Deliverable D1.3

First Prototype Integration (release 1)

Appendix: Prototype Documentation and Tutorial

Status and Version:	Version 1.1, Final	
Date of issue:	14.01.2008	
Distribution:	Project Internal	
Author(s):	Name	Partner
	Sandra Haseloff	UNIK
	Rico Kusber	UNIK
	Nermin Brgulja	UNIK
	Borbála Katalin Benkő	BUTE
	Peter Deussen	FOKUS
	Edzard Höfig	FOKUS
	Antonio Manzalini	TI
	Rosario Alfano	TI
	Antonietta Mannella	TI
	Mario Giacometto	TI
	Marco Mamei	UNIMORE
Checked by:	Franco Zambonelli	UNIMORE

Table of Contents

1 Introduction 4

1.1 Purpose and Scope 4

1.2 Reference Material 4

1.2.1 Reference Documents 4

1.2.2 Acronyms 5

1.3 Document History 5

1.4 Document Overview 6

2 New ACE Features at a Glance 6

2.1 Advanced Logging 6

2.2 Parallel Plans 7

2.3 Advanced Plan Creation and Modification Rules 7
2.4 Specific Functionality Thread Pool 7
2.5 Support for Supervision 8
3 Updated ACE Component Model 8
3.1 Gateway 9
3.1.1 Event-based Communication 9
3.1.2 Contracted Connection Handling 10
3.1.2.1 Contract Establishment 10
3.1.2.2 Contract Usage 11
3.1.2.3 Contract Cancellation 12
3.2 Manager 12
3.2.1 Event Processing 12
3.2.2 Life Cycle Management 13
3.2.3 The Life Cycle Protocol 15
3.3 Executor and Plan 16
3.3.1 The Plan 17
3.3.1.1 Transitions 17
3.3.1.2 Plan executions 17
3.3.2 Executor Architecture 18
3.3.2.1 Impulse flow 19
3.3.2.2 Implementation 19
3.3.3 Parallel Plans 19
3.3.3.1 Semantics 19
3.3.3.2 Context gathering 20
3.3.3.3 Advantages 20
3.4 Facilitator and Self Model 20
3.4.1 Facilitator 21
3.4.2 Self Model 22
3.4.2.1 States and transitions 23
3.4.2.2 Plan creation and modification rules 28
3.4.2.3 Self Model Syntax 32
3.4.2.4 Self Model how to 34
3.5 Functionality Repository 35
3.5.1 Basic Concept of the Functionality Repository 35
3.5.1.1 Purpose 35
3.5.1.2 Connection with other ACE organs and external ACEs 36
3.5.1.3 Design concepts 36
3.5.1.4 Terms 37
3.5.2 The Functionality Model 37
3.5.2.1 Functionality name 38
3.5.2.2 Black box model 38
3.5.2.3 Underlying call sequence 39
3.5.2.4 Event layer 40
3.5.3 Accessing a Service from Another ACE 41
3.5.4 Specific Functionalities and Common Functionalities 42
3.5.4.1 Common functionalities 42
3.5.4.2 Specific functionalities 43

3.5.4.3 Deploying a specific functionality to the Repository 44
3.5.5 Statelessness 44
3.5.6 Accessing the Sessions and the Repository-Internal Data Structures 44
3.5.6.1 CallParameters 45
3.5.6.2 CallWideContext 45
3.5.6.3 Sessions 45
3.5.7 Accessing other ACE Resources 46
3.5.7.1 Logging 46
3.5.7.2 ThreadPool 46
3.5.8 What Happens when a Service is Called? 47
3.5.9 Technical Expectations, Error Handling 47
3.5.9.1 Technical expectations 47
3.5.9.2 Error handling during the deployment 48
3.5.9.3 Error handling during the call 48

4 Integration Features 48

4.1 Knowledge Network Support 48

4.2 Supervision Support 50

4.3 Aggregation Support 53
4.3.1 Group Communication 54
4.3.2 Initialisation 55
4.3.3 Group Communication using REDS 55
4.3.4 Integration of Self-Aggregation in the Gateway Component 55
4.3.5 Updating the Goals on the Neighbour List 55
4.3.6 Using Self-Aggregation Algorithm to Update the Neighbour List 56
4.3.7 Self-Aggregation Issues with Multiple Different Goals 57
4.3.8 Load-balancing Example 57
4.3.9 Discussion 58

4.4 Security Support 58

5 Sample Application Description 59

5.1 Structure of the Application 59

5.2 Supervision in the Advertisement Example 62

5.3 Context Verification in the Advertisement Example 63

6 Tutorial: Programming and Executing ACEs 63

6.1 Developing ACE Specific Functionalities 64
6.1.1 Functionality Java Classes 64
6.1.2 Functionality Descriptor 69

6.2 Creating the ACE Self Model 72

6.3 Executing the ACE Application 75
6.3.1 Setting Up the ACE Application Environment 75
6.3.2 Running the ACE Application 76

7 Conclusion and Outlook 79

1 Introduction

1.1 Purpose and Scope

This document constitutes the deliverable 1.3 "First Prototype Integration (release 1)". It describes the current ACE component model and new features that have been researched, developed, and implemented since the last deliverable D1.2 (cf. [D1.2]) as well as the integration work done between WP1 and the other WPs of the CASCADAS project. Additionally, a sample application is presented and a tutorial is given on how to program and execute ACEs.

Concerning the content, some parts of this document are similar to deliverable D1.2. We reference the work which has already been done in deliverable D1.2 wherever possible. Nevertheless, due to intensive modifications we completely describe every component of the Toolkit in order to keep this document comprehensible and to present the Toolkit in a coherent manner.

1.2 Reference Material

1.2.1 Reference Documents

[ACEL] ACELandic - A Scripting Language for Autonomic Communication Elements, available at: https://www2.mik.bme.hu/repositories/cascadas/trunk/wp1/dev/doc/ACELandic.doc.

[Chsw] Chainsaw log viewer http://logging.apache.org/chainsaw/.

[D1.2] Deliverable D1.2: Prototype implementation (release 1), July 2007.

[D3.1] Deliverable D3.1: Aggregation Algorithms, Overlay Dynamics and Implications for Self-Organised Distributed Systems, December 2006.

[D3.3] Deliverable D3.3: Software Implementation of Modules for Adaptive Aggregation, June 2007.

[D5.3] Deliverable D5.3: The Open Toolkit for Knowledge Networks, January 2008.

[DIET] DIET Agent Platform, available online: http://diet-agents.sourceforge.net.

[Log4J] Log4J - http://logging.apache.org/log4j/.

[OOJD] OOjDREW – Object Oriented Java Deductive Reasoning Engine for the Web, available online: www.jdrew.org/oojdrew/.

[REDS] REDS – A Reconfigurable Dispatching System, available online: http://zeus.elet.polimi.it/reds.

[RuleML] RuleML - Rule Markup Language. RuleML 0.88 stripped syntax is used available online: www.ruleml.org/0.88/.

1.2.2 Acronyms

ACE	Autonomic communication element
DIET	DIET Agent Platform (see [DIET])
FCE	FunctionalityCallEvent
GA	Goal achievable
GN	Goal needed
GUI	Graphical user interface
ID	Identification
No-arg	Without arguments
Log4J	Logging framework (see [Log4J])
PEX	PlanExecutor
REDS	Reconfigurable Dispatching System (see [REDS])
WP	Work package
XML	Extensible Markup Language

1.3 Document History

Version	Date	Authors	Comment
0.1	27/09/2007	Sandra Haseloff	Proposal of initial document structure and responsibilities
0.2	09/10/2007	Sandra Haseloff	Modifications in structure/responsibilities after WP1 PhC
0.9	14/12/2007	All Authors	Contributions
1.0	21/12/2007	Rico Kusber, Sandra Haseloff	Pre-final editing
1.1	14/01/2008	Sandra Haseloff, Nermin Brgulja, Rico Kusber	Final editing after internal review

1.4 Document Overview

In the first part of this document, acronyms, general definitions, and used references are clarified. Chapter two gives an overview of new ACE features which have been integrated during the last research period. Issues like an advanced logging mechanism and parallel Plan execution are explained here. Thereafter, in chapter three the current state of the ACE component model is presented with respect to what was already developed before and what are the changes that have been introduced. All ACE organs are depicted in detail in order to document their features, properties, and the way to use them. Chapter four then describes the integration work that has been and will be done between WP1 and the other WPs of the CASCDAS project. It is explained how the Toolkit can be utilised to implement for example Knowledge Network or Supervision features based on the ACE concept. In order to show the feasibility of the ACE concept the WP1 example application was extended and is described in chapter five. Chapter six then presents a tutorial which can be used to learn how to program and execute ACE based applications. Detailed instructions are provided on developing with the CASCADAS Toolkit. The last section of this document, chapter seven, summarises what is written before and concludes with an outlook on how the research and development of the CASCADAS Toolkit will continue.

2 New ACE Features at a Glance

ACEs have been designed with the goal to fulfil a set of dedicated demands. This includes, for example, exhibiting autonomic and situation-aware behaviour or providing services in a self-similar and extensible manner. How these demands have been addressed is described in section two of [D1.2]. Throughout the whole time the ACE concept has been researched, designed, and implemented, permanent adaptation and extension has taken place. This section shortly describes new features that were integrated into the Toolkit within the last research period.

2.1 Advanced Logging

In order to enable debugging and to keep applications comprehensible, a logging mechanism was introduced. It is useful not only to develop the Toolkit itself, but it can be utilised by ACE application developers as well.

