuples where the value of the "id" attribute is between the previous and the current value of the first output min-time.

```
SELECT * FROM OUTPUT_SCHEMA.CORRESPONDING_TABLE WHERE "_o_id"
    > /*previous tuple_id_seq*/? AND "_o_id" <= /*current
   tuple_id_seq*/? ;
```

   Note: Tuples with `"_o_id"> /*current tuple_id_seq*/?` should not be read from the table.

3. Store the current value of the first output min-times as previous value for the next round

5. The Hello World Example

5.1. Description

The Hello World example realizes a simple politeness rule: Each time I see a person, I should greet that person. Thus the Hello World program consists of a single rule stating that whenever a see_person event arrives, a greet_person event should be derived. Both the see_person as well as the greet_person event carry a "person" attribute which holds the name of the person that has been seen or should be greeted.

5.2. Input

The incoming see_person events have to be inserted into the `INPUT_SCHEMA."see__person"` table. The table has two attributes, namely "_o_id" and "person". The insertion *must not* set the "_o_id" attribute. It is required that the "_o_id" attribute is set to the default value specified in the table definition.

5.3. Output

The outcoming greet_person events are stored in the `OUTPUT_SCHEMA."greet__person"` table.

The greet_person events/tuples can be read incrementally using their "_o_id" attribute and the first output min-time value of the query performing the copying from the `WORKING_SCHEMA."`

Listing 1: The Dura types and rules for the Hello World example

```
DESCRIPTION "Hello World example."

input
EVENT
  see_person{ person{string} }
END

output log
EVENT
  greet_person{ person{string} }
WITH
  DETECT
    greet_person{ person{ var P} }
  ON
    event e: see_person{ person{ var P} }
  END
END
```

greet__person" buffer/table for the greet_person events to the corresponding OUTPUT_SCHEMA."greet__person" table.

This means:

1. Obtain value of first output min-time

   ```
   SELECT "output_min_time_value_1" FROM META_SCHEMA."queries"
      WHERE "type" = 'OUTPUT' AND "output" = 'greet_person' ;
   ```

2. Copy all tuples where the value of the "_o_id" attribute is between the previous and the current value of the first output min-time.

   ```
   SELECT * FROM OUTPUT_SCHEMA."greet__person" WHERE "_o_id" >
      /*previous tuple_id_seq*/? AND "_o_id" <= /*current
      tuple_id_seq*/? ;
   ```

   Note: Tuples with "_o_id"> */*current tuple_id_seq*/*? should not be read from the table.

3. Store the current value of the first output min-times as previous value for the next round

5.4. Input Example – see_person.jar

The see_person.jar realizes a simple console where the names of persons that have been seen can be entered and which delivers corresponding see_person events to the Hello World program. Note that see_person.jar expects that the name of the input schema is "hello_world_input".

5.5. Output Example – greet_person.jar

The greet_person.jar realizes a simple console where the names of persons that should be greeted are displayed. It reads the derived greet_person events and writes the corresponding names to the console. Note that greet_person.jar expects that the name of the output schema is "hello_world_output". greet_person.jar does not look for new greet_person events immediately when being executed, but waits for an explicit "start" command.

5.6. Test Sequence

```
compile example.hello_world, examples/hello_world.dura, examples/hello_world.xml

load spec examples/hello_world.xml
list specs
init example.hello_world:test, hello_world_meta, hello_world_input, hello_world_working
, hello_world_output, hello_world_log
list progs
run example.hello_world:test
start
stop
start
stop
quit
remove prog example.hello_world:test
remove spec example.hello_world
clear progs
uninstall
```

6. The Access Control Example

6.1. Description

The Access-Control example realizes two simple access control rules:

1. If a person requests access and the person is a member of the staff, then the access is granted.
2. If a person requests access but was granted access less than 30 seconds ago the this might indicate

a abuse of its identity and an intrusion warning is raised.

Listing 2: The Dura types and rules for the Access Control example

```
DESCRIPTION "Access control example."

input
EVENT
  request-access{ person{string} }
END

input
STATEFUL OBJECT
  staff{ person{string} }
END

output log
EVENT
  grant-access{ person{string} }
WITH
  DETECT
    grant-access{ person{ var P} }
  ON
    and{
      event e: request-access{ person{ var P} },
      state s: staff{ person{ var P} }
    }
  END
END

output log
EVENT
  intrusion-warning{ person{string} }
WITH
  DETECT
    intrusion-warning{ person{ var P} }
  ON
    and{
      event e: grant-access{ person{ var P} },
      event f: request-access{ person{ var P} }
    } where {{event e, event f} within 30 sec, event e before event
        f}
  END
END
```

6.2. Input

The incoming request-access events have to be inserted into the table with name `INPUT_SCHEMA."request_d_access"`. The table has two attributes, namely "_o_id" and "person". The insertion *must not* set the "_o_id" attribute. It is required that the "_o_id" attribute is set to the default value specified in the table definition.

The staff members which should be granted access have to be stored in the `INPUT_SCHEMA."staff"` table. The table has two attributes, namely "_o_id" and "person". The insertion *must not* set the "_o_id" attribute. It is required that the "_o_id" attribute is set to the default value specified in the table definition.

6.3. Output

The outcoming grant-access events are stored in the `OUTPUT_SCHEMA."grant_d_access"` table.

The grant-access events/tuples can be read incrementally using their "_o_id" attribute and the first output min-time value of the query performing the copying from the `WORKING_SCHEMA."grant_d_access"` buffer/table for the grant-access events to the corresponding `OUTPUT_SCHEMA."grant_d_access"` table. This means:

1. Obtain value of first output min-time

   ```
   SELECT "output_min_time_value_1" FROM META_SCHEMA."queries"
      WHERE "type" = 'OUTPUT' AND "output" = 'grant-access' ;
   ```

2. Copy all tuples where the value of the "_o_id" attribute is between the previous and the current value of the first output min-time.

   ```
   SELECT * FROM OUTPUT_SCHEMA."grant_d_access" WHERE "_o_id" >
      /*previous tuple_id_seq*/? AND "_o_id" <= /*current
      tuple_id_seq*/? ;
   ```

   Note: Tuples with `"_o_id"> /*current tuple_id_seq*/?` should not be read from the table.

3. store the current value of the first output min-times as previous value for the next round

The outcoming intrusion-warning events are stored in the table with name `OUTPUT_SCHEMA."intrusion_d_warning"`.

They can be read analogously to the grant-access events.

6.4. Input Example – access-control-input.jar

The access-control-input.jar realizes a simple console where the names of persons that request access can be entered and which delivers corresponding request-access events to the Access-

Control program. Note that access-control-input.jar expects that the name of the input schema is "access_control_input".

6.5. Output Example – access-control-output.jar

The access-control-output.jar realizes a simple console where grant-access and intrusion-warning events are displayed. It reads the derived grant-access and intrusion-warming events and writes "Grant access to NAME" or "Intrusion warning with name NAME" respectively to the console where NAME denotes the value of the person attribute contained in the event. Note that access-control-output.jar expects that the name of the output schema is "access_control_output". access-control-output.jar does not look for new events immediately when being executed, but waits for an explicit "start" command.

6.6. Test Sequence

```
compile example.access-control, examples/access-control.dura, examples/access-control
.xml
load spec examples/access-control.xml list specs
init example.access-control:test, access_control_meta, access_control_input,
 access_control_working, access_control_output, access_control_log
list progs
```

Execute in sql-client for setting the data for the staff members:

```
INSERT INTO "access_control_input"."staff" ("person" )
VALUES
        ('Simon'),
        ('Steffen'),
        ('Francois')
;
```

```
run example.access-control:test
init
start
stop
start
stop
quit
remove prog example.access-control:test
remove spec example.access-control
clear progs
uninstall
```

7. The Access Control Example & MonetDB

The section illustrates the effect of the commands introduced in Section 3 and Section 3 using the Access Control example. The effect of each command is briefly explained particularly with respect to changes in the database. After that the corresponding SQL statements are shown in a condensed form. Particularly read-only queries are omitted.

