Abstract

This document represents the textual part of the Deliverable D2.4 which includes the 3\textsuperscript{rd} version of the supervision prototype: it highlights the new features of the prototype, describes the implemented components and provides some hints on how to design a supervision system and use the prototype.

Moreover, it describes the activities that WP2 is developing in co-operation with other WorkPackages and some intermediate results, demonstrating the level of integration reached so far with the other parts of CASCADAS Project.

The source code of the prototype is available as open source on SourceForge at http://sourceforge.net/projects/acetoolkit/.
# Table of Content

1 Introduction 4
   1.1 Purpose and scope 4
   1.2 References 6
   1.3 Document History 7

2 Prototype architecture and functional features 8
   2.1 Language for Correlation and Assessment 9
      2.1.1 ACECol 9
      2.1.2 CorAL 10
      2.1.3 AL 12
   2.2 Planning and Effection 13
   2.3 Long Term Supervision – StatePredictor 14
      2.3.1 ACE Types and ACE Ensembles 15
      2.3.2 Execution Models 15
      2.3.3 Passive vs. Active Modelling 17
   2.4 Clustering of supervised ACEs with supervisors ACEs 17
      2.4.1 Placing Supervisors 19
      2.4.2 Aggregating Supervisors 19
      2.4.3 Application Example – Model Based Supervision 19

3 Prototype components 21
   3.1 Supervised-supervisor interfaces 22
   3.2 Sensor 23
   3.3 Correlator 25
   3.4 Assessor 27
   3.5 Planner 28
   3.6 Effector 29
   3.7 Long Term Supervision components 30
      3.7.1 StatePredictor prototyping 30
      3.7.2 Current state of the integration in Supervision prototype 32

4 Process for setting-up a Supervision System 32

5 Co-operation activities with other WPs 36
   5.1 Supervision Prototype as an extension of ACE toolkit 37
   5.2 Self-* features of Supervision 38
Bringing Autonomic Services to Life

5.2.1 ACE toolkit embedded supervision and Self-adaptation 38
5.2.2 ACE toolkit embedded supervision and self-organization algorithms 41
5.2.3 Clustering for Supervisors/Supervision association 43
5.3 Security features of Supervision 44
5.3.1 Self-supervision for monitoring ACEs 44
5.3.2 Reputation to assist the supervision system 44
5.3.3 Application scenario 45
5.4 Supervision and Knowledge Network framework 45
5.5 Supervision in Cascadas demonstration scenario 47

6 Future Work 48

7 Conclusions 49

8 Appendix - User guide 50
8.1 Plan templates for supervision components 50

8.2 The Supervision Languages 55
8.2.1 ACECol 55
8.2.2 CorAL 61
8.2.3 AL 71
Bringing Autonomic Services to Life

1 Introduction

This deliverable presents an intermediate result, planned at M30 [DIP], towards the achievement of the main objective of WP2 for the last year of the Project: complete the development the supervision framework for the CASCADAS toolkit, and demonstrate its validity in an integrated context.

The deliverable consists of both a software part and a textual part (this document).

The software provides the 3rd version of the supervision prototype, developed according to the pervasive supervision model [D2.3]. The source code of the prototype is prepared to be provided as open source on SourceForge at http://sourceforge.net/projects/acetoolkit/.

This companion document highlights the enhancements introduced in the new version of the prototype, describes the implemented components, and provides a guide on how to design a supervision system and use the prototype. Moreover, it describes the activities that WP2 is developing in cooperation with WP1, WP3, WP4, WP5 and WP6 and some intermediate results, demonstrating the level of integration. This part of the document anticipates some of the contents planned to be described in WP2 final deliverable (D2.5, planned at M36).

1.1 Purpose and scope

The software part of Deliverable D2.4 implements the 3rd version of the prototype of the ACE based supervision, according to the pervasive supervision model.

ACE based supervision is the supervision mechanism investigated in CASCADAS which monitors and manages according to some service-oriented management policies, for instance groupings/patterns of components which might be orchestrated by some business logic. It is complementary and co-operate with the ACE tool-kit embedded supervision, oriented to the supervision of huge amount of components, through a highly distributed approach. ACE based supervision is implemented through collaborative supervisors providing a supervision service: each supervisor is an aggregate of ACEs, providing specific supervision components/functions, such as, assessor, correlator, DriftAnalyser, etc. [D2.2]. A detailed description of the supervision approach and mechanisms adopted in CASCADAS are reported in [D2.3].

This new version of the ACE-based supervision prototype enhances the previous one, described in [D2.3], on several features, including:

- integration in some of supervision components of an interpreter for an improved version of the supervision languages to define logic for filtering, correlating, aggregating, and assessing
- introduction of a planner and effector components following the model based approach described in [D2.1m];
- introduction of new features for interrogative monitoring and advanced control into the ACE toolkit, to improve the interworking with the supervision components;
- prototyping (and partial integration in the supervision prototype) of long term supervision components; in particular a new component, the StatePreditor component, was defined and prototyped.

The first part of this document describes the advances on the ACEs based supervision approach (in Section 2) and the components of the 3rd version of the supervision prototype
Bringing Autonomic Services to Life

(in Section 3), implemented through the ACE toolkit developed by WP1. Moreover, Section 2 introduces some considerations concerning the issue of aligning a supervision pervasion with the structure of the system configuration under supervision, by employing the self-aggregation mechanisms developed in WP3. Section 4 describes how to design a supervision system, by using the pervasive supervision approach and the supervision prototype. This first part is complemented by an Appendix providing some details on how programming/configuring the supervision components: in particular, it describes the templates for the definition of self-models of the ACEs implementing the supervision components and details the syntax of the supervision languages.

The second part of this document provides a report on the activities performed in collaboration with other Workpackages, in order to demonstrate the level of integration of the work on supervision. Such cooperation activities are aiming at:

- improving the adoption of solutions elaborated by other CASCADAS WPs (in particular, WP3 and WP4) in the supervision mechanisms;
- demonstrating the applicability of supervision solutions in other contexts, such as in WP4 and WP5;
- guaranteeing a stronger integration of WP2 supervision prototype with the ACE toolkit developed by WP1 and the demonstration scenario elaborated in WP6.

Section 5 presents such activities and some of the preliminary results.

Section 6 and Section 7 complete the document with the description of the future works and some conclusions, respectively.
1.2 References


[D1.2] CASCADAS D1.2, “Prototype implementation (release 1)”, (December 2007).

[D1.3] CASCADAS D1.3, “First Prototype Integration (release 1)”, (December 2007).


## 1.3 Document History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Authors</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>June 10th, 2008</td>
<td>Corrado Moiso</td>
<td>ToC finalised at London meeting</td>
</tr>
<tr>
<td>0.2</td>
<td>June 26th, 2008</td>
<td>Deliverable contributors</td>
<td>First integrated draft for internal reviews</td>
</tr>
<tr>
<td>0.3</td>
<td>July 2nd, 2008</td>
<td>Corrado Moiso, integrating inputs from co-authors</td>
<td>Implementation of comments from internal reviewers and contributors</td>
</tr>
<tr>
<td>1.0 - Final</td>
<td>July 4th, 2008</td>
<td>Corrado Moiso</td>
<td>Final version</td>
</tr>
</tbody>
</table>
2 Prototype architecture and functional features

This section summarizes the new features developed in the 3rd version of the supervision prototype. For a better reference let us first summarize the architecture of the prototype.

Figure 1 – Prototype architecture

Figure 1 shows the principle architecture of the toolkit embedded supervision prototype. For sake of simplicity, only one instance of each component is shown, while in practice there will be always a number of supervised ACEs, sensors, effectors, etc. The use of interaction protocols is indicated by coloured arrows.

The following achievements have been made on the supervision prototype:

1. An implementation of the supervision languages (i.e., ACECol, CorAL, and AL) has been provided (Section 2.1). Interpretation functions for the several language fragments (filtering, correlation, aggregation, and assessment) have been integrated into the prototype architecture. These sublanguages are integrated as interpretation functions in the Sensor, Correlator, and Assessor, respectively; with respect to the versions described in [D2.3] the syntax of the supervision languages was cleaned and evolved towards a Java-like style: this facilitates the designers since they can now adopt a single level of abstraction throughout the entire design process.

2. A planner and effector functions following the model based approach described in [D2.1m] have been provided (Section 2.2). The planner is invoked when the system configuration under supervision enters a critical or otherwise non-desirable state, and provides so-called contingency plans which are supposed to lead the system back into a normal operational state. Contingency plans are deployed by using effectors as controlling functions in the ACEs of the system under supervision. In order to extend the monitoring and control features which an ACE under supervision offers to its supervisor, extensive interrogative features have been implemented; moreover, the notion of passive and active controllers have been introduced.
Bringing Autonomic Services to Life

3. The long term supervision components have been prototyped and partially integrated with the other supervision components; in addition to the DriftAnalysier and Event Predictor components, the StatePredictor component, whose aim is to predict the future direction that an ACE might take, was investigated and prototyped; Section 2.3 specifies in detail the structures and methods used to implement the this new component.

Finally, Section 2.4 reports some considerations concerning the issue of aligning a supervision pervasion with the structure of the system configuration under supervision, obtaining a suitable coverage for control and observation. As the system under supervision is a self-organized entity, self-aggregation mechanisms developed in WP3 (clustering and reverse clustering) have been employed in order to associate supervised ACEs with the supervision structure.

2.1 Language for Correlation and Assessment

The languages have been radically simplified for the third evaluation prototype. The modifications have not compromised the expressive powers of the languages. We have simply cleaned the syntax, providing users with a language that fits better with the ACE paradigm. In the following we will present the languages:

- ACECoL (ACE Constraint Language), for defining filtering properties within the Sensor component;
- CorAL (Correlation ACE Language), for defining correlation properties within the Correlator component;
- AL (Assessment Language), for defining assessment properties within the Assessor component.

We introduce specific examples to highlight the potentiality of the new language but also to facilitate the reader understanding the basic principle behind the definition and the usage of the language. The syntax of the languages is reported in the Appendix.

2.1.1 ACECoL

The goal for ACECoL is to produce a simple Java-like property language that could be used within the Sensor component. The Sensor receives Java objects from other ACEs present in the system. The filtering property defines which of these objects are of interest for supervision and which one can be discarded, i.e., those that do not satisfy this filtering property. A general ACECoL filtering property complies with the following template:

```
filter [event=className: property]
```

A designer is called to define two pieces of information. The first is the name of the event type he/she is interested in capturing by means of the sensor. This is specified in the `className`, which is a fully qualified Java class name. The second is a parametrical property that predicates over a single event of type `className`. The parameter is the event itself, which can be indicated by using the special keyword “event”. For example, to indicate that we are only interested in events of the type “Room” that are sent by the room with ID equal to “living room” or by rooms whose ID contains “bedroom”, we write the following.

```
filter [event=domoticexample.Room :
{event}.getRoomID().equals("living room") == true ||
{event}.getRoomID().contains("bedroom") == true ]
```
Bringing Autonomic Services to Life

The event is of the type “Room” and is defined in the package “domoticexample”. The property is a Java expression. The language we provide is mainly a superset of the Java language, with the addition of the following constructs for dealing with sequences of values:

- **forall** – this construct allows us to state that all the data in a given set need to satisfy a certain property. For example,

  ```java
  filter [event=weatherexample.WeatherSnapshot :
  (forall i = 0 to {event}.getCities().size() -1;
  {event}.getCities().get(i).getTemperature() > 15 )]
  ```

  predicates on an event of type “WeatherSnapshot”. Each object of this type contains a list of “City” objects which can be retrieved by using the method “getCities”. For each city the method “getTemperature” can output the current temperature in that city using method “getTemperature”. The property uses a forall construct to state that all the temperature values must be greater than 15 degrees Celsius. Notice the use of the variable “i” to refer to the i-th element in the list.

- **exist** – this construct is similar to the forall construct except that the sub-property within the exist construct must be true for at least one of the data in the sequence. For example, the following property states that at least one of the cities must have a temperature greater than 20 degrees Celsius.

  ```java
  filter [event= weatherexample.WeatherSnapshot:
  (exist i = 0 to {event}.getCities().size() -1;
  {event}.getCities().get(i).getTemperature() > 20 )]
  ```

  Since all the events in our framework extend a common type (SupervisionData), we can always exploit the following methods when defining our properties:

  - **Date getTime():** returns a Date indicating when the event was first seen by the Sensor component.

2.1.2 CorAL

CorAL is the language we use to correlate events that are received by the Correlator component. The goal of this component is to aggregate events that originate from different sources and that have reason to be treated together when performing supervision. In our framework, designers will typically use Sensor objects as correlation sources. Obviously, each Sensor can provide more than one source for correlation. The result of the correlation is the creation of an aggregate data structure. The structure is a table with \( n \) columns, where \( n \) is the number of sources we are dealing with. Therefore, each row is a tuple made up of one element per source.

Correlation is a two-step procedure. First, the designer must define how to sample the event sequences: one sequence for each event source (Sensor) being used. This can be achieved by complying with the following template:

```java
define {
  sequenceName1: sequence of className in window (timeAmount)
  where sequenceProperty;
  sequenceName2: _
}
```

Each sequence defines the type of event to collect, a time frame for performing the collection, and a sequence filtering property. The goal of the filtering property is to exclude
some of the events in the sequence. When predicating on the events in the sequence, we provide two special keywords. The first is the “index”. It represents the position of a generic event in the sequence. The second is the “length”. It represents the number of events collected in the sequence. For example,

```
define {
    seqA: sequence of domoticexample.Room in window (6 s)
        where index == length -1;
    seqB: sequence of domoticexample.Room in window (3 s)
        where index % 2 == 0;
}
```
defines two event sequences. The first, “seqA”, collects “Room” objects for 6 seconds, and only keeps the last one. The second, “seqB”, collects “Room” objects for 3 seconds, and only keep those that are in an even position within the sequence.

Once we have all the relevant events, we perform a Cartesian product among the events in the sequence, and only keep those that satisfy a correlation property that predicates on events coming from different sources. To define the correlation, designers can use the following template:

```
correlate ( $varName1 in sequenceName, $varName2 in sequenceName, …)
every timeAmount
where correlationProperty;
```

In the first part of the template, the designer defines alias names for the events that will be involved in the Cartesian product. The designer has to define one alias name for each of the sequences involved, whereas the sequence names are the same as those given in the previous definition section. These aliases will then be used to refer to the events in the correlation property. In the second part of the template, the designer specifies how often the correlation needs to take place. Every `timeAmount` that passes, the Correlator component starts collecting new sequences. As soon as the sequences are complete, it checks property `correlationProperty`.

A complete example follows:

```
define {
    seqRooms: sequence of domoticexample.Room in window (10s);
    seqDesired: sequence of domoticexample.DesiredTemperature
        in window (10s) where index = length-1;
}
correlate ($a in seqRooms, $b in seqDesired)
every 2s
where ($a.getTemperature() - $b.getDesiredTemperature()) > 5 ||
    ($b.getDesiredTemperature() - $a.getTemperature()) > 5;
```

In this example, we have defined two event sequences. The first, called “seqRooms”, collects all the “Room” events received by the Correlator component in a 10 seconds time frame. The second, called “seqDesired”, collects “DesiredTemperature” events for 10 seconds, and discards all events but the last one it receives. Every 2 seconds we try to correlate data. This means that every 2 seconds we start collecting two new sequences, one of type “seqRooms” and one of type “seqDesired”. Once the sequences are ready, we perform the Cartesian product between the elements of “seqRooms” and the elements of
Bringing Autonomic Services to Life

“seqDesired”, and only keep those tuples that satisfy the correlation condition. In this example, the condition states that we keep only those tuples in which the element received from “seqRooms” has a temperature that does not differ more than 5 degrees Celsius from the desired temperature read from the event received from “seqDesired”.

2.1.3 AL

AL is the language we use for defining assessment properties for the data correlated by the Correlator component. The correlator correlates data from different sources. Since we cannot know when the data will arrive, the correlator collects the data in sequences. Sequences are limited by time constraints. Once all the sequences are complete, we perform a Cartesian product. At this point we have many tuples. Each tuple is made up of one datum from each sequence. Some of these tuples are thrown away, and some are kept. This depends on a correlation property. Those that are kept are sent to the assessor.

The assessor component can treat the data it receives using a table in which each row is a tuple of events that satisfy the correlation property. This gives designers a means to refer to the entire set of tuples received by the assessor, using the methods it provides. With AL, we provide two kinds of assessments: single or multiple. The difference lies in the object of the assessment. If we are assessing a property of the entire table, our goal is to produce a single truth-value, so we say we perform a single assessment. If we are assessing a property of the rows in the table, our goal is to produce a truth-value for each row in the table, so we say we perform a multiple assessment.

When defining assessments, designers can use the following template:

```
assessmetntType assessment: assessmentProperty ;
```

The assessmentType can be either “single” or “multiple”, while the assessmentProperty is defined by the same filtering language used in ACECoL. The only difference is that, in single assessment we can directly reference the entire table using the special keyword “table”, while in multiple assessment, besides the entire table, we can also directly reference the tuple being treated using the special keyword “row”.