To enable the logger in an ACE based application, a set of configuration inputs has to be declared within the applications `settings.properties` file. By default, logging to port 4000 on localhost is enabled and `INFO` is specified as log level. Changes can individually be made depending on the needs of the person who works with the log output. Beside port 4000, a list of alternative outputs can be defined so that for example logging to the console or a file is possible as well. To display the log output in a well arranged manner, the Chainsaw logging tool can be used (cf. [Chsw]). Chainsaw is a GUI-based log viewer designed to view Log4J messages which, in case of the CASCADAS Toolkit, come from

- the application environment in which ACEs are running,
- all running ACEs and each organ of all ACEs,
- and the specific functionalities of all running ACEs.

Differentiating between sources of log messages on such a fine-grained level as listed above enables the user respectively developer to investigate occurring problems dedicated to their assumed cause. If for example external events get lost for an unknown reason, it may be a good starting point to look at the log messages of the Gateway organ instead of reading all log messages of the affected ACE. Nonetheless, Chainsaw facilitates also to display all log entries of an ACE without separating between different organs. This may be useful in case of investigations where e.g. the log message order is important. Section 6.1.1 describes how a developer can design specific functionalities in a way that they use the logger. So, logging is not limited to the self-similar parts of ACEs which are provided by the Toolkit but can be integrated in any application that is based on the ACE concept.

2.2 Parallel Plans

During the Toolkit development, it turned out to be necessary that ACEs must be able to perform multiple actions at the same time. Those actions could for example be:

- waiting for a response to a service call,
- while performing computationally intensive calculations,
- and reacting to context changes, all at the same time.

Though it is possible to declare these activities as a sequence, it is much more convenient and comprehensible to design a number of independent Plans executed in parallel. Section 3.3.3 explains the usage of parallel Plans.

2.3 Advanced Plan Creation and Modification Rules

To exhibit an autonomic character, ACEs can change their behaviour according to context changes in their environment. Those behavioural changes can be achieved by modifying the Plan currently under execution. Until now, this modification was triggered by context changes only. In addition to that, the feature to be able to react to any event that may occur was introduced. This extends the range of causes an ACE can perceive in order to change its behaviour accordingly. Imagine the following scenario.

> *An ACE performs its normal operation when suddenly one of its contracts breaks for an unforeseen reason. In that case, the ACE gets notified of that by receiving a* `CancelContractEvent`*. Utilising the new Plan modification feature, this event can be used to trigger a change in the Plan in a way that causes the affected ACE to re-establish all its contracts.*

In section 3.4.2.2 the Plan creation and modification rules are presented in detail.

2.4 Specific Functionality Thread Pool

Each ACE is, besides utilising a set of common functionalities, able to include and access a repository of specific functionalities. The code stored in that repository enriches an ACE with individual services it can use and expose to the outside world. However, regarding the ACE's life cycle management, individual code can cause serious problems as for instance to prevent an ACE from shutting down correctly if custom threads are not terminated properly. To avoid those effects, a thread pool was introduced. It provides specific functionalities with the possibility to start individual threads but, at the same time, ensures

that the ACE, respectively the Functionality Repository organ, keeps full control over all of them. Section 3.5.7.2 describes the thread pool.

2.5 Support for Supervision

The Toolkit offers support for Knowledge Networks, Aggregation, Security, as well as Supervision. The last one is pointed out in this section because its integration strongly affects the core of the ACE model. Integration of the other aspects is more encapsulated in functionality instead of architecture itself.

The purpose of the Supervision System is to observe the behaviour of ACEs and to intervene in case any anomaly is detected. Support for supervision is realised by a dedicated organ which observes a set of points that are important for ACE internal behaviour as well as internal and external communication. Those points are for example the message queues of the Bus, i.e. Manager organ, and the Gateway. In addition to check points within the ACE, an external set of supervision ACEs is contracted which includes sensors and effectors. How Supervision works in detail is described in chapter 4.2.

3 Updated ACE Component Model

The ACE component model forms the basis of the CASCADAS Toolkit. By providing a core which is used to create each and every ACE, it enables designing applications in a self-similar manner. During the CASCADAS project, the component model has evolved through different stages, always adapting to current research results and requirements. Originating from the idea to separate a common part, which is identical for each ACE, and a specific part which enables the usage of customised, individual functionality, an organ concept was introduced. We use the term *organ* in analogy to the biological model of the human organism. Assembling an ACE as a set of organs where each of them is dedicated to fulfil clearly defined tasks leads to a well structured and modularised component model (see Figure 1). In accordance with continuously obtained research results, this organ based component model is being adapted and improved.

Compared to the ACE component model described in [D1.2], the main modifications are:

- Bus and Lifecycle Manager were merged to the Manager organ.
- A Supervision organ was introduced.
- The Reasoner was replaced by the Executor.

Currently, an ACE comprises six organs which are able to interact by exchanging events. The **Gateway** is responsible for external communication, i.e. exchanging events with other ACEs, whereas the **Manager** realises, first, internal communication between all organs (this part is called Bus), and second, the life cycle management of the ACE it belongs to. Interaction with the Supervision System takes place utilising the **Supervision** organ. The **Facilitator** uses a Self Model to create and schedule all Plans an ACE can fulfil. Afterwards, these Plans are executed by the **Executor** which may therefore utilise common or specific functionalities provided and organised by the **Functionality Repository**. Figure 1 illustrates the ACE component model and the relations between all organs. The following sections describe the purpose, design, and features of every organ in detail.

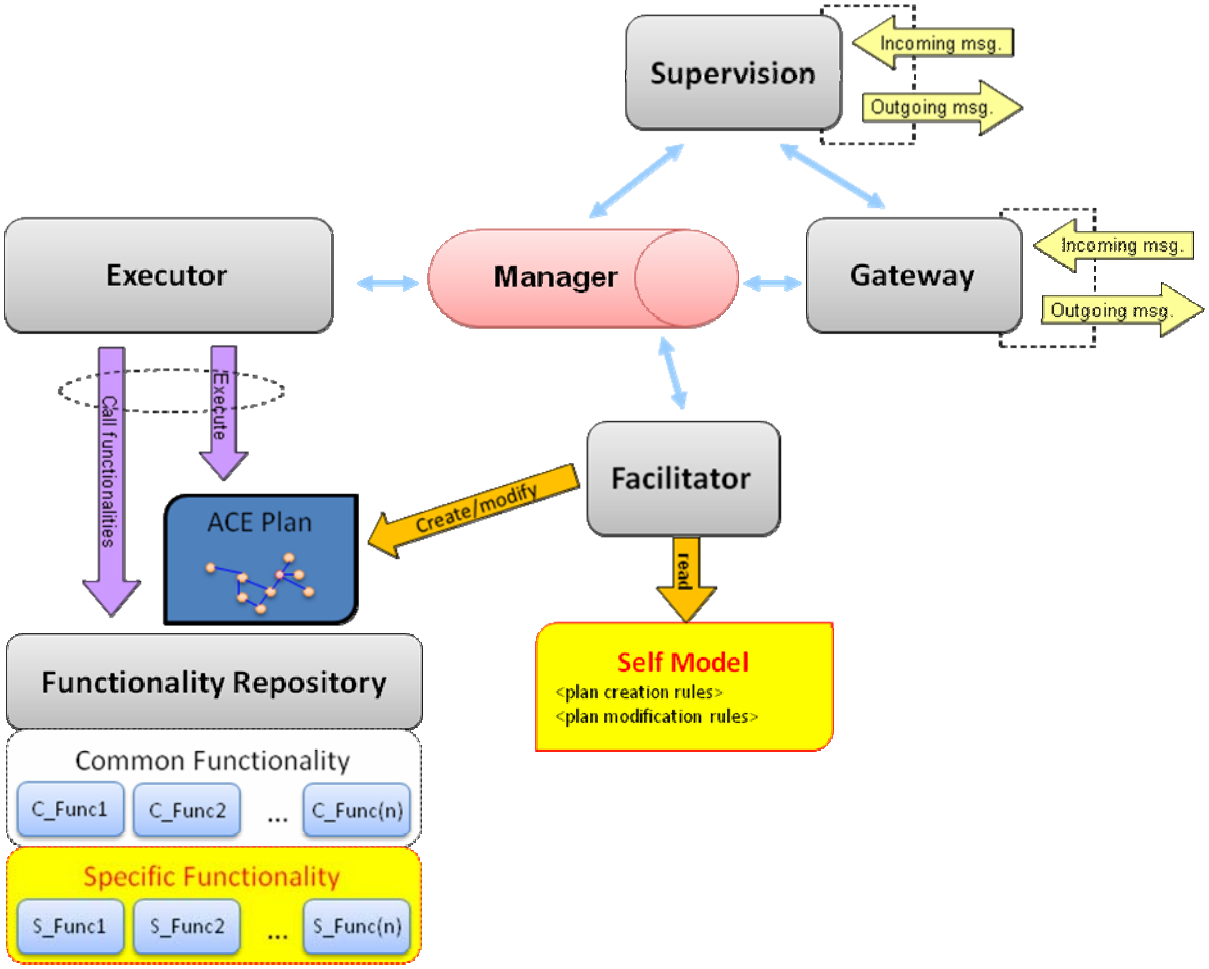


Figure 1: ACE component model showing ACE organs and their relations

3.1 Gateway

The Gateway is the ACE internal organ in charge of communicating with the external world. There are two basic communication mechanisms: anonymous REDS-based GN-GA transport, which is connection-less (cf. [REDS]); DIET-based connection-oriented transport that uses contracts to assure the communicating parties about properties of the underlying communication channel (cf. [DIET]).

3.1.1 Event-based Communication

The GN-GA protocol is supported by a publish-subscribe mechanism where some ACEs subscribe to events of interest and some other ACEs publish information. A suitable routing structure based on the REDS middleware takes care of delivering the events to the proper subscribers. Let us explain this communication more explicitly:

- Messages to be sent to other ACEs via the Gateway are represented by an ACE Envelope object (`cascadas.ace.event.Envelope`). Specific messages (e.g.,

`cascadas.ace.event.GoalAchievableEvent`) subclass from the Envelope class and add variables representing the message payload. The class Envelope contains a String variable `reds_routing` used to route the Envelope across the REDS network.