A complete SQL log can be obtained easily by

1. Prefixing the JDBC url in the Event-Mill configuration file with “jdbc:log4jdbc:” (i.e. “jdbc:log4jdbc:monetdb://localhost/demo” intead of “jdbc:monetdb://localhost/demo”)
2. Setting the log level of the “jdbc.sqlonly” logger from “WARN” to “INFO” (i.e. replace <logger name="jdbc.sqlonly" level="WARN"/> by <logger name="jdbc.sqlonly" level="INFO"/>)

7.1. First Start of Engine

Effect: Creates the schema (`"meta"`) which is configured to contain the meta data of the engine and creates the tables within that schema for storing the meta data. Currently the meta data consists of a table (`"program_specs"`) holding all available (compiled) program specifications and a table (`"program_insts"`) holding all available initialized programs and their location in the database and their current state.

Furthermore functions for computing the maximum and minimum of two attributes of the same tuple are defined.

SQL-Statements:

```
CREATE SCHEMA "meta";

CREATE TABLE "meta"."program_specs" (
        "id" INT AUTO_INCREMENT PRIMARY KEY,
        "namespace" CLOB NOT NULL,
        "simple_name" CLOB NOT NULL,
        "description" CLOB NOT NULL DEFAULT '',
        "xml_spec" CLOB NOT NULL,
        CONSTRAINT "program_spec_unique_name" UNIQUE ("namespace"
           , "simple_name")
);

CREATE TABLE "meta"."program_insts" (
        "id" INT AUTO_INCREMENT PRIMARY KEY,
        "spec_ref" INT REFERENCES "meta"."program_specs" ON
           UPDATE CASCADE ON DELETE CASCADE,
        "instance" CLOB NOT NULL DEFAULT '',
        "buffer_max_number_of_min_times" INT,
```

```
        "query_max_number_of_output_min_times" INT,
        "query_max_number_of_input_min_times" INT,
        "meta_schema" CLOB NOT NULL,
        "input_schema" CLOB NOT NULL,
        "working_schema" CLOB NOT NULL,
        "output_schema" CLOB NOT NULL,
        "log_schema"  CLOB NOT NULL,
        "state" CLOB NOT NULL,
        CONSTRAINT "program_unique_name" UNIQUE ("spec_ref", "
           instance")
);

CREATE FUNCTION GREATEST ( "n0" BIGINT, "n1" BIGINT ) RETURNS
   BIGINT
BEGIN
        IF ( "n1" > "n0" ) THEN SET "n0" = "n1"; END IF;
        RETURN "n0";
END;

CREATE FUNCTION LEAST ( "n0" BIGINT, "n1" BIGINT ) RETURNS BIGINT
BEGIN
        IF ( "n1" < "n0" ) THEN SET "n0" = "n1"; END IF;
        RETURN "n0";
END;
```

7.2. Compile

Command: `compile example.access-control, examples/access-control.dura, examples/access-control.xml`

Effect: Compiles the Access Control program specified in `examples/access-control.dura` and stores the compiled program specification together with the specified name “example.access-control” in the `examples/access-control.xml` file.

SQL-Statements: No effect to the database

7.3. Load

Command: `load spec examples/access-control.xml list specs`

Effect: Loads the compiled Access Control program from `examples/access-control.xml` and store and indexes it in the the `"program_specs"` table of the schema `"meta"` containing the meta information of the engine.

SQL-Statements:

```
INSERT INTO "meta"."program_specs"
   ("namespace", "simple_name", "description",
"xml_spec")
VALUES
   ( 'example', 'access-control', 'Access␣control␣example.',
'<program>
␣␣␣<namespace>example</namespace>
␣␣␣<simple_name>access-control</simple_name>
␣␣␣<description>Access␣control␣example.</description>
␣␣␣<buffers>
␣␣␣␣␣␣...
␣␣␣</buffers>
␣␣␣<queries>
␣␣␣␣␣␣...
␣␣␣</queries>
</program>')
;
```

7.4. Init Program

Command: `init example.access-control:test, access_control_meta, access_control_input , access_control_working, access_control_output, access_control_log`

Effect: Creates all schema and tables needed for the execution of the program instance and initializes the meta data necessary for controlling the incremental evaluation of the queries. Furthermore the program instance is registered in the engine meta data.

SQL-Statements:

Register the program instance

```
INSERT INTO "meta"."program_insts"
   ("spec_ref", "instance", "buffer_max_number_of_min_times", "
      query_max_number_of_output_min_times", "
      query_max_number_of_input_min_times", "meta_schema", "
      input_schema", "working_schema", "output_schema", "
      log_schema", "state")
VALUES
   ( (SELECT "id" FROM "meta"."program_specs"
WHERE
   "namespace" = 'example'
AND
   "simple_name" = 'access-control'
)       , 'test', 1, 1, 2, 'access_control_meta', '
   access_control_input', 'access_control_working', '
   access_control_output', 'access_control_log', 'CREATED')
```

```
;
```

Create the needed schema

```
CREATE SCHEMA "access_control_meta";
CREATE SCHEMA "access_control_input";
CREATE SCHEMA "access_control_working";
CREATE SCHEMA "access_control_output";
CREATE SCHEMA "access_control_log";
```

Create the sequence for producing unique identifiers for events, states and actions

```
CREATE SEQUENCE "access_control_meta"."tuple_id_seq" AS BIGINT
   START WITH 0;
```

Create the tables for storing the meta data

```
CREATE TABLE "access_control_meta"."buffers" (
   "id" INT AUTO_INCREMENT PRIMARY KEY,
   "prog_package" CLOB NOT NULL,
   "simple_name" CLOB NOT NULL,
   "min_times_number" INT NOT NULL,
   "type" CLOB NOT NULL,
   "table" CLOB NOT NULL,
   "has_changed" BOOLEAN DEFAULT TRUE NOT NULL,
   "min_time_name_1" CLOB,
   CONSTRAINT "unique_names" UNIQUE ("type", "prog_package", "
      simple_name"),
   CONSTRAINT "unique_table" UNIQUE ("type", "table")
);

CREATE TABLE "access_control_meta"."queries" (
   "id" INT AUTO_INCREMENT PRIMARY KEY,
   "prog_package" CLOB NOT NULL,
   "type" CLOB NOT NULL,
   "number_of_output_min_times" INT NOT NULL,
   "number_of_input_min_times" INT NOT NULL,
   "is_active" BOOLEAN DEFAULT TRUE NOT NULL,
   "inputs" CLOB NOT NULL,
   "output" CLOB NOT NULL,
   "sql_eval_statement" CLOB NOT NULL,
   "referenced_clocks" CLOB NOT NULL,
   "min_times_computation" CLOB NOT NULL,
   "output_min_time_name_1" CLOB,
   "output_min_time_value_1" BIGINT,
   "input_min_time_name_1" CLOB,
   "input_min_time_value_1" BIGINT,
   "input_min_time_name_2" CLOB,
```

```
   "input_min_time_value_2" BIGINT
);
```

Create tables/buffers needed for storing events, states and actions

```
CREATE TABLE "access_control_input"."request_d_access" (
    "_o_id" BIGINT NOT NULL DEFAULT NEXT VALUE FOR "
       access_control_meta"."tuple_id_seq",
    "person" CLOB NOT NULL
);
```

