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

Project acronym: *EMILI*

Project full title: Emergency Management in Large Infrastructures

Grant agreement no.: 242438

D4.7 Refinement of the Implementation of event processing and ECA Rules for SITE

Due date of deliverable: 30/06/2012
Actual submission date: 29/08/2012
Revision: Version 1.1

Ludwig-Maximilians University Munich (LMU)

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

Author(s)	Steffen Hausmann, Simon Brodt, 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. Dura is the high-level event, state and action language developed within WP4 of the EMILI project. Dura is tailored to the requirements of the three emergency management use-cases of WP3. Event-Mill is the evaluation engine for the Dura language. As Dura, Event-Mill is designed to meet the specific requirements of the WP3 use-cases.

Since the last report on the implementation contained in Deliverable D4.5 has been improved significantly. Most notably are the modularization mechanisms for Dura described in Deliverable D4.6., the "Temporal Stream Algebra" which serves as operational semantics for Dura and forms the basis of the evaluation and the Generic Bus API that was defined to simplify the integration with the SITE component of WP6 and the specific components of the WP3 use-cases. Furthermore the error analysis and the error reporting of the Dura compiler has been improved and two Eclipse plug-ins were adopted to support the development of Dura programs. Finally the integration of the Event-Mill engine with the MonetDB database of WP5 has been continued, the EVent-Mill engine has become more configurable and the capabilities of its command-line interface have been extended.

The report complements the implementation code and the documentation files which are available from `http://www.pms.ifi.lmu.de/cep/releases/`. The implementation code forms the main part of Deliverable D4.7. 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 command-line and Java-API interfaces for providing a Dura program and for compiling and executing that program (Section 1-3). Moreover the SQL, Java-API and command-line interfaces for providing the initial data of states and the stream data of events are explained (Section 4). All interfaces are illustrated by small code snippets. Furthermore the Hello World example in Section 6 shows how to go the whole way from compiling, initializing and running a program to the use of the Generic Bus API for providing input data and retrieving results. This description of the interfaces is particularly important for the integration with the SITE component of WP6, but also for WP3 and WP2.

Second the report provides details on the compilation and evaluation of Dura programs. It introduces Temporal Stream Algebra which is used to specify the operational semantics of Dura and serves as an abstraction barrier for the compilation from Dura to SQL. The report sketches the translation of Dura to Temporal Stream Algebra and shows that expressions of Temporal Stream Algebra can be transformed to parametrized expressions of relational algebra. The final generation of SQL statements based on the parametrized relational algebra expressions is not described in the report because the close relationship between relational algebra and SQL is commonly known. Finally the report describes the principles for the evaluation of a Dura program on top of the MonetDB database (WP5) based on the parametrized SQL statements that are the outcome of the compilation process.

Part I. Manual

1. Setup & Configuration

1.1. Needed Software

- MonetDB SQL database (`http://www.monetdb.org/Downloads`)

  Best known compatible version: July 2012
- Java 1.6 or higher (`http://www.oracle.com/technetwork/java/javase/downloads/index.html`)
- event-mill-0.3.3.zip (or event-mill-and-examples-0.3.3.zip) or higher (`http://www.pms.ifi.lmu.de/cep/releases/`)

1.2. Getting started

1. Install Java 1.6 (or higher)
2. Install MonetDB SQL Server
3. Unzip the event-mill-X.X.X.zip archive
4. Adopt the event-mill.configuration.xml file if necessary.

   The event-mill.configuration.xml file contained in the zip archive 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 MonetDB is running on an external server or ports or login information have been changed then event-mill.configuration.xml needs to be adopted to reflect these changes (see Section 1.3.1). 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 (see Section 1.3.3).
5. Adopt the event-mill.io.configuration.xml file if necessary.

   The event-mill.io.configuration.xml file contained in the zip archive should cooperate by default with a local out of the box installation of MonetDB. If the MonetDB configuartion deviates from the defaults the event-mill.io.configuration.xml file needs to be adopted accordingly (see Section 1.4.1). Furthermore additional message formats and bus implementations need to be registered in the event-mill.io.configuration.xml file (see Sections 1.4.3 and 1.4.4).
6. Execute event-mill.jar. It is recommended to start the MonetDB server manually before executing event-mill.jar, however if the DB-server start-up command is correctly config-

urated this should not be required.

1.3. Main Configuration

1.3.1. The Database Connection

The Event-Mill engine needs to build a JDBC connection to the MonetDB database. For this it needs some information on how to connect to MonetDB. This information can be specified in the following section of the event-mill.configuration.xml file:

```
<configuration>
   <stream-buffer-store ... >
      ⋮
      <connection ... use-prepared-statements="false">
         <server url="localhost"/>
         <database name="demo"/>
         <property key="user" value="monetdb"/>
         <property key="password" value="monetdb"/>
      </connection>
      ⋮
   </stream-buffer-store>
   ⋮
</configuration>
```

The following Information can be specified:

- The url of the database server

  ```
        <server url="URL OF THE DATABASE SERVER"/>
  ```

- The name of the database

  ```
        <database name="NAME OF DATABASE"/>
  ```

- The user-name

  ```
        <property key="user" value="USER NAME"/>
  ```

- The password

  ```
        <property key="password" value="PASSWORD"/>
  ```

- The prepared statement flag

  ```
        <connection ... use-prepared-statements="FLAG">
  ```

  This flag specifies whether the Event-Mill engine should use prepared SQL statements or ordinary (adhoc) SQL statements. Currently the recommended value is `false`.

1.3.2. The Meta Schema

The Event-Mill engine needs one schema in the database, the so-called “meta schema”, where it can create tables for storing meta informations, like the available program specifications, the instances of these specification and the database schemas used by these instances to store their data. The meta schema can be specified using

```
   <meta_schema name="NAME OF THE META SCHEMA"/>
```

in the following section of the event-mill.configuration.xml file:

```
<configuration>
   <stream-buffer-store ... >
      ⋮
      <meta_schema name="meta"/>
      ⋮
   </stream-buffer-store>
   ⋮
</configuration>
```

1.3.3. The start-up and stop commands

The start and stop commands are optional. They are used when the Event-Mill engine finds that the MonetDB database server is not yet running at the start of the engine. In that case the Event-Mill engine attempts to start MonetDB using the start command and to stop MonetDB using the stop commands when the engine is stopped itself. The two commands can be specified using

```
   <start-command>START COMMAND</start-command>
   <stop-command>STOP COMMAND</stop-command>
```

in the following section of the event-mill.configuration.xml file

```
<configuration>
   <stream-buffer-store ... >
      ⋮
      <start-command>START COMMAND</start-command>
      <stop-command>STOP COMMAND</stop-command>
   </stream-buffer-store>
   ⋮
</configuration>
```

1.4. I/O Configuration

The I/O configuration is separated from the main configuration of the Event-Mill engine. The reason is that the I/O only affects the Java I/O API for the Event-Mill. However the Java I/O API is not mandatory for delivering or retrieving data to or from Event-Mill. Delivering or retrieving data can also be done by a direct interaction with MonetDB following some simple rules described in Section 4. Therefore the implementation of the Event-Mill engine and the implementation of the Java I/O API are mostly independent and thus their configuration files should be separated, too.

1.4.1. Database Connection

The Java I/O API needs to build a JDBC connection to the MonetDB database. For this it needs some information on how to connect to MonetDB. This information can be specified in the following section of the event-mill.io.configuration.xml file:

```
<configuration>
   <io>
      <streams ... >
         <stream iri="de.lmu.ifi.pms.cep.event-mill.prototype" ... >
            ⋮
            <connection ... use-prepared-statements="false">
               <server url="localhost"/>
               <database name="demo"/>
               <property key="password" value="monetdb"/>
               <property key="user" value="monetdb"/>
            </connection>
            ⋮
         </stream>
      </streams>
      ⋮
   </io>
</configuration>
```

The following Information can be specified:

- The url of the database server

```
      <server url="URL OF THE DATABASE SERVER"/>
```

- The name of the database

```
      <database name="NAME OF DATABASE"/>
```

- The user-name

```
      <property key="user" value="USER NAME"/>
```

- The password

```
      <property key="password" value="PASSWORD"/>
```

- The prepared statement flag

```
      <connection ... use-prepared-statements="FLAG">
```

This flag specifies whether the Event-Mill engine should use prepared SQL statements or ordinary (adhoc) SQL statements. Currently the recommended value is `false`.

1.4.2. Meta Schema

The Java I/O API needs to access the meta data an available programs and their instances stored by the Event-Mill engine. Thus Java I/O API has to know the database schema where the Event-Mill engine puts this information. The meta schema can be specified using

```
   <meta_schema name="NAME OF THE META SCHEMA"/>
```

in the following section of the event-mill.io.configuration.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
   <io>
      <streams ... >
         <stream iri="de.lmu.ifi.pms.cep.event-mill.prototype" ... >
            ⋮
            <meta_schema name="meta"/>
         </stream>
      </streams>
      ⋮
   </io>
</configuration>
```

1.4.3. Providing Message Formats

Internally the Event-Mill engine represents events and other data as tuples. However, outside of the Event-Mill engine events are represented as some kind of messages with a more or less arbitrary format depending on the use-case ad the other components Event-Mill has to integrate with for that use-case. For this reason the Java I/O API for Event-Mill provides a way to specify new Message formats. A message format basically specifies how a a message is converted into a tuple and conversely how a tuple is converted into a message. In this way Event-Mill can easily integrate with different message formats. The Java I/O API can even be used to integrate with different message formats of multiple proprietary components that might need to cooperate for reaching the goals of some use-case.

New message formats can be defined by implementing the interface specified by the abstract class `de.lmu.ifi.pms.cep.event_mill.io.format.MessageFormatProvider`. The new formats need to be registered in the following section of the event-mill.io.configuration.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
 <io>
```

```
  ⋮
  <formats>
   <format iri="de.lmu.ifi.pms.cep.io.format.semi-structured"
     class="de.lmu.ifi.pms.cep.event_mill.io.format.SemiStructuredMessageFormatProvi
   <format iri="de.lmu.ifi.pms.cep.io.format.xml"
     class="de.lmu.ifi.pms.cep.event_mill.io.format.XmlMessageFormatProvider"/>
   ⋮ further formats
  </formats>
  ⋮
 </io>
</configuration>
```

A format can be referenced using its iri. Note that a format might be parametrized and thus the iri of a format can not be equal to the name of the Java class implementing the format in general.

1.4.4. Providing Bus Implementations

The Java I/O API for Event-Mill allows to integrate with quite arbitrary bus systems. A bus is just a mechanism to transfer messages of some format from one endpoint of the bus to other endpoints of the bus. The Java I/O API makes no implications on the way this transmission is actually done and which policies are used to distribute the messages between the endpoints of the bus. This is fully left to the implementation of the bus system. One prominent example for such a bus system is the Enterprise Service Bus (ESB) proposed in WP6 ([]).

The employed bus system can be specified in the following section of the event-mill.io.configuration.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
   <io>
      ⋮
      <buses class="de.lmu.ifi.pms.cep.event_mill.io.bus.HeapBroadcastMessageBusProv
         <format ref="de.lmu.ifi.pms.cep.io.format.xml"/>
      </buses>
      ⋮
   </io>
</configuration>
```

1.5. Log Configuration

The Event-Mill engine uses the slf4j (`http://www.slf4j.org/`) logging facade together with its standard binding, the logback logging framework (`http://logback.qos.ch/index.html`). In this way the logging of the Event-Mill engine can easily be integrated with the logging external components even if these components use different logging frameworks. Such components could particularly originate from WP2 and WP3.

The Event-Mill engine employs the log4jdbc package(http://code.google.com/p/log4jdbc/) for logging JDBC queries. log4jdbc internally uses the slf4j logging facade.

The logging of Event-Mill can be configured using the logback.xml, logback-test.xml, event-mill.logback.xml and event-mill.logback-test.xml files. The intentions of these files are explained in the following. Details on the syntax and structure of the files and the configuration of logback can be found at http://logback.qos.ch/manual/index.html.

1.5.1. The logback.xml and the logback-test.xml files

The logback logging framework is usually configured using the the logback.xml file. As the Event-Mill engine is likely to be used as one of a number of different components the logback.xml file contained in the Event-Mill installation was designed in such a way that it supports a modular configuration of the logging. Therefore the logback.xml file contains only some very general specifications and delegates the configuration of the logging for the particular components to configuration files corresponding to each component. For the Event-Mill engine this file is called event-mill.logback.xml.

The following shows the structure of the logback.xml file. Note the inclusion of the event-mill.logback.xml file at the end of the configuration.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="false">

   <timestamp key="logFileTimestamp" datePattern="yyyyMMdd'-'HHmmss'-'SSS"/>
   <property name="logFileDir" value="log/" />

   <!-- directs logging to the System.out stream -->
   <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
   ⋮
   </appender>

   <!-- directs logging to the System.err stream -->
   <appender name="STDERR" class="ch.qos.logback.core.ConsoleAppender">
   ⋮
   </appender>

   <root level="WARN">
      <appender-ref ref="STDERR" />
   </root>

   <include resource="event-mill.logback.xml"/>
   ⋮ log configurations for other modules

</configuration>
```

For testing and debugging purposes logback supports an easy mechanism to override the usual log configuration. If a file called logback-test.xml is available then the configuration in that file is preferred before the one in the logback.xml file. Usually the logback-test.xml file will configure a more extensive logging than the logback.xml file.

1.5.2. The event-mill.logback.xml and the event-mill.logback-test.xml files

As mentioned before the event-mill.logback.xml file is referenced from the logback.xml file and actually configures the logging for the Event-Mill engine. The event-mill.logback-test.xml file is used instead if the logback-test.xml file is present.

The following shows the structure of the event-mill.logback.xml file contained in the installation.

```
<?xml version="1.0" encoding="UTF-8"?>
<included>

   <!-- Creates the YearMonthDay-HourMinuteSecond-Millis-event-mill.log file -->
   <appender name="EVENT_MILL_LOG" class="ch.qos.logback.core.FileAppender">
   ...
   </appender>

   <!-- Creates the YearMonthDay-HourMinuteSecond-Millis-event-mill-io.log file -->
   <appender name="EVENT_MILL_IO_LOG" class="ch.qos.logback.core.FileAppender">
   ...
   </appender>

   <!-- Creates the YearMonthDay-HourMinuteSecond-Millisecond-sql.log file -->
   <appender name="SQL_LOG" class="ch.qos.logback.core.FileAppender">
   ...
   </appender>

   <!-- logging of engine execution -->

   <logger name="de.lmu.ifi.pms.cep" level="TRACE">
      <appender-ref ref="EVENT_MILL_LOG"/>
   </logger>

   <logger name="de.lmu.ifi.pms.cep.event_mill.io" level="TRACE" additivity="false">
      <appender-ref ref="EVENT_MILL_IO_LOG" />
   </logger>

   <logger name="de.lmu.ifi.pms.cep.event_mill.sql.io" level="TRACE" additivity="fa
      <appender-ref ref="EVENT_MILL_IO_LOG" />
   </logger>

   <!-- logging of jdbc queries -->

   <!--configuration of the database internal logging-->

   <!-- Creates the YearMonthDay-HourMinuteSecond-Millis-event-mill.log file -->
   <logger name="nl.cwi.monetdb" level="DEBUG" additivity="false"/>

   <!-- log4jdbc logger configuration -->

   <logger name="jdbc" level="OFF" additivity="false">
      <appender-ref ref="EVENT_MILL_LOG" />
      <appender-ref ref="SQL_LOG" />
   </logger>

   <logger name="jdbc.sqlonly" level="INFO"/>
   <logger name="jdbc.sqltiming" level="OFF"/>
   <logger name="jdbc.audit" level="OFF"/>
   <logger name="jdbc.resultset" level="OFF"/>
   <logger name="jdbc.connection" level="OFF"/>

   <!-- log4jdbc internal logging -->
   <!-- logger name="log4jdbc.debug" level="WARN"/-->

</included>
```

The following line in the configuration above has a special meaning, though it follows the usual syntax of configuration files for logback.

```
   <!-- Creates the YearMonthDay-HourMinuteSecond-Millis-event-mill.log file -->
   <logger name="nl.cwi.monetdb" level="DEBUG" additivity="false"/>
</included>
```

The JDBC driver of MonetDB seems to use a proprietary kind of logging. Thus ist could usually not be configured using slf4j and logback. To work around this problem the specification for logger “nl.cwi.montetdb” is internally used to set the log level of the MonetDB JDBC driver. In this way the configuration of the logging remains homogeneous.