- The default value of this variable is 'ace-all'. This expresses that the messages should be broadcasted to all the ACEs in the network. ACEs, at start up, subscribe to all the messages having `reds_routing` = "ace-all" (this subscription is in the `RedsClient` constructor). REDS forwards the messages on the basis of a pattern matching schema. Current implementation (see `subscribe` method in `RedsClient`) supports String contains pattern matching, i.e. an Envelope matches a subscription if its `reds_routing` String is contained in the subscription String.

GN-GA publish-subscribe operations are performed via common functionalities called from the Self Model.

```
<action>gn_sender_service(goalName=dresscode,myAddress=?globalSession://ac
eAddress)</action>


<action>gn_answer_service(goal=?inputMessage://goal,serviceName=hobby_prov
ider_service,myAddress=?globalSession://aceAddress)</action>
```

The `gn_sender_service` and `gn_answer_service` are used to send a GN-GA message respectively.

```
<trigger>cascadas.ace.event.GoalNeededEvent</trigger>
<guard_condition>EQUALS(?inputMessage://goalName,hobby)</guard_condition>


<trigger>cascadas.ace.event.GoalAchievableEvent</trigger>
<guard_condition>EQUALS(?inputMessage://goalName,dress)</guard_condition>
```

The above code waits until a suitable GN-GA message arrives to perform some operations.

3.1.2 Contracted Connection Handling

A contract is seen as an agreement between 2..n partner ACEs about certain constraints for a defined period of time, more precisely: from the establishment of a contract until cancellation of a contract. Currently these constraints are reduced to a set of mappings between a role name and an ACE address, but may include other requirements or conditions in the future (e.g. security constraints, lifelines, or billing information).

3.1.2.1 Contract Establishment

Technically a contract is established by sending an `EstablishContractEvent` through the Gateway of an ACE. The event needs to contain a `Contract` object specifying the role-address mapping, which is used by the Gateway to create connections to all of the partners defined within the contract. Connections are based on DIET mirror agents. For each connection between the initiating ACE and a given contractual partner a DIET mirror agent is created and migrated to the other ACE's environment. The receiving ACE subsequently creates another mirror agent and sends it back to the initiating ACE, resulting in a bi-directional, dedicated connection. The `EstablishContractEvent` is then forwarded to the receiving ACE where it appears on the internal bus (i.e. the Manager

organ) and also triggers the exact same connecting behaviour. At the end a fully-connected mesh has developed where each partner can communicate with every other one using contract-specific connections. We refer to this mesh as the "contractual context".

For an example refer to Figure 2, where a contract between three partners has been created using the mapping triple `{"user",Env2:Foo / "provider",Env3:Bar /"supervisor",Env1:Ding}`, where each entry defines a role (e.g. "user") and an address (e.g. Env2:Foo).

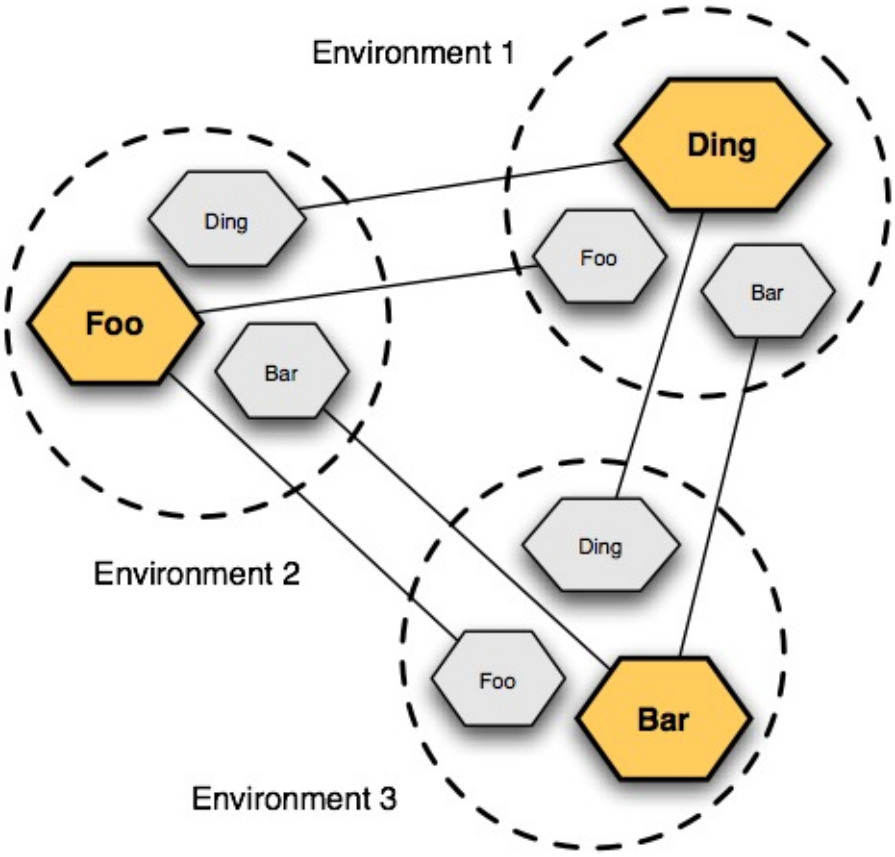


Figure 2: A contract with 3 partners

For the user of the ACE Toolkit contract establishment is trivial: the common functionality *contract_n_establishment_service* (see 3.5.4.1) takes care of sending a properly initialised event. The establishment service takes a mandatory parameter *contractId*, defining an id that references the contract in the execution session after it has been established. All remaining arguments are interpreted as `{name,address}` pairs. Every ACE is able to request the establishment of contracts, normally after receiving matching GA events, which contain the address of the goal-achieving ACE.

3.1.2.2 Contract Usage

Once a contract is in place, so-called *contract events* may be sent using contract roles as destinations. An event that is sent along a contractual connection needs to derive from `cascadas.ace.gateway.ContractEvent`, in most cases this will be through

`cascadas.ace.event.ServiceUsageEvent`. This makes sure that a contract and an optional target role can be associated with the event.

Sending an event may be done using e.g. the *send* common functionality, which takes a Java class name for the *event* parameter, a contract for the *contract* parameter (for example from the current execution session), and an optional *role* parameter that is supposed to contain the role name of the ACE the event is sent to. It is not possible to send events to oneself.[1] If the role is not specified, the event will be forwarded to all connections of the contract.

3.1.2.3 Contract Cancellation

Contracts may be cancelled any time and from any of the participating ACEs. After a contract has been cancelled, all mirror agents have been removed and no direct communication is possible anymore.

There are two reasons a contract is cancelled:

1) The DIET connection has broken down. DIET employs timeout mechanisms that check for lifelines of a connection, proper connection setup, and the like. It might happen that a connection is really broken (e.g. an unplugged cable), but it is also possible that a timeout value is too low for the situation, both lead to cancellation of the contract. Most timeout values can be configured in the DIET section of the global configuration file *settings.properties*.
2) An ACE sends a `CancelContractEvent`, triggering the cancellation of the contract on purpose.

When a contract is cancelled, a `CancelContractEvent` appears on the Bus of each of the participating ACEs and all attempts to subsequently use the contract will result in an error.

The *cancel_contract_service* common functionality can be used to conveniently end a contract. It takes a single argument: the contract to cancel.

3.2 Manager

The Manager is a newly introduced ACE organ which results from merging the Bus and the Lifecycle Manager. Integrating these two organs became necessary as a consequence of further developments concerning the lifecycle and internal event handling of an ACE. The following section describes the tasks and features of the Manager, the protocol necessary to perform a lifecycle action[2] and the way in which other ACE organs are affected by respectively involved into life cycle management.

3.2.1 Event Processing

The messaging functionality of the Bus was completely moved to the Manager so that no changes to the internal event processing mechanism were necessary. Organs are still handling events in the same way as documented in [D1.2].

[1] Use the Manager organ for this!
[2] Lifecycle actions are *clone, destroy, init, move, reset, start,* and *stop.*

An `AceJob` deriving class implements methods with a signature of `handle(`*event type*`)`, where *event type* is of the event class that the organ would like to handle. Organs are free in choosing 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`. Sending of events is accomplished by using the `send` method of any event type.

Note: The `standby()` method of `cascadas.ace.event.Event` has been deprecated, along with all other methods that are used for synchronous sending of events. The usage of synchronous event sending turned out to be a source of problems between different threads, leading to (in the best case) complex architectural solutions and (in the worst case) to deadlocks. Please refrain from using these methods.

Users of the Toolkit will deal with events in three possible ways

1) As a specification to the *trigger* clause in a Plan transitions (see section 3.3.1.1)
2) As a parameter to the generic send functionality (see section 3.5.4.1) used in *action* clauses of a Self Model transition
3) Directly as a Java object for input or output events when writing a *specific functionality* (see section 3.5.2.4)

All of the events that may be processed in one of the three ways given above will need to be derived from either `cascadas.ace.event.AceLocalEvent` or `cascadas.ace.event.ServiceUsageEvent`. `AceLocalEvent` is the basis for all local events transmitted via the Bus and accessible to a user. `ServiceUsageEvent` serves as the basis for all events to be transmitted to other ACEs via the Gateway and subsequently appear on the Bus of the receiving ACE. Both event classes add support for storing and accessing arbitrary parameters from the Self Model.

`ServiceUsageEvent` is only usable in conjunction with a valid contract that needs to be set before sending it (e.g. using the common *send* functionality or by handing it back after finishing a specific functionality block). It then exposes *contract*, *sourceRole* and *destinationRole* parameters that might be queried or set.

Users of the Toolkit are encouraged to write their own specific event classes. It is also possible and sensible to re-use existing events, where adequate (for example `cascadas.ace.event.ServiceCallEvent`).

