Exeta

Martin Procházka

February 2, 2018

Contents

Chapter 1

Introduction

Exeta is a framework for processing mutually dependent *tasks* in a distributed heterogeneous computing environment. The framework consists of the following components:

- a compiler
- a repository
- an engine
- a console

The *compiler* translates task definitions written in Exeta *language* into relational representation which is stored in a *repository.* The repository is utilized by the *engine* that submits tasks to various servers (database, ETL, application, etc.) for execution, then tracks them and updates their *status.* The order of tasks' execution is given by defined *dependencies.* Task processing is monitored and controlled by operations stuff via the operational *console.*

Exeta is designed to minimize both development and operational effort. It is supported by the following features:

- code reusability
- code generation
- tasks grouping
- failure recovery
- easy deployment

- dynamic prioritization
- dynamic capacity planning

Chapter 2

Tutorial

Our tutorial comes from the field of data warehousing and ETL[1] processing in a telecommunication company. Data warehouse stores data from various source systems such as network mediation, billing, CRM[2], ERP[3], external data sources (e.g., regulators, government, statistical office or marketing agencies) or manually maintained code lists. It keeps detailed data (with limited history), its aggregations (with potentially unlimited history) and data derived from it according to defined business rules. This enables reporting, data analysis, forecasting, customer segmentation, data mining, campaign support, etc. All data processing is performed by the ETL tool. This is in particular the extraction of data from source systems, its loading into the data warehouse, and transformation (consolidation, aggregation, and calculation of derived data).

2.1 Architecture

The data warehouse consists of several layers each serving to a different purpose:

- extract layer
- stage layer
- target layer

[1]Extraction, Transformation, Load
[2]Customer Relationship Management
[3]Enterprise Resource Planning

2.1.1 Extract Layer

The extract layer is placed in the file system of the ETL server. It is a place where extracts from the source systems are delivered to. Extracts are loaded into the stage layer. Extract that is successfully loaded form the extract layer into the stage layer is deleted from the file system by the task that is responsible for loading into the stage layer. It means that the production and the consumption of the extracts are asynchronous.

Extracts form agreed interfaces between source systems and the data warehouse. They must meet strictly defined rules. For example:

> *Each extract is a semicolon separated variable length file.*
>
> *Each string field is delimited by* `"`*, encoded in unicode, with* `\` *as an escape character.*
>
> *Each date field is in ISO 8601 format* `YYYY-MM-DD`*.*
>
> *Each timestamp field is in ISO 8601 format* `YYYY-MM-DD HH24:MI:SS`*.*
>
> *Each number uses* `.` *as a decimal point, etc.*

In order to minimize volume of data transferred three types of extracts are used:

- increment,
- delta,
- full.

Incremental extracts are suitable for transactional data that are generated once by the source system and then never changed. An increment needs to be identifiable by a sequence number or by a time stamp. *Delta extracts* are used in cases of frequently modified data. Their first field contains an operation that should by applied to the row to load it into the target table. This operation is `INSERT`, `UPDATE`, or `DELETE`. Each data set populated with delta extracts must have a primary or unique key. *Full extracts* are used for small data sets or for data sets that do not conform to prerequisites on incremental or delta extracts.

2.1.2 Stage Layer

The stage layer is a source for the target layer. It is a database that stores a limited history of extracts. So, it enables reloads of data in the target layer. The stage is not accessible to end users.

2.1.3 Target Layer

2.2 Assignment

Load a company data provided by the Statistical Office into the data warehouse on regular basis. The data consists of two files:

- KLAS80004_CS
- RES_CS

The first one is a code list containing a *classification of economic activities NACE*, the other contains a company information from the *Business Register* (id, name, address, type of ownership, date of establishment and termination, list of NACE activities, and a range of a number of employees). Implement the data processing in the following steps:

1. Transfer files from an agreed place into the extract layer.
2. Load extracts into the stage layer.
3. Transform data from the stage into the 3NF in the target layer.

Use `scp` for file transfers. The data warehouse runs on PostgreSQL database server. Exeta runs on a linux host.

2.3 Solution

2.3.1 Database Part

2.3.2 ETL Part

Exeta Source

generic reusable tasks will be developed that

Task Code

Expanded Instance Tree

The Exeta source files are translated into relational representation which is stored in the database repository. This representation is expanded into the instance tree. Expansion starts with the task main and proceeds from the top to the bottom. During this process, features are substituted with

Task classification		Exeta source file
Generic	shell	`100_sh_generic.e`
	database	`200_db_generic.e`
	stage	`300_edw_stg_generic.e`
	target	`400_edw_tgt_generic.e`
Specific	hierarchy	`500_main.e`
	extract	`600_ext_specific.e`
	stage	`700_edw_stg_specific.e`
	target	`800_edw_tgt_specific.e`

Table 2.1: Exeta source files.

Task classification		Task name
Generic	shell	`Copy`
	database	`AnalyzeTable`
		`DropTable`
	stage	`LoadStgTable.CreateNew`
		`LoadStgTable.LoadNew`
		`LoadStgTable.CopyNew`
		`LoadStgTable.DropOld`
	target	`LoadTgtTable.CreateNew`
		`LoadTgtTable.DropTmp`
		`LoadTgtTable.CreateDel`
		`LoadTgtTable.CreateIns`
		`LoadTgtTable.CreateUpd`
		`LoadTgtTable.LoadDel`
		`LoadTgtTable.LoadIns`
Specific	target	`LoadTgtTable.Transform.Activity`
		`LoadTgtTable.Transform.Party`
		`LoadTgtTable.Transform.PartyActivity.T01`
		`LoadTgtTable.Transform.PartyActivity.New`

Table 2.2: Executable tasks.