1.6. Eclipse Plug-ins

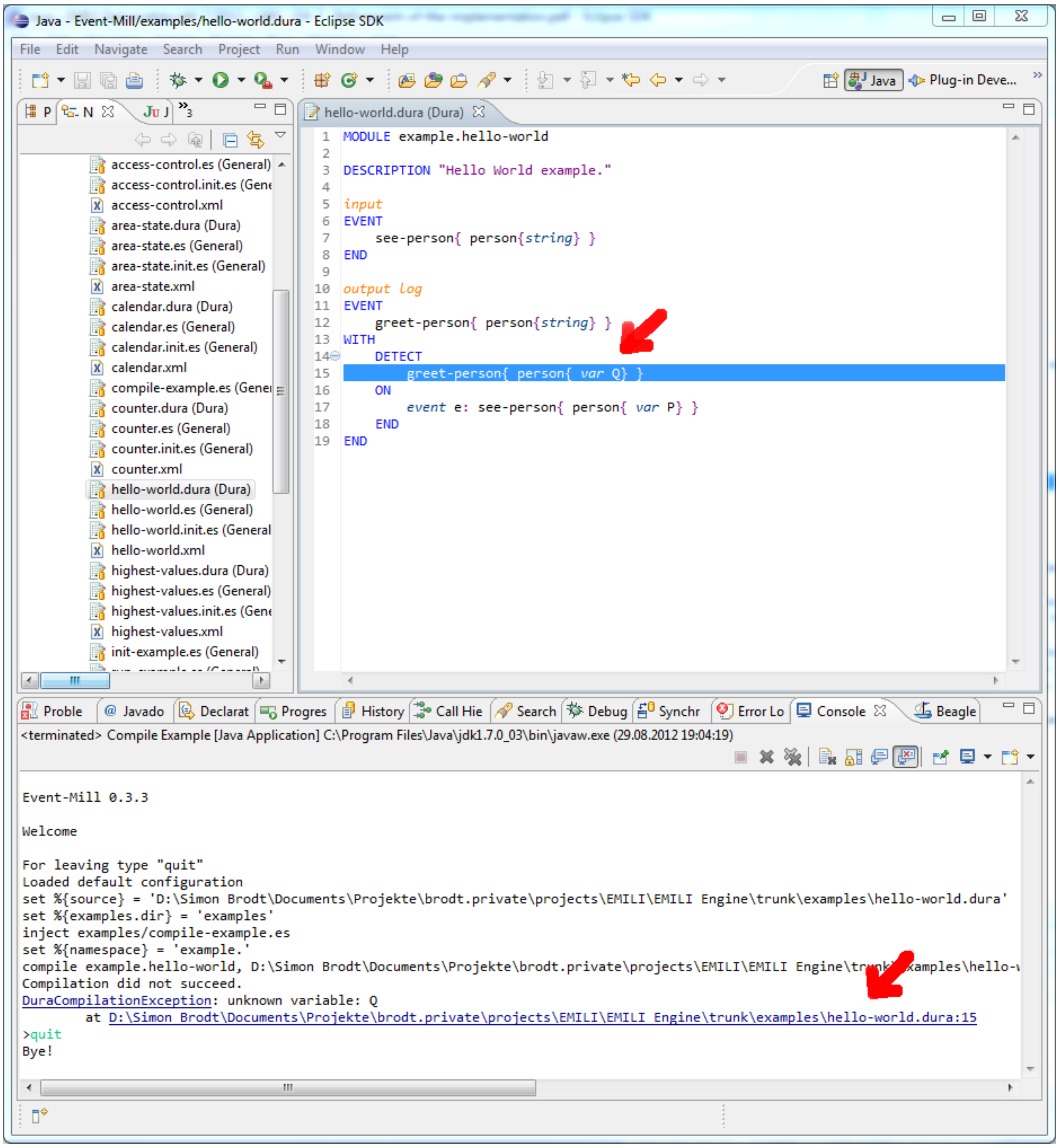
1.6.1. Syntax Highlighting

Syntax highlighting is very useful to show the structure of a program. The Eclipse update site `http://sunshade.sourceforge.net/` provides a modified version of the worldfile editor plug-in for Eclipse (`http://marketplace.eclipse.org/content/wfe-wordfile-editor`) that enables syntax highlighting for Dura.

1.6.2. Error Linking

It is very helpful for the development process of a program if the error messages from the compiler directly point to the location, i.e. the file and line, in a program that cause the error. The Eclipse update site `http://www.pms.ifi.lmu.de/cep/plugins/sunshade/` provides a modified version of the sunshade plug-in for Eclipse (`http://sunshade.sourceforge.net/`) that creates hyperlinks from the error messages of the Dura compiler that allow to directly jump to the file and the line in that file that caused the error.

The following screen-shot shows the appearance of Eclipse with both plug-ins installed. The visible Hello World example contains an error, that is detected by the compiler. The line causing the error is marked as a result of clicking the hyperlink in the error message.


Java - Event-Mill/examples/hello-world.dura - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Java
Plug-in Deve...
hello-world.dura (Dura)
access-control.es (General)
access-control.init.es (General)
access-control.xml
area-state.dura (Dura)
area-state.es (General)
area-state.init.es (General)
area-state.xml
calendar.dura (Dura)
calendar.es (General)
calendar.init.es (General)
calendar.xml
compile-example.es (General)
counter.dura (Dura)
counter.es (General)
counter.init.es (General)
counter.xml
hello-world.dura (Dura)
hello-world.es (General)
hello-world.init.es (General)
hello-world.xml
highest-values.dura (Dura)
highest-values.es (General)
highest-values.init.es (General)
highest-values.xml
init-example.es (General)
Proble Javado Declarat Progres History Call Hie Search Debug Synchr Error Lo Console Beagle

```
1  MODULE example.hello-world
2
3  DESCRIPTION "Hello World example."
4
5  input
6  EVENT
7      see-person{ person{string} }
8  END
9
10 output log
11 EVENT
12     greet-person{ person{string} }
13 WITH
14     DETECT
15         greet-person{ person{ var Q} }
16     ON
17         event e: see-person{ person{ var P} }
18     END
19 END
```

```
<terminated> Compile Example [Java Application] C:\Program Files\Java\jdk1.7.0_03\bin\javaw.exe (29.08.2012 19:04:19)

Event-Mill 0.3.3

Welcome

For leaving type "quit"
Loaded default configuration
set %{source} = 'D:\Simon Brodt\Documents\Projekte\brodt.private\projects\EMILI\EMILI Engine\trunk\examples\hello-world.dura'
set %{examples.dir} = 'examples'
inject examples/compile-example.es
set %{namespace} = 'example.'
compile example.hello-world, D:\Simon Brodt\Documents\Projekte\brodt.private\projects\EMILI\EMILI Engine\trunk\examples\hello-w
Compilation did not succeed.
DuraCompilationException: unknown variable: Q
        at D:\Simon Brodt\Documents\Projekte\brodt.private\projects\EMILI\EMILI Engine\trunk\examples\hello-world.dura:15
>quit
Bye!
```

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

2.1. Important Parameters

- The *module* specifies a kind of entry-point into a Dura program. It refers to the modularization mechanism of Dura (see [].)
- The *source folder* or folders specify the set of Dura files that are searched for specifications contained in or referenced by the specified module.
- The *DTSA file* denotes the file where the outcome of the compilation should be saved. As the compilation of a Dura program involves some sophisticated analysis, loading the outcome of the compilation is much faster than recompiling the program.
- The *instance* of a program is the instantiation of a program specification. The instantiation of a program specification mainly assigns a "memory area" to the program, i.e. a set of database schemata where the program may store its data and perform the evaluation. There may exist multiple instances of the same program specification.
- The *full name of an instance* consists of the name of the program followed by the name of the instance.

```
String sourceFolder = "examples/";
String module = "example.hello-world";
String dtsaFile = "examples/hello-world.xml";
String instance = "test";

final String instanceFullName =
   TSA_SyntaxPatterns.programInstFullName(module, instance);
```

2.2. Creating an Event-Mill instance

The `de.lmu.ifi.pms.cep.event_mill.EventMill` class provides a way to easily interact with the Event-Mill engine using Java. An instance of this class can be created in the following way, where `Configuration.read()` deserializes the event-mill.configuration.xml file into an instance of `de.lmu.ifi.pms.cep.event_mill.configuration.Configuration`.

API call

```
EventMill eventMill = new EventMill(Configuration.read());
```

2.3. The `clear` command

Syntax: `clear`

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

API call

```
eventMill.clear();
```

2.4. The `compile` command

Syntax [Deprecated]: `compile MODULE, SOURCE_FILE.dura`

Compiles the Dura program specified in `SOURCE_FILE.dura` and stores the compiled program specification under name `MODULE`. Fails if there already exists a program specification with name `MODULE`.

This command has become deprecated due to the introduction of the modularization features described in Deliverable D4.6 [8]. According to that feature programs are intended to consist of multiple files. However a generalization of this command using several source files could not be disambiguated from the compile command described below.

API call

```
DTSA_Program dtsa_program = eventMill.compile(module,
   sourceFile);
eventMill.addSpec(dtsa_program);
```

Syntax: `compile MODULE, DTSA_File.xml, SOURCE_PATH_1, SOURCE_PATH_2, ...`

Compiles the Dura program specified in `SOURCE_PATH_1, SOURCE_PATH_2, ...` and stores the compiled program specification together with the specified name `NAMESPACE.PROGRAM_NAME` in the `DTSA_File.xml` file.

API call

```
DTSA_Program dtsa_program =
    eventMill.compile(module, sourcePath1, sourcePath2, ... );
eventMill.save(dtsa_program, dtsaFile);
```

2.5. The `load` command

Syntax: `load spec DTSA_FILE.xml`

Loads the compiled form of a program specification contained in the file `DTSA_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.

API call

```
dtsa_program = eventMill.load(dtsaFile);
eventMill.addSpec(dtsa_program);
```

2.6. The `list specs` command

Syntax: `list specs`

Lists the names of all available program specifications.

API call

```
eventMill.availableProgramSpecifications()
```

2.7. The `remove spec` command

Syntax: `remove spec MODULE`

Removes the program specification with name `MODULE` if such exists.

API call

```
eventMill.removeSpec(module);
```

2.8. The `clear specs` command

Syntax: `clear specs`

Removes all existing program specifications.

API call

```
eventMill.clearSpecs();
```

2.9. The `init` command

Syntax: `init MODULE:INSTANCE, META_SCHEMA, INPUT_SCHEMA, WORKING_SCHEMA, OUTPUT_SCHEMA, LOG_SCHEMA`

Initialize the program instance named `MODULE:INSTANCE` 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.

API call

```
String[] schemas = new String[]{
                     "sample_meta",
                     "sample_input",
                     "sample_working",
                     "sample_output",
                     "sample_log"
                   };

eventMill.instanciate(module, instance, schemas);
```

2.10. The `list progs` command

Syntax: `list progs`

Lists the name of all instantiated program instances.

API call

```
eventMill.availablePrograms()
```

2.11. The `run` command

Syntax: `run MODULE:INSTANCE_NAME`

Loads the program instance named `MODULE:INSTANCE_NAME` and switches to the so-called “Control Dialog” (see Section 3) which serves for controlling the execution of the program instance.

API call

```
//Obtain program instance
ExecutableProgram<?, ?, ?> executableProgram =
   eventMill.getProg(module, instance);

//Create engine for program instance
Engine engine = new Engine(executableProgram);
```

2.12. The `remove prog` command

Syntax: `remove prog NAMESPACE.PROGRAM_NAME:INSTANCE_NAME`

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

API call

```
eventMill.removeProg(module, instance);
```

2.13. The `clear progs` command

Syntax: `clear progs`

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

API call

```
eventMill.clearProgs();
```

2.14. 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.

API call

```
eventMill.uninstall();
```

2.15. The `quit` command

Syntax: `quit`

Exits the Event-Mill engine.

API call

```
eventMill.close();
```

3. The Control Dialog – Running a Dura Program

The Control Dialog is entered using the `run` command and serves for controlling the execution of an program instance.

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).

API call

```
SchedulerFactory scheduler = eventMill.configuration().
   scheduler;
//retrieving the scheduler from the configuration of the
   Event-Mill instance

engine.init(eventMill.configuration().scheduler);
//initializing the engine instance for the given program with
    the assigned scheduler
```

3.2. The `start` command

Syntax: `start`

Starts/Resumes the query execution for the current program.

API call

```
engine.start();
```

3.3. The `stop` command

Syntax: `stop`

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

API call

```
engine.stop();
```

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.

API call

```
engine.close();
```

4. Delivering and Retrieving Data

4.1. The IOConfiguration object

As described in Section 1.4 the event-mill.io.configuration.xml file contains the required information for connecting to MonetDB or the Event-Mill Engine respectively and for attaching to the configured bus system. This configuration can be accessed using an instance of the `de.lmu.ifi.pms.cep.event_mill.configuration.IOConfiguration` class. This instance can be obtained in the following way:

```
    IOConfiguration ioConfiguration = IOConfiguration.read();
```

4.2. The SQL_EngineIO object

For each initialized Dura program an instance of class `de.lmu.ifi.pms.cep.event_mill.sql.io.SQL_EngineIO` can be used to access the data catalog of the Dura program and to generate the relevant SQL statements for delivering and retrieving data to and from the Event-Mill engine. The `SQL_EngineIO` instance for an initialized Dura program can be obtained from the `ioConfiguration` in the following way, where `engineIdentifier` identifies the instance of the Event-Mill engine that should be used and `programInstanceIdentifier` is the name for the stream corresponding to the instantiated Dura program:

```
String engineIdentifier =
   "de.lmu.ifi.pms.cep.event-mill.prototype";
String programInstanceIdentifier = PATTERNS
   .programIdentifier(engineIdentifier, instanceFullName);

SQL_EngineIO sql_engineIO =
   (SQL_EngineIO) ioConfiguration.stream(
                     engineIdentifier,
                     programInstanceIdentifier
                  );
```

4.3. The Generic Bus API

The Generic Bus API is a convenience API that significantly helps to integrate the Event-Mill engine with almost arbitrary bus systems used for distributing event messages like the Enterprise Service Bus (ESB) described in Deliverable D6.3 [27] The generic Bus ABI bases on three concepts, *streams* which carry events represented as tuples, *buses* which distribute events using arbitrarily formatted event messages and *connectors* which serve for interconnecting streams and buses.

4.3.1. Streams

Streams represent events using tuples. Tuple sinks, i.e. instances of the class `de.lmu.ifi.pms.cep.event_mill.io.stream.SinkStream` are used to write tuples to the stream. Conversely tuple sources, i.e. instances of the class `de.lmu.ifi.pms.cep.event_mill.io.stream.SourceStream` are used to read tuples from the stream. Each sink or source is attached to a single event type and a fixed tuple schema.

Instances of the class `de.lmu.ifi.pms.cep.event_mill.io.stream.StreamIO` provide of tuple sinks and tuples sources for a set of input and output event types. Such sets of input and output event types are usually constituted by the definitions of input and output types in a Dura program.

The `StreamIO` object corresponding to an instantiated Dura program named `instanceFullName` can be obtained in the following way:

```
String engineIdentifier =
   "de.lmu.ifi.pms.cep.event-mill.prototype";
String programInstanceIdentifier = PATTERNS
   .programIdentifier(engineIdentifier, instanceFullName);
```

```
StreamIO streamIO = ioConfiguration.stream(
                        engineIdentifier,
                        programInstanceIdentifier
                    );
```

A tuple sink for an input type of the Dura program can be obtained by:

```
String nameOfInputType = ...
SinkStream sink = streamIO.sinkStream(nameOfInputType)
```

A tuple source for an output type of the Dura program can be obtained by:

```
String nameOfOutputType = ...
SourceStream source = streamIO.sourceStream(nameOfOutputType)
```

4.3.2. Buses

Buses represent events by some kind of messages. In principle the messages may have any Java type, structure or format. The only requirement is that the bus must provide a way to read and write messages from and to tuples. More precisely the bus needs to provide an instance of class `de.lmu.ifi.pms.cep.event_mill.io.format.MessageFormatProvider` that allows to convert a tuple into a suitable message object for the bus and vice versa.