New to the event processing system is that the Manager takes care of filtering event classes according to the life cycle state an ACE currently is in. According to Figure 4 (see below), there are two states in which life cycle events have to be handled by each ACE organ, the inactive and the running state. While being in running state, event processing is not restricted. In inactive state only the processing of life cycle events of the type ***init, start*** or ***destroy*** is allowed. The Manager filters all events and dispatches only those, which are valid. For all other ACE organs this implies that they do not have to care about the life cycle state. They can handle life cycle events equally each time only distinguishing between the different types of life cycle actions.

3.2.2 Life Cycle Management

All states and possible actions concerning an ACE's life cycle are summarised in this section.

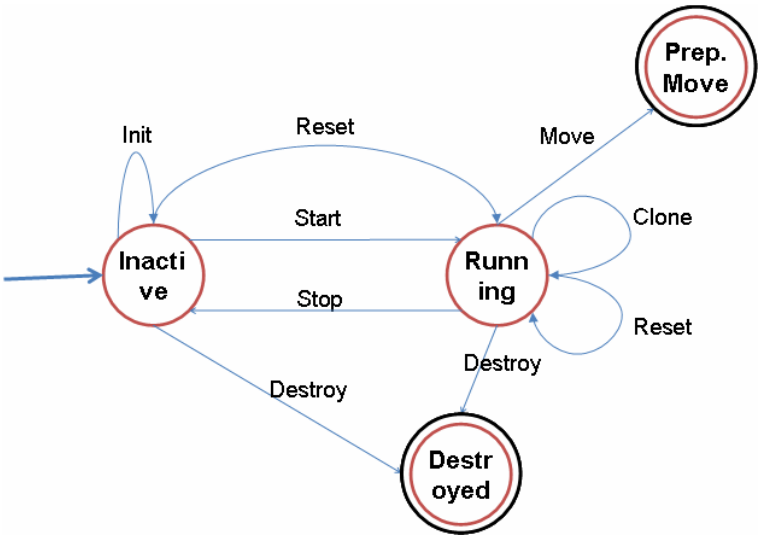


Figure 3: State diagram of the Manager for life cycle action requests

Figure 3 shows a state diagram that represents all possible situations of the Manager in terms of requests of life cycle actions. All other ACE organs must not care about this state diagram. Only the Manager needs to react to life cycle requests in the way the diagram shows.

An ACE can be in four different life cycle states where different life cycle actions are possible. The Manager only reacts to those requests which fit to the state the ACE it belongs to is currently in.

- **Inactive** is the initial state an ACE is in before it starts its operation. This state can also be reached from the Running state if the life cycle action *stop* is requested. In the inactive state the ACE organs, except the Manager, do not perform any actions. They are passive here and only wait for the life cycle actions *init, start* or *destroy* to occur. After the start-up of an ACE, the inactive state is reached automatically. Directly thereafter, the Manager triggers the other organs to initialise themselves by announcing the *init* action. Once this is done, starting the operation and reaching the running state will happen after the Manager sends a life cycle event of the type *start*. This event causes all other ACE organs to start and to perform their normal tasks.

  In order to reach the inactive state, the ACE organs have to stop all their additional threads. To notify cooperating ACEs of the stop of operation, existing contracts have to be cancelled.

- **Running** is the state in which an ACE performs its normal operations. It is reached after the Manager sends a life cycle event of type *start*. Here, the ACE organs have to react to all life cycle events which may be sent by the Manager. These can at the moment be *clone, destroy, move, and stop*. *Clone* and *move* are currently not implemented but planned. Life cycle events of the type *init* and *reset* will not be sent by the Manager when being in running state because *init* is not necessary and *reset* is realised by sending a sequence of *stop* and *start* instead (cf. Figure 4).

  To reach the running state, all organs have to start themselves properly. This may include executing additional threads and leads to normal operation.

- **Prepared to move** is a final state an ACE reaches after a life cycle event of the type *move* is processed. Once this state has been reached, the ACE can never switch to another state. The movement, and therewith the destruction of the ACE instance will follow. The new, moved instance of the affected ACE starts its life cycle in the start state of the life cycle diagram, i.e. inactive. Because *move* is not implemented yet, the definition of this life cycle action may change according to emerging developments.

  When the movement of an ACE is announced, i.e. before the prepared to move state is reached, the organs need to stop all additional threads.

- **Destroyed** is a final state an ACE reaches after a life cycle event of the type *destroy* is processed. The destroyed state represents the end of an ACE's life cycle.

  Preparing to be destroyed includes that the organs stop all additional threads. All contracts should be cancelled and all operations should be stopped.

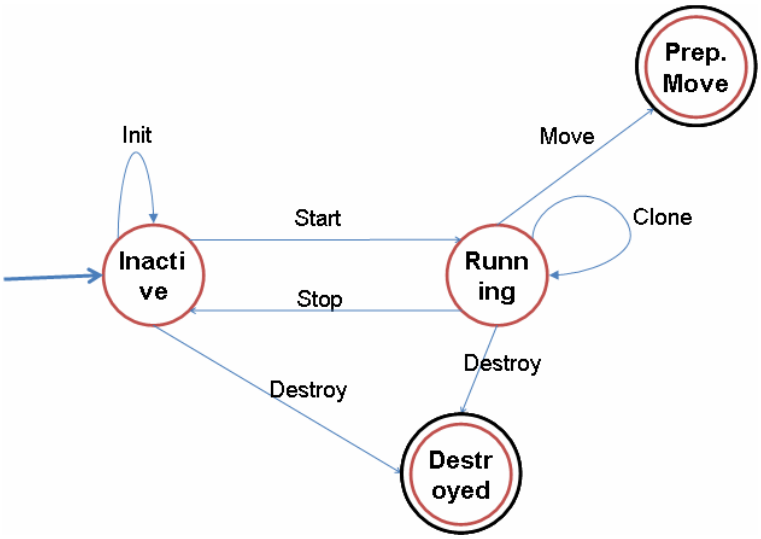


Figure 4: State diagram of all ACE organs for life cycle actions

Figure 4 shows a state diagram that represents all possible situations of all ACE organs in terms of life cycle actions. Organs must not care about requests for life cycle events but they must handle life cycle events, i.e. events of the type `LifeCycleEvent`, as such.

3.2.3 The Life Cycle Protocol

The term ***life cycle protocol*** describes what happens when a life cycle action is requested. In this section we specify what is necessary to cause a life cycle action, which events have to be sent, and how the ACE organs are involved into this process.

In order to cause the ACE to perform a life cycle action, it is necessary to send an appropriate request which specifies the type of life cycle action that should be performed. Such a request can be triggered by an ACE internal organ, the common functionality `request_life_cycle`, or a specific functionality. The Manager organ will take this request, analyse it, and react according to the current life cycle state. If the state the ACE

is currently in forbids performing the requested life cycle action, the request will be ignored. Otherwise the Manager processes it by sending one or more[3] life cycle events which then have to be handled and confirmed by all other ACE organs. Figure 5 shows the work flow when performing a life cycle action.

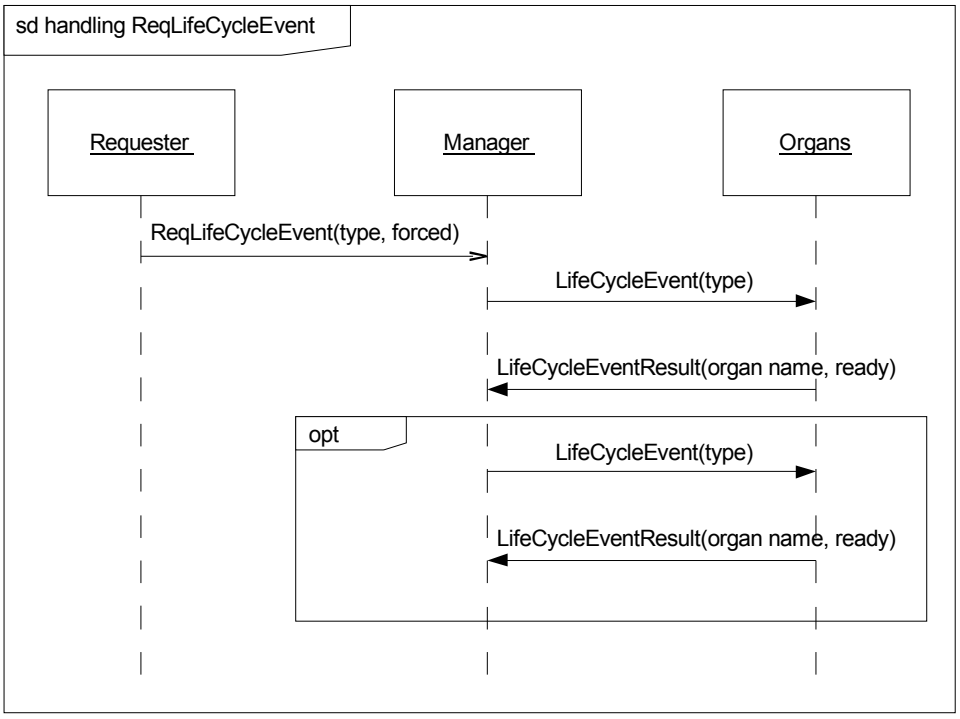


Figure 5: Work flow when performing life cycle actions

3.3 Executor and Plan

The Executor is the organ of the ACE aimed to the execution of Plans elaborated by the Facilitator.

The main role of the Executor is to ensure that any reasoning and decision taken by the Facilitator is put in place in an effective and efficient way ensuring that conditions are verified, actions are executed and proper incoming messages are received.

As far as the developer point of view is concerned the Executor is completely hidden in order to guarantee an efficient and secure execution of the Plan(s). This is achieved mainly by two design principles:

- The encapsulation of any reasoning capability in the Facilitator, the true intelligent core of the ACE.
- The encapsulation of any application logic in the Functionality Repository. The Executor isn't aware of the action semantic. It only ensures that an action is properly executed.