their values. The expanded instance tree resulting from our source files is shown below. Right to each executable task, the link to its source code, generated code, and executable code is placed marked as `src`, `gen`, and `exe` resp. Calling tasks are mentioned in comments.

```
# main
(  # ProvideExtracts
   (  # ProvideExtracts.CzSO
      (  Copy "${EXETA_HOME}/rmt/czso/KLAS80004_CS.csv"
              "${EXETA_HOME}/ext/czso/KLAS80004_CS.${TimeStamp}.dat" src exe
      || Copy "${EXETA_HOME}/rmt/czso/RES_CS.csv"
              "${EXETA_HOME}/ext/czso/RES_CS.${TimeStamp}.dat" src exe
      )
   )
|| # LoadEDW
   (  # LoadStgLayer
      (  # LoadStgLayer.CzSO
         (  # LoadStgTable czso_klas80004_cs
            (  # LoadStgTable.DropNew czso_klas80004_cs
               (  DropTable stg_layer czso_klas80004_cs_20170512000000 src exe
               )
            -> LoadStgTable.CreateNew czso_klas80004_cs src exe
            -> LoadStgTable.LoadNew czso_klas80004_cs src gen exe
            -> LoadStgTable.CopyNew czso_klas80004_cs src gen exe
               # when LoadStgTable.LoadNew czso_klas80004_cs skipped
            -> (  LoadStgTable.DropOld czso_klas80004_cs src gen exe
               || # LoadStgTable.AnalyzeNew czso_klas80004_cs
                  (  AnalyzeTable stg_layer czso_klas80004_cs_20170512000000 src exe
                  )
               )
            )
         || # LoadStgTable czso_res_cs
         (  # LoadStgTable.DropNew czso_res_cs
            (  DropTable stg_layer czso_res_cs_20170512000000 src exe
            )
         -> LoadStgTable.CreateNew czso_res_cs src exe
         -> LoadStgTable.LoadNew czso_res_cs src gen exe
         -> LoadStgTable.CopyNew czso_res_cs src gen exe
            # when LoadStgTable.LoadNew czso_res_cs skipped
         -> (  LoadStgTable.DropOld czso_res_cs src gen exe
            || # LoadStgTable.AnalyzeNew czso_res_cs
               (  AnalyzeTable stg_layer czso_res_cs_20170512000000 src exe
               )
            )
         )
      )
   )
|| # LoadTgtLayer
   (  # LoadTgtTable activity
      # when LoadStgTable czso_klas80004_cs succeeded
      (
         (  (  # LoadTgtTable.DropNew activity
               (  DropTable wrk_area activity_n src exe
               )
            -> LoadTgtTable.CreateNew activity src exe
            )
         || LoadTgtTable.DropTmp activity src gen exe
         )
      -> LoadTgtTable.Transform.Activity src exe
   -> (  (  # LoadTgtTable.DropIns activity
            (  DropTable wrk_area activity_i src exe
            )
         -> LoadTgtTable.CreateIns activity src exe
         )
      || (  # LoadTgtTable.DropUpd activity
            (  DropTable wrk_area activity_u src exe
            )
         -> LoadTgtTable.CreateUpd activity src gen exe
         )
      || (  # LoadTgtTable.DropDel activity
            (  DropTable wrk_area activity_d src exe
            )
         -> LoadTgtTable.CreateDel activity src exe
         )
      )
   -> LoadTgtTable.LoadIns activity src exe
   -> LoadTgtTable.LoadDel activity src exe
   )
|| # LoadTgtTable party
   # when LoadStgTable czso_res_cs succeeded
   (
      (  (  # LoadTgtTable.DropNew party
            (  DropTable wrk_area party_n src exe
            )
         -> LoadTgtTable.CreateNew party src exe
         )
      || LoadTgtTable.DropTmp party src gen exe
      )
   -> LoadTgtTable.Transform.Party src exe
   -> (  (  # LoadTgtTable.DropIns party
            (  DropTable wrk_area party_i src exe
            )
         -> LoadTgtTable.CreateIns party src exe
         )
      || (  # LoadTgtTable.DropUpd party
            (  DropTable wrk_area party_u src exe
            )
         -> LoadTgtTable.CreateUpd party src gen exe
         )
      || (  # LoadTgtTable.DropDel party
            (  DropTable wrk_area party_d src exe
            )
         -> LoadTgtTable.CreateDel party src exe
         )
      )
   -> LoadTgtTable.LoadIns party src exe
   -> LoadTgtTable.LoadDel party src exe
   )
|| # LoadTgtTable party_activity
   # when LoadTgtTable party succeeded
   #   &  LoadTgtTable activity succeeded
   (
      (  (  # LoadTgtTable.DropNew party
         (  DropTable wrk_area party_activity_n src exe
         )
      -> LoadTgtTable.CreateNew party_activity src exe
      )
   || LoadTgtTable.DropTmp party_activity src gen exe
   )
-> # LoadTgtTable.Transform.party_activity
   (
      (  LoadTgtTable.Transform.PartyActivity.T01 src exe
      -> LoadTgtTable.Transform.PartyActivity src exe
      )
   )
-> (  (  # LoadTgtTable.DropIns party
         (  DropTable wrk_area party_activity_i src exe
         )
      -> LoadTgtTable.CreateIns party_activity src exe
      )
   || (  # LoadTgtTable.DropUpd party
         (  DropTable wrk_area party_activity_u src exe
         )
      -> LoadTgtTable.CreateUpd party_activity src gen exe
      )
   || (  # LoadTgtTable.DropDel party
         (  DropTable wrk_area party_activity_d src exe
         )
      -> LoadTgtTable.CreateDel party_activity src exe
      )
```

```
            )
         -> LoadTgtTable.LoadIns party_activity src exe
         -> LoadTgtTable.LoadDel party_activity src exe
         )
      )
   )
)
```

Chapter 3

Concepts

In this chapter, basic Exeta concepts are presented. The Exeta language is introduced – how to define tasks in Exeta language. Compilation – translation of task definitions to task instances forming an instance tree is shown. A processing of task instances (so called task instance runs) is explained with an emphases on statuses they can get into.