The input and output messages for the Event-Mill engine are transmitted on two different different buses (potentially but not necessarily of the same bus system). Though buses are basically bidirectional, different buses for input and output messages have to be used. The reason is that there may be input and output events with the same nominal type but with a slightly different schema for input and output. However the conversion of tuples to messages and from messages to tuples requires a unique schema for the tuples of one type.

The following illustrates the connection to the bus on the side of the Event-Mill engine. Any producer or consumer of event messages can connect to the bus in mostly the same way. The only difference is that he must use a different end point identifier (`busEndpointIdentifier`) than the Event-Mill engine uses for the program instance.

```
String busEndpointIdentifier = programInstanceIdentifier;

String inputBusIdentifier = PATTERNS
   .inputBusIdentifier(engineIdentifier, instanceFullName);
Class<String> inputMessageType = String.class;
String inputMessageFormatIdentifier =
   SemiStructuredMessageFormatProvider.defaultIdentifier;
```

```
MessageBus<String> inputBus =
   ioConfiguration.bus(
      inputBusIdentifier,
      inputMessageType,
      inputMessageFormatIdentifier,
      programInstanceBusEndpoint
   );

String outputBusIdentifier = PATTERNS
   .outputBusIdentifier(engineIdentifier, instanceFullName);
Class<String> outputMessageType = String.class;
String outputMessageFormatIdentifier =
   XmlMessageFormatProvider.defaultIdentifier;

MessageBus<String> outputBus =
   ioConfiguration.bus(
      outputBusIdentifier,
      outputMessageType,
      outputMessageFormatIdentifier,
      busEndpointIdentifier
   );
```

4.3.3. Connectors

Connectors are used to interconnect buses and streams. Connectors work unidirectional. There are connectors that connect streams to buses, buses to streams, one stream to another stream and one bus to another bus. Connectors with a stream on one side and a bus on the other are used to connect the Event-Mill engine to a bus system. Connectors with streams on both sides can be used for coupling multiple instances of the of the Event-Mill instance, e.g. for a distributed processing. Connectors with buses on both sides allow to couple different bus systems. The situation of different bus systems that need to cooperate is likely to occur when an existing system for emergency management should be enhanced by advanced capabilities like the ones developed in the EMILI project.

Connector objects can be obtained by calling the factory methods of the respective classes:

```
BusToStreamConnector<?> inputConnector =
   BusToStreamConnector.create(inputBus, streamIO);
StreamToBusConnector<?> outputConnector =
   StreamToBusConnector.create(outputBus, streamIO);
```

Activating the Connection Connectors are passive by default. In other word, for actually transferring data it is required to frequently call their `nextBlock()` method as shown below.

```
Thread ioThread = new Thread("IO-Thread"){
   @Override
   public void run() {
      while(!this.isInterrupted()){
         inputConnector.nextBlock();
         outputConnector.nextBlock();
         try {
            Thread.sleep(1000);
         } catch (InterruptedException e) {
            break;
         }
      }

   }
};
```

4.4. Exploring the data catalog

4.4.1. Schema

The Event-Mill engine uses five different schema in the database for each initialized Dura 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.

The five schema of an initialized Dura program can be obtained in the following two ways:

SQL

```
SELECT
   "insts"."meta_schema" AS "meta_schema",
   "insts"."input_schema" AS "input_schema",
   "insts"."working_schema" AS "working_schema",
   "insts"."output_schema" AS "output_schema",
   "insts"."log_schema" AS "log_schema"
FROM
   "meta"."program_specs" AS "specs",
   "meta"."program_insts" AS "insts"
WHERE
   "specs"."namespace" = 'example'
AND
   "specs"."simple_name" = 'hello-world'
AND
   "insts"."instance" = 'test'
AND
   "specs"."id" = "insts"."spec_ref"
;
```

API call to the respective `sql_engineIO` object

```
String meta_schema = sql_engineIO.metaSchema();
String input_schema = sql_engineIO.inputSchema();
String working_schema = sql_engineIO.workingSchema();
String output_schema = sql_engineIO.outputSchema();
String log_schema = sql_engineIO.logSchema();
```

4.4.2. 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 modules as mean for structuring a program see Deliverable D4.6 [8]. A simple module 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 module name is an arbitrary long sequence of simple module names.

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

4.4.3. Retrieving the Defined Types of a Dura Program

The names of the types defined in a Dura program can be obtained in the following two ways:

SQL

```
SELECT "prog_package", "simple_name",
FROM "META_SCHEMA"."buffers"
WHERE
        "type" = 'TYPE OF BUFFER'
;
```

where `META_SCHEMA` denotes the name of the meta schema of the respective program instance (see Section 4.4.1) and `TYPE OF BUFFER` is either `INPUT` or `OUTPUT` .

API call to the respective `sql_engineIO`object (see Section 4.2)

```
DTSA_BufferType buffer_type = ...
FixedArrayNavigableSet<String> type_names =
   sql_engineIO.bufferNames(buffer_type)
```

4.4.4. Mapping of Dura Types to SQL 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. As some SQL implementations impose restrictions on the length of table names, the table name produced in the step above will be shortened if necessary.

The SQL table name mapping for some Dura type an instantiated program can be obtained in the following two ways:

SQL

```
SELECT "table"
FROM "META_SCHEMA"."buffers"
WHERE "type" = 'TYPE OF BUFFER'
  AND "prog_package"='MODULE_PART_OF TYPE_NAME'
  AND "simple_name"='SIMPLE_NAME_PART_OF TYPE_NAME'
;
```

where `META_SCHEMA` denotes the name of the meta schema of the respective program instance (see Section 4.4.1), `TYPE OF BUFFER` is either `INPUT` or `OUTPUT`, `MODULE_PART_OF TYPE_NAME` is the Dura type name up to the last dot (inclusive) if such exist and the empty string otherwise and `SIMPLE_NAME_PART_OF TYPE_NAME` is the Dura type name from the last dot (exclusive) .

API call to the respective `sql_engineIO`object (see Section 4.2)

```
DTSA_BufferType buffer_type = ...
String dura_type_name = ...
String table_name = sql_engineIO
   .bufferNameSQLMapping(buffer_type, dura_type_name)
```

4.4.5. 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.

4.4.6. Retrieving the Attribute Names and Types of a Dura Type

The names of the attributes defined for a type of a Dura program can be obtained in the following two ways:

SQL

```
SELECT "attributes"."name" AS "name",
       "attributes"."type" AS "type"
FROM "META_SCHEMA"."buffers" AS "buffers",
     "META_SCHEMA"."attributes" AS "attributes"
WHERE "buffers"."id"="attributes"."buffer_id"
  AND "buffers"."type"='TYPE OF BUFFER'
  AND "buffers"."prog_package"='PACKAGE_PART_OF TYPE_NAME'
  AND "buffers"."simple_name"='SIMPLE_NAME_PART_OF TYPE_NAME'
ORDER BY "attributes"."name" ASC
;
```

API call to the respective `sql_engineIO`object (see Section 4.2)

```
DTSA_BufferType buffer_type = ...
String dura_type_name = ...
FixedArrayNavigableSet<String> attribute_names =
   sql_engineIO.attributeNames(buffer_type, dura_type_name)
```

4.4.7. Mapping of Dura Attribute Names 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. As some SQL implementations impose restrictions on the length of attribute names, the table name produced in the step above will be shortened if necessary.

The SQL attribute name mapping for the attributes of some Dura type of an instantiated program can be obtained in the following two ways:

SQL

```
SELECT "attributes"."sql_name" AS "sql_name"
FROM "META_SCHEMA"."buffers" AS "buffers",
     "META_SCHEMA"."attributes" AS "attributes"
WHERE "buffers"."id"="attributes"."buffer_id"
  AND "buffers"."type"='TYPE OF BUFFER'
  AND "buffers"."prog_package"='PACKAGE_PART_OF TYPE_NAME'
  AND "buffers"."simple_name"='SIMPLE_NAME_PART_OF TYPE_NAME'
ORDER BY "attributes"."name" ASC
;
```

where `META_SCHEMA` denotes the name of the meta schema of the respective program instance (see Section 4.4.1), `MODULE_PART_OF TYPE_NAME` is the Dura type name up to the last dot (inclusive) if such exist and the empty string otherwise , `SIMPLE_NAME_PART_OF TYPE_NAME` is the Dura type name from the last dot (exclusive) and **`TYPE`** `OF BUFFER` is either `INPUT` or `OUTPUT` .

API call to the respective `sql_engineIO`object (see Section 4.2)

```
DTSA_BufferType buffer_type = ...
String dura_type_name = ...
String dura_attribute_name = ...
String attribute_name = sql_engineIO
   .attributeNameSQLMapping(buffer_type,
                            dura_type_name,
                            dura_attribute_name)
```

4.4.8. Mapping of Attribute Types

All Dura types are mapped to flat tuples where the structure is encoded into the (general) attribute names . 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

4.5. Input

4.5.1. 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.

4.5.2. Initialize Static Data

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

4.5.3. Inserting tuples

SQL:

The "_o_block" attribute of the inserted tuple has a special purpose and MUST NOT be set by the payload of an event or initial state. Instead it is required to be set to a special value.

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

```
INSERT INTO
   "sample_input"."example_o_hello_d_world_o_see_d_person"
   ("_o_block", "person")
VALUES
   ((SELECT "block_counter" FROM "sample_meta"."buffers"
     WHERE "id" = '1')
   , 'boy_1')
;
```

Stream API

Initial preparation

```
String dura_type_name = ...
SinkStream sink =
   streamIO.sinkStream(dura_type_name)
```

Actual insert of two tuples

```
sink.next();
sink.setString("person","boy_1");
sink.next();
sink.setString("person","girl_1");

sink.nextBlock();
```

where "person" is the name of an attribute and "boy_1" and "girl_1" are the respective values for this attribute.

Bus API

Initial preparation

```
MessageSink<String> messageSink = emulationOutputBus.sink();
```

Sending a single message

```
String message = "...";
messageSink.write(message);
```

Sending a bulk of messages

```
ArrayList<String> messages = new ArrayList<String>();

String message1 = "...";
String message2 = "...";
messages.add(message1);
messages.add(message2);

messageSink.writeBlock(messages);

messages.clear();
```

4.5.4. The `parse` command

Within the Control Dialog (See Section 3) the `parse` command can be used for initializing static data and sending new events to the Event-Mill engine. The `parse` only puts the messages on the input bus of the Event-Mill engine. A subsequent `write` command is needed to actually insert the corresponding tuples into the tables of the Event-Mill engine. At the moment messages need to be written in a single line.

Syntax: `parse MESSAGE`

Corresponding API call

```
messageSink.write(message);
```

4.5.5. The `write` command

Within the Control Dialog (See Section 3) the `write` command can be used to transfer messages from the input bus to the input tables of the Event-Mill engine. The `write` takes all messages available on the input bus and places the corresponding tuples int the respective tables of the Event-Mill engine.

Syntax: `write`

API call

```
inputConnector.nextBlock();
```

4.6. Output

4.6.1. Retrieving Events and Actions

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

4.6.2. Retrieving Tuples

SQL:

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

Streams

Initial preparation

```
String dura_type_name = ...
SourceStream source =
   streamIO.sourceStream(dura_type_name)
```

Reading a bulk of tuples

```
source.nextBlock();
while(source.next()){
   String person = source.getString("person");
}
```

where "person" is the name of an attribute

Buses

Initial preparation

```
MessageSource<String> messageSource = uiInputBus.source();
```

Reading a single message

```
String message = messageSource.read();
//message can be null
//if there is no message available currently
if(message != null){
   PERFORM ACTION
}
```

Note that the `read()` method is non-blocking. Thus, if no message is available it will return `null`.

Reading a bulk of messages

```
for(String message: messageSource.readBlock()){
   PERFORM ACTION
}
```

4.6.3. The `read` command

Within the Control Dialog (See Section 3) the `read` command can be used to transfer messages from the output tables of the Event-Mill engine. to the output bus. The `read` command takes all tuples from the output tables of the Event-Mill engine and adds the corresponding messages to the output bus.

Syntax: `read`

API call

```
outputConnector.nextBlock();
```

4.6.4. The `print` command

Within the Control Dialog (See Section 3) the `print` command can be used for reading event and action messages from the output bus of the Event-Mill engine and displaying the messages on the console. The `print` command only considers event and action messages that are available from the bus. A preceding `read` command is needed to to add the messages corresponding to the tuples in the output tables of the Event-Mill engine to the bus.

Syntax: `print`

API call

At call

```
for(String message: messageSource.readBlock()){
   System.out.println(message);
}
```

5. Event Scripts

Event Scripts are an easy way to run a sequence of the commands described in Sections 2, 3 and 4. Such sequences are useful for testing a Dura program for example.

5.1. Variables

Event Scripts allow to define variables that are useful if certain parameters are reused over several commands, or if all parameters should be specified at the beginning of the script.

Syntax:

Set Variable
```
   set %{VARIABLE_NAME} = 'VALUE'
```
Drop Variable
```
   set %{VARIABLE_NAME} =
```
Variable Reference
```
   set %{VARIABLE_NAME}
```

5.1.1. Special Variables and Default Values

Some variables have a special meaning:

- `source` - The source file

  Implicitly sets `source.dir`, `result`, `name` and all variables that are implicitly set when setting `name`

- `source.dir` - The source directory
- `result` - The name of the file for the result of the compilation
- `name` - The simple name of the program (The part after the last dot)

  Implicitly sets `meta-schema`, `input-schema`, `working-schema`, `output-schema`, `log-schema`, `instance`
- `meta-schema` - The name of the meta schema
- `input-schema` - The name of the input schema
- `working-schema` - The name of the working schema
- `output-schema` - The name of the output schema
- `log-schema` - The name of the log schema
- `instance` - The name of the program instance
- `namespace` - The namespace of the program (The part preceeding the last dot)

5.2. The `inject` command

The `inject` command executes the commands contained in the specified file name and then continues with the sequence of commands that was available before the execution of the `inject` command, if such commands existed.

Syntax: `inject FILENAME`

5.3. The `pause` command

The `pause` command suspends the execution of further commands for the specified amount of milli seconds. This command is particularly useful in combination with the parse and write commands to define a timing for the incoming events.

Syntax: `pause TIME_IN_MILLIES`

6. The Hello World Example

6.1. 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.

```
MODULE hello-world

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
```

6.2. Input

The incoming see_person events have to be inserted into the `INPUT_SCHEMA."see_d_person"` table. The table has two attributes, namely "_o_block" and "person". The insertion *must not* set the "_o_block" attribute.