[3] A request for reset will cause the Manager to send a sequence of life cycle events of the type stop and start.

3.3.1 The Plan

The Plan is an ACE internal description of the sequence of actions and states needed to reach a given goal. Each Plan is composed of states and the transitions among states.

In order to run a Plan, the states connecting a starting state and a final state have to be reached and the transitions connecting such states have to be executed.

3.3.1.1 Transitions

Each transition is built by the following elements:

- Source: the state the transition starts from;
- Destination: the state the transition goes to;
- Priority: order by which transitions should be evaluated and executed (1: highest priority, >1: lower priority);
- Trigger: establishes the type of trigger that fires the transition. It can have one of the following values:
  - @auto: the transition runs automatically;
  - @wait or <Event type>: an event has to be awaited in order to run the transition;
- Guard Condition: a boolean condition that has to be true in order to run the transition.
- Action: the Functionality Repository function to invoke in order to run the transition.

By default, all transitions are executed in synchronous manner, so the PlanExecutor (cf. section 3.3.2) is blocked until the action is finished.[4]

3.3.1.2 Plan executions

Given the Plan described above the Executor runs it based on the following rules:

As far as the triggering of a transition:

- When the Executor enters a state, all automatic transitions departing from that state are triggered. (Automatic (@auto) transitions have priority over non-automatic ones.)
- When an input message arrives, it triggers those transitions leaving from the actual state where the condition is either @wait or the condition is a message type which matches the type of the inputMessage.

In order to select the transition for execution:

- If a transition is triggered, its guarding condition is evaluated. When the evaluation is positive (guarding condition is true or there's no guard condition at all), the transition is selected for execution.

[4] This model is convenient and useful in most cases but causes limitations in others.
In the new release (2007 December), we added the possibility to denote certain transitions as "asynchronous" ones. These transitions will be executed in a way that the assigned action is started but the PlanExecutor doesn't wait for it to finish and enters the destination state immediately.

- If more than one transitions are triggered, the Executor iterates on them one by one, trying to find one that is selectable for execution (guarding condition is true). Whenever a transition is selected for execution the remaining triggered transitions get neglected (are not considered any more). If none of the triggered transitions is selected for execution the Executor remains in its original state.

Finally to execute a transition:

- When a transition is executed, its action is invoked and the Executor switches to the destination state of the transition.

3.3.2 Executor Architecture

The Executor consists of two packages:

- `cascadas.ace.executor`: main logic classes
- `cascadas.ace.executor.util`: utility classes (e.g. condition evaluator, action parser).

Main logic classes:

Class name	Purpose
ExecutorJob	Accepts the events from the Bus. Creates a new PlanExecutor for each new Plan (and stops the old one if any). Forwards the input messages to the PlanExecutor. Reacts to LifeCycleEvents.
PlanExecutor (PEX)	Executes a Plan: maintains the active state, selects the transitions for execution, invokes the actions assigned to the transition.
CustomComparator	Parent interface for custom guard condition classes.

Utility classes:

Class name	Purpose
FunctionalityCallComposer	Creates a FunctionalityCallEvent object based on the action, inputMessage, and sessions.
GuardConditionChecker	Evaluates the guard condition.
ParameterValueResolver	Resolves the symbolic parameter name into the actual value.
InputEventWrapper	Provides a uniform interface to the input events. (The uniform interface spares code in the ParameterValueResolver class.)

3.3.2.1 Impulse flow

The Executor receives the impulse from the Manager:

Event from the Bus	Meaning	Effect
AnnouncePlanEvent RemovePlanEvent ReplacePlanEvent	Add, remove or replace a Plan.	A new PlanExecutor is created and started, or stopped, or stopped and replaced by a new PlanExecutor.
GoalNeededEvent GoalAchievableEvent ServiceCallEvent ServiceUsageEvent	Events that might trigger transitions.	The event is forwarded to the actual PlanExecutor.
LifeCycleEvent	Life cycle events (stop, move, etc.).	Life cycle related operations.

As seen in the previous table, in the new release (2007 December) the possibility to execute parallel Plans has been added, so in the ExecutorJob it is possible to have several PlanExecutors (running parallel Plans).

3.3.2.2 Implementation

The ExecutorJob is in charge of

- starting, stopping and replacing Plans/PEXs
- forwarding the input messages to the PEXs.

The input scheme of the PlanExecutor has been changed to a producer-consumer model.

- Each PlanExecutor has an input queue (independent of the input queues of the other PEXs).
- The PlanExecutor reads and processes the contents of its input queue.
- When receiving an input message, the ExecutorJob appends it to the end of all PEX input queues.

When starting a new PEX, its input queue is empty.

When stopping a PlanExecutor, its input queue is lost.

When replacing a Plan, the input queue of the old PEX is lost, and a new empty input queue is created for the new PEX.

3.3.3 Parallel Plans

This is a convenience feature to minimise the number of states per Plan. In case an ACE performs independent activities in parallel, such as gathering context information (plan1) and displaying the results (plan2), it can be described either with two independent parallel Plans or with a single Plan (plan1+2) which is a superset of the individual Plans. In other words, parallel Plans do not increase the descriptive power of the system but make it easier to understand, debug, extend, and analyse the behaviour.

3.3.3.1 Semantics

The semantics of the parallel Plans is the following.

- Each Plan is executed by a PlanExecutor (PEX). Plans and PlanExecutors have a one-to-one relationship.
- PlanExecutors are running in parallel (virtual or real parallelism).
- The incoming message is passed for processing to all PEXs. So, the same inputMessage may trigger state change in all Plans, or in some Plans or in no Plan.
- PlanExecutors do not directly communicate with each other, but they may transfer data through the globalSession.
- Plans can be started, stopped and replaced (when a new Plan is started, a new PlanExecutor is created for the Plan. When stopping a Plan, its PEX is stopped and removed. When replacing a Plan, the PEX of the old Plan is stopped and removed and a new PEX is started for the new Plan. The new Plan inherits the executionSession of the old PEX).

3.3.3.2 Context gathering

With the new Parallel Plan model, context gathering has been moved to the surface: it is a parallel Plan now, automatically generated by the Executor.

The context gathering Plan doesn't initiate context gathering; it only processes the gathered information. So, the acquisition process must be started manually (from another Plan), like in the previous releases.

3.3.3.3 Advantages

The new parallel model has got advantages both on the programming level and on the user level.

Some user level advantages:

- The user can extend the ACE with a new parallel functionality without the risk of a state explosion, and without (significantly) modifying the existing Plans.
- Debugging has become easier: It's easy to see which Plan causes unexpected behaviour.

On the programming level, the producer-consumer pattern for the interaction between ExecutorJob and PlanExecutor makes it possible to execute Plans with different throughput in parallel.

3.4 Facilitator and Self Model

The Facilitator is the ACE organ which provides autonomous behaviour to the ACE. It reacts to the changes in the environment and modifies the ACE characteristics with respect to the rules which are specified in the Self Model.

The ACE Self Model is a set of rules and predefined procedures which describes the behaviour or multiple behaviours of an ACE. The Self Model representation is a well formed XML structure that is provided by the ACE developer. Each ACE has its own Self Model which is loaded at the ACE initiation phase, and is continuously analysed and executed by the Facilitator during the ACE life time.

3.4.1 Facilitator

The Facilitator is one of the key organs in the ACE model. It provides an ACE with the capability to autonomously adapt to changes in the context in which it executes. With respect to these changes, the Facilitator will modify the ACE behaviour while adjusting the existing or adding new ACE capabilities.

The characteristics and behaviours of an ACE are specified by the developer within the Self Model. As presented in the figure below, the Facilitator loads the Self Model at the ACE initiation phase. It then continuously reasons upon changes in the environment and creates new or modifies the existing ACE Plans if required. An ACE Plan contains a set of actions which should be performed and is processed by the Executor (cf. Chapter 3.3).

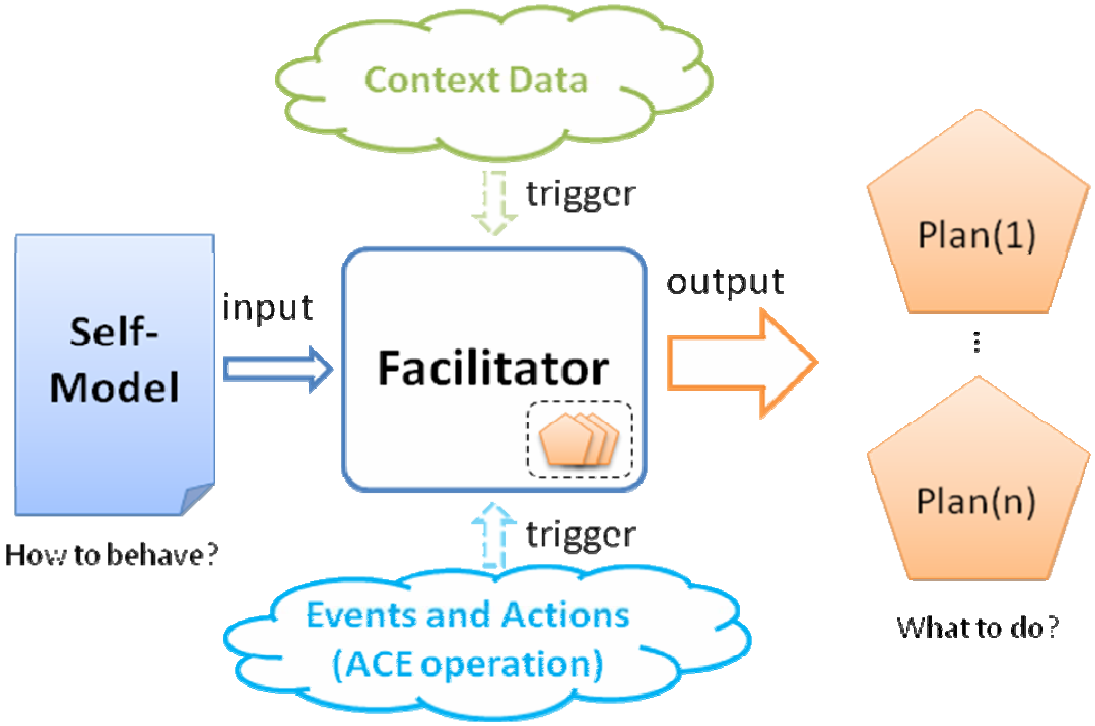


Figure 6: Facilitator general overview

As presented in Figure 6, two types of information are important for the Facilitator: first, the "Context Data" which is any context information that can be gathered from other context provider ACEs and second, the "Events and Actions" which might appear during the ACE runtime. Both of them might cause the creation of a new or termination respectively modification of the existing ACE Plan, as specified in the Self Model.