⋮

```
CREATE TABLE "access_control_working"."request_d_access" (
    "id" BIGINT NOT NULL DEFAULT NEXT VALUE FOR "
       access_control_meta"."tuple_id_seq",
    "person" CLOB NOT NULL,
    "reception_d_time_o_begin" BIGINT,
    "reception_d_time_o_end" BIGINT
);
```

⋮

Register the tables/buffers in the meta data

```
INSERT INTO "access_control_meta"."buffers"
   ( "prog_package", "simple_name", "min_times_number", "type", "
      table" , "min_time_name_1")
VALUES
   ( '', 'request-access', 1, 'INPUT', 'request_d_access', '.id')
;
```

⋮

```
INSERT INTO "access_control_meta"."buffers"
   ( "prog_package", "simple_name", "min_times_number", "type", "
      table" , "min_time_name_1")
VALUES
   ( '', 'request-access', 1, 'WORKING', 'request_d_access', 'id'
      )
;
```

⋮

Prepare queries in the meta data

⋮

```
INSERT INTO "access_control_meta"."queries"
   ( "prog_package", "type", "number_of_output_min_times", "
      number_of_input_min_times", "inputs", "output", "
      sql_eval_statement", "referenced_clocks", "
      min_times_computation" , "output_min_time_name_1", "
      output_min_time_value_1", "input_min_time_name_1", "
      input_min_time_value_1", "input_min_time_name_2", "
      input_min_time_value_2")
VALUES
   ( '', 'INPUT', 1, 1, '<list>
␣␣␣<element␣class="java.lang.String">request-access</element>
</list>', 'request-access',
'
...SQL-STATEMENT...
', '<list>
␣␣␣<element␣class="java.lang.String">system</element>
</list>',
'<time_function>
␣␣␣...
</time_function>', 'id', -9223372036854775807, 'request-access:.
   id', -9223372036854775807, NULL, NULL)
;
```

⋮

```
INSERT INTO "access_control_meta"."queries"
   ( "prog_package", "type", "number_of_output_min_times", "
      number_of_input_min_times", "inputs", "output", "
      sql_eval_statement", "referenced_clocks", "
      min_times_computation" , "output_min_time_name_1", "
      output_min_time_value_1", "input_min_time_name_1", "
      input_min_time_value_1", "input_min_time_name_2", "
      input_min_time_value_2")
VALUES
   ( '', 'WORKING', 1, 2, '<list>
```

```
␣␣␣<element␣class="java.lang.String">request-access</element>
␣␣␣<element␣class="java.lang.String">staff</element>
</list>', 'grant-access',
'
...SQL-STATEMENT...
', '<list>
␣␣␣<element␣class="java.lang.String">tuple_id_seq</element>
</list>', '
<time_function>
␣␣␣...
</time_function>', 'id', -9223372036854775807, 'request-access:id
   ', -9223372036854775807, 'staff:id', -9223372036854775807)
;
```

⋮

7.5. Run

Command: `run example.access-control:test`

Effect: Initializes the Java objects needed for controlling the execution according to the meta data of the program instance stored in the database.

SQL-Statements: Only reading from the database.

7.6. Init Execution

Command: `init`

Effect: Moves the initial static and stateful data from the input buffers/tables to the working buffer/tables and sets the state of the program instance first to initialized. After that the program is locked for the execution.

Note: Remembering the state is particularly important when resuming the execution after an intentional or exceptional stop of the execution.

SQL-Statements:

Moving the initial static and stateful data

```
SELECT get_value_for('access_control_meta','tuple_id_seq');

INSERT INTO "access_control_working"."staff"( "id", "person", "
   valid_d_time_o_begin", "valid_d_time_o_end" )
SELECT
   "_o_id" AS "id",
```

```
   "person" AS "person",
   CAST(/*clock.now:system*/1314712426585 AS BIGINT) AS "
      valid_d_time_o_begin",
   "valid_d_time_o_end" AS "valid_d_time_o_end"
FROM
   "access_control_input"."staff" AS "input0_o_staff"
;

UPDATE "access_control_meta"."queries"
SET   "output_min_time_value_1" = -1,
   "input_min_time_value_1" = -1
WHERE "id" = '1'
;
```

Updating the state of the program instance

```
UPDATE
   "meta"."program_insts"
SET "state" = 'INITIALIZED'
WHERE
   "id" = '1'
;

UPDATE
   "meta"."program_insts"
SET "state" = 'LOCKED'
WHERE
   "id" = '1'
;
```

7.7. Start Execution

Command: `start`

Effect: Starts the evaluation of the program instance and sets the state of the instance to running. The evaluation is done by repeatedly executing incremental SQL statements. Some of the statements reflect the rules in the Dura program (e.g. 3,4), some read the input (e.g. 2) and some write the output and the logs (e.g. 5,6,7,8).

SQL-Statements:

Always consists of pair of statements one computing the increment and one updating the meta data of the query needed for the incremental evaluation

First Round:

```
UPDATE
   "meta"."program_insts"
```

```
SET "state" = 'RUNNING'
WHERE
   "id" = '1'
;
```

Executing the incremental evaluation statement for query 2 (Query one was the query for moving the initial data at `init`)

```
SELECT get_value_for('access_control_meta','tuple_id_seq');

INSERT INTO "access_control_working"."request_d_access"( "id", "
   person", "reception_d_time_o_begin", "reception_d_time_o_end" )
SELECT *
FROM
   (
   SELECT
      "_o_id" AS "id",
      "person" AS "person",
      CAST(/*clock.now:system*/1314712583949 AS BIGINT) AS "
         reception_d_time_o_begin",
      CAST(/*clock.now:system*/1314712583949 AS BIGINT) AS "
         reception_d_time_o_end"
   FROM
      "access_control_input"."request_d_access" AS "
         input0_o_request_d_access"

   ) AS "subquery0"
WHERE /*input.last:request-access:.id*/-9223372036854775807 < "id
   "
  AND "id" <= /*input.now:request-access:.id*/-1
;
```

Updating the meta data for query 2

```
UPDATE "access_control_meta"."queries"
SET   "output_min_time_value_1" = -1,
   "input_min_time_value_1" = -1
WHERE "id" = '2'
;
```

Executing the incremental evaluation statement for query 3

```
SELECT get_value_for('access_control_meta','tuple_id_seq');

INSERT INTO "access_control_working"."grant_d_access"( "id", "
   person", "reception_d_time_o_begin", "reception_d_time_o_end" )
SELECT
   "id" AS "id",
```

```
   "person" AS "person",
   "reception_d_time_o_begin" AS "reception_d_time_o_begin",
   "reception_d_time_o_end" AS "reception_d_time_o_end"
FROM
   (
   SELECT
      "e_o_id" AS "e_o_id",
      "e_o_person" AS "e_o_person",
      "e_o_reception_d_time_o_begin" AS "
         e_o_reception_d_time_o_begin",
      "e_o_reception_d_time_o_end" AS "e_o_reception_d_time_o_end
         ",
      NEXT VALUE FOR "access_control_meta"."tuple_id_seq" AS "id"
         ,
      "e_o_person" AS "person",
      LEAST( "e_o_reception_d_time_o_begin" ) AS "
         reception_d_time_o_begin",
      GREATEST( "e_o_reception_d_time_o_end" ) AS "
         reception_d_time_o_end",
      "s_o_id" AS "s_o_id",
      "s_o_person" AS "s_o_person",
      "s_o_valid_d_time_o_begin" AS "s_o_valid_d_time_o_begin",
      "s_o_valid_d_time_o_end" AS "s_o_valid_d_time_o_end"
   FROM
      (
      SELECT
         "id" AS "e_o_id",
         "person" AS "e_o_person",
         "reception_d_time_o_begin" AS "
            e_o_reception_d_time_o_begin",
         "reception_d_time_o_end" AS "e_o_reception_d_time_o_end"
      FROM
         "access_control_working"."request_d_access" AS "
            input0_o_request_d_access"

      ) AS "subquery1",
      (
      SELECT *
      FROM
         (
         SELECT
            "id" AS "s_o_id",
            "person" AS "s_o_person",
            "valid_d_time_o_begin" AS "s_o_valid_d_time_o_begin",
            "valid_d_time_o_end" AS "s_o_valid_d_time_o_end"
```

```
         FROM
            "access_control_working"."staff" AS "input1_o_staff"

         ) AS "subquery3"

      ) AS "subquery2"
   WHERE "e_o_person" = "s_o_person"

   ) AS "subquery0"
WHERE
   (   /*input.last:request-access:id*/-9223372036854775807 < "
      e_o_id"
   OR /*input.last:staff:id*/-9223372036854775807 < "s_o_id"
   )
  AND "e_o_id" <= /*input.now:request-access:id*/-1
  AND "s_o_id" <= /*input.now:staff:id*/-1
;
```