6.3. Output

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

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."hello_d_world_o_greet_d_person"` buffer/table for the greet_person events to the corresponding `OUTPUT_SCHEMA."hello_d_world_o_greet_d_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" = 'hello-world.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."hello_d_world_o_greet_d_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

6.4. Test Sequence

```
set %{source} = 'examples/hello-world.dura'
set %{examples.dir} = 'examples'
set %{namespace} = ''

compile %{namespace}%{name}, %{result}, %{source}

remove prog %{namespace}%{name}:%{instance}
remove spec %{namespace}%{name}

load spec %{result}
list specs

init %{namespace}%{name}:%{instance}, %{meta-schema}, %{
   input-schema}, %{working-schema}, %{output-schema}, %{
   log-schema}
list progs

run %{namespace}%{name}:%{instance}

init
start

parse hello-world.see-person{ person{ "boy_1" } }
parse hello-world.see-person{ person{ "girl_1" } }
write

pause 1500
```

```
read
print

parse hello-world.see-person{ person{ "boy_2" } }
parse hello-world.see-person{ person{ "girl_2" } }
write

pause 1500

read
print

pause 500
stop
quit

quit
```

6.5. Full API example

```
package de.lmu.ifi.pms.cep.event_mill.sample;

import java.io.IOException;
import java.util.ArrayList;

import de.lmu.ifi.pms.cep.event_mill.EventMill;
import de.lmu.ifi.pms.cep.event_mill.configuration.Configuration;
import de.lmu.ifi.pms.cep.event_mill.configuration.IOConfiguration;
import de.lmu.ifi.pms.cep.event_mill.execution.engine.Engine;
import de.lmu.ifi.pms.cep.event_mill.execution.program.ExecutableProgram
   ;
import de.lmu.ifi.pms.cep.event_mill.execution.scheduler.
   SchedulerFactory;
import de.lmu.ifi.pms.cep.event_mill.io.MessageSink;
import de.lmu.ifi.pms.cep.event_mill.io.MessageSource;
import de.lmu.ifi.pms.cep.event_mill.io.PATTERNS;
import de.lmu.ifi.pms.cep.event_mill.io.bus.MessageBus;
import de.lmu.ifi.pms.cep.event_mill.io.connect.BusToStreamConnector;
import de.lmu.ifi.pms.cep.event_mill.io.connect.StreamToBusConnector;
import de.lmu.ifi.pms.cep.event_mill.io.format.
   SemiStructuredMessageFormatProvider;
import de.lmu.ifi.pms.cep.event_mill.io.format.XmlMessageFormatProvider;
import de.lmu.ifi.pms.cep.event_mill.io.stream.StreamIO;
import de.lmu.ifi.pms.cep.tsa.DTSA_Program;
import de.lmu.ifi.pms.cep.tsa.TSA_SyntaxPatterns;
```

```
public class Main {

public static void main(final String[] args) throws Exception{

   String sourceFolder = "examples/";
   String module = "hello-world";
   String dtsaFile = "examples/hello-world.xml";
   String instance = "test";

   final String instanceFullName = TSA_SyntaxPatterns.
      programInstFullName(module, instance);

   String[] schemas = new String[]{
                     "sample_meta",
                     "sample_input",
                     "sample_working",
                     "sample_output",
                     "sample_log"
                     };



   EventMill eventMill = new EventMill(Configuration.read());

   //clear event mill
   eventMill.clear();

   DTSA_Program dtsa_program = eventMill.compile(module, sourceFolder);

   assert dtsa_program.full_name.equals(module);

   eventMill.save(dtsa_program, dtsaFile);
   dtsa_program = eventMill.load(dtsaFile);

   eventMill.addSpec(dtsa_program);

   //list available program specifications
   System.out.println("Available specifications:");
   for(String spec: eventMill.availableProgramSpecifications()){
      System.out.println(spec);
   }
   System.out.println();

   eventMill.instanciate(module, instance, schemas);

   //list available program instances
   System.out.println("Available program instances:");
   for(String spec: eventMill.availablePrograms()){
      System.out.println(spec);
   }
   System.out.println();
```

```
//Obtain program instance
ExecutableProgram<?, ?, ?> executableProgram = eventMill.getProg(
   module, instance);

//Create engine for program instance
Engine engine = new Engine(executableProgram);


//Connect Engine to message bus

IOConfiguration ioConfiguration;
try {
   ioConfiguration = IOConfiguration.read();
} catch (IOException e) {
   throw new RuntimeException(e);
}

String engineIdentifier = "de.lmu.ifi.pms.cep.event-mill.prototype";

String programInstanceIdentifier = PATTERNS.programIdentifier(
   engineIdentifier, instanceFullName);

StreamIO streamIO = ioConfiguration.stream(engineIdentifier,
   programInstanceIdentifier);

String programInstanceBusEndpoint = programInstanceIdentifier;
final Class<String> messageType = String.class;

final String inputBusIdentifier = PATTERNS.inputBusIdentifier(
   engineIdentifier, instanceFullName);
String inputMessageFormatIdentifier =
   SemiStructuredMessageFormatProvider.defaultIdentifier;
MessageBus<String> inputBus = ioConfiguration.bus(inputBusIdentifier,
    messageType, inputMessageFormatIdentifier,
   programInstanceBusEndpoint);

final String outputBusIdentifier = PATTERNS.outputBusIdentifier(
   engineIdentifier, instanceFullName);
String outputMessageFormatIdentifier = XmlMessageFormatProvider.
   defaultIdentifier;
MessageBus<String> outputBus = ioConfiguration.bus(
   outputBusIdentifier, messageType, outputMessageFormatIdentifier,
   programInstanceBusEndpoint);

final BusToStreamConnector<?> inputConnector = BusToStreamConnector.
   create(inputBus, streamIO);
final StreamToBusConnector<?> outputConnector = StreamToBusConnector.
   create(outputBus, streamIO);

Thread ioThread = new Thread("IO-Thread"){
        @Override
        public void run() {
            while(!this.isInterrupted()){
                inputConnector.nextBlock();
                outputConnector.nextBlock();
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    break;
                }
            }

        }

    };

    //Thread for providing input
    Thread simInputThread = new Thread("Emulation"){

        @Override
        public void run() {

            String emulationBusEndPoint = "de.lmu.ifi.pms.cep.event_mill.
                sample.emulation";
            String emulationMessageFormatIdentifier =
                SemiStructuredMessageFormatProvider.defaultIdentifier;
            MessageBus<String> emulationOutputBus;

            try {

                emulationOutputBus = IOConfiguration.read().bus(
                    inputBusIdentifier, messageType,
                    emulationMessageFormatIdentifier, emulationBusEndPoint);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }

            MessageSink<String> messageSink = emulationOutputBus.sink();

            ArrayList<String> messages = new ArrayList<String>();

            int round = 1;
            while(!this.isInterrupted()){
                messages.add("hello-world.see-person{ person{ \"boy_" +
                    round + "\" } }");
                messages.add("hello-world.see-person{ person{ \"girl_" +
                    round + "\" } }");

                messageSink.writeBlock(messages);
            messages.clear();

            try {
               Thread.sleep(1000);
            } catch (InterruptedException e) {
               break;
            }
            ++round;
         }
      }

   };

   //Thread for reading output
   Thread uiThread = new Thread("UI"){

      @Override
      public void run() {

         String uiBusEndPoint = "de.lmu.ifi.pms.cep.event_mill.sample.ui
            ";
         String uiMessageFormatIdentifier =
            SemiStructuredMessageFormatProvider.defaultIdentifier;
         MessageBus<String> uiInputBus;

         try {
            uiInputBus = IOConfiguration.read().bus(outputBusIdentifier,
                messageType, uiMessageFormatIdentifier, uiBusEndPoint);
         } catch (IOException e) {
            throw new RuntimeException(e);
         }

         MessageSource<String> messageSource = uiInputBus.source();

         while(!this.isInterrupted()){
            for(String message: messageSource.readBlock()){
               System.out.println(message);
            }

            try {
               Thread.sleep(1000);
            } catch (InterruptedException e) {
               break;
            }
         }
      }

   };

   //initialize engine
```

```
    SchedulerFactory scheduler = eventMill.configuration().scheduler;
    //retrieving the scheduler from the configuration of the Event-Mill
        instance

    engine.init(scheduler);
    //initializing the engine instance for the given program with the
        assigned scheduler

    //start program

    engine.start();

    //start IO
    simInputThread.start();
    ioThread.start();
    uiThread.start();

    //wait for user input
    System.out.println("Press [Enter] to stop execution.");
    System.in.read();

    //stop IO
    simInputThread.interrupt();
    simInputThread.join();
    ioThread.interrupt();
    ioThread.join();
    uiThread.interrupt();
    uiThread.join();

    //stop program
    engine.stop();

    //release resources
    engine.close();

    //remove program instance
    eventMill.removeProg(module, instance);

    //remove program specification
    eventMill.removeSpec(module);

    eventMill.close();
}

}
```

7. The Access Control Example

7.1. Description

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

1. If a person requests access and the person is not a member of the staff, then the access is denied.
2. If a person requests access and the person is a member of the staff, but requested access shortly before then the access is denied.
3. If a person requests access and the person is a member of the staff, but requested access for more than tree time within a short period of time then the access is denied.
4. If a person requests access and there is no reason to deny access, then the person is granted access.
5. If a person requests access and belongs to the staff, but the access is denied, then an intrusion warning is raised.

```
MODULE access-control

DESCRIPTION "Access control example."

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

static input
STATEFUL OBJECT
   staff{ person{string} }
END

output log
EVENT
   deny-access{ person{string} }
WITH

   DETECT
      deny-access{ person{var P} }
   ON
      and{
         event e: request-access{ person{var P} },
         not{state s: staff { person{var P} } }
      } where {state s valid-at end(event e) }
   END
   DETECT
      deny-access{ person{ var P} }
   ON
      and{
         event e : request-access{ person{ var P } },
         state s : staff{ person{ var P} },
         event f : request-access{ person{ var P } }
      } //where { state s valid-at end(event e), event f before event e,
           {e , f} within 5 sec}
        where { state s valid-at end(event e), event f before event e,
            end(e) <= begin(f) + 5 sec}
   END

   DETECT
      deny-access{ person{ var P} }
   ON
      and{
         event e: request-access{ person{ var P} },
         state s: staff{ person{ var P} },
         event f: request-access{ person{ var P} }
      } //where {state s valid-at end(event e), end(f) <= end(e), {e, f}
          within 30 sec}
        where {state s valid-at end(event e), end(f) <= end(e), end(e)
           <= begin(f) + 30 sec}
        group by {event e, state s} aggregate {var C = count(event f)}
        where {var C >= 3}
   END

END

output log
EVENT
   grant-access{ person{string} }
WITH

   DETECT
      grant-access{ person{ var P} }
   ON
      and{
         event e: request-access{ person{ var P} },
         not{event f: deny-access{ person{ var P} }}
      } //where {end(f) <= end(e), {e, f} within 5 sec}
        where {end(f) <= end(e), end(e) <= begin(f) + 5 sec}
   END

END

output log
EVENT
   intrusion-warning{ person{string} }
WITH
```

```
    DETECT
        intrusion-warning{ person{ var P} }
    ON
        and{
            event f: deny-access{ person{ var P} },
            state s: staff{ person{ var P} }
        }
    END

END
```

7.2. 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_block" and "person". The insertion *must not* set the "_o_block" attribute.

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.

7.3. Output

The derived grant-access events are stored in the table with name `OUTPUT_SCHEMA."access_d_control_o_grant_d_access"`.

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."access_d_control_o_grant_d_access"` buffer/table for the grant-access events to the corresponding `OUTPUT_SCHEMA."access_d_control_o_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" = 'access-control.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."access_d_control_o_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."access_d_control_o_intrusion_d_warning"`.

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

7.4. Test Sequence

```
set %{source} = 'examples/access-control.dura'
set %{examples.dir} = 'examples'
set %{namespace} = ''

compile %{namespace}%{name}, %{result}, %{source}

remove prog %{namespace}%{name}:%{instance}
remove spec %{namespace}%{name}

load spec %{result}
list specs

init %{namespace}%{name}:%{instance}, %{meta-schema}, %{
   input-schema}, %{working-schema}, %{output-schema}, %{
   log-schema}
list progs

run %{namespace}%{name}:%{instance}

parse access-control.staff{ person{ "staff_1" } }
parse access-control.staff{ person{ "staff_2" } }
parse access-control.staff{ person{ "staff_3" } }
parse access-control.staff{ person{ "staff_4" } }
write

init
start

parse access-control.request-access{ person{ "staff_1" } }
parse access-control.request-access{ person{ "staff_2" } }
parse access-control.request-access{ person{ "visitor_1" } }
write
```

```
pause 1500

read
print

parse access-control.request-access{ person{ "staff_1" } }
parse access-control.request-access{ person{ "staff_3" } }
parse access-control.request-access{ person{ "staff_4" } }
write

pause 1500

read
print

pause 10000

parse access-control.request-access{ person{ "staff_1" } }
parse access-control.request-access{ person{ "staff_3" } }
write

pause 1500

read
print

pause 1500
read
print

pause 500

stop

quit
quit
```

Part II. Building Blocks

8. Overview

The compilation and evaluation of a Dura program involves several implementation layers where each layer takes specific tasks. This section gives a brief overview on the different building blocks and their dependencies.

Dura Dura is the high-level event, state and action language designed in WP4. Dura aims at multiple goals that are important for emergency management:

- ease-of-use
- high-expressivity
- intuitive concepts and syntax
- precise declarative semantics

These goals are discussed more deeply in the Deliverables D4.2, D4.3 and D4.6 [5, 6, 8]. To meet these goals Dura need to offer a flexible syntax that offers different concepts like events, states and actions, different kinds of rules like complex event rules, event-condition-action rules and complex action specifications and numerous syntactic abbreviations for common tasks, usually referred to as syntactic sugar.

Temporal Stream Algebra – TSA Dura programs are translated into Temporal Stream Algebra - TSA. (See Section 10 for an example.) TSA plays the same role for Dura as relational Algebra plays for SQL. Temporal Stream Algebra (TSA) offers a single common concept, temporal streams, to represent the events, states and actions defined in Dura. Furthermore TSA defines a small set of simple operators that nevertheless have the high expressivity required for the translation of the Dura rules. The use of a common concept for events, states and actions and of a small set of simple operators is essential for the in-depth analysis of the temporal relations defined in the Dura rules. Actually this in-depth analysis of the temporal relations is the core reason for the high expressivity that is achievable by Dura and TSA. Moreover the TSA layer is particularly suited for optimizing the compiled program. For details see Section 9.

Differentiated TSA A previous TSA layer breaks a the complex definitions of a Dura program into simple operations. However it does not yet tackle the incremental evaluation of the program. TSA expressions conceptually take an omniscient view, i.e. assume that the whole input streams are known, and derive all consequences at once. This view is not realistic, though. In practice data continuously arrives over time and the consequences of the arriving data should be derived continuously, too. Therefore the TSA expressions from the previous layer need to be transformed into expressions that allow to derive the new consequences resulting from the new data that arrived since the last evaluation step incrementally. This transformation from “omniscient" to incremental TSA expressions bases on the results of the temporal analysis. As the incremental expressions compute the “delta” between the old and the new consequences, they are also called “differentiated TSA expressions, and the transformation process is refereed to as “differentiating” TSA expressions. For details see Section 11.

Parameterized SQL expressions The differentiated are finally translated to parametrized SQL queries which are executable on the MonetDB database. The translation is simple as the TSA operators within a differentiated TSA expression, can directly be interpretated as operators of relational algebra. Note that the translation can be done once at compile time. The only changes for the subsequent steps of the incremental evaluation are the values of the parameters within the SQL expressions. The corresponding functions for computing these values of the parameters are an outcome of the TSA differentiation process in the previous layer.

Event-Mill run-time on MonetDB The actual evaluation of the program against the data/event streams is done by the run-time of the Event-Mill engine on top of MonetDB. The main task of the Event-Mill engine is

The run-time of the Event-Mill engine is not required to see the processed data. The run-time only needs to execute the parametrized SQL expressions from the previous layer in the right way. The execution of these SQL expressions then generates the expected results directly in the MonetDB database. This approach minimizes the amount of data that needs to transferred from and to the MonetDB database. MonetDB stores all data and all computations on that data are also done within MonetDB.

9. Temporal Stream Algebra – TSA

9.1. Introduction

The Temporal Stream Algebra is designed to meet the requirements of detecting and managing emergencies in large infrastructures like metro system, airports or power grids [23, 24, 25]. The requirements from these use-cases are discussed Section 9.2.

TSA contributes the following:

1. A common data model for data streams and database relations
2. An algebra of operators for querying data streams and database relations

Salient properties of TSA are:

- Timestamps after different time lines/time-semantics
- Expressive temporal relations
- Rich negation, grouping and aggregation

The section on the Temporal Stream Algebra (TSA) is structured as follows: Section 9.2 the requirements of the use-cases specific to TSA and motivates our approach using an example from the metro use-case. Section 9.3 introduces temporal streams and the common data model for data streams and database relations. Section 9.6 defines the operators of Temporal Stream Algebra (TSA). Section 9.4 explains how propagated constraints on temporal relations can be used to decide whether an operator application or TSA expression respectively is blocking or not. Section 11 describes the incremental evaluation of TSA.