The *Context Data* is gathered by a common ACE functionality called Context_acquisition_service. In order to invoke this functionality a separate ACE Plan[5] is required. In case the context has changed, the Context_acquisition_service will send a ContextChangedEvent that contains the new context data. The new context values will be extracted and potential modifications of the ACE behaviour (modification of the active Plans) will be performed. More details on this will be provided in the Self Model chapter.

[5] Often called context gathering Plan

The Facilitator listens to the *ACE operation* and modifies its behaviour in case predefined events have been triggered. For example a `CancelContractEvent` which indicates the loss of the connection to another ACE might cause modification of an active ACE Plan that could result in re-establishing the contract to that ACE, or establishing the contract to another ACE which can provide the same goal.

The Facilitator can be used in four different ways:

- Create a new ACE Plan
- Delete an existing ACE Plan
- Modify the ACE Plan(s)
- Request the ACE Self Model
- Submit a new Self Model (replace the current ACE Self Model).

Other ACE organs (e.g. the Executor) could request creation of the new ACE Plan. For this reason they have to create the `CreatePlanRequestEvent` with the ID of the Plan that should be created and send it to the Manager. The Facilitator will create a new ACE Plan for the requested Plan ID. There is a common functionality "start_plan_service" which allows ACE developers to send the `CreatePlanRequestEvent` from the Self Model.

Beside the ACE Self Model, the Facilitator maintains all active Plans. Only the copies of the Plans are submitted to the Executor for processing. In order to stop the Plan execution and delete a Plan from the list of active Plans a `RemovePlanEvent`, containing the ID of the Plan that should be stopped and removed, has to be sent to the Manager. The common functionality "stop_plan_service" allows a developer to send a `RemovePlanEvent` directly from the Self Model.

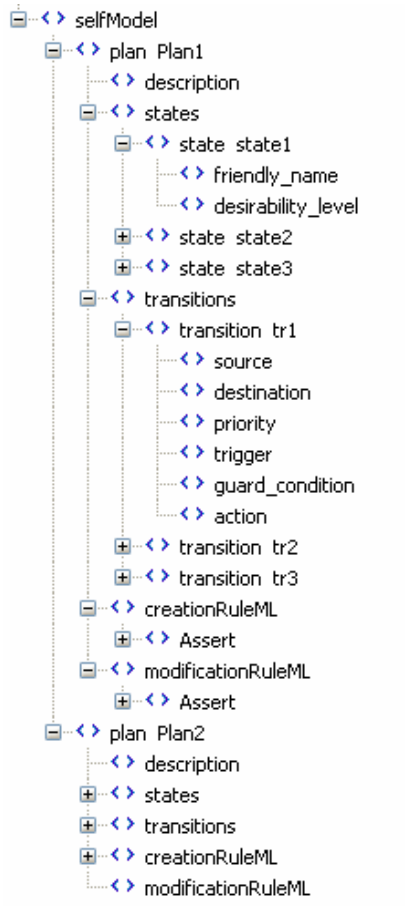
As already described above, the Plan modification can be triggered either by changes to context data (using `ContextChangedEvent`) or by predefined events and actions.

The Facilitator can be requested to deliver the ACE Self Model. For this purpose, another ACE (e.g. Supervision System) has to send the request for the Self Model using `ReqSelfModelEvent`. After receiving this event, the Facilitator answers with a `SelfModelEvent` and will send the ACE Self Model to the requesting party.

In the same way the Self Model can be replaced, after some modification for example. While receiving the `AnnounceSelfModelEvent` which contains the new Self Model, the Facilitator will replace the current Self Model with the new one. The currently used ACE Plan will be recreated according to the new ACE Self Model.

3.4.2 Self Model

The Self Model can be seen as "the brain" of the ACE. All possible ACE behaviours and capabilities as well as the actions to be performed when certain events arise, are specified here. The Self Model is defined in the form of an XML structure and has to be created by the ACE developer. A new approach for Self Model creation, the ACELandic scripting language, is currently under development (cf. [ACEL]).



The ACE Self Model structure is specified in the selfmodel.dtd file and comprises the following rules:

- Every ACE has only one Self Model
- An ACE Self Model contains at least one ACE Plan
- Each ACE Plan performs certain actions and fulfils a certain task
- The default ACE Plan must be specified
- The ACE Plan definition contains
  - Full set of all possible states
  - Full set of all possible transitions
  - Rules for creating the ACE Plan
  - Rules for modifying the ACE Plan
- All ACE Plans must be uniquely identifiable via Plan ID
- All states and transitions that are used within a ACE Plan must be uniquely identifiable via IDs
- Multiple ACE Plans can be started in parallel
- Rules for creating and modifying ACE Plans use standard RuleML syntax

Figure 7: ACE Self Model structure

The Self Model is loaded by the Facilitator at the ACE initialisation phase. The default Plan is the starting point in the ACE execution process. It is created and submitted to the Executor on ACE start-up. Additional Plans can be started from here.

The ACE Plan that is generated from the Self Model is a state machine. It consists of states and transitions which define under which circumstances an action[6] should be performed.

3.4.2.1 States and transitions

As presented in the figure above, a Self Model can contain multiple Plans which must be uniquely identifiable by Plan IDs. Each ACE Plan in the Self Model contains the full set of all possible states and transitions that might appear in the Plan and the rules for their creation and modification.

Each **State** is defined by its *friendly name*, *unique ID* and the *desirability level*. The unique ID must be Plan wide unique. The state's desirability level is an integer value in range 0 to 10. It has to be defined by the developer and is used by the Supervision System. It indicates whether an ACE works confidently or not. If the ACE reaches a state with low desirability level that means it is not operating properly and it requires help from the Supervision System.

[6] Action means invoking a service

```
<state id="state1">
  <friendly_name>Initial State</friendly_name>
  <desirability_level>10</desirability_level>
</state>
```

Each **Transition** is defined by its *unique ID, source, destination, priority, trigger, guard_condition* and *action*. Like for the state, the transition's unique ID must be Plan wide unique. All states and transitions must be uniquely identifiable within the Plan.

Source and destination contain the state IDs of the two states which will be connected by the transition if it gets created. The source state is the state from where the transition can be executed and the destination state is the state where the Executor will move to after performing the transition.

Transitions can be prioritised by setting the `<priority>` parameter. In case a state contains multiple transitions that lead from it, the developer can specify the order in which they will be checked. This mechanism allows developers to execute the transition B rather than the transition A in case both of them satisfy the guard condition. The priority of a transition is defined as a positive integer value and is not mandatory. In case the priority flag has not been specified, it will be set to 0 by default.

Transition prioritisation is specified as

- n (lowest value greater than 0) highest priority transition
- priority > n lower priority transitions
- empty no priority specified (0 by default).

Transitions will be sorted that way that the transition with "priority = n" (n is a positive integer number) will be examined first, the transitions with "priority > n" will be sequentially examined after the first one, and the transitions with non specified "empty priority" are examined at the very end.

The *Trigger* parameter specifies which event has to occur before the related transition should be examined. A developer is allowed to specify the following trigger values:

- `<trigger>@auto</trigger>` (Examine the transition immediately.)
- `<trigger>fully qualified event type</trigger>` (Examine the transition only if particular event type arrives.)

If the trigger contains `@auto`, the Executor will examine the `guard_condition` immediately. If the trigger contains "`fully qualified event type`", then it will be executed only if such an event has arrived. For example a transition that contains `<trigger>cascadas.ace.event.ServiceResponseEvent</trigger>` will be examined only if a `ServiceResponseEvent` has arrived.

Guard Condition `<guard_condition>` specifies the condition under which the transition should be executed. The Executor will run the transition only if the guard condition is satisfied. The developer can specify a guard condition using logical comparators and operators.

The following comparators are supported:

- LT(): less-than

- GT(): greater-than
- EQUALS(): equality check
- OWN_COMP(): own comparator class defined by the developer

The following logical operators are supported:

- AND: conjunction
- OR: disjunction
- NOT: negation

The following data types can be used in the conditions:

- Integer (primitive type)
- Double (primitive type)
- String
- NULL / null (comparison against null object)

Custom comparators (cf. OWN_COMP) must implement the `ConditionChecker` interface. For details please check the javadoc. Custom comparators must be available in the classpath, and must have a default (no-arg) constructor.