Updating the meta data for query 3

```
SELECT get_value_for('access_control_meta','tuple_id_seq');

UPDATE "access_control_meta"."queries"
SET   "output_min_time_value_1" = -1,
   "input_min_time_value_1" = -1,
   "input_min_time_value_2" = -1
WHERE "id" = '3'
;
```

Executing the incremental evaluation statement for query 4

```
SELECT get_value_for('access_control_meta','tuple_id_seq');

INSERT INTO "access_control_working"."intrusion_d_warning"( "id",
    "person", "reception_d_time_o_begin", "reception_d_time_o_end"
    )
SELECT
   "id" AS "id",
   "person" AS "person",
   "reception_d_time_o_begin" AS "reception_d_time_o_begin",
   "reception_d_time_o_end" AS "reception_d_time_o_end"
FROM
   (
   SELECT
      "e_o_id" AS "e_o_id",
      "e_o_person" AS "e_o_person",
      "e_o_reception_d_time_o_begin" AS "
         e_o_reception_d_time_o_begin",
```

```
    "e_o_reception_d_time_o_end" AS "e_o_reception_d_time_o_end
       ",
    "f_o_id" AS "f_o_id",
    "f_o_person" AS "f_o_person",
    "f_o_reception_d_time_o_begin" AS "
       f_o_reception_d_time_o_begin",
    "f_o_reception_d_time_o_end" AS "f_o_reception_d_time_o_end
       ",
    NEXT VALUE FOR "access_control_meta"."tuple_id_seq" AS "id"
       ,
    "e_o_person" AS "person",
    LEAST( "e_o_reception_d_time_o_begin", "
       f_o_reception_d_time_o_begin" ) AS "
       reception_d_time_o_begin",
    GREATEST( "e_o_reception_d_time_o_end", "
       f_o_reception_d_time_o_end" ) AS "reception_d_time_o_end"
FROM
    (
    SELECT
       "id" AS "e_o_id",
       "person" AS "e_o_person",
       "reception_d_time_o_begin" AS "
          e_o_reception_d_time_o_begin",
       "reception_d_time_o_end" AS "e_o_reception_d_time_o_end"
    FROM
       "access_control_working"."grant_d_access" AS "
          input0_o_grant_d_access"

    ) AS "subquery1",
    (
    SELECT *
    FROM
       (
       SELECT
          "id" AS "f_o_id",
          "person" AS "f_o_person",
          "reception_d_time_o_begin" AS "
             f_o_reception_d_time_o_begin",
          "reception_d_time_o_end" AS "
             f_o_reception_d_time_o_end"
       FROM
          "access_control_working"."request_d_access" AS "
             input1_o_request_d_access"

       ) AS "subquery3"
```

```
      ) AS "subquery2"
   WHERE "e_o_person" = "f_o_person"
     AND ( GREATEST( "e_o_reception_d_time_o_end", "
        f_o_reception_d_time_o_end" ) - LEAST( "
        e_o_reception_d_time_o_begin", "
        f_o_reception_d_time_o_begin" ) ) < '30000'
     AND "e_o_reception_d_time_o_end" < "
        f_o_reception_d_time_o_begin"

   ) AS "subquery0"
WHERE
   (   /*input.last:grant-access:id*/-9223372036854775807 < "
      e_o_id"
   OR /*input.last:request-access:id*/-9223372036854775807 < "
      f_o_id"
   )
  AND "e_o_id" <= /*input.now:grant-access:id*/-1
  AND "f_o_id" <= /*input.now:request-access:id*/-1
;
```

Updating the meta data for query 4

```
SELECT get_value_for('access_control_meta','tuple_id_seq');

UPDATE "access_control_meta"."queries"
SET   "output_min_time_value_1" = -1,
   "input_min_time_value_1" = -1,
   "input_min_time_value_2" = -1
WHERE "id" = '4'
;
```

Executing the incremental evaluation statement for query 5

```
INSERT INTO "access_control_output"."grant_d_access"( "_o_id", "
   person", "reception_d_time_o_begin", "reception_d_time_o_end" )
SELECT *
FROM
   (
   SELECT
      "id" AS "_o_id",
      "person" AS "person",
      "reception_d_time_o_begin" AS "reception_d_time_o_begin",
      "reception_d_time_o_end" AS "reception_d_time_o_end"
   FROM
      "access_control_working"."grant_d_access" AS "
         input0_o_grant_d_access"
```

```
   ) AS "subquery0"
WHERE /*input.last:grant-access:id*/-9223372036854775807 < "_o_id
   "
  AND "_o_id" <= /*input.now:grant-access:id*/-1
;
```

Updating the meta data for query 5

```
UPDATE "access_control_meta"."queries"
SET   "output_min_time_value_1" = -1,
   "input_min_time_value_1" = -1
WHERE "id" = '5'
;
```

Executing the incremental evaluation statement for query 6

```
INSERT INTO "access_control_output"."intrusion_d_warning"( "_o_id
   ", "person", "reception_d_time_o_begin", "
   reception_d_time_o_end" )
SELECT *
FROM
   (
   SELECT
      "id" AS "_o_id",
      "person" AS "person",
      "reception_d_time_o_begin" AS "reception_d_time_o_begin",
      "reception_d_time_o_end" AS "reception_d_time_o_end"
   FROM
      "access_control_working"."intrusion_d_warning" AS "
         input0_o_intrusion_d_warning"

   ) AS "subquery0"
WHERE /*input.last:intrusion-warning:id*/-9223372036854775807 < "
   _o_id"
  AND "_o_id" <= /*input.now:intrusion-warning:id*/-1
;
```

Updating the meta data for query 6

```
UPDATE "access_control_meta"."queries"
SET   "output_min_time_value_1" = -1,
   "input_min_time_value_1" = -1
WHERE "id" = '6'
;
```

Executing the incremental evaluation statement for query 7

```
INSERT INTO "access_control_log"."grant_d_access"( "_o_id", "
   person", "reception_d_time_o_begin", "reception_d_time_o_end" )
SELECT *
FROM
   (
   SELECT
      "id" AS "_o_id",
      "person" AS "person",
      "reception_d_time_o_begin" AS "reception_d_time_o_begin",
      "reception_d_time_o_end" AS "reception_d_time_o_end"
   FROM
      "access_control_working"."grant_d_access" AS "
         input0_o_grant_d_access"

   ) AS "subquery0"
WHERE /*input.last:grant-access:id*/-9223372036854775807 < "_o_id
   "
  AND "_o_id" <= /*input.now:grant-access:id*/-1
;
```

Updating the meta data for query 7

```
UPDATE "access_control_meta"."queries"
SET   "output_min_time_value_1" = -1,
   "input_min_time_value_1" = -1
WHERE "id" = '7'
;
```

Executing the incremental evaluation statement for query 8

```
INSERT INTO "access_control_log"."intrusion_d_warning"( "_o_id",
   "person", "reception_d_time_o_begin", "reception_d_time_o_end"
   )
SELECT *
FROM
   (
   SELECT
      "id" AS "_o_id",
      "person" AS "person",
      "reception_d_time_o_begin" AS "reception_d_time_o_begin",
      "reception_d_time_o_end" AS "reception_d_time_o_end"
   FROM
      "access_control_working"."intrusion_d_warning" AS "
         input0_o_intrusion_d_warning"

   ) AS "subquery0"
WHERE /*input.last:intrusion-warning:id*/-9223372036854775807 < "
```

```
   _o_id"
  AND "_o_id" <= /*input.now:intrusion-warning:id*/-1
;
```

Updating the meta data for query 8

```
UPDATE "access_control_meta"."queries"
SET   "output_min_time_value_1" = -1,
   "input_min_time_value_1" = -1
WHERE "id" = '8'
;
```

7.8. Stop/Pause Execution

Command: `stop`

Effect: Stops the evaluation of the program instance and sets the state of the instance back to locked.