9.2. Motivation

Combined data stream and database queries. Use cases from emergency management need combined queries to data stream and database data[1] [23, 24, 25]. Consider a train in which a fire has broken out and that has to stop inside a metro station. Unfortunately the fire is close to one staircase which therefore is likely to quickly fill with smoke and should not be used for evacuation. For correctly assessing the situation and choosing an appropriate reaction, the system needs to combine the alarm events from smoke and temperature sensors with static data about the location of sensors and staircases. This way the system can conclude that one of the staircases being too close to the fire should not be used for evacuation as it will be impassable due to smoke shortly. Since persistent data are required for interpreting the volatile data arriving on the stream, combined data stream and database queries are needed.[2]

Declarativity. We argue that a high-level user-language for combined data stream and database queries should be declarative. A declarative language has a number of well-known advantages, like ease of programming and clear and (relatively) comprehensible semantics. This is particularly important for emergency management where rules have to be written (or at least have to be verified) by security experts which typically have limited programming skills. Furthermore emergency management requires predicable results, thus a clear semantic is mandatory. The work presented in this document is part of the implementation of the event, state and action language Dura [6, 7, 8, 16].

Relational Algebra. For queries to database relations declarative query languages, particularly SQL, are widely used. Their system-level counterpart is Relational Algebra [1, 15]. Temporal Stream Algebra (TSA) generalizes the data model and operators of Relational Algebra to apply to both data streams and database relations. Such a generalization is by no means trivial and, so far, has not been proposed. Preserving the data model and the operators of Relational Algebra has at least two advantages: First the properties of the operators are well understood. For example the laws on operator permutations are very important for the optimization of Relational Algebra expressions. Second, there exist reliable techniques for the evaluation and optimization of Relational Algebra expressions, like specialized join algorithms, heuristics for operator reordering or cost-models, that base on these laws and on other properties of the operators of Relational Algebra. As TSA preserves the operators and their characteristics these techniques

[1]Within this article the terms “database data” or “database relation “ refer to static data as opposed to stream data.

[2] Persistent data is likely to change less frequently than the streaming data itself, but does not have to be completely static. E.g. sensor locations may change, as for instance sensors in a moving train. Using TSA it is possible to handle dynamically changing persistent data in a semantically precise manner (i. e. without race conditions). This also holds for the incremental evaluation of TSA (Section 11). However details on this point it are out of the scope of this article.

and algorithms can be reused or easily adopted for TSA. Third, preserving the operators of Relational Algebra for combined data stream and database queries means that the representation of database queries does not change at all and allows to query data streams in the same way that is already familiar from database relations.

Blocking Operators. Some operators of Relational Algebra, the so called “blocking operators”, like set difference or grouping and aggregation, in general need to process the complete input before the may produce the correct results. For a data stream the complete input is only known at the end of the stream, which in the case of an CEP application is only reached when the application is terminated. However CEP queries are required to deliver results as the data arrives on the stream and not when the application is terminated. Therefore the use of “blocking operators” within CEP queries must be restricted to situations where each result only depends on a limited section of the stream and thus can be produced before the complete stream is known.

Previous approaches [13, 20], including those not related to Relational Algebra [11], tackle this problem on a “syntactical” level. They basically restrict operators in such a way that they may not lead to potentially blocking query expressions. In other words, the “syntax” does not allow to write blocking expressions. However solving this “semantic” problem on a “syntactic” level puts unnecessary limits to expressivity. By contrast TSA does not prevent blocking query expressions on the “syntactical” level. Instead TSA uses a “semantic" analysis of the temporal relations specified in the query expression, to determine whether the expression is blocking or not (See Sections 9.6 and 9.4 for details).

Common Data Model. Keeping the operators of Relational Algebra calls for a common data model for data streams and database relations as the same operators must work on both data sources. Using common operators for both data sources has two advantages: The number of operators is kept low and the common operators allow arbitrary combinations of both stream and database data. Note that a common data model for stream and database data should incorporate constraints on temporal relations providing the information that the temporal analysis needs to determine the validity of an query expression with respect to “blocking” or “non-blocking”. Thus the data model is a key element of TSA (see Section 9.3).

Multiple Time Lines. Modeling the emergency management use cases showed that a single, predefined time model (regardless of being based on time points or intervals) is insufficient. In emergency management even atomic events may refer to at least three times. The first time is the one at which an event, e. g. sensor message, is emitted. This time is known as application time. The second time is set when the event message is received by the CEP system. This time is known as system time. The third time is the time simulated data refer to. Simulated events are used for predictions on the future development of an emergency. For simulated events the time where the underlying data is available (application/system time) differs from the future time for which the event makes a prediction. Whether these times are modeled as time point or intervals depends on their usage and both, the common data model and the algebra proposed in this article, leave this decision open.

Supporting multiple time lines requires more than having multiple timestamp attributes. For example different time lines impose different orders on the events. An in-order processing is therefore only possible for at most one of the time lines. For all other time lines the evaluation is inevitably out-of-order. However previous algebraic approaches [13, 20] assume in-order processing and current DSMS offer only limited support for out-of-order processing, and thus favor a single time line. By contrast TSA treats all time lines equally and imposes no limitation on their number. The presented incremental evaluation of TSA does a bulk-wise out-of-order processing for all time lines (Section 11).

Negation, Grouping and Aggregation. Finally emergency management needs flexible negation and grouping and aggregation that can be controlled, i. e. started and stopped, by events. The “event-controlled” grouping and aggregation is required for queries like “Count the number of people that have left the building from the detection of the fire to the arrival of the fire-brigade”. Similar examples exist for negation. Negation or grouping and aggregation fixed-sized time-windows are too limited.

9.3. Data Model: Temporal Streams

The common data model for data streams and database relations is an essential part of this TSA. Temporal streams generalize the concept of (finite) relations from Relation Algebra so as to include data streams as well. The basic idea behind the definition is that streams may have potentially infinite size, however they are expected to have only a finite history at each point of time, i. e. up to some point of time only a finite amount of data may arrive on the stream.

Definition 1. (Temporal Stream)
A *temporal stream* is a (possibly infinite) relation R with attribute schema $\mathscr{A}(R) \subseteq ATTR$ and timestamp attributes $\mathscr{A}_{temp}(R) = \{p_1, \dots, p_k\}$ that has a finite history at each point in time. I. e. for each point in time $s_1, \dots, s_k \in \mathbb{Q}$ [3] the number of tuples $r \in R$ occurring before that point in time is finite:

$$|\{r \in R \mid r(p_1) \leq s_1 \wedge \cdots \wedge r(p_k) \leq s_k\}| < \infty$$

where $ATTR$ is a set of attribute names. For data streams $\mathscr{A}_{temp}(R)$ contains at least one timestamp attribute. For ordinary (finite) database relation $\mathscr{A}_{temp}(R)$ may be empty. As database relations are always finite they trivially fulfill the condition formulated above.

The definition of temporal streams bases on the observation that each data item in a data stream is carrying temporal information, i.e. timestamps, that is correlated with the sequence order of the data items in the stream.[4] The timestamps may refer to different time-models, none of the timestamps might define a total order on the incoming events and events may arrive out-of-order with respect to each of the timestamps. However as long as the timestamps refer to clocks that all increase over time, i.e. time does not stop for any of the clocks, also the minimum value for

[3] As usually $\mathbb{Q}$ denotes the set of relational numbers.

[4] Even the sequence numbers themselves form a kind of timestamp, admittedly with respect to a quite unusual time model. TSA is able to manage even this special case.

each timestamp of any future event increases over time.[5] In other words, as more and more data of the stream arrives we observe temporal progress with respect to each of the timestamps. TSA is very flexible with respect to the used time models. TSA only requires a total order on the domain of a time model,[6] and supports the simultaneous use of multiple different time models. However, for sake of simplicity and without loss of generality this document only uses a single time model, namely $\mathbb{Q}$.[3]

In Relational Algebra relations are always finite. In some way data streams, and thus the corresponding relations, are finite, too. The reason is, that no application will run forever and so only a finite amount of data may arrive on the stream. However this view is misleading as it wrongly suggests that Relational Algebra could be applied to data streams just as it is (See "Blocking Operators" in Section 9.2).

To stress the fact that we must not wait for the end of a data stream,[7] TSA models data streams as potentially infinite relations adopting the idea in [13, 12]. The payload of each data item including the associated timestamps are stored in attributes of the corresponding type. However those timestamps that are correlated to the sequence order,[5] i.e. refer to the temporal progress of the stream, play special role in the validation and the evaluation of TSA expressions. A so-called "stream bound formula" within the schema remembers the special role of these timestamps (See Definitions 4 and 5).

9.4. Temporal Relations

Temporal relations are an essential part of expressive data stream queries. The temporal relations specified in a query determine whether this query can be evaluated incrementally, i.e. is valid, or not. Their important role is further discussed in Section 9.6 about the TSA operator. Actually the propagation of the temporal relations through the TSA operators and the analysis of the propagated temporal relations as basis for the incremental evaluation is one of the key innovations of TSA. However the definition of the TSA operators require some prior definitions on the terms and formulas for representing these conditions.

Temporal terms are used to specify new timestamps relative to existing timestamps. This is for example needed when defining the timestamps p of a composite event (see Definition 8). Assume that the composite event is composed from two events with timestamps p_1 and p_2. A usual definition for the timestamp p of the composite event would be $p = \max\{p_1, p_2\}$. Temporal terms are also used when establishing a temporal relation, that is not just "equal", between two existing timestamps using the selection operator (see Definition 7). For example if a timestamp p_1 should occur at most 2 seconds after timestamp p_2 then this would translate to $p_1 <= p_2 + 2sec$ where $p_2 + 2sec$ is a temporal term defining a new timestamp relative to p_2. To

[5]The data items may also contain timestamps, that are not correlated to the sequence order, i.e. are not related to the progress of time or the availability of data. Particularly all timestamps of database relations fall into this category. Such timestamps are treated as ordinary data.

[6]This does not impose a total order on the data items.

[7]This implies the need for an incremental evaluation

the best of our knowledge temporal terms allow to realize any of the usual temporal semantics for timestamps (also intervals) of composite events [17, 3, 13, 4].

Definition 2. (Temporal Terms)
The set of *temporal terms* is defined inductively:

1. $v \in ATTR \cup VAR$ is an atomic temporal term
2. $t + c$ is a temporal term
   iff t is a temporal term and $c \in \mathbb{Q}$
3. $min\{t_1, \ldots, t_k\}$, $max\{t_1, \ldots, t_k\}$ are temporal terms
   iff $t_1, \ldots, t_k$ are temporal terms

where *VAR* is a set of temporal variables, $VAR \cap ATTR = \emptyset$.

Temporal relation formulas (TRFs) describe temporal relations between temporal terms, particularly between timestamp attributes and/or variables. Variables are basically required for handling relations on attributes that are discarded by the projection or the grouping operator of TSA (See Definitions 9,10). Temporal relations in TSA are based on timestamps (in contrast to intervals). However time intervals can be defined using two time stamps and a TRF stating that the first timestamp is smaller then the second. Furthermore as TRFs enable conjunctions and disjunctions[8] all 2^{13} interval relations of Allen's interval algebra [2] can be expressed.

Definition 3. (Temporal Relation Formulas)
The set of *temporal relation formulas* (TRFs) is defined inductively:

1. $\top$ and $\bot$ are atomic TRFs
2. $t_1 \,\mathsf{op}\, t_2$ is an atomic TRF for $\mathsf{op} \in \{<, \leq, =, \geq, >, \neq\}$
   iff t_1 and t_2 are temporal terms
3. $G \wedge G'$, $G \vee G'$ and $\neg G$ are TRFs iff G, G' are TRFs

TRFs store the information about temporal relations between timestamp attributes. When using multiple timestamp attributes there shows another effect which is not covered by TRFs, though. The definition of temporal streams (Def. 1) requires that upper limits to the values of all timestamp attributes of a temporal relation result in a finite prefix of the stream. However, if input streams carry more than one timestamp then it often suffices to limit a subset of the timestamp attributes of the temporal stream to obtain a finite prefix. Consider for example, an input stream with application- and system-time timestamps. An upper limit for the values of one of the two timestamps suffice to obtain a finite prefix of the input stream. If two input streams of this kind are combined by the cross-product operator then any subset of timestamp attributes containing at least one timestamp attribute from each stream, can be used to get an finite prefix of the result stream. The purpose of stream bound formulas (SBFs) is to propagate the information which combinations of timestamps are suited for obtaining a finite prefix of a temporal stream. The SBF for a (static) database relation may just be $\top$ as we do not need to obtain a finite prefix in that case.

[8]By contrast Point Algebra [21] allows only conjunctions.

Definition 4. (Stream Bound Formulas) The set of *stream bound formulas* (SBFs) is defined inductively:

1. $\top$ is an atomic SBF
2. $bounded(v,b)$ is an atomic SBF for $v \in ATTR \cup VAR$ where $b \in BOUND$ is a stream bound identifier.
3. $H_1 \wedge H_2$ and $H_1 \vee H_2$ are SBFs iff H_1 and H_2 are SBFs.

Definition 4 assigns so-called "stream bound identifiers" to the atoms of a SBF. As these identifiers do not change during the propagation process (in contrast to the names of attributes and variables) the identifiers allow to identify the input relation that a specific atom in a SBF originates from. This way the temporal analysis of a TSA query expression that is needed for validation and incremental evaluation may base solely on the propagated formulas and does not need to analyze the TSA query expression recursively.

9.5. Temporal Stream Schema

The schema of a relation in Relational Algebra is just its set of attributes. For temporal streams the attribute schema is accompanied by a temporal relation formula and a stream bound formula. The temporal relation formula (Definition 3) describes the temporal relations between the timestamp attributes. The stream bound formula (Definition 4) tells about the ability of timestamp attributes to obtain a finite prefix of a temporal stream matching the schema. Basically a timestamp attribute is able to yield a finite prefix of a temporal stream if it is correlated to the temporal progress of the stream as described above. The two formulas carry the necessary information for the validation and evaluation of a TSA expressions (See Section 9.4).

Definition 5. (Temporal Stream Schema)

1. A *temporal stream schema* is a triple $S = (A,G,H)$ such that A is an attribute schema, $A_{temp} \subseteq A$ is the set of temporal attributes contained in A , G is a temporal relation formula and H is a stream bound formula and all attributes occurring in G and H are in the attribute schema (i.e. $attr(G) \subseteq A_{temp}$ and $attr(H) \subseteq A_{temp}$).
2. A temporal stream schema $S = (A,G,H)$ is *valid* if the set of all timestamp attributes A_{temp} is a stream bound with respect to G and H (See Definition 17).
3. A relation R matches a temporal stream schema $S = (A,G,H)$ if R has attribute schema $\mathscr{A}(R) = A$ and both G and H hold in R (See Definitions 15,17). If S is valid, the latter implies that R is a temporal stream.

9.6. Temporal Stream Definitions and TSA Operators

The common operators for queries to data streams and database relations are the second essential element of TSA. TSA generalizes the operators of Relational Algebra without changing their definition of the result sets. As mentioned in Section 9.2 this introduces the problem of "blocking" operators and query expressions. It is a fundamental observation, that the answer to the question whether a certain query expression is non-blocking, depends on the temporal relations between timestamp attributes imposed by the input streams and the query expression itself. In Relational Algebra this information is not propagated, though. Thus subsequent operators may not rely on the information for determining whether they can be applied correctly.

The solution of this problem (and one of the central ideas of TSA) is the propagation of constraints on temporal relations inside the schema of TSA expressions. The propagation of the "temporal relation formulas" (Definition 3) and of the "stream bound formulas" (SBFs, Definition 4) contained in the schema of TSA expressions is therefore the most important part of the TSA operator definitions.

Within a TSA program, temporal relations are established in several ways. First by constraints that are part of the schema of an temporal stream definitions. Temporal stream definitions are the starting point for all TSA expressions.

Definition 6. (Temporal Stream Definition)
A *temporal stream definition* is a pair $D = (n,S)$ where $n \in STREAM$ is a name for the temporal stream definition, $S = (A,G,H)$ is a temporal stream schema, $STREAM$ is a set of names for temporal stream definitions, $A_{temp} \subseteq A$ is the set of temporal attributes in A and the following restrictions hold:

1. G and H do not contain variables
2. H is a disjunction of atomic SBFs, i. e. has the structure

   $bounded(p_1,b_1) \vee \ldots \vee bounded(p_k,b_k)$

   where $p_1,\ldots,p_k \in A$ and $b_1,\ldots,b_k \in BOUND$
3. The atomic SBFs $bounded(p_i,b_i)$ in H define an injection $b^D : A_{temp} \to BOUND$ with inverse p^D.