The following grammar defines the guarding-condition-language "GCL".

```
CCL = {Expression}
Expression = OPERATOR(Param) | OPERATOR(Param,Param)
Param = Expression | Constant | Reference
Constant = Integer | Double | String
Reference = <reference to the inputMessage, contract & sessions, the same as in
the action>
```

The guard condition comparators are defined as follows:

LT(p1, p2)

- True if (p1 < p2)
- False if (p1 >= p2) or if an error occurs during the comparison (e.g. p1 or p2 is not a number)
- p1, p2: integer, double

GT(p1, p2)

- True if (p1 > p2)
- False if (p1 <= p2) or if an error occurs during the comparison (e.g. p1 or p2 is not a number)
- p1, p2: integer, double

EQUALS(p1, p2)

- True if (p1 == p2)
- False if (p1 != p2) or if an error occurs during the comparison
- p1, p2: integer, double, String, NULL

OWN_COMP(p1)

- True if the custom comparator returns true
- False if the custom comparator returns false or an error occurs during the comparison
- p1: class name (String)

AND(p1, p2)

- True if both p1 and p2 are true
- False otherwise or if an error occurs
- p1, p2: boolean

OR(p1, p2)

- True if p1 or p2 is true
- False otherwise or if an error occurs
- p1, p2: boolean

NOT(p1)

- True if p1 is false
- False otherwise or if an error occurs during the comparison
- p1: boolean

Here are a few examples how to specify guard conditions. Please look at chapter 3.4.2.3 for more details regarding the Self Model syntax.

```
<guard_condition>LT(?inputMessage://age,45)</guard_condition>
```

Runs the transition if the value in the "age" field of the inputMessage is less than 45.

```
<guard_condition>AND(                                    LT(?inputMessage://age,22),
EQUALS(?globalSession://mode,mode17))</guard_condition>
```

The condition is true if the value in the "age" field of the inputMessage is less than 22, and the "mode" variable of the globalSession is equal to "mode17".

```
<guard_condition>NOT(EQUALS(?inputMessage://dress_code,null))</guard_condition>
```

The condition is true if the "dress_code" field of the inputMessage is not null.

Action `<action>` specifies the functionality to be called when a transition is executed. The action contains the ID of the functionality to be called and the input parameters which have to be passed to it. This information must be in sync with functionality description files as described in chapter 3.5.2.

Functionalities might be called with or without parameters. In case a functionality is called without parameters it is sufficient to specify only the functionality ID `<action>functionality_id</action>` whereas if the functionality has input parameters, all parameters with their values have to be listed additionally within the brackets:
`<action>functionality_id(param1=value1,param2=value2)</action>`
The input parameters are initialised with "=" and are comma separated.

Parameter values can be either constants or variables. Constants are set directly in the Self Model and by default are parsed like strings (see chapter 3.4.2.3) whereas variables are gathered from the input message or session objects.

```
<action>wait_service(millis=2000)</action>
```

The functionality with the ID "wait_service" will be called with the `millis` input parameter set to 2000 (a string corresponding to the "java.lang.String" parameter of the black box description of the common functionality). The parameter is initialised by a constant value which is set directly from the Self Model.

```
<action>occassion_display_service(occassion=?inputMessage://dress_code)</action>
```

The functionality occasion_display_service will be called with the parameter `occasion` set to the current value of the `dress_code` parameter from the input message.

```
<action>show_sre_service(sre=?inputMessage)</action>
```

The functionality show_sre_service will be called with the parameter `sre` set to the input message. For exact details on how to pass the entire input message to the functionality please look at chapter 3.4.2.4.

```
<action>my_init_service</action>
```

The functionality my_init_service will be called without passing any parameters to it. This must be in sync with the functionality description file.

```
<action>generic_service_caller_service(contractId=?executionSession://c1,serviceNa
me=random_provider_service,targetRole=provider,r1=0,r2=10)</action>
```

The common functionality generic_service_caller_service will be called with 5 parameters:

- 1 referenced value: ?executionSession://c1, which is of type cascadas.ace.session.Contract;
- 4 string constants: "random_provider_service", "provider", "0" and "10".

In particular, the last two parameters are the input for the functionality that will implement the service named "random_provider_service". If the developer wishes to invoke a service passing constants that are of integer type, he shall write a custom service_caller_service.

The following grammar defines the transition-action-language "TAL".

```
TAL = {Expression}
Expression = FUNCTIONALITY_ID | FUNCTIONALITY_ID(Param1=Value1,Param2=Value2,...)
Param = {input parameters as specified within the functionality description file}
Value = constants | variables {primitive data types| any Java Object}
```

While performing a transition, the Executor will move to the destination state only after the action is accomplished successfully. In some cases the action might require a quite long time for processing a task, but it would be advantageous to continue with the Plan execution even though the task has not been accomplished. In order to specify an asynchronous functionality call, developers have to add the asynchronous="true" flag to the transition.

```
<transition id="tr1" asynchronous="true">
  <source>state1</source>
  <destination>state2</destination>
  ...
  <action>my_action</action>
```

```
</transition>
```

In the example above `my_action` will be called, but the Plan execution will continue and the Executor will move to state2 even though the `my_action` has not been accomplished now. `Asynchronous` is an optional parameter and is set to false by default.

3.4.2.2 Plan creation and modification rules

Beside the full set of states and transitions, each Plan must specify the Plan creation and modification rules. They are two separate sets of predefined facts and rules written in RuleML language.

RuleML "is a Rule Mark-up Language (RuleML), permitting both forward (bottom-up) and backward (top-down) rules in XML for deduction, rewriting, and further inferential-transformational tasks." (cf. [RuleML]).

The basic implementation of the RuleML engine plugged into the Facilitator is the OOjDREW 9.2 engine. *"OOjDREW is a deductive reasoning engine for the RuleML (including its OO extensions), written in the Java programming language."* (cf. [OOJD]).

The Plan creation rules will be executed only at the ACE Plan creation phase. They define how an ACE Plan should be created. As soon as the ACE Plan has been created successfully, the Plan modification rules become active and are continuously evaluated by the Facilitator during the Plan execution time.

Plan creation rules are specified within the `<creationRuleML>` and Plan modification rules within the `<modificationRuleML>` element. Rules are defined in standard RuleML language. RuleML 0.88 stripped syntax is used and predefined keywords which will trigger different actions are specified.

RuleML 0.88 stripped syntax is a lightweight version of the RuleML 0.88 syntax and is processed by the Facilitator pretty fast. For the Toolkit purposes, the following predefined keywords are specified:

Keyword	Description	Creation	Modification
`initState`	Defines the initial state of the Plan. Initial state is the starting point of the Plan execution.	X	X
`createState`	Indicates that a new state should be created.	X	X
`deleteState`	Indicates that an existing state should be deleted. All transitions that lead from that state will be deleted as well.		X
`createTransition`	Indicates that a new transition should be created. Please note that before a new transition can be created, the source and destination states must be available.	X	X
`deleteTransition`	Indicates that an existing transition should be deleted.		X

`startPlan`	Indicates that an additional Plan should be started.		X
`stopPlan`	Indicates that an active Plan should be stopped.		X

Table 1: Predefined keywords for Plan creation and modification rules

RuleML consists of rules and facts. Rules define conditions under which certain assertions are true, and facts define statements that are true. Developers are allowed to use statements defined in the table above in both, facts and rules. The difference is that the statements defined as facts will be always applied, whereas the statements defined within the rules will be applied only if the predefined conditions are satisfied.

RuleML 0.88 starts always with "Assert" that contains the "And" child. Facts are specified within the "Atom" elements and rules within "Implies" elements. This is briefly depicted in Figure 8. For more information on RuleML 0.88 standard please look at [RuleML].

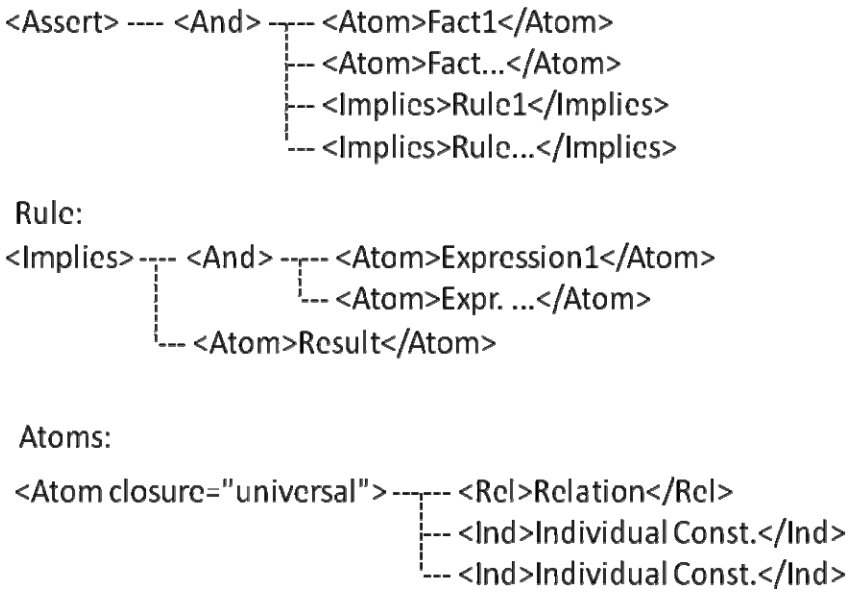


Figure 8: Basic RuleML structure

When writing the RuleML you have to consider that you must use the keywords that are listed in Table 1 in order to apply any actions to the Plan creation or modification. Any variable names etc. can be used for rules and facts, but only the ones with the predefined keywords as specified in the table above will be applied for creating or modifying the Plan.

As already mentioned, RuleML specifies Rules and Facts. Plan actions as specified in the table above can be applied within both: Rules and Facts. For example, if a transition is always a part of the Plan, then it makes more sense to specify its creation within the facts instead of specifying it within the rules. In the following example, the Plan creation RuleML consists only of facts.

```
<creationRuleML>
  <Assert>
    <And>
      <Atom closure="universal">
```

```
        <Rel>createState</Rel>
        <Ind>state1</Ind>
        <Ind>state2</Ind>
      </Atom>
      <Atom closure="universal">
        <Rel>initState</Rel>
        <Ind>state1</Ind>
      </Atom>
      <Atom closure="universal">
        <Rel>createTransition</Rel>
        <Ind>tr1</Ind>
        <Ind>tr2</Ind>
      </Atom>
    </And>
  </Assert>
</creationRuleML>
```

Figure 9: Sample Plan creation RuleML

While executing this RuleML, the Facilitator will create a new ACE Plan which contains two states (state1 and state2) and two transitions (tr1 and tr2). The initial state of the Plan will be set to state1. Please note that each Plan must have one initial state.