When dealing with the table, we can call on the following methods:

- `int getNumberOfRows()`: returns the number of rows in the table
- `int getNumberOfColumns()`: returns the number of columns in the table
- `String[] getColumnLabels()`: returns an array of String with the names of the columns in the table
- `CorrelatedDataRow getRow(int i)`: returns the i-th row in the table
- `SupervisionData getData(int row, int column)`: returns the SupervisionData at the position <row, column> in the table
- `SupervisionData getData(int row, String columnName)`: returns the SupervisionData in row number row and under the column named columnName

When dealing with a single row, we can call on the following methods:

- `int getNumberOfColumns()`: returns the number of columns in the row
- `SupervisionData getData(int column)`: returns the SupervisionData in column number column
Bringing Autonomic Services to Life

- SupervisionData getData(String columnName): returns the SupervisionData under the column named columnName
- SupervisionData[] toArray(): returns an array of SupervisionData with all the data belonging to that row

Two examples taken from our ongoing room temperature example follow:

- Single assessment:
  ```java
  !(exist i=0 to {table}.getNumberOfRows();
  {table}.getData(i,0).getTemperature() > {table}.getData(i,1).getDesiredTemperature() );
  ```

- Multiple assessment:
  ```java
  {row}.getData(0).getTemperature() > {row}.getData(1).getDesiredTemperature();
  ```

In the first example, we state that in the entire table there should never be a row in which the “Room” object in the first column has a temperature that is greater than the desired temperature contained in the “DesiredTemperature” object of the second column. This is expressed by using a negated exist construct, whose variable i can vary between 0 and the number of rows in the table.

In the second example, we declare our interest in obtaining one truth-value for each row in the table. Indeed, we want to know which rows in the table have a “Room” object in the first column whose temperature is greater than the temperature contained in the “DesiredTemperature” object in the row’s second column.

2.2 Planning and Effection

[Deu07, D2.3] introduced a planning algorithm based on the computation of so-called pomtrees as contingency plans to lead a system under supervision from a non-desirable state back to a desirable one. In the course of the Project, a number of adjustment to this first algorithm has been made to take into account the newly introduced mechanisms for controlling ACEs under supervision based on passive and active controllers.

Since transitions are enabled in specific plan states, a contingency plan, which is local to a certain plan execution can be expressed as a pair of mappings \((p, a)\), where \(p: State \rightarrow Set<Transition>\) maps a plan state into the set of transitions allowed to be executed in this state, and \(a: State \rightarrow Set<Event>\) defines the set of events to be issued by a supervisor in order to trigger certain transitions.

**Remark:** It should be noted this is mostly a notationa l difference. We sketch the mapping between a passive controller and a pomtree plan as follow: Let \(x = (E, <, \lambda)\) be a pomtree and let \(x_p\) be the pomtree obtained by restricting \(x\) to the events of an ACE local plan \(\pi\) (in fact, \(x_p\) is a pomset). Then since ACE plans are executed in a sequential way, \(x_p\) is linearly ordered, thus can be expressed as a set of sequences \(e_1, e_2, \ldots, e_n\) of events forming a decision tree. Let \(s_0\) be the state of \(x_p\) which is reached after executing \(e_1, e_2, \ldots, e_k\) for \(k \leq n\) as a start state \(s_0\) (the state at which the planning activities are started) and let \(f_1, f_2, \ldots, f_m\) be the events which are direct successors of \(e_k\). Then \(p(s_0) = \{f_1, f_2, \ldots, f_m\}\).

The algorithm maintains a list of plans which are currently executed by the ACE ensemble under supervision, together with references to the ACEs to which they belong, and the current contracting structure. This information is requested from the respective Sensor ACEs which use the newly implemented interrogation function (compare Section 3.1) to obtain it from the ACEs under supervision. Moreover, for each plan an input queue and a current state are represented in the data structures of the algorithm. Finally, an integer
Bringing Autonomic Services to Life

value is maintained for the plan which is used as depth parameter to terminate the recursive search – we currently set this value to the total number of states of the associated plan.

In each of its recursive instances the algorithm checks whether all the states of all considered plans have a desirability value which equals normal operation (i.e. equals 1 using the conventions introduced in Section 4.1). If so, it returns true. Otherwise, it iterates through the list of plans and checks if the depth value for each plan equals the maximum depth value. If not, the algorithm considers the event at the front of the input queue of the plan. If this event triggers a transition, then the algorithm checks:

1. Is the event a supervision control event? If so, it is added to the active controller mapping for the current state.
2. Is the action of the considered transition a send action? If so, the event to be send is included into the input queues of the receiving plans: for this, both the type of the event [ServiceUsageEvent for inter-ACE communication and ACELocalEvent for intra-ACE communication] and the currently maintained contracts need to be considered.
3. Then the current state of the current plan is updated, the depth value increased, and another recursive call of the algorithm is issued.

Next, we iterate through all transitions departing from the current state which are either automatic or triggered by an event which is not presented in the input queue of the current plan. We need to consider these transitions also because we do not know whether their trigger events are probably issued from the environment of the ACE ensemble, or by repository functions different from the send action. Again, the current state is updated, the depth value is increased, and another recursive call is done.

2.3 Long Term Supervision – StatePredictor

Concepts relevant for long term supervision are threefold and are covered by three types of components which are summarised as follows:

- **DriftAnalyser**: The DriftAnalyser is the first component that has been realised and it is specifically targeted for the observation, analysis and forecasting of quantifiable concepts such as numeric properties of the system under supervision or of the environment it is operating in. The DriftAnalyser component has already been defined in detail in previous deliverables (e.g., in [D2.3]), and will not be discussed further.

- **EventPredictor**: The focus of the EventPredictor is to predict the time window in which a certain event is most likely to occur. Based on the monitoring of past events it provides a static as well as a dynamic time statement around which a given type of event may reoccur. Again, this component has already been defined and discussed in detail in previous deliverables (e.g., in [D2.3]).

- **StatePredictor**: The StatePredictor is specifically designed to observe and predict the execution logic of ACEs as represented by their selfmodel. To be more precise the StatePredictor component is designed to (a) monitor the execution of ACEs, (b) build an execution model that is based on such observations and reflects mass behaviour and (c) predict the potential next states an ACE can step into based on the developed execution model. For instance, these potential candidate states an ACE can step into may then be used to e.g. prevent illegal or dangerous behaviour of an ACE or to optimise the ACE’s behaviour in the long term.
Bringing Autonomic Services to Life

An initial prototype of the StatePredictor and details about the integration of all components into the overall supervision architecture are discussed in Section 3.7. Whereas the remainder of this section will specify in detail the structures and methods used to implement the StatePredictor component, which aim is to predict the future direction that an ACE might take with respect to its plan of execution as represented by its self-model, its own past behaviour or the behaviour of any other ACE that is somehow regarded as similar. This relation of ACEs with respect to observation will be discussed; then two types of execution models will be presented: both of them are suitable to model the execution of ACEs over time with the objective to predict potential future states. Finally, a passive as well as an active modelling method will be discussed.

2.3.1 ACE Types and ACE Ensembles

One aspect of the modelling is the way in which the model is constructed. Another aspect is which ACEs, or groups of ACEs, are to be represented by the same model. There are essentially three alternatives here. The first is to model each ACE separately, collecting data only on its individual use. This reflects the most specific use case in which each ACE is observed individually and for which any recommendation is based only on the behaviour of the component it is actually made for. A second option is to group ACEs of the same type together (e.g. ACEs that share the same self-model). This option would reflect more accurately mass behaviour as not only multiple users of the same ACE are analysed together but all available instances of the same type of service with all associated users. However, recommendations are no longer instance based (that is on the past behaviour of a single component) but they are based on the business logic that is shared between all ACE’s that are under supervision. The third option is to group ACEs based on other requirements, e.g., ACEs that perform a particular operation or service. These might be ACEs of different types that provide different functionalities but collectively perform a specific task. In this case it may be more difficult to collect information reflecting mass behaviour. This is based on the simple fact that, in this case, ACE’s are analysed together that actually have different self-models (or different transactions for that matter). Nevertheless, considering that ACEs may be ensembled frequently to provide specific services the observation thereof can prove useful to identify, e.g. performance variations as well as fraudulent behaviour.

2.3.2 Execution Models

While the probability of the prediction of future states will be based on the past behaviour, the use of the ACEs under supervision, potential next states, plan violations etc. will also depend on the execution model chosen to model the behaviour of ACEs. In order to model the execution logic of an ACE or any orchestration thereof, two different models are envisioned and illustrated in Figure 2. As depicted on the right side, a meshed model may be used to represent ACE specific execution. The rationale for this is based on the stateless method invocation of specific and common service functionality. This immediately relates to an undirected graph in which a collection of states may form short sequences to reflect individual service execution rather than long-term business goals. In fact only the current state transition is of interest independently of which is the state of the overall execution. Such a loose model of observation is ideal for short lived, stateless services where previous conditions are irrelevant.
In contrast, a directed graph, as depicted on the left of Figure 2, may be used to model more specific behaviour or cross ACE interactions, which are likely to be state dependent and, as such, they require a more rigid model. Within this model, the path of execution is also taken into account when registering a state change and also when predicting potential future states. Thus this model is designated for services that have to be observed and analysed as a whole rather than only on their current transition. That is form whenever they are invoked until they have finished, independently of their success.

For ACE ensembles that provide specific services at runtime, a combination of both models is feasible that utilises the meshed execution model at ACE level whereas the directed execution model is used for cross ACE interactions. Such a configuration is depicted in Figure 3, where the top represents the directed execution model, which is composed of entry and exit states of individual ACEs as represented by the meshed models at the bottom. Notable for such a configuration is that no interaction between individual predictor components is required as this is embedded within the logic of the ACE that provides the overall service. This is the only component that needs to be modelled using the directed execution model as its internal logic simply utilises the logic of other ACEs to provide a
Bringing Autonomic Services to Life

specific service for which each sub-service needs to be executive in sequence hence the directed execution model. Nevertheless, how feasible such a configuration is within the context of the ACE toolkit still needs to be explored.

In any case, the core objective for both execution models is the registration of individual state changes within ACEs in order to determine their occurrence on a global level. Other aspects of interest that could also be observed include:

- Reason for State Change
- Frequency (How often a state becomes active. This is different to a state change as a state can be reached from multiple previous states)
- Volatility (How long a state remains active)
- Validity of a state

Monitoring the occurrence of state changes forms the core objective, as subsequent predictions will be based upon this value. How probability measures can be calculated for each given state has already been outlined in Section 3.3.4 of [D2.3].

2.3.3 Passive vs. Active Modelling

Different modelling techniques could be envisaged:

- **Passive Modelling** relates to an a-priori translation of each self-model into a dedicated execution model. The structure and content of this plan will then remain immutable for the lifetime of the observation. This model will directly reflect the design features of an ACE and as such would be able to detect transitions that lead to illegal states.

- **Active Modelling** means that the execution model is created at runtime with no or only a reference to the actual self-model of an ACE. Starting with an empty model, subsequent states and their relations to previous states are constructed at runtime. Before added to the execution model individual states may be validated against the self-model to check for illegal or unwanted transitions.

- **Mixed Modelling** refers to a combination of both techniques. Considering that the logic of an ACE or any number of ACEs, change and evolve over time the underlying execution model may need to be extended or even pruned to accommodate for new or obsolete behaviour that was initially not foreseeable. A simplified execution model may be first created based on the self-model but instead of being immutable (as for the passive modelling) the model may be updated at any time. While this certainly promotes the dynamic and evaluable aspect of ACE behaviour it does make it more difficult to differentiate between valid and invalid transitions. Nevertheless, safeguarding mechanism, which validates new and suspicious transitions before they are fully registered with the execution model, could be implemented.

In general, neither of the described modelling techniques is ideal for all cases and their use will depend mainly on the requirements of the system under supervision. As already discussed in [D2.3], the same statement holds for the use of either the execution models. Thus, utilisation of either modelling techniques as well as their associated execution plan will depend on the requirements of the application itself. A prototype implementation is provided for in Section 3.7.

2.4 Clustering of supervised ACEs with supervisors ACEs

The clustering and reverse clustering approaches, described in the Deliverable [D3.1], aim at providing a mechanism to self-organize the configuration of a service, which is build up from a number of ACEs delivering basic services and functions. [D3.1] does not make any
Bringing Autonomic Services to Life

assumptions about the service logics but it uses an abstract notion of types or colors of nodes (here: ACEs or ACE ensembles) to define a collaborative cluster. In this section, we use a similar approach to describe the service ensemble under supervision.

The material presented in this section advances the considerations on decentralized supervision, described in [D2.3], by several aspects:

- we provide a concrete scheme to apply self-aggregation mechanisms developed in WP3 to organize and to link a network of supervisors to an ensemble of ACEs which is organized into clusters by the same mechanisms;
- we establish the connection to the model based approach in Section 2.4.3 by using the supervisor network to establish a cognitive backbone structure for supervision using the results established in [D21m].

Definitions: Hence, to define the notion of a cluster more precisely, let \( G = (V, E, C) \) be a graph with (finite) node set \( V \), symmetric edge relation \( E \subseteq V \times V \) (i.e. \( v \ E w \iff w \ E v \) for all \( v, w \in V \)), and a symmetric and reflexive match relation \( C \subseteq V \times V \) (i.e. \( v \ C v \ \& \ v \ C w \iff w \ C v \) for all \( v, w \in V \)). A sub-graph of a graph \( G = (V, E, C) \) is a graph \( G' = (V', E', C') \) with

\[
V' \subseteq V, \ E' = E \cap (V' \times V'), \text{ and } C' = C \cap (V' \times V') \text{ for all } v \in V'.
\]

As usual we write \( G' \subseteq G \) if \( G' \) is a sub-graph of \( G \). A graph \( G = (V, E, C) \) is called complete if \( E^* = V \times V \). A cluster of \( G \) is a complete sub-graph \( D = (V', E', C') \) of \( G \) such that \( v \ C' w \) for all \( v, w \in V' \) does hold.

Remark: The notion introduced in [D3.1] differ from our formalization by the fact that a coloring function \( c(v) \) is used to assign a color \( c(v) \) to each node \( v \in V \). The match relation is then given by \( v \ C w \iff c(v) = c(w) \). But since the clustering algorithm presented in [D3.1] make no use of the coloring of nodes but only it applies the matching rule given above, the more general notion given in this report does not invalidate the results in [D3.1]. However, it allows us to work with match relations which depend on the topology of the underlying graph more directly, as it does not require any encoding of topology information into a finite set of colors. There is however one possible pitfall: the results on clustering assume that nodes do not change their color during clustering. Therefore, we cannot use a match relation with depends on the actual clusters which are constructed using this relation. If we however make the match relation dependent on a fixed graph topology (e.g. the outcome of a preceding clustering step) we remain on the safe side.

A basic mapping of these graph theoretic notions into the terminology of ACE communication is to associate an ensemble of ACEs with each cluster, which contributes to one (or several) service contracts. Thus, a cluster comprises an ensemble of collaborating ACEs which provide a common service. The fact that two clusters of the same graph do not necessarily have disjoint node sets means that the same ACE may contribute to several services.

Figure 4 – Placement of supervisors and supervisor network aggregation
Bringing Autonomic Services to Life

The clustering algorithm, presented in [D3.1], now enables us to use a mechanism to rewire a given graph $G$ according to its match relation to obtain another graph $G'$ with the same node set and match relation which is clustered. We now discuss how this mechanism can be used to place supervisor nodes into a clustered graph in a way that a supervisor is exclusively associated with a specific cluster (there might be however more than one supervisor per cluster). In a second step, we use the aggregation mechanism again to build up a network of supervisors according to whether their supervised clusters overlap. Figure 4 illustrates the approach.

2.4.1 Placing Supervisors

Hence, let $G = (V, E, C)$ be a graph with clusters $D_1, D_2, \ldots, D_k \subseteq G$, with $D_i = (V_i, E_i, C_i)$. We aim on an application of the clustering algorithm which associates a supervisor node $s_i$ with each of these clusters. To this end let $S$ be a set of supervisors ($V \cap S = \emptyset$) of sufficient size (e.g. $|S| \geq k$). We define $G_s = (V_s, E_s, C_s)$ with $V_s = V \cup S$, $E_s \supseteq E$ such that $s \in S$ and $v, w \in V$, and

$$v C_s w \Leftrightarrow (v, w \in V \land v E w) \lor (v \in S \land (\exists u \in V)[u C w \land v E u]) \lor (w \in S \land (\exists u \in V)[u C v \land w E u])$$

The last condition defines a match between nodes $v$ and $w$ if there is already an edge between these nodes (we are not going to alter the original service topology), or $v = s_i$ is connected to some node $u$, and $u$ belongs to the same cluster than $w$ (and vice versa for $w = s_i$). Hence if we chose the edge relation $E_s$ in a way that in each cluster $D_i$ there is at least one node connected to a supervisor $s_i$, then the aggregation algorithm approximately connects each node in the cluster to $s_i$.

2.4.2 Aggregating Supervisors

Let $s_i$ be a supervisor of the cluster $D_i$ in the clustered graph $G_s$. We now modify the match relation of $G_s$ to

$$s C s' \Leftrightarrow (\exists v \in V)(\exists v' \in V)[s E_s v \land s' E_s v' \land v E v']$$

for all $s, s' \in S$. With this match relation, supervisors of neighboured clusters (those which contain nodes maintaining an edge relation and therefore a match among each other).

Since the supervisor network which results in the application of the aggregation algorithm to the set of supervisors is again a clustered graph (with the match relation corresponding to its edge relation, i.e. clusters are cliques in this graph), we can apply the placement and the aggregation steps again and obtain a supervision meta-structure on top of the supervision network.

We conclude that the aggregation mechanisms provided by WP3 equip us with a generic approach to align supervision networks according to the structure of the underlying service ensemble under supervision. We now discuss how to use this network to organize the based supervision model.