3.1 Definition

The main Exeta asset is a *task*. Each task has its header consisting of a *name* and a set of *identifiers*, a set of *features*, a *body*, and *rules*. Tasks can be called from other tasks with identifiers' and features' values assigned.

Both identifiers and features are used to transfer an information from outside of a task call into its inside and to parametrize a code executed by the task. They differ in one important aspect: The task name together with its identifier values uniquely identifies a task instance. There cannot be more task instances of the same name and the same identifier values in Exeta. The importance of identifiers in the identification of task instances is discussed in more detail further on.

The task body determines what the task actually does. There are following three possibilities:

- The task calls other tasks.
- The task executes its code.
- The task generates a code first and then executes the generated code.

A server that generates a code is called *generator* and it is set by a system feature `generator`. A server that executes a code is called *executor* and it is set by a system feature `executor`.

If the code of a generating task contains not escaped `${TimeStamp}` (the only dynamic feature that contains a value that is known at run-time, not any sooner) then the task is generated each time just before it is executed. Otherwise, it is generated just once at the time of deployment.

To declare which tasks should be executed in the computing environment there is one special task called `main`. This is the task all other tasks that should be executed are called from, either directly or transitively.

A task `LoadTable` that loads data into a Data Warehouse table is defined in the following way:

```
task LoadTable Schema Table
  call PrepareWorkArea.${Method} ${Schema} ${Table}
    -> LoadWorkTable.${Schema}.${Table}
    -> LoadTargetTable.${Method} ${Schema} ${Table}
;
```

It has two identifiers `Schema` and `Table` that uniquely identify what table is loaded, and two parameters `IdList` and `Method` that specify how the table is loaded. Further, there is a *with* clause that lists task features. In our case, there is just one feature `recovery`–a system feature defining Exeta behavior in case of task's failure (for details, see Subsection 3.1.3).

The first kind of task's body is demonstrated by the task `LoadTable` mentioned above. This task *calls in sequence* following three other tasks:

- `PrepareWorkArea.${Method}`,
- `LoadWorkTable.${Schema}.${Table}`, and
- `LoadTargetTable.${Method}`

with identifier and parameter values passed to them. Values of both of them are denoted using Unix shell notation, i.e., the value of an identifier `table` is denoted by `${table}`. Calls of all tasks in this example demonstrate another powerful Exeta feature–called task names can contain variable values. It enables *polymorphism* introduced by object-oriented programming languages. In our case, there is one generic task for a load of any table. What code is actually executed depends on specific values of identifiers or parameters `Schema`, `Table`, and `Method`. A different code is executed for a different method (such as `SCD1`, `SCD2`, `Increment`, etc.) and different tables in different schemata. See the following code for examples of `LoadTable` task calls:

```
task LoadTargetDaily
with executor = mypsql
  , schedule = " 0 0 * * * "
call LoadTable DW_DB DW_ACCOUNT
     with IdList = ( ACCOUNT_ID )
       , Method = SCD2
     when LoadTable STG_DB S_ORG_EXT succeeded
  || LoadTable DW_DB DW_SUBSCRIBER
     with IdList = ( SUBSCRIBER_ID )
       , Method = SCD2
     when LoadTable DW_DB  DW_ACCOUNT succeeded
       & LoadTable STG_DB S_ASSET    succeeded
;
```

Identifiers, parameters, and features are inherited from task's ancestors and they can be re-assigned by children.

A task name together with identifiers and an executor the task is assigned to uniquely identifies the task instance as well as task script and its location in the Exeta file system repository.

When condition and time related system features:

```
task LoadTargetMonthly
with schedule = " 0 0 * * 0 "
call LoadTable DW_DB DW_SEGMENTATION
;
task LoadTargetDaily
with schedule = " 0 0 * * * "
call LoadTable DW_DB DW_CAMPAIGN
     when LoadTable DW_DB DW_SEGMENTATION succeeded
;
```

The check if the load of the table `DW_CAMPAIGN` can be started with a time stamp given by a schedule `Daily` is done in following steps:

1. Get a maximum time stamp from the schedule `Monthly` assigned to the task call `LoadTable DW_DB DW_SEGMENTATION` that is less or equal to the current time stamp of the task call `LoadTable DW_DB DW_CAMPAIGN`.

2. If the task call `LoadTable DW_DB DW_SEGMENTATION` finished with the required status for this time stamp (i.e., succeeded in our example) then start the task call `LoadTable DW_DB DW_CAMPAIGN`, otherwise postpone it.

In our example, `main` could look like this:

```
task main
call LoadDW
;
task LoadDW
with executor = mypsql
submit same after 5 m 5 times then suspend when failed
submit next when succeeded
submit next when skipped
call LoadDWStg
  || LoadDWTgtDaily
  || LoadDWTgtMonthly
;
```

3.1.1 Features

Custom

System

There are several features–so called *system features*–that have a special meaning in Exeta and influence the way tasks are executed. System features are the following:

- `executor`
- `generator`
- `timestamp`
- `schedule`

Some of them have been already mentioned in the text above. Here, their complete list with a brief explanation of each of them is provided. When not defined explicitly a feature is set to its default value. Default values are underlined.

Any system feature can be set for a task instance by its parental instance or it can be passed into a task instance as a parameter.

Exeta recognizes two different types of executing servers defined by the following two system features:

- `executor`
- `generator`

The feature `executor` determines a server or application that *executes* the task code.