The first restriction allows to validate whether the schema of a query expressions matches the schema of the output stream. The third restriction is merely technical, it is important when generating the necessary formulas and functions for the incremental evaluation. The second restriction in the above definition expresses that each stream bounds (see Definition 17) of a temporal stream definitions must contain a stream bound that consists only of a single timestamp attribute.[9] For example, if you would like to express that both system- (p_{sys}) and application-time (p_{app}) are stream bounds of a stream $D = (n,S)$ then the stream bound formula H of schema S would look like this:

$$H = bounded(sys, n{:}sys) \vee bounded(app, n{:}app)$$

All TSA expressions E have an associated schema $\mathscr{S}(E)$. The schema of a temporal stream definition $D = (n,S)$ is $\mathscr{S}(D) = S$. The schema $\mathscr{S}(E)$ of composite TSA expressions is defined inductively in the definitions of the TSA operators.

[9] Stream bounds with multiple elements that do not fulfill this condition are allowed for the schema of TSA expressions. They result from the application of binary operators like cross-product, set-difference or union.

Definition 7 shows that temporal relations may also be derived from conditions on timestamp attributes imposed by the selection operator. The definition of the resulting relation/temporal stream is the same as for Relational Algebra.

Definition 7. (Selection σ) Let E be a TSA expression with schema $\mathscr{S}(E) = (A, G, H)$ and let R be a temporal stream which matches the schema of E and C a set of conditions with $domain(C) \subseteq A$.

$$\sigma[C](R) := \{r \in R \mid r \text{ satisfies } C\}$$
$$\mathscr{S}(\sigma[C](E)) := (A, (G \wedge C_{temp}), H')$$

where C_{temp} is the TRF that results from C when replacing every non-temporal atom by $\top$ if it occurs with positive polarity or by $\perp$ if it occurs with negative polarity.

The definition of C_{temp} extracts the maximum temporal information from the condition C. Basically C_{temp} results from C by replacing all non-temporal atoms in such a way that the condition C is fulfilled as much as possible.

Finally temporal information is introduced by the definition of new timestamp attributes relative to existing ones. At that point the presentation of the TSA operators slightly differs from the usual presentation of the operators of Relational Algebra. The usual projection operator of Relation Algebra allows both discarding attributes and defining new attributes based on existing ones. In TSA we split these two tasks into two operators: The projection operator of TSA which only discards attributes and the imbed operator which allows to define new attributes. These changes in the presentation help to keep the definitions simple, as each definitions only cares for a single aspect of the propagation of temporal relation formulas and stream bound formulas. The following definition uses the notion of "temporal terms" that are formally introduced in definition 2. Temporal terms are a powerful mean to define relative timestamps.

Definition 8. (Imbed ι)

Let E be a TSA expression with schema $\mathscr{S}(E) = (A, G, H)$ and let R be a temporal stream which matches the schema of E. Let $a' \notin A$ be a new attribute, t the term defining a' and $a_1, \ldots, a_k \in A$ the attributes occurring in t.

$$\begin{aligned}
\iota[a' = t](R) := \{r' \in dom(A \cup \{a'\}) \mid \exists r \in R \ & \text{such that} \\
& r'(a) = r(a) \text{ for } a \in A \text{ and} \\
& r'(a') = f_t(r(a_1), \ldots, r(a_k))\} \\
\mathscr{S}(\iota[a' = t](E)) := (A \cup \{a\}, G', H) & \\
G' := \begin{cases} G \wedge (a' = t) & \text{if } t \text{ is a temporal term} \\ G & \text{else} \end{cases} &
\end{aligned}$$

where $f_t : dom(a_1) \times \ldots \times dom(a_k) \to dom(a')$ is the function of the values of $a_1, \ldots, a_k \in A$ defined by t. If t is a temporal term, then t defines a relative timestamp.

The main point of the above definition is, that the definitions of relative timestamp attributes are preserved within the temporal relation formula G' of the schema.

For projection and grouping the definition of the resulting temporal stream is almost the same as for Relational Algebra. The only difference is that the projection may not introduce new attributes. For both operators the temporal relation formula and the stream bound formula are propagated in the same way: The discarded timestamp attributes are replaced by temporal variables. The simple definition of grouping might be surprising, as grouping is one of the "blocking" operator which are usually considered as "problematic". But as TSA solves the problem of blocking query expressions on the basis of the propagated temporal constraints and not on the level of the operators, the definition of the grouping operator can actually be straightforward.

Definition 9. **(Projection π)** Let E be a TSA expression with schema $\mathscr{S}(E) = (A, G, H)$ and let R be a temporal stream which matches the schema of E and $A_1 \subseteq A$ the set of retained attributes.

$$\begin{aligned}
\pi[A_1](R) &:= \{r' \in dom(A_1) \mid \exists r \in R \text{ such that} \\
&\qquad\qquad\qquad\qquad r'(a) = r(a) \text{ for } a \in A_1\,\} \\
\mathscr{S}(\pi[A_1](E)) &:= (A_1, \xi(G), \xi(H))
\end{aligned}$$

where $A_{temp} \subseteq A$ and $A_{1temp} \subseteq A_1$ are the sets of temporal attributes in A and A_1, respectively, and ξ is an injective substitution of the discarded timestamp attributes in $A_{temp} \setminus A_{1temp}$ by variables that do not occur in G or H.

Definition 10. **(Grouping γ)** Let E be a TSA expression with schema $\mathscr{S}(E) = (A, G, H)$ and let R be a temporal stream which matches the schema of E and $A_1 \subseteq A$ the set of grouping attributes. Let $a_1, \ldots a_k \in A \setminus A_1$ and $a'_1, \ldots, a'_k \notin A_1$ and $F_1, \ldots, F_k$ aggregation functions like min, max, sum or avg.

$$\begin{aligned}
\gamma[A_1][a'_1 = F_1(a_1), \ldots, a'_k = F_1(a_k)](R) &:= \{r' \in dom(A_1 \cup \{a_1, \ldots, a_k\}) \mid \exists r \in R \\
&\qquad \text{such that } r'(a) = r(a) \text{ for } a \in A_1 \\
&\qquad \text{and } r'(a'_i) = F_i((r_\gamma(a_i))_{r_\gamma \in R_r}) \quad \} \\
\mathscr{S}(\gamma[A_1][a'_1 = F_1(a_1), \ldots, a'_k = F_1(a_k)](E)) &:= (A_1, \xi(G), \xi(H))
\end{aligned}$$

where $R_r = \{r_\gamma \in R \mid r_\gamma(a) = r(a) \text{ for } a \in A_1\}$ and ξ is an injective substitution of the discarded timestamp attributes $A_{temp} \setminus A_{1temp}$ by variables that do not occur in G or H.

The next definition is the second deviation from the usual presentation of operators in Relation Algebra. The basic TSA operators do not include a join but a cross-product operator.Thus using the basic operators of TSA, a join must be expressed as a combination of cross-product and selection. Again this change helps to keep the definitions simple.

However omitting the join as basic operator does not restrict the expressivity or efficiency of TSA in any way. Further operators, like the usual projection, renaming, join, semi-join and anti-join, can easily be added to TSA. Such easy extensions are not described in this article for pace reasons. All these operators can be expressed as combinations of the basic operators presented in this document. Sections 9.7 and 11 show that the incremental evaluation of TSA solely depends on the correct propagation of the temporal constraints, but is independent from the concrete operators contained in an TSA expression. Thus, defining a composite operator with respect to semantics, constraint propagation and incremental evaluation is just done by providing its decomposed representation by means of basic operators.

Definition 11. (Cross Product $\times$) Let E_1 and E_2 be TSA expressions with schema $\mathscr{S}(E_1) = (A_1, G_1, H_1)$ and $\mathscr{S}(E_2) = (A_2, G_2, H_2)$ that have disjoint attributes ($A_1 \cap A_2 = \emptyset$) and let R_1 and R_2 be temporal streams which match the schema of E_1 and E_2 respectively.

$$\begin{aligned} R_1 \times R_2 &:= \{r' \in dom(A_1 \cup A_2) \mid \exists r_1 \in R_1, r_2 \in R_2 \text{ with} \\ &\qquad\qquad r'(a) = r_1(a) \text{ for } a \in A_1 \\ &\qquad\qquad r'(a) = r_2(a) \text{ for } a \in A_2\} \\ \mathscr{S}(E_1 \times E_2) &:= (A_1 \cup A_2, \xi_1(G_1) \wedge \xi_2(G_2), \xi_1(H_1) \wedge \xi_2(H_2)) \end{aligned}$$

where ξ_1 and ξ_2 are injective substitutions, such that the $\xi_1(G_1)$, $\xi_1(H_1)$ and $\xi_2(G_2)$, $\xi_2(H_2)$ use disjunct sets of temporal variables.

The definition of the cross-product operator is straight-forward. The renaming of the temporal variable avoids unintended interferences between the temporal relation formulas and stream bound formulas of the two subexpressions.

Definition 12. (Union $\cup$) Let E_1 and E_2 be TSA expressions with schema $\mathscr{S}(E_1) = (A, G_1, H_1)$ and $\mathscr{S}(E_2) = (A, G_2, H_2)$ that have the same attributes and let R_1 and R_2 be temporal streams which match the schema of E_1 and E_2 respectively.

$$\begin{aligned} R_1 \cup R_2 &:= \{r \in dom(A) \mid \text{with } r \in R_1 \text{ or } r \in R_2\} \\ \mathscr{S}(E_1 \cup E_2) &:= (A, G', (\xi_1(H_1) \wedge \xi_2(H_2))) \end{aligned}$$

$$\begin{aligned} G' &:= (G'_1 \wedge G''_1 \wedge \xi_1(G_1)) \vee (G'_2 \wedge G''_2 \wedge \xi_2(G_2)) \\ G'_1 &:= (\xi_1(p_1) = p_1) \wedge \ldots \wedge (\xi_1(p_k) = p_k) \\ G''_1 &:= \bigwedge_{v \in attr(H_2) \cup vars(H_2)} (\xi_2(v) \leq p_1 + -\infty) \wedge \ldots \wedge (\xi_2(v) \leq p_k + -\infty) \\ G'_2 &:= (\xi_2(p_1) = p_1) \wedge \ldots \wedge (\xi_2(p_k) = p_k) \\ G''_2 &:= \bigwedge_{v \in attr(H_1) \cup vars(H_1)} (\xi_1(v) \leq p_1 + -\infty) \wedge \ldots \wedge (\xi_1(v) \leq p_k + -\infty) \end{aligned}$$

where ξ_1 and ξ_2 are injective substitutions, that replace all timestamp attributes $p_1, \ldots, p_k \in A$ by temporal variables and substitute temporal variables by others such that $\xi_1(v) \neq \xi_2(w)$ for any two timestamp attributes or temporal variables v and w occurring in A, G_1, G_2, H_1 or H_2.

It is far from obvious why the definition of the union is not as simple as the definition of the cross-product. With regards to the temporal relation formulas (TRFs) only, it would in fact be possible to define $G' = \xi_1(G_1) \vee \xi_2(G_2)$ analogously to the definition in the cross-product operator.[10] The crucial point is that the attributes from the two subexpressions, despite their identical names, are actually different, as they originate from different inputs. This does not matter for

[10] The definition for the TRF in the result schema is equivalent to the simple definition w.r.t. Definition 15.

the TRFs at the first place. However it does matter for the SBFs. Identifying the attributes from the different inputs could result in a wrong analysis of the stream bounds (Definition 17) and other properties of the TSA expression. Therefore the substitutions ξ_1 and ξ_2 and the formulas G_1' and G_2' are used to decouple the formulas from the two inputs. The intuitive meaning of G_1'' and G_2'' is that the part of G' corresponding to the first subexpression should not care about the requirements $\xi_2(H_2)$ on stream bounds from the the second subexpression and vice versa.

Definition 13. (Set Difference $\setminus$) Let E_1 and E_2 be TSA expressions with schema $\mathscr{S}(E_1) = (A, G_1, H_1)$ and $\mathscr{S}(E_2) = (A, G_2, H_2)$ that have the same attributes and let R_1 and R_2 be temporal streams which match the schema of E_1 and E_2 respectively.

$$\begin{aligned}
R_1 \setminus R_2 &:= \{r_1 \in R_1 \mid \neg\exists r_2 \in R_2 \text{ with } r_1 = r_2\} \\
\mathscr{S}(E_1 \setminus E_2) &:= (A, G_1 \wedge (G_1' \vee (G_2' \wedge \xi_2(G_2))), (H_1 \wedge \xi_2(H_2))) \\
G_1'' &:= \bigwedge_{v \in attr(H_2) \cup vars(H_2)} (\xi_2(v) \le p_1 + -\infty) \wedge \ldots \wedge (\xi_2(v) \le p_k + -\infty) \\
G_2' &:= (\xi_2(p_1) = p_1) \wedge \ldots \wedge (\xi_2(p_k) = p_k)
\end{aligned}$$

where ξ_2 is an injective substitution for timestamp attributes and temporal variables, such that the substituted variants $\xi_2(G_2)$, $\xi_2(H_2)$ of G_2, H_2 and the original formulas G_1, H_1 use disjunct sets of temporal variables.

The constraint propagation for set-difference is a little bit tricky, too. On the one hand, the TRF G_2 does not impose any constraints on the output tuples of the expression. Thus we could just choose the TRF G_1 of the first subexpression E_1 as TRF for the schema of the full expression. On the other hand the information in G_2 needs to be propagated to correctly determine the stream bounds with respect to the second subexpression E_2. The solution is as follows: First G_1 is assumed to hold for the negative subexpression as well, as those tuples from any stream produced by the second subexpression that do not fulfill G_1 are irrelevant anyhow. Second $\xi_2(G_2)$ is combined disjunctively with G_1'. As in the definition for the union G_1'' basically tells that in case of the positive subexpression one does not need to care about the requirements $\xi_2(H_2)$ on stream bounds from the negative subexpression. A symmetric counterpart of G_1'' for the negative subexpression is not necessary, as G_1 also holds for the negative subexpression. However G_1'' does not impose any temporal relations between the timestamp attributes of E. Thus the same holds for $G_1'' \vee (G_2' \wedge \xi_2(G_2))$. TSA makes only weak, “syntactic” checks when defining temporal streams or TSA expressions. Further “semantic” checks are done at a later stage (see Section 9.4 about temporal analysis).

Definition 14. (TSA Query) A TSA query is a pair $q = (D, E)$ such that E is a TSA expression with schema $\mathscr{S}(E) = (A, G_E, H_E)$ and D is a temporal stream definition for the output stream with schema $\mathscr{S}(D) = (A, G_D, H_D)$ which both have the same attributes.

9.7. Temporal Analysis

As stated throughout the whole TSA section, TSA analyses the temporal constraints for a number of reasons, e. g. for the decision on the validity of a query expression and for its incremental evaluation. This section provides the relevant definitions and briefly describes the employed algorithms.