SQL-Statements:

```
UPDATE
   "meta"."program_insts"
SET "state" = 'LOCKED'
WHERE
   "id" = '1'
;
```

7.9. Quit Program

Command: `quit`

Effect: Exits the evaluation of the program instance sets the state of the instance back to initialized and frees all resources and releases the Java objects used for controlling the execution.

SQL-Statements:

```
UPDATE
   "meta"."program_insts"
SET "state" = 'INITIALIZED'
WHERE
   "id" = '1'
;
```

Part II.
An Incremental Approach for Compiling Dura Queries

Dura[2] is a high level, declarative and uniform reactive event query language. It is designed to ease the development and implementation of versatile and advanced emergency management related scenarios.

Dura has been carefully designed to meet the requirements of the use cases and to enable an efficient and robust implementation of programs. The language combines event queries, queries of stateful objects and the specification of complex actions in a homogeneous and integrated fashion. These concepts are highly desirable for modern emergency management systems in order to obtain a reactive and dynamic behavior. Moreover, Dura comes with various high-level language constructs, as for instance composite and time aware actions, which are intended to support programmers by simplifying the development of rules, but do not increase the expressiveness of the language itself. However, a language that combines a wide range of high-level constructs and integrates reactive rules and declarative event queries in a uniform and time aware fashion is rather unique in the field of complex event processing. Hence, it enables new approaches for modern and more reliable emergency management systems.

For the compilation of Dura queries we decided to use an incremental compilation approach. Dura queries are not directly compiled to an algebra for temporal streams named temporal stream algebra (TSA) that can be evaluated by Event-Mill, which would yield a very complex and monolytic compiler that is hard to program, understand and to maintain. Instead, Dura queries are translated to a simplified variant of Dura called Dura_C (pronounced “Dura Core”) in a first step and then translated from Dura_C to TSA in a second step. The simplified language Dura_C contains all mandatory constructs to remain as expressive as Dura whereas syntactic sugar is omitted. However, transforming Dura to Dura_C and subsequently compiling Dura_C to TSA is substantially easier than compiling Dura directly into TSA. A complete description of Dura_C is given in section 9.

8. A Schema for Events, Stateful Objects and Actions

Having a notion of schema that describes the properties and characteristics of available objects is very common in high level programming languages. There are several reasons which makes a schema desirable for emergency management applications.

First of all, a schema defines which kinds of (atomic) events are needed for a program and how

[2]The language has been formerly known as DEAL. Due to name conflicts with other event processing languages it has been renamed to Dura.

they look like, that is, what data they carry, which type the data has, etc. Therefore, rules that are querying unavailable event types or attributes of events that do not exist and thus cannot derive any events can be detected during compile time.

Second of all, a schema simplifies the development of large programs. It describes the characteristics of derived events. As a consequence, even if rules that are deriving a particular event are modified, the described characteristics of the event in the schema remain the same. Therefore, the schema can be regarded as some kind of interface mechanism. Modification of a rule have only local effect on the program as long as the schema remains the same. The schema only defines which kinds of events are available and how they look like but it does not specify how events are derived. Consequently, the information given in the schema provides a good and concise overlook of available events without going into the detail of how the events are actually derived. Furthermore, schemas can be easily extended towards a sophisticated module system.

Third, the availability of a schema simplifies the handling of rules not only for humans but also for the compiler. Especially if modules are considered, it is important for the programmer as well as for the compiler to be able to quickly determine where rules that derive certain events are located and how the derived events look like. Because a module mechanism for Dura is due by the next round of deliverables, these requirements are already considered in order to enable a smooth transition to programs with modules and to reduce potentially required adaption of rules.

8.1. A Schema for Dura

In Dura, the schema describes the payload of events, stateful objects, and actions including the type of their attributes without giving concrete rules or definitions of how these objects are actually derived or executed.

A schema is given by a semi-structured expressions that specifies the name and type of all available attributes. These semi-structured expressions are tree-like structures with certain restriction that enable an efficient mapping of events to tuples of a database.

Listing 3: A sample schema for `temp` events

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

The purpose of using semi-structured expressions is to give programmers a mean to structure, for instance, the payload of events in a manner similar to composite data type structures (structs for short) which are known from other programming languages, like `C` or `C++`. Therefore, the schema definitions of Dura are a compromise between flat tuples that bear no structure at all

but can be implemented very efficiently and generic semi-structured expressions which have an arbitrary structure but cannot be easily represented in a relational database.

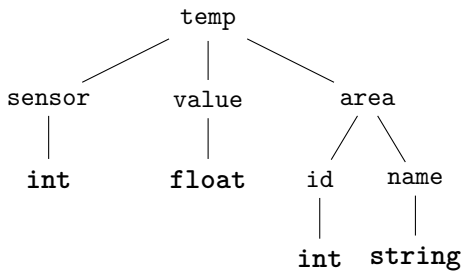

Figure 1: Tree representation of the schema given in listing 3

Figure 1 contains the tree representation of the schema given in listing 3. In order to be a valid schema definition, the following constraints need to hold for the corresponding tree-like structure:

- The leaf nodes of the tree need to contain types and must not have any siblings,
- for all inner nodes it must hold that all their siblings contain different labels, and
- the tree-like structure must be a proper tree without references, that is, definitions must not be cyclic.

This characterisation ensures that the path to each leaf, which can be described by the labels of the nodes on this path, is unique. Accordingly, the schema given in listing 3 is a valid schema definition, whereas `temp{ sensor{int, int} }` and `temp{ value{float}, value{int} }` are not.

Because each leaf node of the schema has a unique id and only leaf nodes are actually carrying data, the schema or respectively data instances of the schema can be efficiently mapped to flat tuples in a database by storing only the leaf nodes and referencing them with their path.