The feature `generator` specifies a server or application that *generates* (and possibly installs) a code to be executed. It is used by tasks that execute a generated code. Such tasks are executed in two steps:(i) first, the task code is executed by the generator; then (ii) the output from the first step is executed by the task executor.

Run-time

TOTO BUDE NASTAVITELNÉ AŽ V RUN-TIME REPOSITORY. IMPORTANCE A CAPACITY SE TOTIŽ VÁŽE NA TASK INSTANCE.

Usually, there is more candidates to run on the same executor at the same time, but not all of them can be executed at once as resources of each executor are limited. Selection which candidate will be passed to its executor is controlled by two features

- capacity, and
- importance.

Task instance's capacity tells us how many per cents of executor's resources are consumed by the task instance.

Which task instances should be passed to a given executor with a capacity C_E for execution?

1. Let C_R be a sum of capacities of all task instances running on this executor.
2. If there is a submitted task instance with this executor and a capacity C such that $C \leq C_E - C_R$, then run this instance on the executor.
3. Repeat previous steps until no task instance is selected.

3.1.2 Body

Execute

Until now, all tasks called other tasks within their bodies. No task executed any piece of executable code directly. Now, we will show how to link an Exeta task to a code to be executed.

Let us see the following task:

```
task LoadTable.DW_DB.DW_ACCOUNT execute ;
```

There is just a note in the task's body that it executes some code, but the code is missing there. Instead of including the code in Exeta definition file the code is placed in separate file the path and name of which is fully determined by task's *executor* and *name*. In our case, as `executor=mypsql` and `mypsql` file exists in `${EXETA_HOME}/rep/psql/srv` folder, it is

`${EXETA_HOME}/rep/psql/src/LoadTable.DW_DB.DW_ACCOUNT`

where usually `EXETA_HOME=/opt/exeta`. Separating Exeta definition file from executable codes makes a development easier as the executable code can be developed using development tools specific for a given executor.

Generate an Execute

Call

3.1.3 Rules

Recovery

Recovery of failed task instances is controlled by a *recovery rule*. It determines what happens after task's failure. A time between task's failure and its recovery attempt is controlled by the next item *wait*. Recovery rule has the follewing syntax

```
when failed
{ then
  [ wait ⟨Number⟩ ( s | m | h ) and ]
  ( fail caller | submit same )
  ( once | twice | ⟨Number3+⟩ times )
}
  then ( succeed | skip | suspend | fail caller | submit same )
```

where *Number* is a natural number; *Action* takes one of the following values:

`succeed`	– leaves the failed task in the status `succeeded`,
`skip`	– leaves the failed task in the status `skipped`,
`suspend`	– leaves the failed task in the status `suspended` and waits for manual resolution of the problem,
`fail caller`	– raises the failure to a calling task,
`submit same`	– re-submits (restarts) the failed task;

Submission

Schedule features together determine(i) which time stamps are passed to a task instance, and (ii) when a task instance is started. It is controlled by the following two features:

- `schedule`
- `timestamp`

The feature `schedule` is a list of one of the following forms:

`( submit same when succeeded )`

or

`( submit same when skipped )`

or

`( submit next by` *plan* `after` *base* `)`

or

`( when` *final-status* `then submit next by` *plan* `after` *base* `)`

or

`( submit next by` *plan* `after` *base* `when` *final-status-1*
`( submit next by` *plan* `after` *base* `when` *final-status-2*

consisting of the following objects: the *final-status* – specifies the final status (`succeeded` or `skipped`); the *base* – determines the base the next value of time stamp is derived from; the *plan* – specifies the sequence of time stamps passed to the task and all its children as a value of the feature `timestamp`. It uses extended `cron` notation as you can see in the following example:

```
schedule = ( "14 3 1 * *" "0 0 2 * *" )
```

This schedule generates following sequence of time stamps:

```
       ⋮
2015-05-01 03:14
2015-05-02 00:00
2015-06-01 03:14
2015-06-02 00:00
       ⋮
```

There are two ways how to determine the next value of this system feature `timestamp` that is passed to a given task instance:

- `submit next`

  A new task instance will be submitted with a time stamp given by its schedule that immediately follows the time stamp of the *last* (just finished) task call;

- `submit future`

  A new task call will be submitted with a time stamp given by its schedule that immediately follows the current (real) time.

The features `schedule` can be set in the task declaration only (not in the task call) and once defined it cannot be redefined in successor tasks. The feature `timestamp` is set internally by Exeta system; it can be set in no Exeta feature declaration.

3.1.4 Code

```
${EXETA_HOME}
+-bin
+-doc
+-rep
  +-⟨server type name⟩
    +-bin
    | +-fail
    | +-generate
    | +-run
    | +-status
    +-src
    | +-⟨task name 1⟩
    | +-⟨task name 2⟩
    | | ···
    | +-⟨task name i⟩
    +-srv
    | +-⟨server name 1⟩
    | +-⟨server name 2⟩
    | | ···
    | +-⟨server name j⟩
    +-wrk
      +-⟨task name 1⟩.f
```

```
+-⟨task name 1⟩.g
+-⟨task name 1⟩.s
+-⟨task name 2⟩.f
+-⟨task name 2⟩.g
+-⟨task name 2⟩.s
| ...
+-⟨task name k⟩.f
+-⟨task name k⟩.g
+-⟨task name k⟩.s
```

3.2 Compilation

3.3 Processing

3.3.1 Statuses

Processing of any *task instance* begins when it is *submitted* with all its parameters and features substituted with values. The task instance can be submitted either(i) manually by operational support (typically the new task instance), or (ii) automatically when the previous instance of the same task has just finished successfully.

The submitted instance of the task does not start immediately. More accurately, it is ready to run and it is actually started when all task instances it depends on get into required statuses as specified by its start condition. At this moment the status of the task instance is changed to *running* and its code is passed to the executor for execution.