Normalization of Temporal Relation Formulas.
Temporal relation formulas (TRFs) can be normalized using the following equivalences in ($\mathscr{NORM}$).[11] After normalization the TRF is equal to $\top$ or $\bot$ or all atomic subformulas have the form $v \leq w + c$ and the normalized TRF does not contain negation. This normalization is essential for following definitions and the algorithmic analysis of TRFs.

$$\begin{aligned}
Neg1 &: \neg(G \wedge G') \Leftrightarrow (\neg G \vee \neg G') \\
Neg2 &: \neg(G \vee G') \Leftrightarrow (\neg G \wedge \neg G') \\
Neg3 &: \neg\neg G \Leftrightarrow G \\
Neg4 &: \neg(t_1 \,\mathsf{op}\, t_2 + c) \;\Leftrightarrow\; t_1 \,\mathsf{op}^{-1}\, t_2 + c \\
Neg5 &: \neg\top \Leftrightarrow \bot \text{ and } \neg\bot \Leftrightarrow \top \\
Top &: G \wedge \top \Leftrightarrow G \text{ and } G \vee \top \Leftrightarrow \top \\
Bot &: G \wedge \bot \Leftrightarrow \bot \text{ and } G \vee \bot \Leftrightarrow G \\
Zero &: v \,\mathsf{op}\, w \;\Leftrightarrow\; v \,\mathsf{op}\, w + 0 \\
Eq &: t_1 = t_2 + c \;\Leftrightarrow\; (t_1 \leq t_2 + c) \wedge (t_1 \geq t_2 + c) \\
Neq &: t_1 \neq t_2 + c \;\Leftrightarrow\; (t_1 < t_2 + c) \vee (t_1 > t_2 + c) \\
Geq &: t_1 \geq t_2 + c \;\Leftrightarrow\; t_2 + c \leq t_1 \\
Gr &: t_1 > t_2 + c \;\Leftrightarrow\; t_2 + c < t_1 \\
Less^{12} &: t_1 < t_2 + c \;\Leftrightarrow\; t_1 \leq t_2 + (c - \varepsilon) \\
Arith1 &: t_1 + c \,\mathsf{op}\, t_2 \;\Leftrightarrow\; t_1 \,\mathsf{op}\, t_2 + (-c) \\
Arith2 &: t_1 \,\mathsf{op}\, (t_2 + c) + d \;\Leftrightarrow\; t_1 \,\mathsf{op}\, t_2 + (c + d) \\
Min1 &: t \leq \min\{t_1, \ldots, t_k\} + c \;\Leftrightarrow \\
&\quad\; t \leq t_1 + c \wedge \ldots \wedge t \leq t_k + c \\
Min2 &: \min\{t_1, \ldots, t_k\} \leq t + c \;\Leftrightarrow \\
&\quad\; t_1 \leq t + c \vee \ldots \vee t_k \leq t + c \\
Max1 &: t \leq \max\{t_1, \ldots, t_k\} + c \;\Leftrightarrow \\
&\quad\; t \leq t_1 + c \vee \ldots \vee t \leq t_k + c \\
Max2 &: \max\{t_1, \ldots, t_k\} \leq t + c \;\Leftrightarrow \\
&\quad\; t_1 \leq t + c \wedge \ldots \wedge t_k \leq t + c
\end{aligned}$$

for temporal terms $t_1, \ldots, t_k$ and $c, d \in \mathbb{Q}$ and TRFs G, G' and $v, w \in ATTR \cup VAR$ and $\mathsf{op} \in \{<, \leq, =, >=, >, \neq\}$ and $<^{-1} \mapsto >$, $\leq^{-1} \mapsto \geq$, $=^{-1} \mapsto \neq$, $\geq^{-1} = \leq$, $>^{-1} \mapsto <$, $\neq^{-1} \mapsto =$

[11] In case of the algorithmic analysis, the equivalences are to be read from left to right.
[12] Imagine ε as infinitely small value with $(c - \varepsilon) + (d - \varepsilon) = ((c + d) - \varepsilon)$ and $c - \varepsilon < c$ but $d < c - \varepsilon$ if $d < c$.

Definition 15. (Temporal Relations) A temporal relation formula G holds in R, denoted $R \models G$, iff for all tuples $r \in R$ the instantiation $\sigma_r(G)$ of G is satisfiable in $\mathbb{Q}$:

$$\models_{\mathbb{Q}} \exists v_1, \ldots, v_l : \sigma_r(G)$$

where $A_{temp} = \{p_1, \ldots, p_k\}$ and $\{v_1, \ldots, v_l\} = vars(G)$
and $\sigma_r := \{p_1 \mapsto r(p_1), \ldots, p_k \mapsto r(p_k)\}$

The temporal distance[13] of two temporal variables or timestamp attributes is the maximum that the value of the second temporal variable is smaller than the value of the first [9]. This information is essential for the incremental evaluation (see Section 11) and for automatic garbage collection[14].

Definition 16. (Temporal Distance) Let G be a temporal relation formula and R be a temporal stream with $\mathscr{A}(R) = A$. The *temporal distance* of two attributes or variables $v, w \in ATTR \cup VAR$ with respect to G is

$$\begin{aligned} dist_G(w,v) &:= \max_{C \in dnf(\overline{G})} \{dist_C(w,v)\} \\ dist_C(w,v) &:= \min\{c \mid \mathscr{TD}, C \models_{\overline{\mathbb{Q}}} v \leq w + c\} \end{aligned}$$

where $u, v, w \in ATTR \cup VAR$ and $c, d, +\infty, -\infty \in \overline{\mathbb{Q}}$ and $\overline{G}$ is the normalized form of G and C is a conjunction of atomic TRFs of the form $v \leq w + c$ and $\mathscr{TD}$ contains

$$\begin{array}{ll} Ref: & v \leq v \\ Trans: & u \leq v + c \wedge v \leq w + d \Rightarrow u \leq w \pm (c+d) \\ Inf: & v \leq w + +\infty \end{array}$$

The algorithmic analysis of the temporal distances is closely related to the simple temporal problem (STP) [22] and the disjunctive temporal problem DTP [19]. Basically the (naive) analysis algorithm is as follows: The TRF is normalized and converted into disjunctive normal form. Each conjunction is an STP instance. The distance of all pairs of attributes and variables for this instance can determined using any algorithm for the all-pair shortest path problem, e. g. the Floyd Warshall algorithm [14], or specialized algorithms for STP. The distance of two attributes or variables for the whole TRF is then the maximum distance of the two attributes or variables in any of the conjunctions.

The following definition is about so-called “stream bounds” that can be derived from a SBF and its corresponding TRF. Stream bounds are those sets of timestamp attributes of a temporal stream that are suited to limit the stream to a finite prefix. Intuitively speaking, stream bounds do not let any part of the stream pass infinitely. The decision whether some TSA query is blocking or not depends on the analysis of stream bounds.

[13]Note that the temporal distance is usually asymmetrical.
[14]Automatic garbage collection is not described in this document, see [12, 9] for the idea

Definition 17. (Stream Bounds) Let G be a temporal relation formula and H be a temporal relation constraint and let R be a temporal stream with $\mathscr{A}(R) = A$.

1. A set $\{p_1,\ldots,p_k\} \subseteq \mathit{ATTR}$ is a *stream bound* with respect to G and H iff

   $$\mathit{bounded}(p_1,b_1),\ldots,\mathit{bounded}(p_k,b_k),G,\mathscr{T}\mathscr{D},\mathscr{S}\mathscr{B} \models H$$

   for any $b_1,\ldots,b_k \in \mathit{BOUND}$[15] and

   $$\mathscr{S}\mathscr{B} : v \leq w + c,\ c < +\infty,\ \mathit{bounded}(w,b_w) \Rightarrow \mathit{bounded}(v,b_v)$$

   for $v,w \in \mathit{ATTR} \cup \mathit{VAR}$, $b_v,b_w \in \mathit{BOUND}$ and $c \in \overline{\mathbb{Q}}$
2. A set $\{p_1,\ldots,p_k\} \subseteq \mathit{ATTR}$ is a *stream bound* with respect to schema $S = (A,G,H)$ if $\{p_1,\ldots,p_k\}$ is a stream bound with respect to G and H.
3. The set $\{p_1,\ldots,p_k\} \subseteq \mathscr{A}_{temp}(R)$ of timestamp attributes is a *stream bound* for R iff every upper limit $s_1,\ldots,s_k \in \mathbb{Q}$ for the values $p_1,\ldots,p_k$ yields a finite prefix of R:

   $$|\{r \in R \,|\, r(p_1) \leq s_1, \ldots r(p_l) \leq s_l\}| < \infty$$

4. H holds in R with respect to G, denoted $R \models_G H$, iff all stream bounds $\{p_1,\ldots,p_l\} \subseteq A_{temp}$ with respect to G and H are stream bounds of R.

The algorithmic analysis of the stream bounds is similar to the one for temporal distances. The TRF G is normalized and converted into disjunctive normal form. For each conjunction C the following is done: First the distance between all attributes and variables in the conjunction is computed. Second for each atom in the SBF H, the atom is set to true if the distance from one of the attributes of the potential stream bound $\{p_1,\ldots,p_k\}$ to the attribute or variable in the atom is finite. Otherwise the atom is set to false. If the H holds under this interpretation, then $\{p_1,\ldots,p_k\}$ is a stream bound with respect to C. If $\{p_1,\ldots,p_k\}$ is a stream bound with respect to all conjunctions, then $\{p_1,\ldots,p_k\}$ is a stream bound with respect to G and H.

9.8. Validity of TSA Queries

Definition 18. (Validity of Temporal Stream Definitions)
A temporal stream definition D with schema $\mathscr{S}(D) = (A,G,H)$ is *valid* iff the set of all timestamp attributes A_{temp} is a stream bound with respect to G and H. The definition implies that any relation R matching the schema $\mathscr{S}(D)$ is a temporal stream.

[15] Actually the stream bound identifiers of atomic SBFs do not play a role here.

Definition 19. (Validity of TSA Queries)
Let $q = (D, E)$ be a TSA query as defined in Definition 14 and let $D_1, \dots, D_k$ be the temporal stream definitions occurring in E. The TSA query q is *valid* iff

1. $D_1, \dots, D_k$ are valid.
2. The schema $\mathscr{S}(E)$ of the TSA expression E matches the schema $\mathscr{S}(D)$ of the definition for the output stream, i. e. G_E implies[16] G_D and all stream bounds with respect to G_D and H_D are stream bounds with respect to G_E and H_E.

The most important property of valid TSA queries is, that they are non-blocking, i. e. can be evaluated incrementally.

Proposition 20. (Non-Blocking Queries)
Let $q = (D, E)$ be a TSA query with schema $\mathscr{S}(D) = (A, G_D, H_D)$ and let $D_1, \dots, D_k$ be the temporal stream definitions occurring in E.

If q is *valid* then q is *non-blocking*. This means, for any temporal streams $R_1, \dots, R_k$ matching the schema of $D_1, \dots, D_k$ respectively and any stream bound $\{p\} \subseteq A_{temp}$[17] with respect to $\mathscr{S}(D)$, the finite prefix for limit $s \in \mathbb{Q}$

$$\begin{aligned}&\{r \in E(R_1, \dots, R_k) \mid r(p) \leq s\} = \\ &\{r \in E(\{r_1 \in R_1 \mid r_1(p_{1,1}) \leq s_{1,1} \vee \dots \vee r_1(p_{1,l_1}) \leq s_{1,l_1}\}, \\ &\qquad\qquad\qquad\qquad \vdots \\ &\qquad \{r_k \in R_k \mid r_1(p_{k,1}) \leq s_{k,1} \vee \dots \vee r_1(p_{k,l_k}) \leq s_{1,l_k}\} \\ &\qquad) \mid r(p) \leq s\}\end{aligned}$$

where $\{p_{i,1}\}, \dots, \{p_{i,l_i}\}$ are stream bounds for $\mathscr{S}(D_i)$ and $s_{i,1}, \dots, s_{i,l_i} \in \mathbb{Q}$ for $1 \leq i \leq k$, i. e. the prefix depends only on finite prefixes of $R_1, \dots, R_k$.

(Sketch). Let $\mathscr{S}(E) = (A, G_E, H_E)$. Without loss of generality one may assume that the temporal stream definitions $D_1, \dots, D_k$ use different stream bound identifiers, as Proposition 20 and none of its indirectly referred Definitions depend on the actual names of stream bounds.

As q is a valid query, any stream bound $\{p\} \subseteq A_{temp}$ with respect to $\mathscr{S}(D)$ is a stream bound with respect to $\mathscr{S}(E)$.

Let $\overline{G}_E$ be the normalized form of G_E (see Section 9.7) and the DNF of $\overline{G}_E$ of G_E be $dnf(\overline{G}_E) = C_1 \vee \dots \vee C_l$, where $C_1, \dots, C_l$ are conjunctions of normalized atomic temporal relation formulas (Definition 3).

For each D_i, $1 \leq i \leq k$ and each C_j there must exist at least one[18] atomic stream bound formula $bounded(v_{i,j}, b_{i,j})$ in H_E[19] (Definition 4) where $v_{i,j} \leq p + dist_{C_j}(v_{i,j}, p)$ (Definition 16) and $b_{i,j}$

[16] Can be checked if right side does not contain variables. By definition G_D and H_D do not contain variables.

[17] The 2. restriction in Definition 14 and Proposition 20 imply the analogous proposition for arbitrary stream bounds.

[18] If D_i describes a static relation this does not hold. But in that case there is nothing to show. One can choose $l_i = 0$.

[19] Note that H_E is in CNF with exactly one clause per $D_1, \dots, D_k$. This follows immediately from the definition of temporal stream definitions (Definition 6)) and of TSA operators.

belongs to D_i.

However $v_{i,j}$ is actually a renamed version of the attribute $p_{i,j} := p^{D_i}(b_{i,j})$ associated to $b_{i,j}$ in D_i. Thus if $p \leq s$ in the result relation then $p_{i,j} \leq s_{i,j} := s - dist_{C_j}(v_{i,j}, p)$ in the source relation R_i, at least for "case" C_j of G_H.

Proposition 21 allows to apply the formal results from the formulas to the actual relations.

Finally the disjunction $r_i(p_{i,1}) \leq s_{i,1} \vee \cdots \vee r_1(p_{i,l}) \leq s_{i,l}$ provides the condition required for the proof for each temporal stream definition D_i. This disjunction could be condensed but this is needed for the proof. □

The proof of Proposition 20. bases on the following proposition on the well-definedness of the TSA operators.

Proposition 21. (Well-Definedness) Let E be a valid TSA expression where $D_1, \ldots, D_k$ are temporal stream definitions occurring in E. If $R_1, \ldots, R_k$ are temporal streams matching the schema of $D_1, \ldots, D_k$ respectively then the $E(R_1, \ldots, R_k)$ is a temporal stream and matches the schema $\mathscr{S}(E)$.

(Sketch). The proof is straight-forward by induction over the structure of E. □

Proposition 22. (Relational Completeness) TSA is *relational complete* [10] for finite (non-stream) relations.[20]

Proof. On finite non-stream relations, TSA is equivalent to Relational Algebra. □

10. Translating Dura to TSA

In the following a small Dura examples illustrates the translation of *Dura* to TSA. The example simple rule detects a fire if smoke and high temperature are detected in the same area at almost the same time, i.e. within 5 seconds.

```
DETECT
    Fire{area{var A}}
ON
    event s: Smoke{area{var A}}
    event t: High-Temp{area{var A}}
WHERE
    {s, t} within 5 sec
END
```

[20] The term "relational complete" compares the expressive power of some formalism for querying *finite relations* to the expressive power of Relational Algebra. A generalization to temporal streams is, if at all possible, non-trivial.

In the case of that simple example the translation is straight forward: `and{...}` is translated to a cross-product followed by a selection for unifying the corresponding attributes. A further selection translates the conditions/temporal relations in the `where{...}` part of the query. an imbedding operator is used to define the attributes of the result as defined in the head of the rule. Another imbedding operator adds the implicitly defined time-stamp attributes. Finally the superfluous attributes from the inputs are discarded using a projection.

$$
\begin{array}{l}
\pi[reception-time, area](\\
\quad \iota[r-t.begin = \min(s.r-t.begin, t.r-t.begin), r-t.end = \max(s.r-t.end, t.r-t.end)](\\
\quad\quad \iota[area = s.area](\\
\quad\quad\quad \sigma[\max(s.r-t.end, t.r-t.end) - \min(s.r-t.begin, t.r-t.begin) < 2sec](\\
\quad\quad\quad\quad \sigma[s.area = t.area](\\
\quad\quad\quad\quad\quad \delta[* \to s.*](Smoke) \times \delta[* \to t.*](High-Temp)\\
)))))
\end{array}
$$

The δ operators at the very Bottom of the query serve for renaming attributes or in the case of the example for adding a prefix to the attribute names. The *delta* operator is a convenience operator and could be replaced by an imbedding operator followed ba a projection. However the *delta* operator simplifies the translation and is more efficient than the combination of an imbedding and a projection operator.