2.4.3 Application Example – Model Based Supervision

The collective maintenance of knowledge about the underlying service providing components enables to place supervisors and connect them to a supervisor network. The model based approach [D2.1m] aims on the construction of a cognitive backbone structure. This structure is built on demand and provides a suitable degree of granularity to perform
Bringing Autonomic Services to Life

perception and planning tasks such as the StatePredictor function described in Section 3.7 and the planning algorithm of Section 2.2.

We now give a very brief summary of the model framework for supervision presented in [D2.1m], to establish a simplified notation suitable for the current discussion; a complete description of the model is reported in [D2.1m, D2.3, Deu07].

A model is basically a (not necessarily finite) state machine. Models provide an abstracted view of collections of ACE execution plans (either running as parallel plans within one ace, or distributed over several ACEs). Operations for composing models have been discussed abstractly in [D2.1m] and in relation to the concrete formalism used to express ACE execution plans in [D2.3]. The model framework for supervision uses embedding morphisms to relate a model $M_C$ to a composition of models $M$ containing $M_C$ as component; for the sake of brevity let us denote such an embedding by $\xi: M_C \rightarrow M$. Thus if for instance the model $M = M_1 \parallel M_2 \parallel \ldots \parallel M_k$ is the result of the (e.g. synchronous) composition of state machines $M_1$, $M_2$, ..., $M_k$, then we have embeddings $\xi_i: M_i \rightarrow M$ for each of these components. It has been discussed in [D2.3] how asynchronous communication by means of input queues, as used by the ACE toolkit, can be modeled as embedding morphisms.

Another morphism used in [D2.1m] is the abstraction arrow. Abstraction means that we restrict a model to its observable contents. In particular that means that an abstract state corresponds to a set of concrete ones (which cannot be distinguished on the given level of abstraction), and abstract system actions (approximate: state machine transitions) have a set of concrete realizations. Some concrete actions however may not have an abstract counterpart. Abstraction arrows are denoted as $\alpha: M_C \rightarrow M_A$.

Then we have the following facts on what we have called zooms.

Any pair $(\xi, \alpha)$ of an embedding and an abstraction arrow can be extended in a unique way by another pair $(\xi', \alpha')$ of an embedding and an abstraction such that the diagram

![Diagram](image)

commutes. Basically, this result can be interpreted as follows: If we first use an abstract version of $M_3$, namely $M_1$, and embed this abstraction into a composite environment $M_2$, than we can replace $M_1$ by $M_3$ in $M_2$ in the most abstract way (i.e. without adding information which is not already present in $M_2$ and $M_3$), obtaining am model $M_4$ which differs from $M_2$ only in the degree of concreteness w.r.t. the abstraction arrow $\alpha$. Thus we can use the arrow to zoom into the model $M_2$, unfolding all the internal details of $M_3$.

Let us now investigate on how we can use the notion of zooms to collaboratively maintain a model structure by using a network of supervisors. Redrawing the situation pictured in Figure 4 from the perspective of one of the supervisors reveals the existence of two “areas” within the supervised ensemble: A controlled area that comprises the service nodes to which a supervisor is directly connected to, and a collaborative area that comprises the service nodes under control of another supervisor to which the supervisor node under consideration is connected to in the supervisor network. Figure 5 illustrates these areas for supervisor node $s$. 

---

Page 20 of 76
We now use a model $M_s$ to describe the controlled area of the supervisor $s$, which is basically the (asynchronous) composition of the execution plans of the ACEs in this area. The cooperative area is expressed by abstractions of the models for the controlled area of the supervisors in the collaborative area, i.e. related to the current example: abstractions $N_s'$ and $N_s''$ of the models $M_s'$ and $M_s''$ of the controlled areas of $s'$ and $s''$. A possible way to obtain those abstractions is to restrict $M_s'$ and $M_s''$ to those actions which attend in a communication with $M_s$ and to introduce an internal action $\tau$ to which all other actions are mapped – an operation with is well known as *hiding* in the literature on concurrency theory (see, e.g., [Mil80] for an early account). Another approach would be to abstractions defined by the ACECol filtering expressions as described in [D2.3]. Thus from the perspective of $s$ we end up with a supervision model $M$ and embeddings $\xi_1: M_s \rightarrow M$, $\xi_2: N_s' \rightarrow M$, and $\xi_3: N_s'' \rightarrow M$, as well as abstractions $\alpha_1: M_s' \rightarrow N_s'$, and $\alpha_2: M_s'' \rightarrow N_s''$.

We now have the appropriate tools to increase the level of control if a supervision action cannot be performed on an (ACE or contract) local level. To this end, let us consider again the example configuration described in the discussion of Figure 5 and suppose that for instance the computation of a contingency plan (Section 2.2) cannot be performed because of the lack of sufficient control of the ACEs in the collaborative area of $s$.

Now, a placement and aggregation step performed on the level of $s$, $s'$, and $s''$ could invoke a supervisor $s'$ which is connected to these three supervisors. A planning attempt of $s'$ will use the model $M$ with zoomed concretisations along the abstractions $\alpha_1: M_s' \rightarrow N_s'$, and $\alpha_2: M_s'' \rightarrow N_s''$.

### 3 Prototype components

In the third supervision prototype we have made substantial improvements to the ACE-based implementation of supervision key components. The changes we introduced help to clarify the run-time behaviour of each component, and simplify the chore of stringing them together for collaboration.

In the following, we will provide high-level descriptions of each component’s goals, as well as an indication of the rationale behind their implementation. We will then proceed to give concrete details on the functionalities they provide (interfaces), and on how these functionalities are used to reach the component’s goals. The latter information is given under the form of component plans expressed by using the ACELandic.

Section 3.7 reports the current state of the prototyping of the long term supervision components, and of their integration in the supervision prototype, planned at M33.
3.1 Supervised-supervisor interfaces

The deployment and function of Supervision Checker Objects has been already described in previous WP1 and WP2 deliverables, we thus give just a brief summary of the approach. ACEs base internally on a communication infrastructure comprising of (a) an internal communication bus, and (b) a gateway, which handles the inter-ACE communication. The bus is in particular responsible for disseminating events between the several ACE organs. Thus, observing and controlling the bus and the gateway provides sufficient information to understand and to influence all ongoing processes within an ACE. For this purpose, Bus Checker Objects (BCO) and Gateway Checker Objects (GCO) implement a method

```
Result check(Event e)
```

which is called as a call-back for each event e traveling over the associated infrastructure element. If the method returns a positive result, the event will be distributed to all its subscribers (all organs who implement an appropriate handle method for this event), otherwise it is dropped. Additionally, checkers can use a method

```
void insert(Event e)
```

which allows to insert additional events into the associated infrastructure elements.

Supervision Checker Objects communicate to the Sensor and Effector supervisor ACEs by using a simple request/reply protocol scheme. A previous implementation which made use of DIET mirror agents to implement these connections has been replaced by the DIET message channels.

In the current implementation, the check/insert pair of methods has been used to implement interrogative monitoring and advanced control, which are described as follows.

**Interrogative Monitoring**

Interrogation functions make use of the check/insert functions provided by the BCO (Bus Checker Object) by defining a number of events containing requests for information which are handled by the Executor and – in the case of getting information about the possible events a repository function can send (see belos) – by the Repository organ. These organs respond by issuing events containing the requested information. These responsive events are again intercepted by the BCO and delivered to a Sensor ACE associated with the supervised ACE under consideration. Requests are message of type BusSupervisionEvent which a type parameter indicating the requested information:

1. Currently executed plans
2. States in which those plans are
3. State of the input queues of each of the executed plans
4. Global session and execution session
5. Events which are possibly issued by a specific repository function (those requests are handled by the Repository organ utilizing a special interface provided by the Output Mapper of a user defined function).

**Advanced Control**

Model based supervision results in the computation of non-local coordinated contingency plans to define a collective course of actions which are intended to lead an ACE based configuration from a problem state back into a normal operation. Actions may be
Bringing Autonomic Services to Life

1. Passive: Prohibit a certain transition from being executed
2. Active: Enforce a certain transition by issuing a triggering event

Since transitions are enabled in specific plan states, a contingency plan local to a certain plan execution can be expressed as a pair of mappings \((p, a)\), where \(p: State \rightarrow Set<Transition>\) maps a plan state into the set of transitions allowed to be executed in this state, and \(a: State \rightarrow Set<Event>\) defines the set of events to be issued by a supervisor in order to trigger certain transitions. These events are derived from a dedicated base class (namely SupervisionControlEvent) which is also used by the planning algorithm to identify those events which are intended to be actively controlled by the supervisor.

Passive control (i.e. an interpretation of the function \(p\)) is realized by an extension of the plan executor implementation, which checks for the current state \(s\) whether a transition to be executed is in the set \(p(s)\). Active control (i.e. an interpretation of the function \(p\)) is done by the BCO which checks for the current state \(s\) whether the set \(a(s)\) is not empty, and if so, it introduces events for all elements of \(a(s)\) into the internal communication Bus, where they will travel to the plan executor instances and handled there.

3.2 Sensor

The Sensor component uses a “SupervisionDataFilter” component to perform the actual filtering. The component performs event selection based on a property expressed in ACECol, and exposes the following API:

- **public SupervisionDataFilter(String filteringProperty) throws RecognitionException**
  this method is a constructor method. It takes a property given in the ACECol and initiates the “SupervisionDataFilter” object with an appropriate parser. The parser is then used to create a tree structure representing the property and capable of evaluating this property over a received event.

- **public boolean filter(SupervisionData event) throws ACEColException**
  this method performs the actual filtering. It receives an event under the form of a “SupervisionData” object and returns a boolean truth-value. The value is true if the event satisfies the filtering property, meaning it should be kept, and false if it does not, meaning it should be discarded.

To make these functionalities available from the self-model of a Sensor ACE, we use the following “wrapper” functions:

- **public void compileFilter(String store, String property, String result)**
  This function compiles a filter property, and stores the resulting SupervisionDataFilter object in the execution session under the key store. If the compilation process is unsuccessful, the local session contains the value “no” at the key result, otherwise “yes”.

- **public void filter(String property, SupervisionData event,**
Bringing Autonomic Services to Life

    String result)
This function applies the filter stored under the key property in the execution session to an input event and returns the result in the execution session under the key result.

A self-model which makes use of these functions is shown below. It assumes a publish-subscriber based cooperation scheme: Supervision components which are interested in data filtered by a certain property subscribe an associated topic. We use "$pubTopic" to indicate the topic associated with a filter expression, i.e. components such as correlators which are interested in events fulfilling this property have to describe this topic. The following template assume moreover that the supervision checker objects which correspond to the executing Sensor ACE deliver their results using a simple notification (a SupervisionEvent with type field set to notification).

    call init[filteringProperty <- local/filteringProperty,  
              filterLocation <- local/filterLocation,  
              propertyCorrect <- isPropertyCorrect];  
    if local/isPropertyCorrect = yes {  
      forever {  
        receive SupervisionEvent [data -> sensed-data]  
        when message/type = notification;  
        call filter[filterLocation <- local/filterLocation,  
                    data <- local/sensed-data,  
                    result <- result];  
        if local/result = yes {  
          call publish[topic <- $pubTopic,  
                       data <- local/sensed-data];  
        }  
      }  
    }

As a second functionality, Sensors need to support interrogative monitoring (compare e.g. Section 3.5). To this end, a parallel plan is started after the Sensor accepts a supervision contract:

    plan interrogate {  
      forever {  
        choice {  
          alternative {  
            receive Request <= internal.supervision-contract  
            when type = get-plans;  
            disseminate BusSupervisionEvent[type <= get-plans];  
            receive BusSupervisionEvent[data -> plans]  
            when type = plans;  
            send Reply[type <= plans, data <- local/plans]  
            => internal.supervision-contract;  
          }  
          alternative {  
            receive Request <= internal.supervision-contract  
            when type = get-states;  
            disseminate BusSupervisionEvent[type <= get-states];  
            receive BusSupervisionEvent[data -> states]  
            when type = states;  
            send Reply[type <= states, data <- local/states]  
            => internal.supervision-contract;  
          }  
          alternative {  
            receive Request <= internal.supervision-contract  
            when type = get-contracts;  
            disseminate BusSupervisionEvent[type <= get-contracts];  
            receive BusSupervisionEvent[data -> contracts]  
            when type = contracts;  
          }  
        }  
      }  
    }
Bringing Autonomic Services to Life

```java
send Reply[type <- contracts, data <- local/contracts] => internal.supervision-contract;
}
alternative {
    receive Request <= internal.supervision-contract
    when type = get-sessions;
    disseminate BusSupervisionEvent[type <- get-sessions];
    receive BusSupervisionEvent[data -> sessions]
    when type = sessions;
    send Reply[type <- plans, data <- local/sessions] => internal.supervision-contract;
}
alternative {
    receive Request <= internal.supervision-contract
    when type = get-queues;
    disseminate BusSupervisionEvent[type <- get-queues];
    receive BusSupervisionEvent[data -> queues]
    when type = queues;
    send Reply[type <- queues, data <- local/queues] => internal.supervision-contract;
}
```

The class GenericSupervisionChecker which is proposed as base class for all supervision checker objects has been extended to support this interrogation protocol. An implementation which overwrite the method

```java
public void receiveFromSupervisor(SupervisionEvent event)
```

therefore has to call` super.receiveFromSupervisor(event)` when the event is not handled in the new code.

### 3.3 Correlator

The Correlator component implements the CorAL language described in Section 2.1. It receives “SupervisionData” objects from appropriately configured sources, and performs correlation. The result is a table structure containing multiple tuples, where the cardinality of each tuple is the number of sources being considered. Indeed, the goal is to take one object from each source and to create tuples. The decision to place objects together in a tuple is made depending on a correlation property given in CorAL.

The Correlator component exposes the following methods:

```java
public void init(
    String bufferLocation,
    String CorALProperty,
    String correlationManagerLocation,
    String samplingPeriodLocation,
    String timerEventLocation,
    String startTimestampLocation)
```

This method performs initialization. The parameters that are passed to this method are: 1) the key in the ACE’s global map under which to find the buffer used for collecting the received events, 2) the CorAL property that will guide the correlation, 3) the key in the ACE’s global map under which to find the CorrelationManager (the actual Java object that implements CorAL and performs the correlation), 4) the key in the ACE’s local map under which to find the sampling period (which is read from the CorAL property), 5) the key in the
Bringing Autonomic Services to Life

ACE’s local map under which to find the timer that is used for sampling, and 6) the key in the ACE’s local map under which to find the timestamp at which the latest sample started.

```java
class Correlator {  
  public void addFilteredData(  
      SupervisionNotification filteredData,  
      String bufferLocation)  
  {  
    // Add filtered data to buffer  
  }  
}  
```

This method is called to add a newly received SupervisionData to a buffer. The contents of this buffer will be then filtered and used for correlation as described in the CorAL specification.

```java
class Correlator {  
  public void createSample(  
      String bufferLocation,  
      String startTimestampLocation)  
  {  
    // Create a sample from the buffer  
  }  
}  
```

This method is called to create a sample from the contents of the buffer. Since the buffer is continuously filled with new SupervisionData, and since each sample is defined as the set of data received within a given time-frame (see CoAL), this method needs to know both where to find the buffer, and where to place the time-stamp of the moment in which the sampling is started.

```java
class Correlator {  
  public void retrieveSample(  
      String bufferLocation,  
      String correlationManagerLocation,  
      String correlationPossible)  
  {  
    // Retrieve a previously created sample  
  }  
}  
```

The method is called to retrieve a previously created sample. In this case the parameters indicate 1) the location of the buffer being used to collect the received data, 2) where to find the correlation manager, and 3) where to state if sampling has been completed and if the actual correlation can be started.

```java
class Correlator {  
  public void correlate(  
      String correlationManagerLocation,  
      String result,  
      String dataAvailable)  
  {  
    // Perform actual correlation  
  }  
}  
```

This method is called to perform the actual correlation between the sampled data. The parameters provide 1) the location of the correlation manager, 2) where to place the result, and 3) where to place an indication of the fact that correlation has completed.

In accordance with the two main activities performed by the Correlator component (sampling and correlation), the component executes two main ACELandic plans.

The first plan-fragment is responsible for performing sampling:

```java
forever {  
  choice {  
    alternative {  
      receive Notification[data -> filtered-data]  
        <= internal.supervision-contract  
        when message/topic = $subTopic;  
      call add-filtered-data[filtered-data <- local/filtered-data,  
                              bufferLocation <- global/bufferLocation];  
    }  
    alternative {  
      receive TimerExpired;  
      call create_sample[  
                              bufferLocation <- global/bufferLocation,  
                              startTimestampLocation <- local/startTimestampLocation];  
      disseminate SampleReady;  
    }  
  }  
}
```

Page 26 of 76
 Bringing Autonomic Services to Life

This plan states that the Correlator component continuously waits for either a new Notification, or its internal timer to expire. In the first case, the Correlator calls method "add-filtered-data". In the second case, it calls method "create_sample" and disseminates the notion that a new sample is ready for actual correlation. $subTopic indicates the topic the Correlator is waiting for.

The second plan-fragment is responsible for executing the actual correlation:

```java
forever {
    receive SampleReady;
    call retrieve_sample[
        bufferLocation <- global/bufferLocation,
        correlationManagerLocation <- global/correlationManagerLocation,
        correlationPossible <- local/correlationPossible];
    if local/correlationPossible = yes {
        call correlate[
            correlationManagerLocation <- global/correlationManagerLocation,
            result <- local/result,
            dataAvailable <- local/dataAvailable];
        if local/dataAvailable = yes {
            call publish[topic <- $pubTopic, data <- local/result];
        }
    }
}
```

This plan continuously waits for new samples to be ready. When a new sample is available, the Correlator calls the method "retrieve_sample" to retrieve it. At that point it checks in the global map to see if the correlation is possible. If this is the case, it calls the method "correlate" and, as soon as the correlation results are ready, it publishes them by calling method "publish". $pubTopic indicates the topic on which the result are published. The complete Correlator plans are given in Appendix.