If the processing is finished successfully without any error the status is set to `succeeded` and the next run of this task instance is scheduled and submitted. Otherwise, the status is changed to `failed`.

The failed task instance can be *recovered.* It depends on a selected recovery method and a number of recovery attempts. The recovery means that the task instance status is changed back to `submitted`. However, if a number of retries has been already exhausted the task instance is switched into the status `suspended`.

A task instance in any of statuses except the `running` status can be *blocked.* Once blocked the instance can be *unblocked.*

If the task instance need to succeed the number of its recovery attempts must be set to `unlimited`.

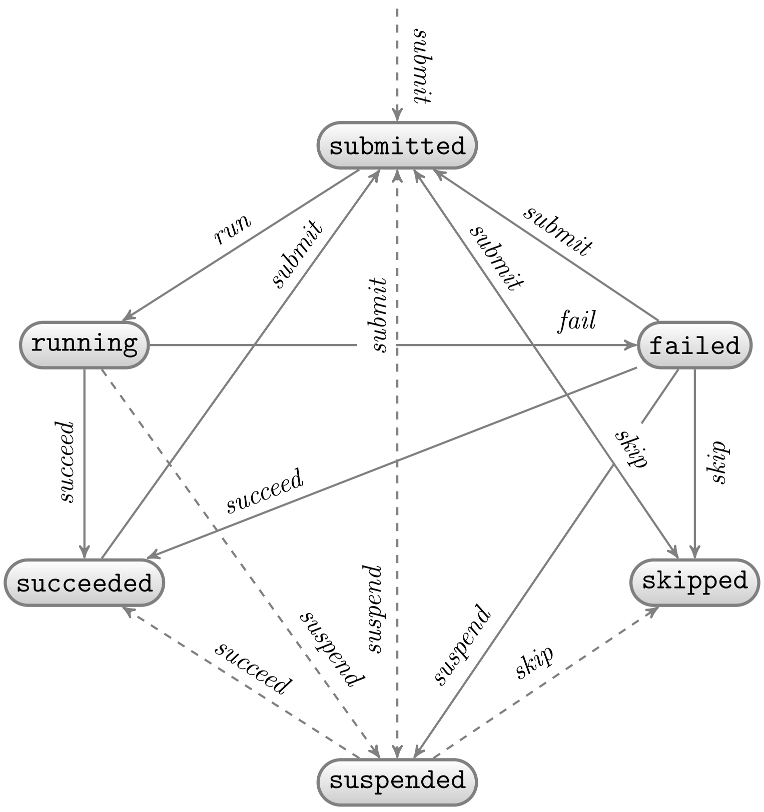


Figure 3.1: Run status transition diagram.

3.3.2 Caller Status Evaluation

ATTENTION: If

$$\mathtt{A}_1 \ \texttt{->}\ \mathtt{A}_2 \ \texttt{->}\ \ldots \ \texttt{->}\ \mathtt{A}_n,$$

then $\mathtt{A}_i$ cannot run with the next value of `TimeStamp` before $\mathtt{A}_n$ has succeeded or skipped for any $i \in \{1, \ldots, n\}$. Otherwise, an object created by $\mathtt{A}_i$ and used by some $\mathtt{A}_j$, $i < j$, would be recreated by a new run of $\mathtt{A}_i$.

A || B		B						
		submitted	running	failed	suspended	succeeded	skipped	–
A	submitted	submitted	running	running	running	running	running	submitted
	running	running	running	running	running	running	running	running
	failed	running	running	failed	failed	failed	failed	failed
	suspended	running	running	failed	suspended	suspended	suspended	suspended
	succeeded	running	running	failed	suspended	succeeded	succeeded	succeeded
	skipped	running	running	failed	suspended	succeeded	skipped	skipped
	–	submitted	running	failed	suspended	succeeded	skipped	–

Table 3.1: Status of `A || B`.

A -> B		B					
		submitted	running	failed	suspended	succeeded	skipped
A	submitted	submitted	–	–	submitted	–	–
	running	running	–	–	running	–	–
	failed	running	–	–	failed	–	–
	suspended	suspended	–	–	suspended	–	–
	succeeded	running	running	failed	suspended	succeeded	succeeded
	skipped	running	running	failed	suspended	succeeded	skipped

Table 3.2: Status of `A -> B`.

In case of `||` tasks `A` and `B` can have different schedules.
Example:

```
schedule Daily   0 & 0 & * & * & * & *
;
schedule Monthly 0 & 0 & * & 1 & * & * # the last day of a month
;
task main
   submit same after 5 m 12 times then suspend when failed
   submit next when succeeded
   call AddDayToTemp || AddMonth
;
task AddDayToTemp
   with schedule = Daily
   ...
;
task AddMonth
   with schedule = Monthly
   run when AddDayToTemp succeeded
   call AddTempToStats -> DelTemp
;
task AddTempToStats
...
;
```

3.3.3 When Condition Evaluation

if `A.timestamp` $<$ `B.timestamp` $=$ `main.timestamp`

condition	status of `A`					
	`submitted`	`running`	`failed`	`suspended`	`succeeded`	`skipped`
`A submitted`	`run`	`wait`	`wait`	`wait`	`skip`	`skip`
`A running`	`wait`	`run`	`wait`	`wait`	`skip`	`skip`
`A failed`	`wait`	`wait`	`run`	`wait`	`skip`	`skip`
`A suspended`	`wait`	`wait`	`wait`	`run`	`skip`	`skip`
`A succeeded`	`wait`	`wait`	`wait`	`wait`	`run`	`skip`
`A skipped`	`wait`	`wait`	`wait`	`wait`	`skip`	`run`

Table 3.3: Evaluation of a condition.