Grammar for Schema Definitions Please note that the given grammars in this text are only used to to illustrate the most important aspects of the language and are thus not intended to be complete. The complete grammar is contained in appendix A.

```
schemaDefinition ::= label '{' basicType '}'
                   | label '{' compositeType '}'
                   | label '{' schemaDefinition (',' schemaDefinition)* '}'

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

```
compositeType ::= ID

label ::= ID
```

8.2. Types and Type Definitions

For the definition of schemas and subsequently the mapping of values to tuples, types are required that indicate which kind of values are expected for certain attributes of the data. To this end, a simple type system is introduced to the language. Note that, although types which can be defined by the user are supported, they are only a mean to give names to frequently used composite types. A more powerful type system can be elaborated in the future, but this is rather a computer science than an emergency management issue.

Dura comes with the following (basic) types: `boolean`, `int`, `long`, `float`, `double`, `string`, `identifier`, `duration`, and `timestamp`. Further types can be defined using the `TYPE <name> IS <type definition> END` statement. However, each type definition needs to have a unique name and although type definitions may refer to (other) user defined types, there must not be any cyclic type definitions.

Listing 4: Type definition for type `area` and `room-location`

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

Note that, because the applications we consider in the use cases do not require type specific functions and overloading of operators a sophisticated type system is not required in the language. Therefore, user defined types should be considered as a mean to give names to commonly used composite type definitions, like for instance the type `area` which is defined in listing 4 and which is used in numerous event definitions.

The type definition in listing 4 ties multiple attributes together without naming their parent node explicitly. Hence, the defined type can only be stated in leaf nodes of a schema or type definition.

Listing 5: An similar definition of `room-location`

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

A, from a low level perspective, equivalent definition of the type `room-location` is given in listing 5. In the alternative definition, the user defined types are substituted such that only basic types remain in the definition of `room-location`. Therefore, the composite type `area` is substituted by its definition `id{ int }, name{ string }`. However, with this kind of definition,

fewer type checks can be carried out and thus the definition of listing 4 should be chosen over the latter.

Adding Constraints to Types Constraints can be added to types which is particularly useful to specify temporal relationships between different attributes. The given information of temporal dependencies may either be required for the evaluation of queries or can support the engine in deriving certain events earlier.

Constraints are for instance used to define the type `time-interval` which is given in listing 6. The constraint specifies that the timestamp of the attribute begin needs to be lower or at most equal to the timestamp of the attribute end.

Listing 6: Definition of the build-in type `time-interval`

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

The same mechanism is not only used for (data) type definitions, but also for the definition of events and stateful object which is discussed in section 8.4.

Grammar for Type Definitions

```
typeDefinition ::= 'TYPE' ID 'IS' basicType 'END'
                 | 'TYPE' ID 'IS' schemaDefinition (',' schemaDefinition)*
                     typeSupplement? 'END'

typeSupplement ::= 'where' '{' pathFormula (',' pathFormula)* '}'

pathFormula ::= path arithmeticRelation path
```

8.3. Constant Definitions

Constants can be defined in a manner similar to that of 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. This increases the readability of the code, because so-called magic numbers are substituted by more meaningful identifiers. Furthermore, the code becomes easier to maintain, because if the value needs to be adapted, the value of the constant definition needs to be modified only once whereas otherwise the value needs to modified throughout the whole code.

Listing 7 contains the constant definition `max-temp` for a temperature threshold. Wherever a number can be specified in a rule, the name of the constant preceded by the keyword `const` can be specified instead.

Listing 7: Definition of the constants `max-temp` and `office-location`

```
CONST max-temp IS 42.5 END
CONST office-location IS
  room{ roon-number{7}, location{ id{23}, name{"E4"} } } END
```

Definition of constants are not limited to basic types as the definition of `office-location` demonstrates. However, defining constants for basic types, that is, giving names to instances of a basic type, seems to be way more important for the use cases than the definition of constants for user defined types. Note that `office-location` is an instance of the type `room-location` which is defined in listing 4.

Grammar for Constant Definitions

```
constDefinition ::= 'CONST' ID 'IS' constTerm 'END'
```

8.4. Event Definitions

Event definitions bracket the schema of events and rules that derive the corresponding event type together. Event definitions are similar to the definition of classes in object oriented languages such as Java. In a Java class definition, class variables define the properties of the class whereas functions specify how the specified (and only the specified) class variables are altered. In Dura, the schema defines the properties of the event without specifying how events are derived, whereas the rules define how events are derived and how the concrete values for their properties are computed.

All rules that are deriving the same event need to be grouped together in the same event definition. The definition includes not only rules but also a schema that specifies the properties of the derived event. For all rules that are contained in an event definition, the derived events need to be in accordance with the given schema of the definition. Furthermore, for every event type that is used in the program there must be exactly one event definition.

Grouping rules according to a schema, that is, by their derived event type, has several advantages. Event definitions can thus be regarded as some kind of interface mechanism. As long as the schema is not modified, rules can be added and removed from the definition without changing the properties of the event. Without an explicit schema, the properties of an event need to be automatically reconstructed by the compiler. Hence, adding a single rule to a program can substantially alter the properties of an event and thus break accidentally other rules that are located in a totally different part of the program. Therefore, an explicit schema brings stability and continuity to the program because as long as the schema remains untouched the impact of local modifications of rules remains local to the corresponding event definition.

Furthermore, grouping similar rules makes large programs clearer and easier to understand. One can quickly get an overview of how a certain event type is derived by looking at a small and

Listing 8: A complex event definition including a schema and declarative rules

```
EVENT
  temp{
    sensor{int},
    value{float}
  }
WITH
  DETECT
    temp{ sensor{var Id}, value{var C} }
  ON
    measurement{ temperature{var C},
                 sensor{var Id}, unit{"centigrade"} }
  END

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

  ...
END
```

particularly coherent part of the program. Moreover, because rules are clustered in a small region of the program instead of being scattered throughout the whole program, rules that derives a certain event are easier to find and to compare.

In addition, we consider the concept of event definition beneficial when modules are included in the language which is due to deliverable D4.6. When modules are introduced, the potential problems that are described above become even more severe, because without a schema, rules are not only scattered within a single file but also among several files.

Modifiers for Event Definitions Event definitions are also used to define external and atomic events, respectively. The `WITH` part of an event definition is optional and can be omitted. Therefore, the only given information about an event are its properties and hence the event cannot be derived by the event processing system itself.

There are three different modifiers that specify further properties of event definitions, namely `input`, `output`, and `log`. These modifiers correspond to the schema types of Event-Mill that are discussed in section 4.6. Each event definition can be preceded by multiple modifiers, for instance to specify that an external event should be stored in the long-term archive.

External events that are sent to the event processing system, for instance by a sensor, need to be marked with the `input` keyword. (Derived) events that should be sent to components which are outside the event processing system need to be marked with `output`. Finally, `log` results in the

storage of events in a long-term archive for later analysis.[3]

Listing 9: An atomic event specification

```
input log
EVENT
  measurement{
    sensor{int},
    temperature{float},
    unit{string}
  }
END
```

The information about external events is used during compile time to identify rules which query events that either do not exist or do not correspond the specified schema. In both cases unintended programming flaws in the rules are likely because the body of the rules can never be satisfied and thus according rules will never derive any event and is thus redundant.

Especially in emergency management applications robust and error tolerant programs are highly desirable and thus the described analysis during compile time is a auxiliary mean that supports the programmer during the development of rules in writing less error prone rules and programs.

Implicit Attributes of Events Each event has some implicit attributes for the reception time and the event reference in addition to its explicitly specified attributes. The implicit attributes of the event definition given in listing 9 are made explicit in listing 10.

Listing 10: Event specification including the implicit attributes

```
input log
EVENT
  measurement{
    id{identifier},
    reception-time{time-interval},
    sensor{int},
    temperature{float},
    unit{string}
  }
END
```

Note that although this is a valid event definition, implicit attributes do not need to be specified by programmers. Moreover, the reception time and the event identifier are implicitly set by the event processing system for every event, regardless whether it is received from an external source or derived by a rule.

[3]Recall that events, unless they are marked with the `log` keyword, are automatically deleted when they are no longer needed by the event processing system.

Static Type Checks Listing 11 contains a Dura program that looks reasonable in the first place, but reveals some flaws when it is examined more closely.

Listing 11: A (statically) incorrect Dura program

```
TYPE area IS { id{int}, name{string} } END

EVENT
  temp-average{ area{area}, value{float} }
WITH
  DETECT
    temp-average{ area{var A},
                  value{avg(var T)} } group by {event i}
  ON
    and{
      event i: temp{ area{var A} },
      event j: temp{ area{var A}, valeu{var T} }
    } where { {i,j} within 2 min, j before i}
  END
END