### 3.4 Assessor

The Assessor component implements the AL language. It receives a table of correlated data from the Correlator component, and has two modus operandi. The first is called "single assessment", in which the Assessor tries to assess a single property that predicates over the contents of the entire table. The result, therefore, is a single truth-value. The second is called "multiple assessment", in which the Assessor tries to assess a single property: one for each row in the correlated table. The result, therefore, is a set of truth-values, as well as a sub-table containing only those rows for which the property was false.

The Assessor component exposes the following API methods:

```java
public void init(
    String assessmentProperty,
    String assessorLocation,
    String propertyCorrect)
```

This method performs initialization. The parameters that are passed to this method are: the property expressed in AL that is supposed to be assessed, the key in the ACE’s local map under which to place the Assessment Manager component (the actual Java object that implements AL and performs the assessment), and the key in the ACE’s local map under which to place the result of the assessment.

```java
public void assess(
    String assessorLocation,
    SupervisionNotification correlatedData,
    String result,
    String tableLocation)
```
Bringing Autonomic Services to Life

This method activates the actual assessment. Its parameters are: 1) the location in which to find the AssessmentManager component; 2) the data on which to perform the assessment; 3) the result of the assessment; and 4) the location in which to place the sub-table with the rows for which the assessment was not true.

The Assessor component is guided by the following plan-fragment expressed in ACELandic:

```java
forever {
    receive Notification[data -> correlated-data] <= internal.supervision-contract
    when message/topic = $subTopic;
    call assess[
        assessmentManagerLocation <- local/assessmentManagerLocation,
        correlatedData <- local/correlated-data,
        result <- result,
        tableLocation <- local/tableLocation];
    if local/result = "no" {
        call publish[topic <- $pubTopic, data <- local/tableLocation];
    }
}
```

The plan states that every time a Notification is received, the Assessor must call the method "assess". If the result of the assessment (placed under the key "result") is false, the Assessor publishes the negative results calling the method "publish". $subTopic indicates the topic the component is waiting for, while $pubTopic indicates the topic on which the result are published. The complete Assessor plan is given in Appendix.

3.5 Planner

The Planner component exposes the following API methods:

```java
public void init(String plans, String states, String contracts)
```

This method initializes using the values stored in the execution session under the keys plans (set of plans executed in the ACEs under supervision), states (set of states in which these plans currently are), and contracts (set of currently active contracts). These values are obtained by an interrogative request from the sensor ACE (Section 3.2).

```java
public void plan(String c-plans, String outcome)
```

This method performs the actual planning activity and stores the resulting contingency plans (both the passive and active controller part) under the key c-plans in the execution session. The string outcome indicates the outcome of the planning activity (i.e. whether it had been possible to construct a contingency plan at all).

```java
public void project(String c-plans, ACEAddress target, String local-c-plans)
```

This method constructs a local projection of the contingency plan stored under the key c-plans in the execution session for an ACE with address target, and stores the result under the key local-c-plans.

The following self-model fragment should be used as a template for coordinating the planning activities:

```java
// receiving the ace address of a supervised ACE
```
Bringing Autonomic Services to Life

```plaintext
receive Notification[supervised -> supervised] <= internal.supervision-contract : controller when message/type = config;

forever {
  // getting a notification from assessor: something’s wrong
  receive Notification <= internal.supervision-contract : assessor;
  send Request[type <- get-plans] => internal.supervision-contract : sensor
  where message/type = plans;
  send Request[type <- get-states] => internal.supervision-contract : sensor
  where message/type = states;
  send Request[type <- get-contracts] => internal.supervision-contract : sensor
  where message/type = contracts;
  call init[
    plans <- local/plans,
    states <- local/states,
    contracts <- local/contracts];
  call plan[c-plans <- c-plans, outcome <- planning-possible];
  if local/planning-possible = yes {
    call project[
      c-plans <- local/c-plans,
      target <- supervised,
      local-c-plan <- local/c-plan];
    send Notification[type <- act, data <- local/local-c-plan] => internal.supervision-contract : effector;
  } else {
    send Notification[type <- no-planning-possible] => internal.supervision-contract : controller;
  }
}
```

To keep the template simple we have considered only one ACE under supervision, in practice. The planner has to deal with a number of those ACEs. The planner starts with receiving the address of the ACEs under supervision from the controller. Then it awaits a notification from the Assessor about a detected problem. It continues with requesting all the information, required by the planning, from the sensor associated with the ACE under supervision (the example assumes that the association supervised ↔ sensor ↔ effector is known). After performing the planning activities, it checks whether a contingency plan has been successfully constructed. If so it is sent to the effector. If not, a notification is sent to the controller.

3.6 Effector

The Effector provides only interface functions; thus, it does not involve any repository functions. It is controlled by the following self-model fragment:

```plaintext
forever {
  receive Notification[c-plan => data] <= internal.supervision-contract when message/type = act;
  disseminate BusSupervisionEvent{
    type <- deploy-controller,
    controller <- local/c-plan;
  }
}
Bringing Autonomic Services to Life

Note that the BusSupervisionEvents are handled by the supervision organ of the Effector and thus automatically sent to the associated ACE under supervision.

3.7 Long Term Supervision components

3.7.1 StatePreditor prototyping

Prototypes of the StatePreditor component, based on the two types of execution models specified in Section 2.3, have been implemented and partially tested.

A visualisation of the first prototype of the StatePreditor component, covering the meshed execution model, is shown in Figure 6, where the model itself has been created a priori. The left screenshot, (a), shows the model immediately after it has been created with no state changes registered. The thin lines between individual states represent valid transitions including cyclic references as for "state5". The right side, (b), shows a simulation where 1) state changes are registered with the execution model, 2) potential future states are identified and 3) their probability calculated. For each registered transition, the model shows the “from state” (blue), the “to state” (green) and all available “next states” (shades of red) ranked in the order of their likelihood to occur next. The order of next states is also marked numerically inside each node with 1 as the most likely, 2 as the second most likely candidate and so on. The frequency, indicating the number of times each transition has been traversed, is shown by the thickness of the lines connecting valid states.

![Figure 6 – Meshed execution model (Passive Modelling)](image)

Similar to the above, Figure 7 shows the meshed execution model. However, instead of using the passive modelling technique, the execution model is created actively at runtime. As shown by the six individual screenshots (organised from the top left to the bottom right) each transition does add relevant states as well as their relation to the execution model and simultaneously updates relevant occurrence values. Recommendations of potential next states are then based on the current model, which can never be assumed to be complete. Thus subsequent recommendation may differ depending on how the model develops over time. Furthermore, illegal transitions, if they occur, cannot be detected but instead are added to the model normally and treated as valid. Nevertheless, shown by the last screenshot (bottom right) and by assuming that only valid transitions are added to the model, the entire execution logic of any given ACE can be modelled, and potential candidates for future state changes can be identified.
Bringing Autonomic Services to Life

Figure 7 – Meshed execution model (Active Modelling)

The second prototype of the StatePreditor component, which is still under development reflects the directed execution model and is visualised in Figure 8. As shown, the general plan is organised from left to right with a collection of states available at each level of execution. Similar to the meshed execution model states are linked together by valid transition as indicated by the thin lines that connect each state. In order to predict potential next states the full execution path has to be provided, clearly identifying the current point of execution within the model. Based on the current point of execution, the probability of next states can be calculated in the same way as for the meshed execution model. Once known, the probability of next states can be calculated in the same way as for the meshed execution model. Clearly, this model is useful if the execution logic has to follow a strict sequence where only forward forks are allowed. However, this also implies that cyclic references are not allowed. Instead the execution path needs to be "reset" to a valid start state from which the execution can be monitored again. This limitation may prove to be difficult to implement, as it requires the ACE(s) that are under supervision to have a clear understanding about their own logic.
3.7.2 Current state of the integration in Supervision prototype

With respect to overall integration, both the DriftAnalyser and the EventPredictor components have been fully implemented and realised as ACEs. Relevant interfaces and communication mechanism have already been outlined in [D2.3]. However, current work focuses on a better integration with other supervision components to allow for more flexible component orchestration. In particular, the translation of current self-models of either component, to be expressed in ACELandic, is currently being investigated. Required modifications to either component are ongoing.

While the directed state predictor is currently under development, a prototype of the meshed state predictor is already available. A simulation engine has been developed to test this prototype by simulating the execution logic of ACEs as defined by their self-model. Both active as well as passive modelling have been implemented for this prototype. Subsequent work will concentrate on the finalisation of the directed state predictor as well as the integration of both components into the ACE toolkit. The latter will be achieved by wrapping each predictor by an ACE component that follows the interfaces as discussed in Section 3.1.

Both activities are expected to be finalised by Month 33 of the project.

4 Process for setting-up a Supervision System

This Section provides an example of the process to setup a supervision system for an ACE based system, by using the pervasive supervision approach and the supervision prototype. Since the supervision architecture currently provided supports a great variety of configurations, the process has to be adapted to the concrete application case.

Stage 1 – Define the Supervision Tasks and Architecture
Bringing Autonomic Services to Life

The first step comprises in identifying the supervision tasks to be performed, and the general set-up of the control loop architecture to be provided by the supervision system. This step comprises pretty much a software design phase and is somewhat out of the scope of this report (which does not cover general software design methodologies), but some guidelines can be defined.

- Identify the problem to be solved by the supervision system. For instance, in the case of fault management, which problems should be considered? Crashes of elements in the supervised system, mal-functioning ACE organs and repository code, interaction problems such as lifelocks and deadlocks, etc. A number of approaches including classification trees [Gro93, Ost88], boundary value analysis (an introduction can be found at [BVA]), etc. is available to perform these analyses.

- How can those problems be detected and what are appropriate reactions? What values and events need to be accessed and taken into account in order to identify a problem? How to correlate this information to get an appropriate picture of the overall system state? How to react on a deviation from the set of desired states and how to enforce this reaction?

Alleyne et al. conclude in [All03] that it is not only necessary to understand the function of the deployed control loops in order to enforce efficient control, but it is also necessary to understand and to deal with the additional complexity which results from the interaction of several control loops and the mutual influence they impose to each other. Thus it seems to be advisable to make sure that interactions between control loops (in particular implicit ones that results from low level dependencies within the ACEs under supervision) take place only if well-understood.

After the high level description of control loops has been done, a break-down of the supervision task to the level of supervision components has to be done. In order to give some guidelines on how to do this, we repeat the responsibilities of the several components:

- **Sensors** are responsible to gather data from the ACEs under supervision. These data base on observations done by supervision checker objects deployed in these ACEs, there is a one-to-one correspondence between supervised ACEs and sensor ACEs. Sensors are additionally responsible to filter and to classify gathered data and to distribute to other components (in general correlators) according to this classification.

  Sensors are also responsible to deploy supervision checker objects into the associated ACE under supervision, which boils down to setting up a separate contract between the sensor, the associated effector, and the supervised ACE, and to send a StartSupervisionEvent containing the checker objects to the supervised ACE using this triangular contract.

- **Correlators** are responsible to process data received from other supervision components (usually sensors and other correlators) and correlate them – those data are assumed to come from distributed sources. The correlated results are aggregated to new data and distributed to other components (usually other correlators and assessors).

- **Predictive Components** (DriftAnalyzer, State/EventPredictor) components are intended to perform long-term observation of an ACE or a class of ACEs and thus have to be invoked only if the ACE ensemble under supervision is likely to maintain a permanent interaction. A different way to of invocation is however to setup a dedicated supervision systems for long-term observation (then lacking planner,
Bringing Autonomic Services to Life

assessors, and effectors) and provide the obtained information to other (supervision) system as supplementary service.

- **Assessors** use correlated (or just filtered) data to judge whether the operation of the system under supervision in normal or the invocation of a reaction pattern is needed. Usually we assume that there is one assessor per supervision task.

- **Planners** have the purpose to compute a course of actions which will lead the supervised ACE ensemble back into a normal operational state. Since in general we consider a set of ACEs under supervision and not just a single one the planner has to break-down those contingency plans into action sequences performed on the level of local ACE execution plans before distributing them to the Effectors. We usually assume that there is one planner per supervision task.

- **Effectors** have a pure interface function. Their purpose is to deploy local contingency plans to their associated ACEs. There is a one-to-one correspondence between effectors and their associated ACEs.

- **Controller**. For initial setup and configuration purposes a controller ACE comes in handy. A controller ace reveals a goal for its associated supervision task and thus can be contracted by an ACE configuration which is to put under supervision. After contracting the controller, the initiating ACE sends configuration information to the controller, in particular information about the other ACEs in the configuration to be supervised, contracts, etc. The controller then uses this information to discover and to contract the ACEs with will form the supervision system. After doing so, it awaits the end of the contract with the supervised ACEs and disseminates this information to the rest of the ACEs in the supervision system.

Stage 2 – Continuous and Interrogative Interactions

Continuous interactions are those which based on a reactive reception of incoming events by the supervision system components, initiated by a request/reply based sampling of data by the Sensors. Observations of an ACE ensemble under supervision can be done using this interaction pattern. A publish/subscribe interaction style is appropriate as main interaction mechanism:

- Sensors will publish the gathered data
- Correlators, drift analysers, and state predictors will subscribe on data from analytical supervision components and publish processed data
- The assessor and the state predictor will subscribe data from analytical supervision components, but send (unsubscribed) notifications to the planner (if involved) or directly to the effector.

Thus the continuous coupling requires the definition of an appropriate set of subscription topics to coordinate the message flow. If ACECol is used to filter, correlate, and to assess, topics should be associated with filter and correlation expressions.

In opposite to that, the interrogative interaction style bases on a request/reply message exchange to obtain specific information which will generally not distributed in a continuous way. An example is the request for getting the currently executed plans by the planner as described in Section 2.2. Interrogative interactions should be applied only in exceptional cases (detected problems, inconsistencies, etc.) as they interrupt the normal MAPE-like processing chain of the supervision ensemble.

Stage 3 – Define the Supervision Logic
Bringing Autonomic Services to Life

After performing the necessary design steps 1) and 2), it is now time to design self-models and command items to set-up and to coordinate the interaction of the several supervision components. Self-models of those ACEs should comprise the following plan structure (see Appendix for self-model templates):

A setup plan should be used to handle the initial GN/GA exchange, receive configuration information, and to start operational plans. Additionally, if the component under consideration subscribes for a topic provided by another supervision ACE, the associated subscription requests has to be sent. Finally, the setup plan should remain active to receive and to distribute cancellation events which indicate to terminate the supervision activities.

The operational plan should follow a three step approach, namely (1) receiving data, (2) processing them, and (3) distributing the results to other ACEs according to the interaction patterns defined in Stage 2. To support a publish/subscribe scheme, a parallel plan to handle incoming subscription requests have to be provided. Targets of interrogative interactions (i.e. in particular sensors) should implement another parallel plan to handle incoming requests.

After self-models have been defined, the final step aims on providing a system of messages to put the interaction between the supervision ACEs into life. In [D2.3] we already described an extensive protocol message and data system. If ACELandic is used as self-model specification language, the “declare [ ... ]” construct comes in handy when fixing the Java based type system for message in a last step.

Stage 4 – Implement Supervision Checker Objects

The final task is to implement the Supervision Checker Objects. Basically, two methods are to be implemented by the supervision system designer:

```java
public void receiveFromSupervisor(SupervisionEvent event)
```

this method is called if an event is incoming from the sensor or effector, thus it should contain the command interpretation logic. It comes in handy to have a method ready to send a reply to the supervisor, i.e.

```java
private void sendSupervisionEvent(
    Connection connection,
    String type,
    SupervisionData data)
```

The implementation of the Check Object Interface GenericSupervisionChecker already contains two Connections, namely one for the sensor and one for the effector, hence it is advised to use it as a base class for supervision checker objects.