&	run	wait	skip
run	run	wait	skip
wait	wait	wait	skip
skip	skip	skip	skip

Table 3.4: Evaluation of &.

|	run	wait	skip
run	run	run	run
wait	run	wait	wait
skip	run	wait	skip

Table 3.5: Evaluation of |.

Conditions on non-final statuses (i.e., `submitted`, `running`, `failed`, or `suspended`) have *time variants*, such as

```
A running longer then 1 h
```

Just mentioned condition is evaluated to `run` if `A` is running longer then 1 hour; to `skip` if it has already succeeded or skipped; and to `wait` otherwise. Similar definitions hold for other non-final statuses as well.

Conditions on non-final statuses can be used for processing diagnostics. For example, abortion of a task running longer than expected can be done together with some cleaning. A task recovery then makes a task to run once again.

EXAMPLE!!! NUTNÝ JINÝ SCHEDULE WHEN SUCCEEDED A JINÝ WHEN SKIPPED. WHEN SKIPPED BUDE STEJNÝ SCHEDULE JAKO PRO TAKTO OŠETŘENÝ TASK (FROM LAST), WHEN SUCCEEDED BUDE NAPŘ. PO 5-TI MINUTÁCH FROM NOW.

CO KDYŽ TIMESTAMP ZOTAVUJÍCÍHO TASKU PŘEDBĚHNE NÁSLEDUJÍCÍ TIMESTAMP ZOTAVOVANÉHO TASKU? NUTNO NĚJAK ZAŘÍDÍT OPĚTOVNÉ SPUŠTĚNÍ ZOTAVUJÍCÍHO TASKU SE STEJNÝM TIMESTAMP. NAPŘ. TAK, ŽE ZOTAVUJÍCÍ TASK PO ÚSPĚŠNÉM ZOTAVENÍ ZOTAVOVANÉHO TASKU PŘEJDE DO STAVU FAILED. (TO JE MATOUCÍ, BYL BY FAILED I KDYŽ ÚSPĚŠNĚ ZOTAVIL. LEPŠÍ BUDE SCHEDULE WHEN SUCCEEDED THEN SAME, NEBO NEPOSUNOVAT TIMESTAMP, KDYŽ NENÍ DEFINOVÁN SCHEDULE WHEN SUCCEEDED.)

3.3.4 When to start a new task instance run

conjunction of the following conditions holds:

- current (i.e., the last) task instance run (if exists) is succeeded or skipped
- when condition is fulfilled
- all dependent (both horizontaly and verticaly) task instance runs are succeeded or skipped
- current (i.e., the last) task instance run (if exists) cannot be failed (by fail caller when failed rule)

3.3.5 Error Handling–when failed Rules

3.3.6 Submission–when succeeded/skipped Rules

Chapter 4

Development

4.1 Servers

4.2 Recursion

Exeta offers a possibility to assemble a task of different subtasks dynamically based on a list of values passed to the task by its call in the last identifier.

This feature is demonstrated here by the following example: Let's assume that `DW_PRODUCT` is a data warehouse table that integrates data from several sources (let's say CRM, ERP, Billing System) using a different historization method for each source. We would like to have just one generic task called `LoadTgtTable` with a parameter `Table` and one feature `SMList` that calls different subtasks

- `PrepWrkTab.${Method} ${Table} ${Source}`
- `LoadWrkTab.${Table}.${Source}`
- `LoadTgtTab.${Method} ${Table} ${Source}`

based on pairs of `Source` and `Method` in `SMList`. It can be accomplished in the following way:

```
task LoadTargetDaily
call LoadTgtTable DW_PRODUCT
     with SMList =
          (  CRM SCD2
             ERP SCD1
             BS  SCD2
          )
;
```

```
task LoadTgtTable Table
call LoadTgtTabClone ${Table} ${SMList}
;

task LoadTgtTabClone Table SMList : Source Method
call LoadTgtTabClone ${Table} ${SMList}
  || (  PrepWrkTab.${Method} ${Table} ${Source}
     -> LoadWrkTab.${Table}.${Source}
     -> LoadTgtTab.${Method} ${Table} ${Source}
     )
;
```

4.3 Error Handling and Recovery

```
task RecoverHangingTasks
call fail ( T1 Id1 Id2 Id3 ) when T1 Id1 Id2 Id3 running 30 m
  || fail ( T2 A B )         when T2 A B         running  1 h
  || ...
;
task fail T
with executor = exeta
succeed when failed
submit same when succeeded
submit next when skipped
execute
;
```

`fail` is a system task that aborts and fails a task specified as its identifier.

Chapter 5

Operations

5.1 Monitor

`submitted` *instance*

`running` *instance*

`failed` *instance*

`suspended` *instance*

`succeeded` *instance*

`skipped` *instance*

`status` *instance*

`tree` *instance*

`predecessors` *instance*

`successors` *instance*

5.2 Control

`submit` [{ *timestamp* } [`with` [`all`] { `predecessors` | `successors` }]] *instance*

This operation submits all suspended instance runs in a subtree rooted in the task *instance*. (If there is any suspended instance run in the subtree then its root is suspended too.)

If the instance run determined by the *timestamp* is not suspended, then the operation submits a new run with `timestamp` set to the *timestamp* (if specified) or to the next timestamp prescribed by instance's schedule.

If the *timestamp* precedes the current timestamp then all already run instances following the *timestamp* are *forgotten*.

If the *timestamp* matches no scheduled timestamp then the nearest following timestamp is used.

`fail` *instance*

`suspend` *instance*

Cancels all submitted, running, or failed task instances in a subtree rooted in the *instance*.

`succeed` *instance*

Succeeds all suspended task instances in a subtree rooted in the *instance*.