The translation Dura of queries with negation, and particularly queries referring to stateful objects and complex action definitions is much more challenging. Basically Dura rules containing references to stateful objects and complex action definitions need to be rewritten to (potentially recursive) rules using events with modified definitions for the implicit timestamps, only. As translation may involve recursive rules, special care has to be taken that the resulting TSA program is still valid/non-blocking, i.e. can be evaluated incrementally.

11. Differentiating TSA Programs

An incremental evaluation is obviously crucial for TSA. The incremental evaluation allows to derive results continuously as the data arrives on the stream. Without an incremental evaluation results could only be derived at the end of the stream which is too late for CEP application and particularly for emergency management. Proposition 20 in Section 9.7 shows an important property of valid TSA queries: Any prefix of the result stream of a query q depends only on finite prefixes of the input streams of q.

Increment formulas are the basis for the increment conditions that are used to transform a usual TSA expression into an incremental expression. Basically the increment conditions are derived from the increment formulas, by replacing names $b \in BOUND$ by their current value at run-time. In other word, Increment formulas are the parametrized versions of the increment conditions.

Definition 23. (Increment Formulas) Let H be a SBF. The *increment formula* $\triangle H$ to H is defined recursively:

$$\begin{aligned}
\triangle\top &= \top \\
\triangle\, bounded(p,b) &= p \leq b \\
\triangle(H_1 \wedge \ldots \wedge H_k) &= \triangle H_1 \wedge \ldots \wedge \triangle H_k \\
\triangle(H_1 \vee \ldots \vee H_k) &= \triangle H_1 \vee \ldots \vee \triangle H_k
\end{aligned}$$

where $p \in ATTR \cup VAR$ and $b \in BOUND$. $\triangle H$ results from H by replacing each atom $bounded(p,b)$ in H by $p \leq b$.

Stream bound functions are used to compute the progress of an output stream of a TSA query with respect to the progress of the input streams of the query (see Section 11). Stream bound function formulas are an intermediate step in the generation of these functions. Due to their special structure, stream bound function formulas are probably a good basis for optimizing stream bound functions using techniques for the optimization of Boolean functions [18].

Definition 24. (Stream Bound Functions) Let G_E be a TRF, H_E and H_D be SBFs, $b \in BOUND$ the name for a stream bound and $p \in ATTR$ a timestamp attribute that is assigned to b (e. g. by $bounded(p,b)$ in a temporal stream definition).

1. The *stream bound function formula* for $\triangle H_D$, G_E and $\triangle H_E$ is defined recursively:

$$\begin{aligned}
F_{\triangle H_D, G_E, \triangle H_E} &= \bigwedge_{C_E \in dnf(\overline{G_E})} F_{\triangle H_D, C_E, \triangle H_E} \\
F_{\triangle H_D, C_E, \top} &= \top \\
F_{\top, C_E, \triangle H_E} &= \bot \text{ for } H_E \neq \top \\
F_{(b_D \leq p), C_E, (v \leq b_E)} &= \begin{cases} \top & \text{if } dist_{C_E}(p,v) = -\infty \\ \bot & \text{if } dist_{C_E}(p,v) = +\infty \\ b_D \leq b_E - dist_{C_E}(p,v) & \text{else} \end{cases} \\
F_{\triangle H_D, C, (\triangle H_{E1} \wedge \ldots \wedge \triangle H_{Ek})} &= F_{\triangle H_D, C, \triangle H_{E1}} \wedge \ldots \wedge F_{\triangle H_D, C, \triangle H_{Ek}} \\
F_{\triangle H_D, C, (\triangle H_{E1} \vee \ldots \vee \triangle H_{Ek})} &= F_{\triangle H_D, C, \triangle H_{E1}} \vee \ldots \vee F_{\triangle H_D, C, \triangle H_{Ek}} \\
F_{(\triangle H_{D1} \wedge \ldots \wedge \triangle H_{Dl}), C, \triangle H_E} &= F_{\triangle H_{D1}, C, \triangle H_E} \vee \ldots \vee F_{\triangle H_{Dl}, C, \triangle H_E} \\
F_{(\triangle H_{D1} \vee \ldots \vee \triangle H_{Dl}), C, \triangle H_E} &= F_{\triangle H_{D1}, C, \triangle H_E} \wedge \ldots \wedge F_{\triangle H_{Dl}, C, \triangle H_E}
\end{aligned}$$

   where $p, v \in ATTR \cup VAR$ and $b' \in BOUND$ and $\overline{G_E}$ is the normalized form of G_E and C_E is a conjunction of atoms of the form $v \leq b_E$.
   The basic idea is, to enforce $v \leq b_E$ using p and b_D and the inequation
   $v \leq p + dist_{C_E}(p,v) \leq b_D + dist_{C_E}(p,v) \overset{!}{\leq} (b_E - dist_{C_E}(p,v)) + dist_{C_E}(p,v) \leq b_E$.
   The $\bigwedge$ in the first case results from the fact that H_E must hold in each case of G_E (compare Definition 4).

2. Let $F = F_{\triangle H_D, G_E, \triangle H_E}$ be the stream bound function formula for $\triangle H_D = (p \leq b_D)$ and, G_E, $\triangle H_E$. The *stream bound function* f_F of F is defined recursively:

$$\begin{array}{ll} f_\top = +\infty & f_\bot = -\infty \\ f_{b_D \leq b_E - d} = b_E - d & \\ f_{F_1 \wedge \ldots \wedge F_k} = \min\{f_{F_1}, \ldots, f_{F_k}\} & \\ f_{F_1 \vee \ldots \vee F_k} = \max\{f_{F_1}, \ldots, f_{F_k}\} & \end{array}$$

   where $b_E \in BOUND$ and $d \in \overline{\mathbb{Q}}$.

3. The *stream bound function* for $\triangle H_D = (p \leq b_D)$ and, G_E, $\triangle H_E$ is

$$f_{(p \leq b_D), G_E, \triangle H_E} = f_{F_{(p \leq b_D), G_E, \triangle H_E}}$$

4. The *stream bound function* for a TSA query $q = (D, E)$,with $\mathscr{S}(D) = (A, G_D, H_D)$ and $\mathscr{S}(E) = (A, G_E, H_E)$, and a stream bound identifier $b_D \in BOUND$, with $bounded(p^D(b), b)$ is an atom in H_D, is

$$f_{b_D, q} = f_{(p^D(b) \leq b), G_E, \triangle H_E}$$

   where p^D is the function defined in Definition 6 that returns the timestamp attribute $p \in A$ that is assigned to b within D.

The stream bound function formula F can be normalized to be $\top$ or $\bot$ or not to contain $\top$ or $\bot$ at all. In that case f_F is either $-\infty$ or $+\infty$ or does not contain $-\infty$ or $+\infty$ at all. If $F = \bot$, thus $f_F = -\infty$, then p is not a stream bound with respect to G_E and H_E. If $f_F = +\infty$ then G_E and H_E belong either to a static relation or a stream that is actually empty because its imposed temporal relations are unsatisfiable.

Without loss of generality the definitions for the incremental evaluation of TSA assume that all temporal stream definitions of a TSA program use different stream bound identifiers $b \in BOUND$. This can easily be realized if $BOUND = STREAM \times ATTR$ and for temporal stream definition $D = (n, S)$ and stream bound $\{p\}$ of S the stream bound identifier $b = b^D(p)$ corresponding to p in D has the form $b = n : p$.

The incremental evaluation of TSA allows for an asynchronous evaluation of the queries of a TSA program.[21] However the queries are not fully independent from each other, but need to exchange information on the progress of their respective output streams. This is necessary as the maximum achievable progress of of the output stream of a query depends on the progress of the referred input streams.

The incremental evaluation uses values assigned to the stream bound identifiers occurring in the temporal stream definitions and queries, to propagate this progress information. These values are used to instantiate the increment formula (Definitions 23) for the current increment computation of a query. The values for stream bound identifiers serve for a similar purpose as the "punctuations" of [26]. However they are not only used for input streams but are propagated through the evaluation process.

[21] Actually every fair sequence for the increment computations will compute the correct result. The actual sequence may significantly affect the efficiency and the response-time of the evaluation, though.

Definition 25. (Stream Bound Values)
Let P be a set of TSA Queries. For a TSA Query $q = (D,E)$ with $\mathscr{S}(D) = (A, G_D, H_D)$ let $s_{q,i} \in \mathbb{Q}$ and $e_{q,i} \in \mathbb{Q}$ denote the start and end time of the ith ($i \geq 1$) increment computation for q. The following is defined simultaneously:

1. Value of b for query q at the ith increment computation for q and $i \geq 0$

$$\begin{aligned} v_{q,0}(b) &= -\infty \\ v_{q,i}(b) &= f_{b,q}(v_{s_{q,i}}(b_1), \ldots, v_{s_{q,i}}(b_l)) \end{aligned}$$

   where b occurs in H_D and $f_{b,q}$ is the function term from Definition 24 and $b_1, \ldots, b_l$ are the stream bound names occurring in $f_{b,q}$.

2. Value of b for query q at time $t \in \mathbb{Q}$

$$v_{q,t}(b) = \begin{cases} v_{q,0}(b) \text{ for } t < e_{q,1} \\ v_{q,i}(b) \text{ for } e_{q,i} \leq t < e_{q,i+1} \end{cases}$$

3. Value of b for temporal stream definition D at time t

$$v_{D,t}(b) = \min_{\substack{q' \in P \\ \mathrm{out}(q') = D}} \{v_{q',t}(b)\}$$

   The value $v_{D,t}(b)$ for input streams must be provided. The values for each D must not decrease and should exceed every limit after some finite amount of time. If so, then the same holds for the derived values, assuming a fair execution sequence for the queries. [22]

4. Value of b at time t

$$v_t(b) = v_{D_b,t}(b)$$

   where D_b is the temporal stream definition corresponding to b.

The definition is well defined as $s_{q,i} < e_{q,i}$ for $i \geq 1$.

The incremental evaluation is able to work with conservative approximations of the stream bound values, too. This is particularly useful for a parallel or even distributed evaluation of a TSA program, as the values for the stream bound identifiers do not need to be perfectly synchronized. Furthermore one could use conservative approximations for the functions $f_{b,q}$. The functions $f_{b,q}$ are optimal with respect to the achievable progress within one step, however approximate,i. e. simpler versions of the functions may help to reduce the computational overhead that is potentially introduced by the computation of the stream bound values.

As the each step of the incremental evaluation of a query only depends on the current values for the stream bound identifiers, it is easily possible to stop and resume the incremental evaluation by storing the values persistently. This is particularly useful for crash recovery, but also allows to switch the scheduling strategy or even to re-optimize the executed program.

[22] In case of system time this is basically the current value of the local clock minus 1."Minus 1" ensures that definitely all new data item will receive a greater timestamp than the returned value. This assumes that the value of a clock is never decreasing. For application time, a mechanism similar to "punctuation" [26] had to be used.

Definition 26. (Increment Expression)
Let $q = (D, E)$ be a *valid* TSA query with output schema $\mathscr{S}(D) = (A, G_D, H_D)$. The *incremental expression* for the $i+1$th ($i >= 0$) increment computation for q is:

$$\triangle E_{i+1} = \sigma[\triangle H_{Di+1} \wedge \neg \triangle H_{Di}](E)$$

where $\triangle H_D$ is the increment formula for H_D from Definition 23, $\triangle H_{Di+1}$ results from $\triangle H_D$ by replacing each stream bound name b by its new value $v_{q,i+1}(b)$ and $\triangle H_{Di}$ results from $\triangle H_D$ by replacing each stream bound name b by its old value $v_{q,i}(b)$ from the ith execution of $\triangle E$.

The increment expression for a query can be seen as parametrized expression. This is very useful as it avoids to compile each of the increment expressions at the run-time of the TSA program. Instead the parametrized version of the increment expression can be compiled only once.

Furthermore, the increment expression is in fact an ordinary Relational Algebra expression, i.e. the incremental evaluation reduces the evaluation of TSA expressions to the evaluation of Relational Algebra expressions. In other words, TSA generalizes Relational Algebra to data streams and the incremental evaluation of TSA reduces the evaluation of TSA to the evaluation of Relational Algebra. Thus an efficient implementation of Relational Algebra could easily be enhanced to an implementation of TSA that allows for processing database relations *and* streams.

Proposition 27. (Correctness) The consecutive execution of the increment statements $\triangle E_1, \triangle E_2, \triangle E_3, \ldots$ for a TSA query $q = (D, E)$ yields the same result as if E had been applied to the whole stream at once.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

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

[3] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[4] R. S. Barga, D. Gannon, and D. A. Reed. The client and the cloud: Democratizing research computing. *IEEE Internet Computing*, 15(1), 2011.

[5] S. Brodt, S. Hausmann, and F. Bry. Deliverable D4.2: Reactive Rules for Emergency Management. `www.emili-project.eu`, 2010.

[6] S. Brodt, S. Hausmann, and F. Bry. Deliverable D4.3: Dura – Concepts and Examples. `www.emili-project.eu`, 2011.

[7] S. Brodt, S. Hausmann, and F. Bry. Deliverable D4.5: Implementation. `www.emili-project.eu`, 2011.

[8] S. Brodt, S. Hausmann, and F. Bry. Deliverable D4.6: Modularization Mechanisms. `www.emili-project.eu`, 2012.

[9] F. Bry and M. Eckert. On static determination of temporal relevance for incremental evaluation of complex event queries. In *Proc. Int. Conf. on Distributed Event-Based Systems*, pages 289–300. ACM, 2008.

[10] E. F. Codd. Relational completeness of data base sublanguages. *In: Database Systems, Prentice Hall and IBM Research Report RJ 987, San Jose, California*, 1972.

[11] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White. Cayuga: A general purpose event monitoring system. In *CIDR*, pages 412–422, 2007.

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

[13] M. Eckert, F. Bry, S. Brodt, O. Poppe, and S. Hausmann. Two Semantics for CEP, no Double Talk: Complex Event Relational Algebra (CERA) and its Application to XChangeÊQ. In S. Helmer, A. Poulovassilis, and F. Xhafa, editors, *Reasoning in Event-based Distributed Systems*, volume 347 of *Studies in Computational Intelligence*, chapter 4. Springer, 1 edition, 2011.

[14] R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5:345–, June 1962.

[15] H. Garcia-Molina, J. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2001.

[16] S. Hausmann and F. Bry. Complex Actions for Event Processing. Submitted for publication, 2012.

[17] M. Hong. *Expressive and Scaleable Event Stream Processing*. PhD thesis, Cornell University, 2009.

[18] B. R. King, S.-V. A. L., M. C. T., and H. G. D. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, Norwell, MA, USA, 1984.

[19] M. K. Kostas Stergiou. Backtracking algorithms for disjunctions of temporal constraints. *Artif. Intell.*, 120, June 2000.

[20] J. Krämer and B. Seeger. Semantics and implementation of continuous sliding window queries over data streams. *ACM Trans. Database Syst.*, 34, April 2009.

[21] P. v. B. Marc Vilain, Henry Kautz. *Constraint propagation algorithms for temporal reasoning: a revised report*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[22] J. P. Rina Dechter, Itay Meiri. Temporal constraint networks. *Artif. Intell.*, 49:61–95, May 1991.

[23] N. Seifert and M. Bettelini. Deliverable D3.1: Use Cases Requirements Analysis and Specification (Main Report). `www.emili-project.eu`, 2010.

[24] N. Seifert, M. Bettelini, and S. Rigert. Deliverable D3.2: Concrete Use Case Models. `www.emili-project.eu`, 2011.

[25] N. Seifert, M. Bettelini, S. Rigert, S. Vraneš, V. Mijović, N. Tomaševiś, G. Konečni, V. Janev, L. Kraus, P. L. Kroner, D. Siller, A. Braun, Y. Leontyeva, J. L. M. E. nol, and J. R. H. Gonzales. Deliverable D3.3: Use case modelling for implementation in SITE. `www.emili-project.eu`, 2012.

[26] P. Tucker and D. Maier. Exploiting punctuation semantics in data streams. In *Proc. ICDE*, page 279, 2002.

[27] S. Vraneš, M. Stanojević, V. Janev, N. Tomaševiś, and V. Mijović. Deliverable D6.3 Design of the first prototype of the integrated EMILI-SITE system. `www.emili-project.eu`, 2011.