```java
public Result check(Event event)
```

This method is called as a call-back for each message travelling over the associated infrastructure element. Result is an enumeration type comprising the elements DENY and ALLOW. If DENY is returned, then the event is not allowed to travel over the infrastructure element.

Additionally, a call-back function is available so signal to the implementation code of the checker object that the deployment process is completed (this includes establishment of the connections to the sensor and the effector). This method is

```java
public void deployed()
```

and should be used for initialisation tasks. For instance, the heartbeat generator thread used in the WP2 standalone demo and the supervision part of the project wide demo is started in the deployed() method of the GCO.
5 Co-operation activities with other WPs

This section presents the activities that WP2 is performing in cooperation with other Workpackages and some preliminary results.

The cooperation activities are represented in Figure 9.

Some activities investigate how WP2 solutions could be applied in other contexts of CASCADAS:

- WP2 is investigating with WP4 on how to use of the supervision infrastructure to monitor the behaviour of ACEs for the definition of the reputation information; the specific use case is the auction model present in the pervasive advertisement scenario proposed in WP6 (see Section 5.3.1);
- WP2 is analysing with WP5 the use of supervision solutions for the validation of liveliness of knowledge atoms (e.g., periodic “test queries”, deregistration if no answer is received in time) (see Section 5.4);

Other cooperation activities aim at integrating and improving the solutions developed in other WPs in the framework of the supervision mechanisms:

- WP2 is investigating, with the support of WP3, how using the self-organisation algorithms to place supervisors and to build up a supervision pervasion (see Section 5.2.2), and to set-up the overlay for exchange of supervision information in the ACE toolkit embedded supervision (see Section 5.2.3);
Bringing Autonomic Services to Life

- WP2 is investigating, with the support of the WP1, how applying self-* features of ACE toolkit for implementing local supervision logic for the ACE toolkit embedded supervision (see Section 5.2.1);
- WP2 is studying with WP4 how introducing reputation information in the events sent by supervised ACEs to their supervisors (see Section 5.3.2);

Finally, other activities have the objectives to enforce the integration of supervision solutions in the CASCADAS toolkit and the demonstration scenarios:

- WP2 is jointly working with WP1 for the compliance and the integration with the ACE toolkit of the supervision prototype (see Section 5.1);
- WP2 is jointly working with WP6 for the extension of the Auction scenario to demonstrate the capabilities of the supervision prototype (see Section 5.5).

5.1 Supervision Prototype as an extension of ACE toolkit

In order to improve the integration with the toolkit developed in WP1, WP2, jointly with WP1, improved the interface between the tool kit and the supersion functions by providing basically the common development of the Supervision Organ within the ACE architecture. In the latest release of the toolkit, this organ provides interrogative monitoring and advanced control of the execution of ACE plans which has been described in detail in Section 3.1.

In addition to that, the meaning of the desirability value associated with the states present in the ACE self-model has been justified and mapped to failure modes as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Operational Mode</th>
<th>Description</th>
<th>Reaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Normal</td>
<td>Normal operation, no problem has been detected</td>
<td>None</td>
</tr>
<tr>
<td>2</td>
<td>Minor failure</td>
<td>Decrease of performance or service quality, but service execution is granted</td>
<td>ACE internal repair actions</td>
</tr>
<tr>
<td>3</td>
<td>Marginal failure</td>
<td>Impact to service execution, problem is assumed to be resolvable by internal management activities</td>
<td>ACE internal repair actions</td>
</tr>
<tr>
<td>4</td>
<td>Critical failure</td>
<td>Impact to service execution, problem is not resolvable without appropriate coordinated activities other ACEs (global fault)</td>
<td>External supervisor assumes control (execution plan level)</td>
</tr>
<tr>
<td>5</td>
<td>Catastrophic</td>
<td>System wide problem</td>
<td>External supervisor assumes control on ACE infrastructure level</td>
</tr>
</tbody>
</table>

Failure modes 2 and 3 are intended to be handled by supervision activities on self-model level, which critical failures are intended to trigger the model based planning approach (Section 2.2). Thus in the above example the statement in the critical { } block define the actions which are available to the supervisor (using passive and active control) to resolve the problem. No generic solution for catastrophic failures has been discussed yet.

A dedicated language construct which allows defining failure modes has been introduced in ACELandic.
5.2 Self-* features of Supervision

This section analyses current activities on the usage of Self-* features developed in WP1 and WP3 to implement the supervision mechanisms.

In particular, Section 5.2.1 provides initial results on the exploitation of self-adaptation in the ACE toolkit-based supervision, and its implementation through self-models. Section 5.2.2 analyses the application of self-organisation algorithms in order to create and maintain the aggregations of ACEs used for exchanging information among ACEs in the ACE toolkit-based supervision. Finally, Section 5.2.3 summarizes the application of clustering algorithms for supervisors/supervised ACEs association, already described in Section 2.4.

5.2.1 ACE toolkit embedded supervision and Self-adaptation

The main goal of the ACE tool-kit embedded supervision is to analyze how distributed supervision can be realized by applying both self-organization techniques developed in WP3 and exploiting the ACE model developed in WP1 at the maximum extend. We consider the ACE tool-kit embedded supervision applied at two different levels of granularity as a kind of Bottom-up process: ACE will be supervised per se by using the self differentiation algorithm [D3.2] and in case of unsuccessful effort the problem could scale to the ACE “neighbors” through the rearrangement of the links; in fact, we could consider the ACE in its role of service provider to be member of (at list) one group of ACEs providing/using the same service. This point is discussed in Section 5.2.2.

In this section we analyze the ACE level supervision which basically involves all the organs of the ACE where each organ fulfils a certain purpose in order to keep the entity in a working state. Nevertheless the same mechanism we are going to describe for supervising the organs of an ACE could be used to supervise any “custom” ACEs he/she implements. Any events of failure recorded should enforce the ACE to (Self) adaptation applying self-differentiation mechanisms.

According to the WP3 definition [D3.2] the differentiation phenomena are that the system (or the individuals) can exist in multiple states among which individuals (or the whole system) can switch. As the ACE behavior both internal (i.e., life cycle) or custom (i.e., the service provided) is represented as a series of states then it’s realistic to think the change of state for an ACE in terms of “(self-) differentiating”. The idea is to manage this change by modeling alternative self-models [D1.3] to mould the ACE behavior and to apply them anytime the functional reliability level of an ACE is at risk. ACEs should be able to modify their behavior by taking a decision locally, according to some “supervision” logic processing their internal state. As a matter of fact what we envision is a kind of inner closed control loop in the ACE as a feature available by the toolkit.

The ACE self-model will be used to instruct the Facilitator to modify the ACE behavior in relation to the context information collected during the supervision as well as the Events and Actions which might be monitored during the ACE runtime, be analyzed according to the rules defined to supervise “logically” the ACE and to modify the “normal” activity of the ACE.

This set of rules is translated in a kind of “supervision plan” that runs in parallel with the normal plans for the activities that the ACE is suppose to carry on. For this reason the supervision feature could be released as a part of the ACE toolkit so that the developer has the opportunity to activate it or to extend the basic supervision solution embedded in the toolkit when necessary.
Bringing Autonomic Services to Life

In the following we describe how this inner supervision should work with respect to the internal event processing mechanism. From a theoretical point of view we:

- identify probes to monitor the ACE behavior;
- organize and evaluate the data collected from the field;
- plan and put in place the necessary countermeasures.

From a Toolkit point of view we could implement supervision as a threat devoted for supervising the ACE as a part of the common functionalities that basically:

- collect data from the organ(s);
- analyzing it;
- in case a supervision event could be risen trigger, the Facilitator to manage the supervision event.

The basic supervision plan, which the ACE has to deal with, is related to its own life cycle. The ACE toolkit gives local mechanisms to manage the ACE life cycle in a way completely transparent to the ACE developer. However, all the life cycle actions are supposed to succeed if any organs start and work correctly. For instance this is what happens if the interaction between the organs and the Manager that has in charge to manage the organs communication, cannot take place. In this case to enforce the ACE to exhibit a real autonomic character we have to force it to trigger a corrective action, or better to force a change of its state. We define a self-model for the lifecycle state machine for the ACE, defining the rules required to trigger the corrective measures when necessary. This makes sense only if it is possible to supersede the Manager behavior to force the state machine to a stable and correct life cycle state.

Organs are free to choose the type of event that they would like to be notified of; the only constraint is that the event type needs to be derived from cascadas.ace.event.Event. For this reason an event cascadas.ace.event.AdaptationEvent will be created between the Manager and the Facilitator as the first is in charge of the ACE life cycle while the latter is responsible for the autonomous change of the ACE behavior. Any Organ must be equipped with a technical means (e.g., an interface) in order to be supervised and helped in case of resource leakage, i.e., it has a problem with requests frequencies and time of service. The Manager asks periodically any organ about its state and notifies this information to the Facilitator, which performs its analysis and decides whether to react or not on it according to the rules coded in the self-model. Therefore, a parallel supervision self-model should be run in order to instruct the Facilitator about the plan to be executed.

In the following an example of Self Model is presented with some generic rules to force the change state of the organ:

```
<selfModel>
  <plan id="Plan1" default="true">
    <description>Self Model for reconfigure the organs</description>
    <states>
      <!-- All possible states of the Plan1 are defined here -->
      <state id="state1">
        <friendly_name>Inactive</friendly_name>
        <desirability_level/>
      </state>
      <state id="state2">
        <friendly_name>Start</friendly_name>
        <desirability_level>10</desirability_level/>
      </state>
      <state id="state3">
        <friendly_name>Running</friendly_name>
        <desirability_level>10</desirability_level/>
      </state>
    </states>
  </plan>
</selfModel>
```
This rule says that if a AdaptationEvent arrives, and the info carried on “?event://state,time,organ_type” is exactly the same saved in the global session somewhere before “?globalSession://mystate,mytime,myorgan_type” then probably the ACE is not able to refresh this information and the createTransition action is called. Then the two transitions tr1 (i.e., stop) and tr2 (i.e., start) for the organ type are executed. An issue under evaluation with WP1 is if this “warm” shutdown/restart for the organs is possible without compromising the ACE performance.

Further analysis and simulation of this approach will be carried on during the last months of the project.
5.2.2 ACE toolkit embedded supervision and self-organization algorithms

The second approach we want to analyze for the ACE embedded tool-kit supervision is the level of interaction between ACEs providing a complex service.

The ACE is not an island but, as its role of “service provider”, it takes part to a kind of ACE community that fulfills a certain goal.

The Plan modification can be triggered by either changes to context data or events/actions coming from other ACEs.

In this context the “failed” ACE could be helped through the interaction of the ACEs belonging to the same “clique” (i.e., offering the same GA or alternatively asking for the same GN). This could be done by eventually forcing to rewire the links among them to find a substitute to the “wrong” ACE. In this situation the appliance of the WP3 algorithm is challenged to overcome the local dimension of supervising the ACE behavior as explained in the previous section.

For sake’s clarity a simple example could be derived if we consider the supervision from FCAPS, as defined in [D2.3], perspective defining some performance parameters.

We could calculate the min-max threshold values (i.e., MaxT, MinT), according to the performance parameters chosen, and raise a supervision event every time there is a discrepancy between values collected at run time and the need ones.

From a pragmatic point of view, we want to monitor and eventually modify the ACE behavior in relation to the rules defined in the self-model for every range of values of the performance parameters. When the ACE reaches the defined “limit” value, it should force to put in place the modification planned by looking for another ACE able to take some load or to supply a service in case of a received contract cancel request.

From a general point of view we want to use the “clustering” functionality provided by the ACE toolkit to settle a neighbors’ list for each ACE and to apply the self-organization principles [D3.1] to enforce a change in the ACEs’ links.

Two main phases are envisioned in our supervision parallel plan:

- the first one is devoted to instruct the Gateway to set up the overlay between the “contracted ACEs” in order to define and maintain a neighbors’ list for every ACE, e.g.: a list of ACEs offering the same service (contract provider); a list of ACEs asking for the same service (contract user);
- the second one is to establish the criteria to modify the ACE links, i.e., to find the “substitute” ACE through a rearrangement of the ACE overlay. This reorganization is done according to the rewired rules of the “on-demand” clustering algorithm described in the document [D3.1].

For the first point we use the “clustering_service” functionality provided by the ACE toolkit [D3.3] that will work transparently to create a list of neighbors that can be used later to establish and rearrange GA/GN contracts.

We suppose to have a Clustering contract for each ACE, in which:

- the initial neighbor list has been initialized by the invocation of the Clustering functionality;
- the reference to the neighbor list has been sent to the Gateway which: 1) recognizes and manages internal/external events related to the clustering algorithms; 2) manages the neighbor list and the clustering contracts;
Bringing Autonomic Services to Life

- the cluster is rewired when needed.

Consider an overlay of ACEs (Figure 10): the cluster of ACEs labeled “A” which use a service provided by the cluster of ACEs named B. We suppose that the "B" ACEs enforce load-balancing and status information by communicating among each other. Moreover, these ACEs are supposed to be altruistic in the sense that they volunteer in taking tasks (i.e., contracts) to unburden those ACEs, which are in a status of high utilization.

In this way we have to store a maximum number of contracts as "?globalSession://MaxActive parameter and compared this value with ?executionSession://nrActiveContracts obtained at runtime with a specific functionality e.g., ‘CountContract’. Then any CreateContract event received triggers the facilitator to evaluate the plan modification rules, where the “?executionSession://nrActiveContracts” value is compared with “?globalSession://MaxActive”; if first value is less than the latter value then B is able to take some load from the neighbors and advertise the other clustering contract members (e.g., sending a GA).

---

**Figure 10 – ACE aggregations for ACE toolkit embedded supervision**

In the following we describe the Self-Model for the “B” ACE used to rewire the cluster:

```xml
<selfModel>
  <plan id="Plan2" default="true">
    <description>Self Model for reconfigure the ACEs Overlay</description>
    <states>
      <!-- All possible states of the Plan1 are defined here -->
      <state id="state1">
        <friendly_name>Initial State</friendly_name>
        <desirability_level/>
      </state>
      <state id="state2">
        <friendly_name>Final State</friendly_name>
        <desirability_level>10</desirability_level/>
      </state>
    </states>

    <transition id="tr1">
      <source>s1</source>
      <destination>s2</destination>
      <priority/>
    </transition>
  </plan>
</selfModel>
```
Bringing Autonomic Services to Life

```xml
<trigger>@auto</trigger>
<guard_condition></guard_condition>
&action>
  event = cascadas.Ace.event.ClusteringPropagationEvent,
  role = MatchMaker,
  contract=?globalSession://my-clusteraggregation,
  aces-address)
</action>
</transition>