`skip` *instance*

5.3 Deploy

```
suspend main
status main
```

Wait until `status main` returns `succeeded` for all `main` instance runs. Then

```
deploy main
submit main
```

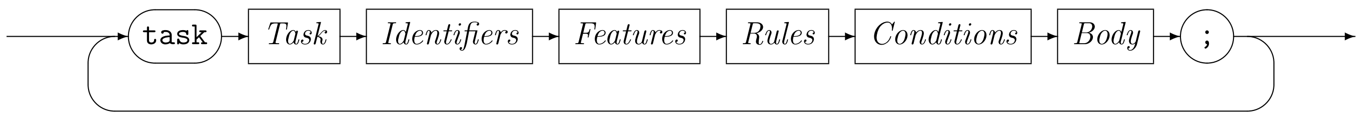
5.4 Reload

`submit with all successors` *timestamp instance*

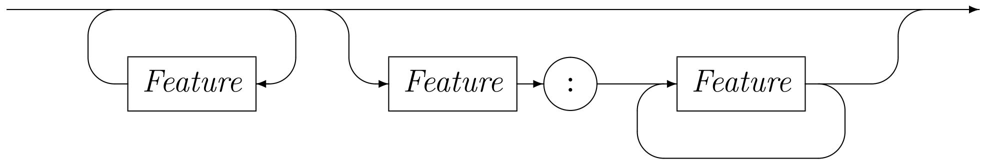
Appendix A

Syntax

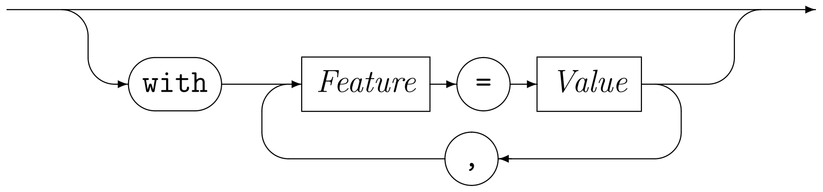
Exeta



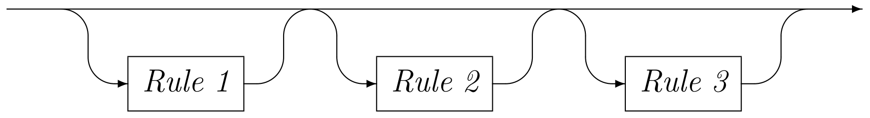
Identifiers



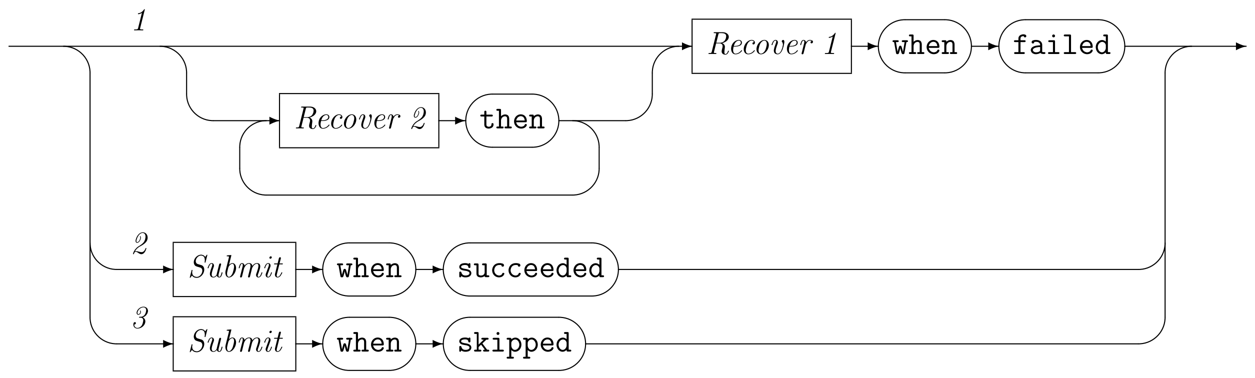
Features



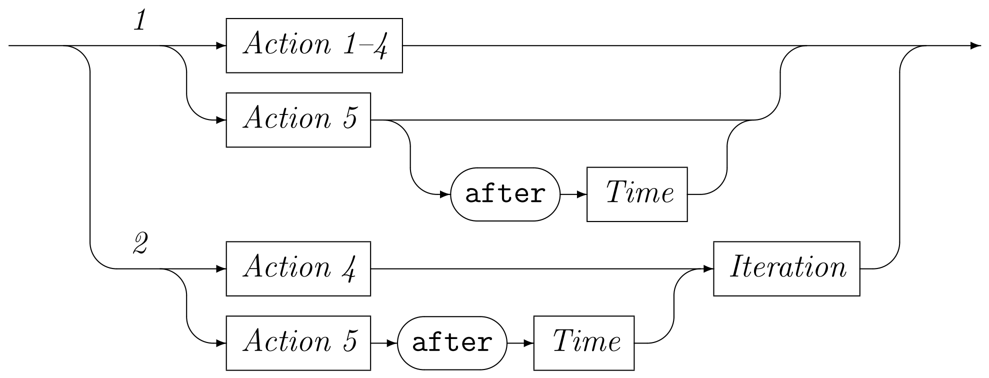
Rules



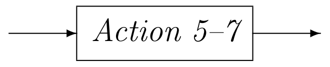