EVENT
  temp{ sensor{int}, area{int}, value{float} }
END
```

First, line 12 contains a query for `temp` event. However, the attribute of the event is not spelled correctly. Therefore, the query will never match any `temp` event, because none of them has a `valeu` attribute.

In addition, `var A` is bound to an integer value (line 11 and 12), namely the id if the sensor's area, and therefore the `area` attribute of the derived event is bound to the id of the area as well. However, in the schema of the event definition the `area` attribute has the type `area`. Therefore, `temp-average` events do not have an `id` and a `name` attribute, and every rule that queries those attributes will never match any `temp-average` event.

Both errors occurred very likely accidentally and were not intended by the programmer in the first place, however they render all rules that depend on `temp-average` events useless. With a statical analysis of the rules, both errors can be detected and reported to the programmer during compile time, that is, before the program is actually executed and thus helps to write less error prone and more robust programs.

Grammar for Event Definitions

```
eventDefinition ::= modifiers 'EVENT' schemaDefinition typeSupplement?
                    ('WITH' eventSpecification* )? 'END'

modifiers ::= modifier*
```

```
modifier ::= 'input' | 'output' | 'log'
```

8.5. Introducing More Versatile Subqueries to Dura

Having considered the first actual queries of the use cases, we recognized that allowing aggregation and mathematical expressions only in the head of a Dura rule can be cumbersome and too restrictive. Especially if subqueries are considered, that is, compound queries without a head that are used in the body of rules, it is more convenient to be able to specify aggregation also in the body of an event query.

To this end, two new constructs are added to Dura which can be specified, similar to the already existing `where` construct, after atomic queries, conjunctions, and disjunctions. `let` is used to create new variable bindings based on the value of other variables and `group by` /`aggregate` is used to aggregate values.

After each atomic query, conjunction, and disjunction an arbitrary number of `let`, `where` and `group by`/`aggregate` constructs can be specified. Listing 12 contains an example that uses a subquery to determine the id of all temperature sensors that reported a temperature in the same area of the alarm's origin that is 50% above the mean temperature of the last two minutes.

Listing 12: A Dura rule with an anonymous subquery

```
DETECT
  mean-temp-exceeded{ id{var Id}, value{var Temp} }
ON
  and{
    event e: temp{ area{var A}, id{var Id}, value{var Temp} },
    event f: and{
      event g: alarm{ area{var A} },
      event h: temp{ area{var A}, value{var AggTemp} }
    } where { {g,h} within 2 min, h before g }
      group by {event g} aggregate { var Mean = mean(var AggTemp) }
  } where { e during f, var Temp >= var Mean*1.5 }
END
```

A similar behaviour can be realized by two separate rules that contain no subqueries. The usage of anonymous subqueries keeps single rules shorter and clearer whereas explicit rules without anonymous subqueries increase the reusability of code among several rules. Depending on the concrete situation one or another way may be more suitable.

8.6. Action and Stateful Object Definition

External actions and stateful objects are defined in a manner similar to the definition of (external) events. The properties of available objects and external actions are specified by means of a

stateful object/action definition without specifying how the stateful object is derived from other stateful objects or how actions are actually executed, respectively.

Note that the modifiers of actions and stateful object are limited to `log`. In contrast to event definitions they cannot be preceded by `input` or `output`.

Listing 13: Definition of Stateful Objects and (External) Actions

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

ACTION
  open-dampers{ area{string} }
END

ACTION
  turn-on-lights{ area{string} }
END
```

Build-In Events and Actions Dura already comes with some atomic actions which deal with the manipulation of stateful objects. These internal actions are implicitly defined by the definition of stateful objects and hence do not need to be defined manually by the programmer. The same applies for events that are automatically derived either as a result of the execution of actions or as a consequence of manipulating of stateful objects.

More precisely, actions and events which do not need to be defined by the programmer using action or event definitions include `create/delete/update-object` actions, `object-created /deleted/updated` events, and `action-initialized/succeeded/failed` events.[4]

Grammar for External Action and Stateful Object Definition

```
stateDefinition ::= 'log'? 'STATEFUL OBJECT' schemaDefinition
                      ('WITH' stateSpecification*)? 'END'

actionDefinition ::= 'log'? 'ACTION' schemaDefinition
                       ('WITH' actionSpecification*)? 'END'
```

9. Dura$_C$

Dura$_C$ (pronounced "Dura Core") is a simplified subset of the Dura language. Dura$_C$ contains all mandatory constructs of Dura which are needed to gain a high expressiveness, whereas syntactic

[4]Note that external actions are not executed by the event processing system itself but rather by external actuators. Therefore external actions are only initiated by the event processing system.

sugar is omitted. As a result, both languages are equally expressive, but the compilation of Dura$_C$ is far easier because only less complicated and fewer constructs need to be translated.

For the compilation of Dura programs, Dura is translated to Dura$_C$ in a first step and subsequently compiled to the temporal stream algebra (TSA) in a second step. TSA is finally evaluated by the event processing system on top of a relational database.

The stepwise compilation of Dura has several advantages over a direct compilation of Dura to TSA. Because the basic constructs that are supported by Dura$_C$ and TSA are quite similar, the complexity of compiling Dura$_C$ to TSA is moderate and therefore the the implementation of the compiler is straight forward. Although translating Dura to Dura$_C$ is more complicated, the transformation is still easier than directly compiling Dura to TSA. Therefore, a stepwise respectively incremental compiler will be easier to develop and to maintain and thus the whole compilation process will be less error prone and more robust. Furthermore, the different compilation steps can be approached in parallel which leads to a faster results and further functionality can be subsequently added if required.

9.1. Event Queries

One major difference between event queries of Dura and Dura$_C$ is the nesting of queries. Dura allows (almost) arbitrarily nested anonymous queries whereas Dura$_C$ supports only flat conjunctions and disjunctions in combination with negated queries.

Furthermore, the query head of Dura$_C$ queries may not contain computation of new values. If desired such computations need to be specified in the `let` part of the query body. However, in contrast to Dura, variables that are introduced in a `let` or `aggregate` part of a query need to explicitly specify their type.

Listing 14: A sample Dura$_C$ event query

```
DETECT
  temp-stats{ area{var A}, avg{var MinMaxAvg} }
ON
  and{
    event e: alarm{ area{var A} },
    event f: temp{ area{var A}, value{var Temp} }
  } where { {event e, event f} within 2 min,
            event f before event e}
    group by {event e} aggregate { var float Max = max(var Temp),
                                   var float Min = min(var Temp) }
    let { var float MinMaxAvg = (var Max + var Min)/2 }
END
```

Grammar for Dura$_C$ Event Queries

```
eventSpecification ::= 'DETECT' term 'ON' query 'END'

query ::= 'and' '{' flatQuery (',' flatQuery)* '}' querySupplement?
        | 'or' '{' flatQuery (',' flatQuery)* '}' querySupplement?
        | flatQuery (',' flatQuery)*

flatQuery ::= 'not' '{' atomicQuery '}' querySupplement
            | 'not' atomicQuery
            | atomicQuery

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

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

9.2. Range Restriction of Rules

All rules of a program need to be range restricted which means that variables which occur either in the query head or in the supplement of a query, that is, a combination of `where`, `let`, and `group by` /`aggregate` constructs, need to be positively bound.

For now, lets assume that the body of rules consists only of con/disjunctions of queries and `where` constructs. Then, every variable and reference of an event, stateful object or action that is used in either the head of the rule or in a `where` part of a con/disjunction needs to occur at least once in a positive query in case of a conjunction of queries and in every positive query in case of a disjunction of queries. Note that queries with a preceding `not` or `exists` are not considered as being positive queries. Furthermore, `where` constructs that are part of a negation can contain all variables of the negated query and positive variables and references of the conjunction they are contained in.[5]

Therefore, both rules given in listing 15 are *not* range restricted. In the first rule, `var T` is used in the `where` part in line 7, although the variable is only bound in one query (and not as requested in both queries) of the disjunction in the rule body. In the second rule, the variable `var Id` is used in the query head even though it is only bound in a negated query in the conjunction of the rule body.

Things get slightly more difficult if `let` and grouping is considered. If `let` occurs in the supplement of a query, the newly introduced variables can be used in every subsequent query supplement and in the head of the rule. However, if grouping occurs in the supplement of a query, then only variables and identifiers that occur either in the `group by` or variables that are introduced in the corresponding `aggregate` part can be used in subsequent query supplements and in the query head. In general one can say that `let` increases the amount of variables that can be used

[5] Using negated queries in combination with disjunctions makes no sense for continuous queries on unbounded event streams.

Listing 15: Examples of rules that are *not* range restricted