<Implies closure="universal">
  <And>
    <Atom closure="universal">
      <Rel>?event</Rel>
      <Ind>cascadas.ace.gateway.CancelContractEvent</Ind>
    </Atom>
    <Atom closure="universal">
      <Rel>?event://AceAddress</Rel>
      <Ind>?globalSession://myAceAddress</Ind>
    </Atom>
    <Atom closure="universal">
      <Rel>createTransition</Rel>
      <Ind>tr1</Ind>
    </Atom>
  </And>
  <Atom closure="universal">
    <Rel>createTransition</Rel>
    <Ind>tr1</Ind>
  </Atom>
</Implies>
```

This rule says that if a CancelContractEvent event arrives, and the contract canceled is related to the ACE whose address in "?event://AceAddress" is the same saved in the global session, somewhere before "?globalSession://myAceAddress", then invoke the rewiring of the cluster through the ClusteringPropagationEvent event described in the <action> tag that specifies the node’s requests to start a clustering algorithm.

Further analysis and simulation of this approach will be carried on during the last months of the project.

### 5.2.3 Clustering for Supervisors/Supervision association

An approach to utilize self-organisation mechanisms as developed in WP3 has been discussed in Section 2.4. It makes use of the idea that the system under supervision is itself a self-aggregated system structured into several clusters. The aim is to place supervisors in a way that each cluster gets supervised, departing from an initial topology which assigns a supervisor to at least one node in each cluster. In a second step, a supervisor network is constructed and allows disseminating information about ACEs of one cluster from a supervisor to another responsible to supervise an adjacent cluster.

We have discussed the model based supervision as an application example for this approach. The constructed supervisor network allows to perform observation and planning for multiple clusters, to increase the so-called controlled area of a supervisor and the level of granularity of the supervision models in use on demand.

---

1 In the CASCADAS approach, this initial topology can be constructed using a GN/GA based discovery of the supervisor
5.3 Security features of Supervision

Security and supervision are two important properties that need to be considered all together to tackle down possible attacks by malicious and faulty components so that self-preservation and self-healing can be guaranteed in the CASCADAS framework.

The integration goes beyond the use of cryptographic algorithms for the protection of the monitoring messages or authentication of the components that provide supervision capabilities. What we foresee consists of providing functionalities to avoid and limit a-priori the impact of attacks while, at the same time, of exploiting the monitoring capabilities of the self-supervision system to collect information on the behaviour of the ACEs. In these directions we follow two approaches which are complementary and described in the following sections.

5.3.1 Self-supervision for monitoring ACEs

The ACE tool-kit embedded supervision system presented in [D2.3] exploits the self-organization capabilities of ACEs to perform supervision in a distributed fashion at the level of single ACEs and interaction contracts. Each ACE collects information on its internal state as well as the one received by external entities so that faulty components can be detected and the failure state can be recovered, e.g. by aggregating with other ACEs that provide the same service of the faulty one.

Within this framework, we have described in [D4.1] and [D4.3], how reputation management systems can be used to drive the interactions between ACEs to guarantee self-preservation of the system. To function properly, the reputation system requires the collection of information about the past behavior of ACEs to detect the presence of selfish ACEs, which deny supplying services by not answering the GN messages or exploit resources without contributing to the system, and of malicious ACEs, which aim at attacking the system.

The self-supervision system can be served as a valuable tool to monitor the behavior of ACEs in transactions, or negotiations for aggregation, and to collect the necessary data, given by the loopback control mechanism implemented in every single ACE. With this information, the reputation system is able to predict the trustworthiness of ACEs, which can be seen as the expected quality of service that these ACEs can provide to the whole system.

5.3.2 Reputation to assist the supervision system

In a complementary setting, the information provided by the reputation management system can be exploited to filter the information received by single ACEs. The supervision model presented in [D2.3] consists of a pool of ACEs which are supervised, monitored and managed, in a federated fashion. The main role is assumed by a Macro ACE, named supervisor and formed by simple ACE components, which interprets the sensed data according to requirements like safety, functional correctness, consistency, performance, reliability, etc., and enforces corrective measures if a violation of these requirements is detected, or elaborated according to some planning.

In a nutshell, the supervision system to correctly detect violations relies on the sensed data provided by single components, which might be malicious by nature. Indeed, these ACEs can provide false data so that either a good component is excluded from the system, as it is recognized as a faulty one, or faulty components are not detected immediately.

The reputation management system can be used to filter out false data in such a way that
the prediction of the supervisor is not biased. For instance, to each ACE is associated a reputation value which is a global measure of the trustworthiness of the ACE and it is used by the supervisor to weigh the information received. This reputation value is updated after each report by the supervisor and distributed in the system by using the distributed supervision framework described in [D2.3]. The supervisor computes a credibility value, which is locally stored and continuously updated. This credibility value is based on the correctness of the information provided, which is measured by correlating the reports of different agents.

5.3.3 Application scenario

In this section, we present the foreseen integration of the reputation management system and the supervision system in the context of the pervasive advertisement scenario designed in WP6. We consider two types of roles for an ACE: seller and bidder. Each ACE can be part of both groups, but for simplicity of presentation we consider the two groups to be separated without losing generality.

Bidders and sellers are managed by two different reputation management systems that work in parallel without any specific cross point. When a seller announces for an item, bidders get information about the seller’s reputation value to compute the risk of a transaction and decide if to bid for the item. In a symmetric way, sellers get information on the ACE that has just placed the bid to decide whether they accept the offer based on the price, the reputation value of the bidder and the accepted risk level that the seller wants to take from selling the item. For instance, for valuable items, the seller wants to limit the risk at a minimum level, thus it will accept bids only from reputable nodes. While for low price items, it is much important to complete the transaction fast so that it can dedicate more resources to other functionalities.

5.4 Supervision and Knowledge Network framework

Simplified, a knowledge network, KN, is a generic structure that organises distributed knowledge of any format into a system that will allow it to be retrieved efficiently. The rationale of the knowledge network is to act as a middle layer that connects to a multitude of sources, organises them based on various concepts and finally provides well-structured, pre-organised knowledge to individual services and applications. The core requirement for such a vehicle to work is obviously the availability of data. Within the context of KNs, such data are provided by a concept called knowledge atoms, KA, which reflects a dedicated interface any ACE can incorporate to publish any sort of knowledge into the scope of the knowledge network. Supporting autonomic aspects this publication mechanism is realised as a self-registration mechanism that is embedded within each KA. Consequently, the KN has no control over individual KA’s and as such cannot influence if a KA becomes unavailable either voluntarily or unexpected. That is that a KA may decide to deregister itself from a KN or that it may encounter a problem which in the worst case could terminate a KA without proper deregistering itself from the KN. It is the latter case for which the supervision of KN components such as the collection of all KA’s is important in order to constantly validate the organisational structures of the KN and also to provide continuing QoS with respect to content and response time.

As depicted in Figure 11, any number of KAs can register with any number of network components within which they are further references to build up organisational structures. Please see [5.3] for more details on how knowledge networks operate. In order to validate the availability of such components their livelihood may be monitored at certain intervals. If determined to be unavailable, that is that a dedicated method capable of analysing the livelihood of a component returns false, then appropriate action may be taken including the
Bringing Autonomic Services to Life

deregistration of the component form all other network component, self-healing where possible or forcing the component to be restarted.

Figure 11 – Knowledge Network

Depending on the requirement of the knowledge source the supervision of liveliness can be performed in different ways, which can be summarised as follows:

- **Embedded**: By ACE toolkit embedded supervision, each network component that internally, via its own self-model, validates each component registered with it. If it is detected that a component has become unavailable, then the network component itself can execute relevant procedures to remove this component concerned with the network and could also notify other components that may take further action in an attempt to recover the component.

- **Partially Distributed**: For this a dedicated supervision ACE monitors the liveliness of all components within the KN or even within the ACE universe as a whole and takes the appropriate action if a component becomes unavailable. This solution however is very heavyweight on the supervision ACE and actually reflects almost a centralised solution to a distributed problem.

- **Fully Distributed**: A fully distributed solution means that each component has a supervisor component attached to it that specifically monitors the liveliness of the component it is attached to. In turn, other components may register with the supervision component to be notified if the component under supervision becomes unavailable. While this solution reflects the most robust solution, it also implies a certain overhead with respect to the additional components required and the contracts to be managed.

The embedded supervision mechanisms seem to be the most suitable considering that the overall goal is to validate the organisational structures of the KN. Consequently, only this mechanism will be further investigated. For this the atom interface has been extended providing a “isAlive” function that can be called by other components. By default this function will always return true indicating that if the function call was successful then the overall component is functioning correctly. This however may be modified for specific KA’s to include a more detailed self-testing procedure. In a secondary step, a dedicated plan will be developed, to be included within each KN component, capable of periodically validating each registered component. The functionality to remove a component from the scope of a KN component is already provided for and will be executed if a component is found to be no longer available. It is expected that this functionality is fully implemented and tested by Month 33.
5.5 Supervision in Cascadas demonstration scenario

To demonstrate the enhancement of an ACE based system by a supervision pervasion, we employ the following scenario in the context of the project wide demonstration application. Subject of the supervision activities is the lifelines of the connection between the incorporated ACEs – to simplify the situation we concentrate on the contract between the Seller ACE and the Auction Centre ACE in the distributed auction part of the demonstration.

Supervision is done by issuing an exchange of heartbeat signals between these two ACEs, hence, if the connection is malfunctioning in one or both directions, this fault can be detected by comparing the time points of sending and receiving a heartbeat signal (concerning the fact that sending and receiving of messages is done on different levels in the ACE architecture, the case that only one direction of the communication between two ACEs is lost is not an unrealistic scenario). Heartbeats are handled by supervision checker objects deployed into Seller and Auction Centre by the supervising ACEs, hence the liveliness validation mechanism is not visible on the level of the self-models (or repository functions) of the supervised ACEs. The deployment comprises the following steps:

1. The Seller ACE contracts a supervisor and disseminates the details of the contract under supervision (namely the ACE addresses of the Seller and the Auction Centre) to the supervisor.
2. The supervisor sets up a supervision pervasion comprising a Sensor and an Effector for both the Seller and the Auction Centre, two Correlators (each one responsible for the correlating sending and receiving of heartbeats in one of the two directions), and an Assessor.
3. The Sensor ACEs deploy a Gateway Supervision Checker object in the supervision organs of their associated ACEs under supervision. These checkers start threads to generate the heartbeat signals.
4. The Sensor ACEs start interrogating the ACEs under supervision by requesting and receiving information about the number of outgoing and incoming heartbeats within a given sampling period from the checker objects. This information is distributed to the Correlators.
5. The Correlators compute the difference between the time points of sending event and last receiving event of heartbeats by one of the ACEs under supervision, and send this information to the Assessor ACE. If this delay exceeds a certain threshold for one or both of the supervised directions, the Assessor invokes the respective Effectors to resolve the problem.
6. Error insertion is done in the same way as in the WP2 standalone demonstration by disabling the sending functionality of one of the supervised ACEs, namely the Seller ACE in the current case (a small GUI has been provided for this purpose).
7. The Effectors, if invoked, send a corrective action to their associated Gateway Checker Object which results in a reset of the gateway of the associated ACE. This simplified fault detection and reaction model can be extended easily to perform a full reset of the ACE under supervision by triggering its lifecycle management.

In this scenario, care must be taken to handle the potential disruptions arising from the Seller ACE entering a faulty state, which cannot be avoided and will inevitably affect the application level. However, the application is developed in such a way to limit the inconveniences to the actual auction affected by ACE disruptions, which will therefore not
Bringing Autonomic Services to Life

affect the following auction transactions. From the point of view of bidders and Auction Centre, affected by seller’s disruption, the fault causes one or more messages not being received. With respect of them the auction freezes at the state just before the faulty one. For example, considering a fault that takes place in the central phase of the auction, when bids are submitted and evaluated, the disruption will cause the seller’s inability to notify (to both bidders and Auction Centre) changes in the price of the item under auction. Bidders will therefore obtain, from the Auction Centre, stale information on the current auction and will keep submitting prices under the current one, with the consequence of effectively stopping the competition (i.e. the price growth). Moreover, if the fault is still present when the auction expires, the disruption will also affect the termination notification, with the consequence of bidders keeping submitting offers even after termination. Both cases are handled by the application in a simple an linear way, with the high dynamics nature of the application allowing the situation to advance to a correct state as fresh information becomes available (i.e. when the supervisor reacts to the detection of seller’s fault).

As an evolutionary step, we are looking at ways to exploit the information made available by the supervisor (i.e. the detection of an ACE entering a faulty state). This information results in fact very useful in the system, and its exploitation is expected to lead to (application-level) sophisticated reaction policies. As an example, the supervisor might decide to notify all other supervised ACEs of the faulty state of one or more specific ACEs, leaving operative supervised ACEs the freedom to react autonomously according to the peculiar needs of each of them. Reflecting this hypothetical scenario in the CASCADAS application, some bidders might use that information to give up the current auction and find another seller, while others might decide to suspend temporarily the submission of offers until another supervisor’s notification confirming the seller’s recovery. On the other hand, the Auction Centre(s) might use the supervisor’s notification to simply invalidate the auction (also waiting for another supervisor’s notification confirming the new seller’s operative state, just to notify the seller of the invalidation of its auction), thus enhancing coherence in the AWP, or simply “quarantine” the auction, by hiding the auction to bidders, in such a way to avoid unnecessary communication traffic.

6 Future Work

The main activities planned for completing the implementation of the prototype of the ACE based supervision and for its evaluation and demonstration are reported in the following list:

- completion of the integration of long term supervision components;
- definition of scenarios for testing and evaluating the features of the prototype;
- introduction of reverse clustering algorithms for associating supervised ACEs to the supervisors;
- refinement/enrichment and implementation of the scenarios aiming at demonstrating some of the supervision features in the Auction demonstration developed in WP6.

In addition, WP2 is planning to finalize the definition of the ACE toolkit embedded supervision and its specification for some specific scenarios; the approach will be evaluated through simulations.

Finally, cooperation activities with WP4 and WP5 concerning the application of supervision mechanisms to security and knowledge management contexts will be completed.
Bringing Autonomic Services to Life

7 Conclusions

This document is the textual part of Deliverable D2.4, which includes also a new version of the software implementing the supervision prototype.

The main purpose of this document was to describe the new features of the ACE based supervision and the new version of ACE-based components included in the 3rd version of the supervision prototype, developed according to the pervasive supervision model.

The main enhancements w.r.t. the previous version concern the introduction in some of the components of an interpreter of the language for configuring the supervision steps (e.g., filtering, correlation, aggregation, and assessment), the integration of the planner and effector components, the development and partial integration of Long Term Supervision components.

Moreover, the document describes the process to set-up a pervasive supervision system, by using the proposed supervision prototype.

Finally, in order to demonstrate the level of integration with the other WorkPackages, this document describes the activities that WP2 is developing in co-operation with them. Such cooperation activities allow on one hand to improve the adoption of solutions elaborated by other CASCADAS WPs in the supervision mechanisms, and on the other hand demonstrate the applicability of supervision solutions in other contexts. Finally, they guarantee a stronger integration with the ACE toolkit developed by WP1 and the demonstration scenario elaborated by WP6.
8 Appendix - User guide

This appendix explains, in detail, how to program/configure the supervision components of the prototype. Section 8.1 provides templates for configuring the plan of the ACEs implementing the supervision components. Section 8.2 reports the complete syntax of the supervision languages.

8.1 Plan templates for supervision components

The configuration of supervision components heavily depends on the application the user is trying to build. To this end we have prepared plan templates for the supervision ACEs. Users can adopt them as starting points when defining their system. Indeed, here we give the complete plan templates, and provide users with all the information they need to fill them out correctly.

The templates are given as ACELandic plans and refer to the APIs presented in Section 3.

**Sensor:**

```plaintext
declare [ 
SupervisionSubscription <- cascadas.supervision.interaction.protocol.SupervisionSubscription, 
SupervisionUnSubscription <- cascadas.supervision.interaction.protocol.SupervisionUnSubscription, 
Notification <- cascadas.supervision.interaction.protocol.SupervisionNotification ]; 

repository { 
    init [ACEColProperty: java.lang.String, 
    filterLocation: java.lang.String, 
    propertyCorrect: java.lang.String] -> 
    cascadas.supervision.collection.filter.FilteringService: init; 
    filter [filterLocation: java.lang.String, 
    data: cascadas.supervision.interaction.data.SupervisionData, 
    result: java.lang.String] -> 
    cascadas.supervision.collection.filter.FilteringService: 
    filter; } 

selfmodel sensor { 
    initial plan filter { 
    reveal sensor; 
    accept -> internal.supervision-contract; 
    local filteringProperty <- $filteringProperty, 
    local filterLocation <- $filterLocation, 
    run handle-subscription; 
    call init[filteringProperty <- local/filteringProperty, 
    filterLocation <- local/filterLocation, 
    propertyCorrect <- isPropertyCorrect]; 
    call subscribe[topic <- $subTopic1, 
```
Bringing Autonomic Services to Life

sub-contract <- internal.supervision-contract,
publisher <- $pubTopic1;

if local/isPropertyCorrect = yes {
    forever {
        receive Notification[data -> sensed-data] <=
        internal.supervision-contract
            when message/topic = $subTopic1;
        call filter[filterLocation <- local/filterLocation,
            data <- local/sensed-data,
            result <- result];
        if local/result = yes {
            call publish[topic <- $pubTopic,
                data <- local/sensed-data];
        }
    }
}

plan handle-subscription {
    forever {
        choice {
            alternative {
                receive SupervisionSubscription[message -> subscription]
                    when message/topic = $pubTopic do {
                        call process-subscription[subscription <- local/subscription];
                    }
            }
            alternative {
                receive SupervisionUnSubscription[message -> subscription]
                    when message/topic = $pubTopic do {
                        call process-unsubscription[
                            subscription <- local/subscription];
                    }
            }
        }
    }
}

- $filteringProperty is the ACECol property that will be used by the Sensor component;
- $filterLocation is the key under which the filter will be placed;
- Each couple [$subTopicX, $pubTopicX] indicates a subscription's topic, and who is publishing data for that subscription;
- $pubTopic is the name of the topic on which the Sensor will publish its results.
Correlator:

declare [
    Notification <- cascadas.supervision.interaction.protocol.SupervisionNotification,
    TimerExpired <- cascadas.ace.event.TimerExpiredEvent;
    SampleReady <- cascadas.supervision.correlation.SampleReadyEvent;
];

repository {
    init[bufferLocation : java.lang.String, CorALProperty : java.lang.String,
    correlationManagerLocation : java.lang.String,
    samplingPeriodLocation : java.lang.String,
    timerEventLocation : java.lang.String,
    startTimestampLocation : java.lang.String] |->
    cascadas.supervision.correlation.CorrelationService:init;

    add-filtered-data[filteredData
    : cascadas.supervision.interaction.protocol.SupervisionNotification,
    bufferLocation : java.lang.String] |->
    cascadas.supervision.correlation.CorrelationService:addFilteredData;

    create-sample[bufferLocation : java.lang.String,
    startTimestampLocation : java.lang.String] |->
    cascadas.supervision.correlation.CorrelationService:createSample;

    retrieve-sample[bufferLocation : java.lang.String,
    correlationManagerLocation : java.lang.String,
    correlationPossible : java.lang.String] |->
    cascadas.supervision.correlation.CorrelationService:retrieveSample;

    correlate[correlationManagerLocation : java.lang.String,
    result : java.lang.String,dataAvailable : java.lang.String] |->
    cascadas.supervision.correlation.CorrelationService:correlate;
}

selfmodel correlator {
    initial plan sample {
        reveal correlate;
        accept -> internal.supervision-contract;

        global bufferLocation <- $bufLocation;
        local property <- $corProperty;
        global correlationManagerLocation <- $corLocation;
        local samplingPeriodLocation <- $sampPeriod;
        local timerEventLocation <- $timerLocation;
        local startTimestampLocation <- $startTimestampLocation;

        run handle-subscription;
        run correlate;

        call init[bufferLocation <- global/bufferLocation,
        CorALProperty <- local/property,
        correlationManagerLocation <- global/correlationManagerLocation,
        samplingPeriodLocation <- local/samplingPeriodLocation,
        timerEventLocation <- local/timerEventLocation,
        startTimestampLocation <- local/startTimestampLocation];

        call periodic_event_scheduler_service[
            periodInMillis <- local/samplingPeriodLocation,
            event <- local/timerEventLocation];

        call subscribe[topic <- $subTopic1,
            contract <- internal.supervision-contract,
            publisher <- $pubTopic1];

        call subscribe[topic <- $subTopic2,
            contract <- internal.supervision-contract,
            publisher <- $pubTopic2];

        … //add any number of subscriptions

        forever {
            choice {
                receive Notification[data -> filtered-data] <=
Bringing Autonomic Services to Life

```plaintext
internal.supervision-contract
  when message/topic = $subTopic1;
  call add-filtered-data[filtered-data <- local/filtered-data,
    bufferLocation <- global/bufferLocation];