Rule



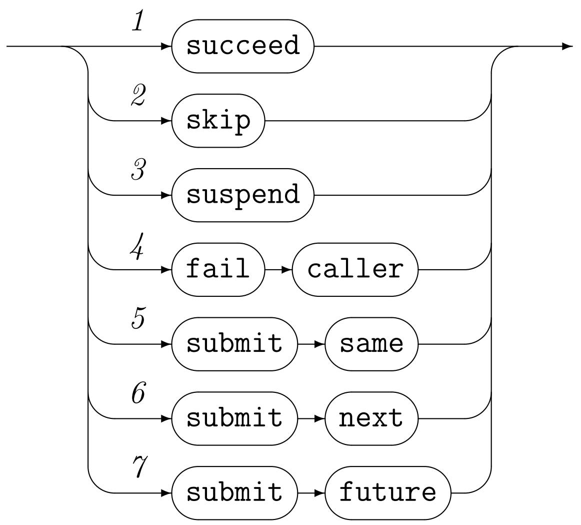
Recover



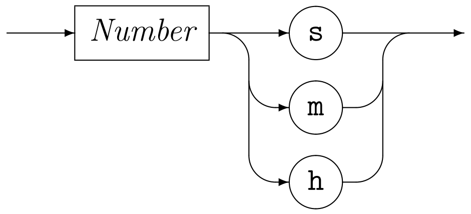
Submit



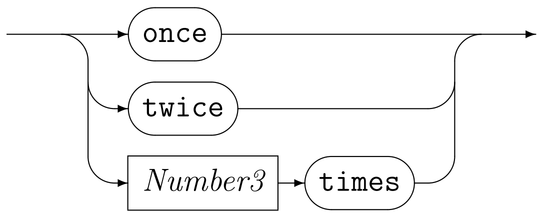
Action


1
succeed
2
skip
3
suspend
4
fail
caller
5
submit
same
6
submit
next
7
submit
future

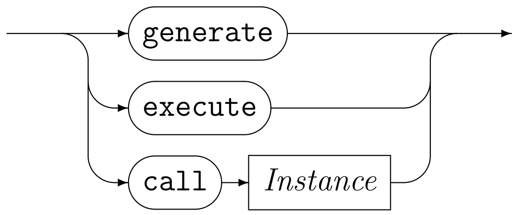
Time


Number
s
m
h

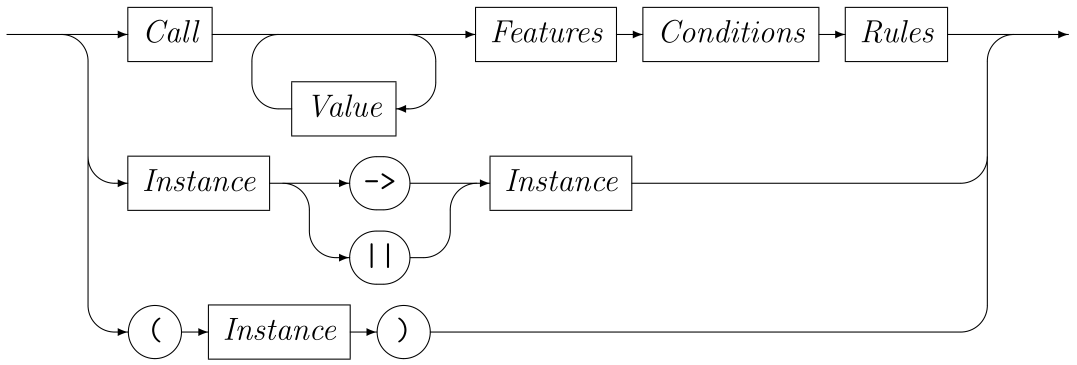
Iteration



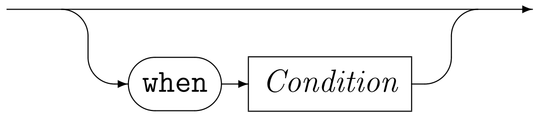
Body



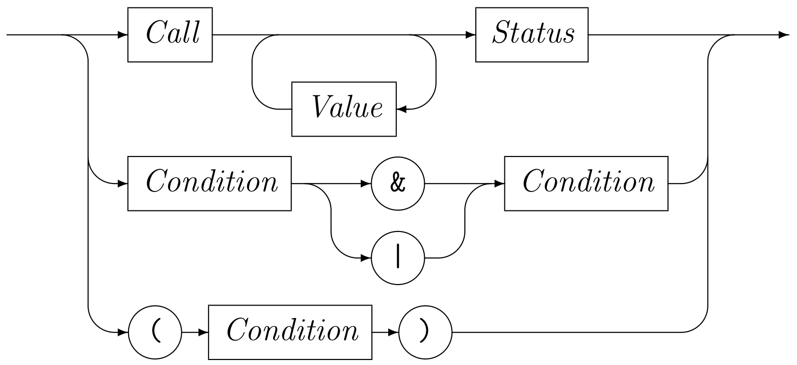
Instance



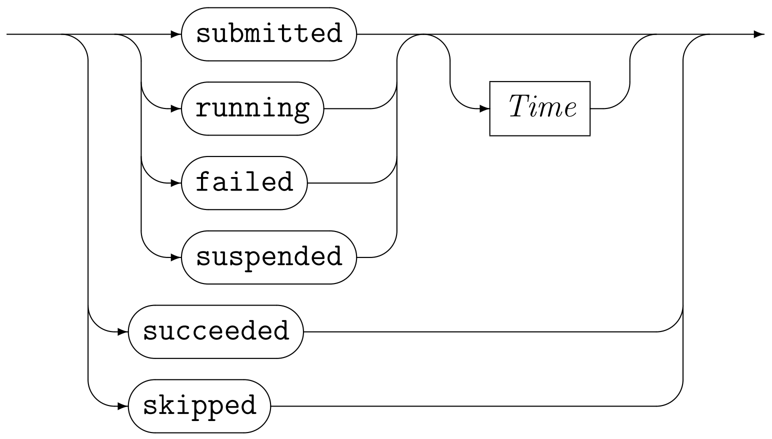
Conditions



Condition



Status


submitted
running
failed
suspended
Time
succeeded
skipped