```
DETECT
  uncertain-alarm{ area{var A} }
ON
  or{
    event: temp{ area{var A}, value{var T} },
    event: smoke{ area{var A} }
  } where { var T >= const max-temp }
END

DETECT
  sensor-malfunction{ id{var Id} }
ON
  and{
    event e: uncertain-alarm{ area{var A} },
    not { event f: temp{ area{var A}, id{var Id} } }
      where { {e,f} within 1 min }
  }
END
```

in the head of a query whereas `aggregate` replaces variables by other variables that contain aggregated values.

Accordingly, the rule in listing 16 is not range restricted for two reasons. Although the variable `var Temp` which is used in the `where` part in line 9 occurs in a positive query in line 6, it does neither occur in the `group by` nor is it defined in the `aggregate` part of line 8. Likewise, `var Id` is used in the query head but it does not occur in the grouping of line 8. This conflict can be resolved by introducing a subquery which yields a query that is similar to the one of listing 12.

Listing 16: Another *non* range restricted rule

```
DETECT
  mean-temp-exceeded{ sensor-id{var Id} }
ON
  and{
    event e: alarm{ area{var A} },
    event f: temp{ area{var A}, value{var Temp} }
  } where { {e,f} within 2 min, f before e }
    group by {event e} aggregate {var float Mean = mean(var Temp)}
    where { var Temp >= var Mean*1.5 }
END
```

9.3. Reactive Rules

Reactive rules in Dura$_C$ are a restriction of reactive rules that are available in Dura. Only the concurrent execution of atomic actions is possible in Dura$_C$. Sophisticated constructs for the specification of complex actions, such as `FOR ... DO ...` and `IF ... THEN ...`, are only available in Dura.

Listing 17: Concurrent execution of multiple actions

```
ON
  event e: uncertain-alarm{ area{var A} }
DO
  concurrent{
    action a: open-dampers{ area{var A} },
    action b: turn-on-lights{ area{var A} }
  }
END
```

The reactive rule in listing 17 refers to the atomic actions that have been given in listing 13. Whenever an uncertain alarm occurs, both actions are initiated by the event processing system and subsequently concurrently executed by some external actuators.

Reactive rules make a transition from declarative rules without side effects to the actual execution of actions. They do not need to be clustered together like event, stateful object, and action definitions.

Grammar for Dura$_C$ Reactive Rules

```
reactiveRule ::= 'ON' query 'DO' action 'END'

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

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

9.4. Dura and Dura$_C$ compared

As already pointed out, Dura$_C$ contains only a subset of the constructs of Dura. However, both languages are equally expressive, that is, the language differ only in the amount of syntactic sugar. In the following, the differences of both languages are pointed out and discussed.

Rule Heads Rule heads in Dura$_C$ may only contain variables, constants, identifiers and literals, such as, strings, numbers, etc. Hence, arithmetic expressions and groupings cannot be specified in a rule head. They need to be specified in the supplement of the rule body instead.

Subqueries and Existential Quantification Subqueries are not allowed in the body of Dura$_C$ rules. They need to be made explicit by introducing further rules to the program. Nevertheless, negation can be used in (conjunction) of queries. As with subqueries, existentially quantified queries are not supported by Dura$_C$ but they can be realized by introducing further queries.

Typing of Variables In Dura$_C$, each variable that is defined in a `let` or `aggregate` part of a query needs to be explicitly typed.

Action Composition and Complex Actions In Dura$_C$, actions can only be executed concurrently. Further constraints, for instance on the execution order of actions or additional specifications which state when actions are considered to be executed successfully, cannot be be specified. Furthermore complex action specifications and the `IF` statement are not supported by Dura$_C$.

Constructs for Stateful Objects The two constructs `WHILE` and `DERIVE` which are intended to simplify the handling of stateful objects are not available in Dura$_C$.

Variables and Identifiers In Dura$_C$, identifiers, variables, and constants always need to be preceded by an according keyword. Moreover, variables can only be bound to basic types, such as `int` and `string`.

Note that the described restrictions of Dura$_C$ are only of a syntactical nature. Hence, Dura rules can always be rewritten to (semantically) equivalent Dura$_C$ rules. An automatic transformation of Dura to Dura$_C$ will be implemented by the Dura compiler.

Part III.
Appendix

A. Dura$_C$ EBNF Grammar

```
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 (',' 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 typeSupplement?
                      ('WITH' eventSpecification* )? 'END'

/** constraint: each modifier occurs at most once */
modifiers ::= modifier*

modifier ::= 'input' | 'output' | 'log'

/** constraint: stateful object definition schemaDefinition is unique */
stateDefinition ::= 'STATEFUL OBJECT' schemaDefinition typeSupplement? 'END'

/** constraint: action definition schemaDefinition is unique */
actionDefinition ::= 'ACTION' schemaDefinition 'END'
```

```
/** constraint: query is range restricted with respect to term */
eventSpecification ::= 'DETECT' term 'ON' query 'END'

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

/** constraint: at least one event query in every disjunct */
query ::= 'and' '{' flatQuery (',' flatQuery)* '}' querySupplement?
        | 'or' '{' flatQuery (',' flatQuery)* '}' querySupplement?
        | flatQuery (',' flatQuery)*

flatQuery ::= 'not' '{' atomicQuery '}' querySupplement
            | 'not' atomicQuery
            | atomicQuery

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

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

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

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

actionSupplement ::= where+

typeSupplement ::= 'where' '{' pathFormula (',' pathFormula)* '}'

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

unification ::= typedVariable '=' expr

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

binding ::= variable
          | identifier

aggregation ::= typedVariable '=' aggregationOp '(' variable ')'

where ::= '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

pathFormula ::= path arithmeticRelation path

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

/** constraint: variable is of type time-interval */
timeInterval ::= variable
               | identifier
               | relativeTimerOp '(' timeInterval ',' duration ')'

/** constraint: variable is of type timepoint */
timePoint ::= variable
            | 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 ::= variable
           | constant
           | identifier
           | STRING
           | NUMBER

/**
 * constraints:
 *   all constTerm labels are pairwise distinct
 *   no cyclic definitions
 */
constTerm ::= STRING
            | NUMBER
            | 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 ::= (atom) (('*' | '/') atom)*

/**
 * constraints:
 *   constants used in aritmetic expressions need to be a number
 */
atom ::= '(' mathExpr ')'
       | intervalOp '(' timeInterval ')'
       | variable
       | constant
       | (NUMBER) (timeUnit time*)?
```

```
       | STRING

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

label ::= aggregationOp
        | intervalOp
        | ID

type ::= basicType
       | compositeType

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

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

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

variable ::= 'var' ID

typedVariable ::= 'var' type 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')

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


/* TOKENS */
```

```
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')
```