  } alternative {
    receive TimerExpired;
    call create_sample[bufferLocation <- global/bufferLocation,
      startTimestampLocation <- local/startTimestampLocation];
    disseminate SampleReady;
  }
}

plan handle-subscription {
  forever {
    choice {
      alternative {
        receive SupervisionSubscription[message -> subscription]
        when message/topic = $pubTopic do {
          call process-subscription[subscription <- local/subscription];
        }
      }
      alternative {
        receive SupervisionUmSubscription[message -> subscription]
        when message/topic = $pubTopic do {
          call process-unsubscription[subscription <- local/subscription];
        }
      }
    }
  }
}

plan correlate {
  forever {
    receive SampleReady;
    call retrieve_sample[bufferLocation <- global/bufferLocation,
      correlationManagerLocation <- global/correlationManagerLocation,
      correlationPossible <- correlationPossible];
    if local/correlationPossible = yes {
      call correlate[
        correlationManagerLocation <- global/correlationManagerLocation,
        result <- result,
        dataAvailable <- dataAvailable];
      if local/dataAvailable = yes {
        call publish[topic <- $pubTopic,
          data <- local/result];
      }
    }
  }
}
```

- `$bufLocation` is the key under which to find the location of the buffer being used by the Correlator component;
- `$corProperty` is the correlation property itself;
- `$corLocation` indicates the key under which to find the location of the actual correlation manager;
- `$sampPeriod` indicates the key under which to find the sampling period;
- `$timerLocation` is the key under which to find the timer;
- `$startTimestampLocation` is the key under which to find when the timestamp last started;

• $bufLocation$ is the key under which to find the location of the buffer being used by the Correlator component;
• $corProperty$ is the correlation property itself;
• $corLocation$ indicates the key under which to find the location of the actual correlation manager;
• $sampPeriod$ indicates the key under which to find the sampling period;
• $timerLocation$ is the key under which to find the timer;
• $startTimestampLocation$ is the key under which to find when the timestamp last started;
Bringing Autonomic Services to Life

- Each couple [$\text{subTopicX}, \text{pubTopicX}$] indicates a subscription’s topic, and who is publishing data for that subscription;

- $\text{pubTopic}$ indicates where the correlation results are published.

Assessor:

declare [Notification <- cascadas.supervision.interaction.protocol.SupervisionNotification, TimerExpired <- cascadas.ace.event.TimerExpiredEvent];

repository {
}

selfmodel assessor {
  initial plan assess {
    reveal assess;
    accept -> internal.supervision-contract;
    local property <- $assessProperty;
    local assessmentManagerLocation <- $assessManagerLocation;
    run handle-subscription;
    call init[property <- local/property, assessmentManagerLocation <- local/assessmentManagerLocation, propertyCorrect <- propertyCorrect];
    call subscribe[topic <- $subTopic1, contract <- internal.supervision-contract, publisher <- $pubTopic1];
    call subscribe[topic <- $subTopic2, contract <- internal.supervision-contract, publisher <- $pubTopic2];
    ... //add as many subscriptions as desired
    forever {
      receive Notification[data -> correlated-data] <= internal.supervision-contract
        when message/topic = $subTopic1;
      call assess[
        assessmentManagerLocation <- local/assessmentManagerLocation, correlatedData <- local/correlated-data, result <- result, tableLocation <- tableLocation];
      if local/result = "no" {
        call publish[topic <- $pubTopic, data <- local/tableLocation];
      }
    }
  }
}

plan handle-subscription {
  forever {
    choice {
      alternative {
        receive SupervisionSubscription[message -> subscription] when message/topic = $pubTopic do {
          call process-subscription[subscription <- local/subscription];
        }
      }
      alternative {
        receive SupervisionUnSubscription[message -> subscription] when message/topic = $pubTopic do {
          ...
        }
      }
    }
  }
}
Bringing Autonomic Services to Life

call process-unsubscription [subscription <- local/subscription]:

• $assessProperty is the assessment property we want to check;
• $assessManagerLocation is the key under which to find the actual assessment manager;
• Each couple [subTopicX, pubTopicX] indicates a subscription's topic, and who is publishing data for that subscription;
• $pubTopic indicates where the correlation results are published.

8.2 The Supervision Languages

This section provides the complete grammars for ACECol, CorAL, and AL languages. The grammars are written in Antlr [Par06]. Antlr (ANother Tool for Language Recognition) provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a target language (in our case Java).

8.2.1 ACECol

grammar ACECol;

@header {
package cascadas.supervision.collection.filter;
import cascadas.supervision.interaction.data.SupervisionData;
import cascadas.supervision.common.nodes.*;
import cascadas.supervision.common.exceptions.*;
import java.util.ArrayList;
}

@lexer::header {
package cascadas.supervision.collection.filter;
}

@members {
  private ArrayList<VarNode> varNodes = new ArrayList<VarNode>();

  public String getErrorMessage(RecognitionException e, String[] tokenNames) {
    String msg = super.getErrorMessage(e,tokenNames);
    if (msg == null) {
      if (e instanceof SemanticActionException)
        msg = e.getMessage();
      }
    return msg;
  }

  protected void mismatch(InputStream input, int ttype, BitSet follow) throws RecognitionException {
    MismatchedTokenException mte = new MismatchedTokenException(ttype, input);
    throw mte;
  }

  @rulecatch {
    catch (RecognitionException ex) {
      throw ex;
    }
  }
}
Bringing Autonomic Services to Life

filteringProperty returns [ACEColNode node]
:    "filter" '[' 'event' '=' className ':' formula[null] ']' 
    { 
        RootNode root = new RootNode($className.value,varNodes);
        $node = root;
        $node.addChild($formula.node);
    }

formula [ArrayList<String> idList] returns [ACEColNode node]
:    orFormula[idList] 
    { 
        $node = $orFormula.node;
    } | '(' quantifier Identifier 
    { 
        ArrayList<String> temp = new ArrayList<String>();
        if (idList != null)
            for (int i=0;i< idList.size();i++) {
                if (idList.get(i).equals($Identifier.text))
                    throw new DuplicateIdentifierException($Identifier.text + " already exists");
                else
                    temp.add(idList.get(i));
            }
        idList = temp;
        idList.add($Identifier.text);
    }
    "='e1=expr[idList] 'to' e2=expr[idList] '; p=formula[idList] ')
    { 
        $node = new QuantifierNode($quantifier.text,$Identifier.text);
        $node.addChild($e1.node);
        $node.addChild($e2.node);
        $node.addChild($p.node);
    }

quantifier
:    'forall' | 'exists'

orFormula [ArrayList<String> idList] returns [ACEColNode node]
scope { 
    ACEColNode firstAndFormula;
}
:    and=andFormula[idList]
    { 
        $orFormula::firstAndFormula = $and.node;
    } |'['] and=andFormula[idList] 
    { 
        if ($orFormula::firstAndFormula != null) 
            $node = new OrNode();
        $node.addChild($orFormula::firstAndFormula);
        $orFormula::firstAndFormula = null;
    } 
    $node.addChild($and.node);
} 
    { 
        if ($orFormula::firstAndFormula != null)
            $node = $orFormula::firstAndFormula;
    } 
}
Bringing Autonomic Services to Life

```java
andFormula [ArrayList<String> idList] returns [ACEColNode node]
scope {
    ACEColNode firstSubformula;
}
	sub=subFormula[idList]
    {
        $andFormula::firstSubformula = $sub.node;
    }
    ('&&' sub=subFormula[idList]
    {
        if ($andFormula::firstSubformula != null) {
            $node = new AndNode();
            $node.addChild($andFormula::firstSubformula);
            $andFormula::firstSubformula = null;
        }
        $node.addChild($sub.node);
    }
    )*
    {
        if ($andFormula::firstSubformula != null)
            $node = $andFormula::firstSubformula;
    }
}

subFormula [ArrayList<String> idList] returns [ACEColNode node]
: e1=expr[idList] (rel operator e2=expr[idList])?
    try {
        $node = new RelopNode($relop.text);
        $node.addChild($e1.node);
        $node.addChild($e2.node);
    } catch (NullPointerException e) {
        $node = $e1.node;
    }
    | '!' '(' orFormula[idList] ')' {
        $node = new NotNode();
        $node.addChild($orFormula.node);
    }
}

relop :
    '<' | '<=' | '==' | '>=' | '>

expr [ArrayList<String> idList] returns [ACEColNode node]
scope {
    ACEColNode root;
}
    '(' {
        $expr::root = new UnaryMinusNode();
    }
    | ? t=term[idList]
        if ($expr::root != null)
            $expr::root.addChild($t.node);
        else
            $expr::root = $t.node;
    | '+' t=term[idList]
        ACEColNode left = $expr::root;
```
Bringing Autonomic Services to Life

```java
$expr::root = new PlusNode();
$expr::root.addChildren(left);
$expr::root.addChildren($t.node);
}

<table>
<thead>
<tr>
<th>'-' t=term[idList]</th>
</tr>
</thead>
</table>
ACEColNode left = $expr::root;
$expr::root = new MinusNode();
$expr::root.addChildren(left);
$expr::root.addChildren($t.node);
|

| '*' t=term[idList] |
ACEColNode root;
{|             |
$term::root = $f.node;
|
| '*' f=factor[idList] |
ACEColNode left = $term::root;
$term::root = new MultNode();
$term::root.addChildren(left);
$term::root.addChildren($f.node);
|
| '/' f=factor[idList] |
ACEColNode left = $term::root;
$term::root = new DivNode();
$term::root.addChildren(left);
$term::root.addChildren($f.node);
|
| '%' f=factor[idList] |
ACEColNode left = $term::root;
$term::root = new ModNode();
$term::root.addChildren(left);
$term::root.addChildren($f.node);
|
| '*' |
|      |
$node = $term::root;
|

factor [ArrayList<String> idList] returns [ACEColNode node] scope {
    ACEColNode root;
    |             |
    var[idList] |
    |             |
    $node = $var.node;
    |
    constVal |
            |
    $node = new ConstValNode($constVal.value);
    |
    Identifier |
            |
    boolean idExists = false;
    if (idList != null) {
        for (int i=0; i<idList.size() && !idExists; i++) {
            if (idList.get(i).equals($Identifier.text)) {
                idExists = true;
                break;
            }
        }
    }
    if (!idExists) {
        throw new UnknownIdentifierException($Identifier.text + " is not defined");
    }
}
```

Page 58 of 76
$node = new IdentifierNode($Identifier.text);
}

| '(' orFormula[idList] ')' |
| $node = $orFormula.node;

constVal returns [Object value]
| StringConst |
| $value = $StringConst.text.substring(1,$StringConst.text.length() - 1); |
| NumConst |
| try { $value = Integer.parseInt($NumConst.text); } catch (NumberFormatException e) { try { $value = Long.parseLong($NumConst.text); } catch (NumberFormatException e1) { $value = Double.parseDouble($NumConst.text); }

var [ArrayList<String> idList] returns [ACEColNode node]
| '{' 'event' '}'
| $node = new VarNode(); |
| selector[idList]
| $node.addChild($selector.node);
| '}'
| varNodes.add((VarNode)$node);

className returns [String value]
| id=Identifier |
| $value = $id.text;
| '.' id=Identifier|
| $value = $value + '.' + $id.text;

selector [ArrayList<String> idList] returns [ACEColNode node]
| '.' Identifier |
| (arguments[idList])? |
| try { $node = new SelectorNode($Identifier.text,true,$arguments.list.size()); for (int i=0; i<$arguments.list.size(); i++) $node.addChild($arguments.list.get(i)); } catch (NullPointerException e) { $node = new SelectorNode($Identifier.text,false,0); }
| '{' expression[idList] '}'
| $node.addChild($expression.node);
Bringing Autonomic Services to Life

```java
arguments [ArrayList<String> idList] returns [ArrayList<ACEColNode> list]
{
    $list = new ArrayList<ACEColNode>();
}
'(' (expressionList[idList]
{
    $list = $expressionList.list;
}
)? ')
;

expressionList [ArrayList<String> idList] returns [ArrayList<ACEColNode> list]
:
e=expression[idList]
{
    $list = new ArrayList<ACEColNode>();
    $list.add($e.node);
}
',' e=expression[idList]
{
    $list.add($e.node);
}
)*
;

expression [ArrayList<String> idList] returns [ACEColNode node]
:
orFormula[idList]
{
    $node = $orFormula.node;
}
;

StringConst :
""(Letter|JavaIDDigit|' '|''|'	'|''|'
')"";

NumConst :
('0'..'9')+ ('.' ('0'..'9')*)?
;

Identifier :
Letter (Letter|JavaIDDigit)*
;

fragment
Letter :
    'u0024'
    | 'u0041'..'u005a'
    | 'u005f'
    | 'u0061'..'u007a'
    | 'u00c0'..'u00d6'
    | 'u00d8'..'u00f6'
    | 'u00fb'..'u00ff'
    | 'u0100'..'u1fff'
    | 'u3040'..'u31ff'
    | 'u3300'..'u337f'
    | 'u3400'..'u347d'
    | 'u4e00'..'u9fff'
    | 'uf900'..'ufaff'
;
```
8.2.2 CorAL

grammar CorAL;

@header {
package cascadas.supervision.correlation;

import cascadas.supervision.correlation.nodes.*;
import cascadas.supervision.common.nodes.*;
import cascadas.supervision.common.exceptions.*;
import java.util.ArrayList;
import java.util.Hashtable;
}

@lexer::header {
package cascadas.supervision.correlation;
}

@members {
private ArrayList<AliasNode> aliasNodes = new ArrayList<AliasNode>();
private ArrayList<String> sequenceNames = new ArrayList<String>();
private Hashtable<String, String> aliases = new Hashtable<String, String>();
private ArrayList<SequenceBuilder> builders = new ArrayList<SequenceBuilder>();
private double samplingPeriod;

public String getErrorMessage(RecognitionException e, String[] tokenNames) {
    String msg = super.getErrorMessage(e, tokenNames);
    if (msg == null) {
        if (e instanceof SemanticActionException)
            msg = e.getMessage();
    }
    return msg;
}

protected void mismatch(IntStream input, int type, BitSet follow) throws RecognitionException {
    MismatchedTokenException mte = new MismatchedTokenException(type, input);
    throw mte;
}

public ArrayList<SequenceBuilder> getBuilders() {
    return builders;
}

public Hashtable<String, String> getAliasMap() {
    return aliases;
}
Bringing Autonomic Services to Life

```java
public double getSamplingPeriod() {
    return samplingPeriod;
}

@rulecatch {
    catch (RecognitionException ex) {
        throw ex;
    }
}

correlationProperty returns [ACEColNode node] {
    defineSection correlateSection 'every' numConst 's' (whereSection {
        samplingPeriod = Double.parseDouble($numConst.text);
        $node = $whereSection.node;
    })?
;

defineSection :
    'define' '{' (sequenceDefinition {
        builders.add($sequenceDefinition.builder);
    })+ '}'
;

sequenceDefinition returns [SequenceBuilder builder] scope {
    SequencePropertyNode propertyRoot;
}

Identifier {
    if (sequenceNames.contains($Identifier.text))
        throw new CorALException("Duplicate sequence identifier: " + $Identifier.text);
    sequenceNames.add($Identifier.text);
} ':' 'sequence' 'of' className 'in' 'window' '(' numConst 's' ')' ('where' sequenceProperty {
    $sequenceDefinition::propertyRoot = $sequenceProperty.node;
})? ':

Number window = null;
try {
    window = Integer.parseInt($numConst.text);
} catch (NumberFormatException e) {
    window = Double.parseDouble($numConst.text);
}
builder = new SequenceBuilder($Identifier.text,$className.text,window,$sequenceDefinition::propertyRoot);
;

correlateSection :
    'correlate' '(' a=Alias 'in' id=Identifier {
        if (!sequenceNames.contains($id.text))
            throw new CorALException("Unknown sequence: " + $id.text);
        sequenceNames.remove($id.text);
        aliases.put($a.text,$id.text);
    })? ',' a=Alias 'in' id=Identifier {
        if (aliases.containsKey($a.text))
            throw new CorALException("Duplicate alias: " + $a.text);
        if (!sequenceNames.contains($id.text))
```
throw new CorALException("Illegal sequence: " + $id.text + "(can be used only once)");
sequenceNames.remove($id.text);
aliases.put($a.text,$id.text);
}
'*'}

whereSection returns [ACEColNode node]

'where' filteringProperty
{
    $node = $filteringProperty.node;
}

sequenceProperty returns [SequencePropertyNode node]

a=andIndex
{
    $node = new UnionNode();
    $node.addChild($a.node);
}
'(\'| a=andIndex
{
    $node.addChild($a.node);
}
)*'

andIndex returns [SequencePropertyNode node]

r=relopIndex
{
    $node = new IntersectionNode();
    $node.addChild($r.node);
}
'(\&\& r=relopIndex
{
    $node.addChild($r.node);
}
)*'

relopIndex returns [SequencePropertyNode node]

scope {
    String op;
}

i1=indexExpr ('<='
{
    $relopIndex::op = "<=";
} ['<
{
    $relopIndex::op = ";<";
} [=='
{
    $relopIndex::op = ";==";
} [>
{
    $relopIndex::op = ";>";
} [">=
{
    $relopIndex::op = ";>=";
}
} i2=indexExpr
{
    $node = new SPRelopNode($relopIndex::op);
    $node.addChild($i1.node);
Bringing Autonomic Services to Life

```java
$node.addChild($i2.node);

indexExpr returns [SequencePropertyNode node]
scope {
    SequencePropertyNode root;
}

`-`
{  
    $indexExpr::root = new SPUnaryMinusNode();
}

? t=indexTerm
{  
    if ($indexExpr::root != null)
        $indexExpr::root.addChild($t.node);
    else
        $indexExpr::root = $t.node;
}

`+` t=indexTerm
{  
    SequencePropertyNode left = $indexExpr::root;
    $indexExpr::root = new SPPlusNode();
    $indexExpr::root.addChild(left);
    $indexExpr::root.addChild($t.node);
}

`-` t=indexTerm
{  
    SequencePropertyNode left = $indexExpr::root;
    $indexExpr::root = new SPMinusNode();
    $indexExpr::root.addChild(left);
    $indexExpr::root.addChild($t.node);
}

`*` t=indexTerm
{
    $node = $indexExpr::root;
}

indexTerm returns [SequencePropertyNode node]
scope {
    SequencePropertyNode root;
}

f=indexFactor
{
    $indexTerm::root = $f.node;
}

`*` f=indexFactor
{
    SequencePropertyNode left = $indexTerm::root;
    $indexTerm::root = new SPMultNode();
    $indexTerm::root.addChild(left);
    $indexTerm::root.addChild($f.node);
}

`/` f=indexFactor
{
    SequencePropertyNode left = $indexTerm::root;
    $indexTerm::root = new SPDivNode();
    $indexTerm::root.addChild(left);
    $indexTerm::root.addChild($f.node);
}

`%` f=indexFactor
{
    SequencePropertyNode left = $indexTerm::root;
    $indexTerm::root = new SPModNode();
    $indexTerm::root.addChild(left);
    $indexTerm::root.addChild($f.node);
}

}`
```
Bringing Autonomic Services to Life

```java
$node = $indexTerm::root;
}

indexFactor returns [SequencePropertyNode node]
:
  | 'index'
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
  |
Bringing Autonomic Services to Life

orFormula [ArrayList<String> idList] returns [ACEColNode node] scope {
    ACEColNode firstAndFormula;
}
    and=andFormula[idList]
    {
        $orFormula::firstAndFormula = $and.node;
    }
    (')[' and=andFormula[idList]
    {
        if ($orFormula::firstAndFormula != null) {
            $node = new OrNode();
            $node.addChild($orFormula::firstAndFormula);
            $orFormula::firstAndFormula = null;
        }
        $node.addChild($and.node);
    }
};

andFormula [ArrayList<String> idList] returns [ACEColNode node] scope {
    ACEColNode firstSubformula;
}
    sub=subFormula[idList]
    {
        $andFormula::firstSubformula = $sub.node;
    }
    ('&&' sub=subFormula[idList]
    {
        if ($andFormula::firstSubformula != null) {
            $node = new AndNode();
            $node.addChild($andFormula::firstSubformula);
            $andFormula::firstSubformula = null;
        }
        $node.addChild($sub.node);
    }
};

subFormula [ArrayList<String> idList] returns [ACEColNode node]
    e1=expr[idList] (relop e2=expr[idList])?
    {
        try {
            $node = new RelopNode($relop.text);
            $node.addChild($e1.node);
            $node.addChild($e2.node);
        } catch (NullPointerException e) {
            $node = $e1.node;
        }
    } | '!' '(' orFormula[idList] ')'
    {
        $node = new NotNode();
        $node.addChild($orFormula.node);
    }
};

relop :
Bringing Autonomic Services to Life

expr [ArrayList<String> idList] returns [ACEColNode node]
scope {
    ACEColNode root;
}

    |<
    |<=
    |==
    |>=
    |>
    ;

    expr [ArrayList<String> idList] returns [ACEColNode node]
scope {
    ACEColNode root;
}

    |
    |
    

    expr [ArrayList<String> idList] returns [ACEColNode node]
scope {
    ACEColNode root;
}

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |

    f=factor[idList]
    |
Bringing Autonomic Services to Life

```java
 factor [ArrayList<String> idList] returns [ACEColNode node]
  var[idList]
  { $node = $var.node; }
  constVal
  { $node = new ConstValNode($constVal.value); }
  Identifier
  { boolean idExists = false;
    if (idList != null)
      for (int i=0;i<idList.size() && !idExists;i++)
        if (idList.get(i).equals($Identifier.text))
          idExists = true;
    if (!idExists)
      throw new UnknownIdentifierException($Identifier.text + " is not defined");
    $node = new IdentifierNode($Identifier.text);
  }
  | '(' orFormula[idList] ')' 
  { $node = $orFormula.node; }
  |
  constVal returns [Object value]
  | StringConst
  { $value = $StringConst.text.substring(1,$StringConst.text.length() - 1); }
  | numConst
  { try {
    $value = Integer.parseInt($numConst.text);
  } catch (NumberFormatException e) {
    try {
      $value = Long.parseLong($numConst.text);
    } catch (NumberFormatException e1) {
      $value = Double.parseDouble($numConst.text);
    }
  }
  |
  var [ArrayList<String> idList] returns [ACEColNode node]
  | Alias
  { if (!aliases.containsKey($Alias.text))
    throw new CorALException("Unknown alias: " + $Alias.text);
    $node = new AliasNode($Alias.text);
  }
  | (selector[idList]
  { $node.addChild($selector.node); }
  |
  aliasNodes.add((AliasNode)$node);
  ```
Bringing Autonomic Services to Life

className returns [String value]
:
    id=Identifier
    { id=id=Identifier}
    { $value = $id.text; }
        (" id=Identifier) 
    { $value = $value + "." + $id.text; }
    ;

selector [ArrayList<String> idList] returns [ACEColNode node]
:
    " id=Identifier
    (arguments[idList])?
    try {
        $node = new SelectorNode($Identifier.text,true,$arguments.list.size());
        for (int i=0;i<$arguments.list.size();i++)
            $node.addChild($arguments.list.get(i));
    } catch (NullPointerException e) {
        $node = new SelectorNode($Identifier.text,false,0); 
    }
        (" expression[idList] ")
    { $node.addChild($expression.node); }
    )
;

arguments [ArrayList<String> idList] returns [ArrayList<ACEColNode> list]
:
    { $list = new ArrayList<ACEColNode>(); }
        (" (expressionList[idList]
    { $list = $expressionList.list; }
    )? " )
;

expressionList [ArrayList<String> idList] returns [ArrayList<ACEColNode> list]
:
    e=expression[idList]
    { $list = new ArrayList<ACEColNode>(); $list.add($e.node); }
        (" e=expression[idList]
    { $list.add($e.node); }
    )
;

e=expression [idList]
:
    orFormula[idList]
    { $node = $orFormula.node; }
;

numConst
: IntConst | DoubleConst 
;

Alias
Bringing Autonomic Services to Life

$ Letter (Letter|JavaIDDigit)*$

StringConst:

""{(Letter|JavaIDDigit|\'|\t|\r|\n|u000C)*}""

IntConst:

('0'..'9')+

DoubleConst:

('0'..'9')\'.\'('0'..'9')+

Identifier:

Letter (Letter|JavaIDDigit)*

fragment Letter:

\'u0024\'

| \'u0041..u005a\'
| \'u005f\'
| \'u0061..u007a\'
| \'u00c0..u00d6\'
| \'u00d8..u00f6\'
| \'u00f8..u00ff\'
| \'u0100..u1fff\'
| \'u0300..u33ff\'
| \'u0400..u9fff\'
| \'uf900..uafff\'

fragment JavaIDDigit:

\'u0030..u0039\'

| \'u0060..u0069\'
| \'u060..u069\'
| \'u0966..u096f\'
| \'u09e6..u09ef\'
| \'u0ae6..u0aef\'
| \'u0be6..u0bef\'
| \'u0ce6..u0ceff\'
| \'u0de6..u0def\'
| \'u0e50..u0e5f\'
| \'u0ed0..u0ed9\'
| \'u1040..u1049\'

WS:

\("\\r\\n|u000C\\n\)\{/channel=HIDDEN;}\n
Bringing Autonomic Services to Life

8.2.3 AL

grammar AL;

@header {
package cascadas.supervision.assessment;
import cascadas.supervision.interaction.data.SupervisionData;
import cascadas.supervision.common.nodes.*;
import cascadas.supervision.common.exceptions.*;
import java.util.ArrayList;
}

@lexer::header {
package cascadas.supervision.assessment;
}

@members {
    private ArrayList<VarNode> varNodes = new ArrayList<VarNode>();
    private boolean isSingleAssessment = true;

    public String getErrorMessage(RecognitionException e, String[] tokenNames) {
        String msg = super.getErrorMessage(e,tokenNames);
        if (msg == null) {
            if (e instanceof SemanticActionException)
                msg = e.getMessage();
        }
        return msg;
    }

protected void mismatch(IntStream input, int type, BitSet follow) throws RecognitionException {
    MismatchedTokenException mte = new MismatchedTokenException(type, input);
    throw mte;
}

} catch (RecognitionException ex) {
    throw ex;
}

assessmentProperty returns [ACEColNode node] :
    ('single' | 'multiple' {
        isSingleAssessment = false;
    })
    'assessment' ':' formula>null
{
    ALRootNode root = new ALRootNode(isSingleAssessment,varNodes);
    $node = root;
    $node.addChild($formula.node);
}

form ula [ArrayList<String> idList] returns [ACEColNode node]
:
    orFormula[idList]
{
    $node = $orFormula.node;
}|
    '(' quantifier Identifier
{
    ArrayList<String> temp = new ArrayList<String>();
    if (idList != null)
        for (int i=0; i<idList.size(); i++) {
            if (idList.get(i).equals($Identifier.text))
                throw new DuplicateIdentifierException($Identifier.text + " already exists");
            else
    }{
Bringing Autonomic Services to Life

```java
\[
temp.add(idList.get(i));
\]

idList = temp;
idList.add($Identifier.text);

'=\ e1=expr[idList] \to\ e2=expr[idList] \; p=formula[idList] \;')

\[
$node = new QuantifierNode($quantifier.text,$Identifier.text);
$node.addChild($e1.node);
$node.addChild($e2.node);
$node.addChild($p.node);
\]

quantifier

| 'forall'
| 'exists'

orFormula [ArrayList<String> idList] returns [ACEColNode node]

scope {
    ACEColNode firstAndFormula;
}

and=andFormula[idList]

| ()" and=andFormula[idList]

| if ($orFormula::firstAndFormula != null) {
    $node = new OrNode();
    $node.addChild($orFormula::firstAndFormula);
    $orFormula::firstAndFormula = null;
}

$node.addChild($and.node);

)

)*

| if ($orFormula::firstAndFormula != null)
    $node = $orFormula::firstAndFormula;

andFormula [ArrayList<String> idList] returns [ACEColNode node]

scope {
    ACEColNode firstSubformula;
}

sub=subFormula[idList]

| $andFormula::firstSubformula = $sub.node;

| '&&' sub=subFormula[idList]

| if ($andFormula::firstSubformula != null) {
    $node = new AndNode();
    $node.addChild($andFormula::firstSubformula);
    $andFormula::firstSubformula = null;
}

$node.addChild($sub.node);

)*

| if ($andFormula::firstSubformula != null)
    $node = $andFormula::firstSubformula;

subFormula [ArrayList<String> idList] returns [ACEColNode node]

```
Bringing Autonomic Services to Life

```java
e1=expr[idList] (relop e2=expr[idList])?
    {
        try {
            $node = new RelopNode($relop.text);
            $node.addChild($e1.node);
            $node.addChild($e2.node);
        } catch (NullPointerException e) {
            $node = $e1.node;
        }
    } |
    ! '(' orFormula[idList] ')
    {
        $node = new NotNode();
        $node.addChild($orFormula.node);
    }

relop :
    '<' |
    '<=' |
    '==' |
    '>=' |
    '>' ;

expr [ArrayList<String> idList] returns [ACEColNode node]
scope {
    ACEColNode root;
}

( '-' ? t=term[idList]
    {
        if ($expr::root != null)
            $expr::root.addChild($t.node);
        else
            $expr::root = $t.node;
    }?
     t=term[idList]
    {
        ACEColNode left = $expr::root;
        $expr::root = new PlusNode();
        $expr::root.addChild(left);
        $expr::root.addChild($t.node);
    }|
     ? t=term[idList]
    {
        ACEColNode left = $expr::root;
        $expr::root = new MinusNode();
        $expr::root.addChild(left);
        $expr::root.addChild($t.node);
    }
    )|

term [ArrayList<String> idList] returns [ACEColNode node]
scope {
    ACEColNode root;
}

? f=factor[idList]
    {
        $term::root = $f.node;
    }
( '*' ? f=factor[idList]
    {
        ACEColNode left = $expr::root;
        $expr::root = new PlusNode();
        $expr::root.addChild(left);
        $expr::root.addChild($f.node);
    }|

? f=factor[idList]
    {
        ACEColNode left = $expr::root;
        $expr::root = new MinusNode();
        $expr::root.addChild(left);
        $expr::root.addChild($f.node);
    }|

? f=factor[idList]
```

Bringing Autonomic Services to Life

```java
{  
    ACEColNode left = $term::root;
    $term::root = new MultNode();
    $term::root.addChild(left);
    $term::root.addChild($f.node);
}
```

```
| */ f=factor[idList]  
| {  
    ACEColNode left = $term::root;
    $term::root = new DivNode();
    $term::root.addChild(left);
    $term::root.addChild($f.node);
}
| */ f=factor[idList]  
| {  
    ACEColNode left = $term::root;
    $term::root = new ModNode();
    $term::root.addChild(left);
    $term::root.addChild($f.node);
}
| */ f=factor[idList]  
| {  
    $node = $term::root;
}
```

```
factor [ArrayList<String> idList] returns [ACEColNode node] 
:  
    var[idList]  
    {  
        $node = $var.node;
    }
| constVal  
| {  
    $node = new ConstValNode($constVal.value);
}
| identifier  
| {  
    boolean idExists = false;
    if (idList != null)  
        for (int i=0;i<idList.size() && !idExists;i++)  
            if (idList.get(i).equals($Identifier.text))  
                idExists = true;
    if (!idExists)  
        throw new UnknownIdentifierException($Identifier.text + " is not defined");
    $node = new IdentifierNode($Identifier.text);
}
| '(' orFormula[idList] ')'
| {  
    $node = $orFormula.node;
}
```

```
constVal returns [Object value] 
:  
    StringConst  
    {  
        $value = $StringConst.text.substring(1,$StringConst.text.length() - 1);
    }
| NumConst  
| {  
    try {
        $value = Integer.parseInt($NumConst.text);
    } catch (NumberFormatException e) {
    }
    try {
        $value = Long.parseLong($NumConst.text);
    } catch (NumberFormatException e1) {
    }
    try {
        $value = Double.parseDouble($NumConst.text);
    }
```

```
```
Bringing Autonomic Services to Life

```java
var [ArrayList<String> idList] returns [ACEColNode node]
:
  ("\rrow\r")
  {
    if (isSingleAssessment)
      throw new ALException("Invalid keyword 'row' in single assessment");
    $node = new VarNode();
  }
  ("\rtable\r")
  {
    if (!isSingleAssessment)
      throw new ALException("Invalid keyword 'table' in multiple assessment");
    $node = new VarNode();
  }

  selector[idList]
  {
    $node.addChild($selector.node);
  }

  varNodes.add((VarNode)$node);

className returns [String value]
:
  id=Identifier
  {
    $value = $id.text;
  }
  (\r.\r id=Identifier)*
  {
    $value = $value + "." + $id.text;
  }

selector [ArrayList<String> idList] returns [ACEColNode node]
:
  \r.\r Identifier
  (arguments[idList])?
  {
    try {
      $node = new SelectorNode($Identifier.text,true,$arguments.list.size());
      for (int i=0;i<$arguments.list.size();i++)
        $node.addChild($arguments.list.get(i));
    } catch (NullPointerException e) {
      $node = new SelectorNode($Identifier.text,false,0);
    }
  }

  (\r[expression[idList]\r]\r)
  {
    $node.addChild($expression.node);
  }

arguments [ArrayList<String> idList] returns [ArrayList<ACEColNode> list]
:
  {
    $list = new ArrayList<ACEColNode>();
  }
  (\r(\rexpressionList[idList]\r)\r)
  {
    $list = $expressionList.list;
  }
)? ";"

expressionList [ArrayList<String> idList] returns [ArrayList<ACEColNode> list]
:
```

Page 75 of 76
Bringing Autonomic Services to Life

```java
e=expression[idList]
{
    $list = new ArrayList<ACEColNode>();
    $list.add($e.node);
}

{, e=expression[idList]
    {$list.add($e.node); })*
;

expression [ArrayList<String> idList] returns [ACEColNode node]
{
    orFormula[idList]
    { $node = $orFormula.node; };

    StringConst :
        ""{Letter}"\n{Letter}"
;

    NumConst :
        ('0'..'9')+ ('.' ('0'..'9')+)?
;

    Identifier :
        Letter (Letter)*)
;

    fragment
    Letter :
        'u0024'
        | 'u0041'..'u005a'
        | 'u006f'
        | 'u0061'..'u007a'
        | 'u00d0'..'u00d6'
        | 'u00d8'..'u00e6'
        | 'u00e8'..'u00ef'
        | 'u0100'..'u1ff'
        | 'u0340'..'u034f'
        | 'u0340'..'u034f'
        | 'u0340'..'u034f'
        | 'u0340'..'u034f'
        | 'u0340'..'u034f'
        | 'u0900'..'u09ff'
;

    fragment
    JavaIDDigit :
        'u0030'..'u0039'
        | 'u0066'..'u006f'
        | 'u06f0'..'u06f9'
        | 'u0966'..'u096f'
        | 'u09e6'..'u09ef'
        | 'u0a66'..'u0a6f'
        | 'u0a66'..'u0a6f'
        | 'u0b66'..'u0b6f'
        | 'u0be7'..'u0bef'
        | 'u0c66'..'u0c6f'
        | 'u0ce6'..'u0cef'
        | 'u0d66'..'u0d6f'
        | 'u0d66'..'u0d6f'
        | 'u0e50'..'u0e5f'
        | 'u0eda0'..'u0ed9'
        | 'u1040'..'u1049'
;

    WS :
        ""{'\n'\r'\t'\n'}\n{channel=HIDDEN;}
;
```