The following example specifies the Plan modification RuleML. It consists only of one rule (user is at home AND weather is rainy then create transition tr3). No facts have been specified. The facts will be automatically applied by the Facilitator as the context data values change during the ACE runtime. In the example presented in Figure 10, transition tr3 will be created as soon as user_location changes to "at_home" and weather is "rainy".

```
<modificationRuleML>
  <Assert>
    <And>
      <Implies closure="universal">
        <And>
          <Atom closure="universal">
            <Rel>?contextData://user_location</Rel>
            <Ind>at_home</Ind>
          </Atom>
          <Atom closure="universal">
            <Rel>?contextData://weather</Rel>
            <Ind>rainy</Ind>
          </Atom>
        </And>
        <Atom closure="universal">
          <Rel>createTransition</Rel>
          <Ind>tr3</Ind>
        </Atom>
      </Implies>
    </And>
  </Assert>
</modificationRuleML>
```

Figure 10: Sample Plan modification RuleML

The rules can depend on two types of data:

- Changes to context data, using "**?contextData://**context_name"
- ACE execution process, using "**?event**"

How to modify an ACE Plan with regard to context changes?

In order to modify an ACE Plan with regard to context changes, you have to ensure two things: First, you have to apply "**?contextData://**context_name" syntax in your Plan creation or modification rules. Second, you have to create a separate Plan that will gather the context data. This Plan must be executed at the beginning, before the context data are used. Context data are periodically gathered using the common functionality Context_acquisition_service. The Context_acquisition_service will periodically poll the context information in predefined intervals, and will store the information in the global session under the specified name. For example:

```
Context_acquisition_service(query_interval=1000,
service_name=?inputMessage://serviceName,variable_local_name=user_location,
contract=?executionSession://userLocationProviderContract)
```

The Context_acquisition_service will query a User Location Provider ACE which is a context provider ACE in 1000 ms intervals and will save the values as context data in the global session under "user_location". If context data change, the Facilitator gets notified about the change and gets the new value. It will apply it to the rules and perform actions if required.

Please note that the local_name parameter which is used for the Context_acquisition_service must be the same as the context name used in the RuleML **?contextData://**context_name. Otherwise the facts which will be applied by the Facilitator won't affect the rules.

Please note as well that context data might be also used by the specific functionality of the ACE. Therefore, the ACE might use the Context_acquisition_service in order to gather more context data than specified in the RuleML but which might be used by the application. These additional context data will not affect the creation or modification rules if they are not used there.

How to modify an ACE Plan with regard to the ACE execution process?

The ACE Execution process is characterised by sending events of a certain type. Events are either sent by the ACE itself during the execution process (e.g. CancelContractEvent which indicates cancelling of a contract with a certain ACE) or from a common or specific functionality.

In order to modify an ACE Plan with regard to the ACE execution process, the ?event syntax has to be applied in the RuleML rules. For example:

```
<modificationRuleML>
  <Assert>
    <And>
      <Implies closure="universal">
        <And>
          <Atom closure="universal">
            <Rel>?event</Rel>
            <Ind>cascadas.app.MyEvent</Ind>
          </Atom>
        </And>
        <Atom closure="universal">
          <Rel>createTransition</Rel>
          <Ind>tr3</Ind>
        </Atom>
```

```
      </Implies>
    </And>
  </Assert>
</modificationRuleML>
```

The rules above specify that the transition tr3 should be created if "cascadas.app.MyEvent" has arrived at the Manager. The following RuleML syntax has to be used in order to invoke the actions that are specified in Table 1:

Rules:

```
Relation = ?contextData://context_variable | ?event
IndConstant = {context value} | {event type}
```

Facts and Rule Results

```
Relation = createState | deleteState | initState | createTransition |
           deleteTransition | startPlan | stopPlan
IndConstant = {state Id} | {transition Id} | {plan Id}
```

Note: The Facilitator uses OOJDREW 0.92 Bottom-Up Reasoning Engine [OOJD]. In order to test and verify the Plan creation or modification RuleML developers can use the Java Web Application available on the website (www.jdrew.org/oojdrew/demo/bottomup92.jnlp).

3.4.2.3 Self Model Syntax

This part describes constants and variables that can be used within the Self Model. They are either used as input parameter for the service call within the `<action>` part or are used within the creation or modification RuleML.

Constants

Developers can specify either variables or constants within the Self Model. Constant values are defined as constant strings. For example, `ace_name=ACE1` defines a constant value "ACE1" that is defined within the Self Model and will be used directly.

An example of using constants as input parameters to a functionality call within the transition:

```
<action>start_plan_service(planId=Plan2)</action>
<action>wait_service(millis=2000)</action>
```

Variables

Variables on the other hand are specified using "?". An ACE developer is allowed to use the following predefined variables within the Self Model:

- `?inputMessage` Input message
- `?executionSession` Plan execution session
- `?globalSession` ACE global session
- `?contextData` Context parameters (are continuously updated)

The variables specified within the Self Model can only be read. For example, in order to access parameters from the execution session `executionSession://parameterName`, the parameters must be first made available within the session object either using a common functionality like generic_add_to_execution_session_service or while writing them within the specific functionality `executionSession.put(parameterName, parameterValue);`.

In order to access parameters from the variables specified above, ACE developers have to use the following syntax: "`?storageName://parameterName`" where `?storageName` can be one of the variables listed above and `parameterName` the name of the parameter that is available in the parameters list.

`?inputMessage` represents the incoming message that has been received by the ACE. An input message can be received upon a service call to another ACE on behalf of actions in previous transitions.

```
<transition id="tr1">
  <source>state1</source>
  <destination>state2</destination>
  <priority>1</priority>
  <trigger>@auto</trigger>
  <guard_condition></guard_condition>
  <action>gn_sender_service(goalName=age,myAddress=?globalSession://aceAddress)
  </action>
</transition>
<transition id="tr2">
  <source>state2</source>
  <destination>state3</destination>
  <priority>1</priority>
  <trigger>cascadas.ace.event.GoalAchievableEvent</trigger>
  <guard_condition></guard_condition>
  <action>contract_establishment_service(user=?globalSession://aceAddress,
provider=?inputMessage://providerAddress,contractId=ageContract)</action>
</transition>
```

Explanation: While executing transition tr1, the `gn_sender_service` will send a GN message searching for ACEs which can provide "age". The Executor will then move to state3 and will wait for the first positive answer (`GoalAchievableEvent`). The `providerAddress` is carried by the `GoalAchievableEvent` and can be accessed using `?inputMessage://providerAddress`.

`?inputMessage` corresponds always to the external / incoming message, whose type is specified within `<trigger>`. Developers can access any parameter that is available within the input message.

`?executionSession` represents the Plan Execution Session. ACE developers are allowed to access all parameters which are made available within the execution session. Variables have to be stored to the execution session first, in order to enable access to their values within the Self Model. Parameters can be written to the execution session within the specific functionality or using the common functionality generic_add_to_execution_session_service.

One can write a parameter into the execution session in the following way:

`executionSession.put("parameterName", value)`

The value can be used within the Self Model using:

`?executionSession://parameterName`

`?globalSession` represents the Global Session of the ACE. Similar to the execution session, all parameters that are available within the global session can be accessed in the Self Model using: `?globalSession://parameterName`.

Please note that the execution session and the global session with all their parameters can be accessed directly within the specific functionality as well. In order to make your specific

functionality session aware (i.e. allowing direct access to the session objects within the specific functionality), it must implement the `SessionAware` interface.

`?contextData` represents context information [7] that is available within the ACE. Developers can specify context data as the input parameters to services or to the ACE creation and modification rules. Before the context parameters can be read from `?contextData`, the `context_acquisition_service()` has to be called first.

The context values are stored within the global session from where they can be accessed. Within the Self Model, ACE developers can access the context data using `?contextData://paramName`

How to call the context acquisition service?

```
<transition id="tr3">
     <source>state3</source>
     <destination>state4</destination>
     <priority>1</priority>
     <trigger>@auto</trigger>
     <guard_condition></guard_condition>
     <action>Context_acquisition_service(query_interval=2000,
               service_name=?inputMessage://serviceName,
               variable_local_name=age,
               contract=?executionSession://ageContract)
     </action>
</transition>
```

In the example above, while running transition tr3, the Executor will call `Context_acquisition_service` which will then periodically (every 2 sec.) request the context date "age" form the context provider ACE which has been contracted before.

`Context_acquisition_service` requires four parameters:

- `query_interval` (Context pulling interval)
- `service_name` (Service name, extracted from the GA message.)
- `variable_local_name` (Name of the context variable where to store the value. It should be unique.)
- `contract` (The contract established upon GN-GA.)

The context values can be read within the Self Model using `?contextData://variable_local_name.` They can be applied to both, functionality input parameters and the RuleML for specifying certain rules.

3.4.2.4 Self Model how to

How to start multiple ACE Plans in parallel?

There are two possible ways to start an additional ACE Plan in parallel. The first solution is to specify "`startPlan`" with the Plan ID of the additional Plan to be started within the Plan modification rules. In the modification rules you have to ensure that the Plan is created only once. The second possibility is to start a parallel Plan from another ACE Plan using the common functionality `start_plan_service`.

[7] The Context is implemented as Map<String,Object> in the globalSession. It is also available for those specific functionalities that are session aware.

How to stop an ACE Plan?

Same as for starting an additional ACE Plan, you can stop an already executing ACE Plan from the Plan modification rules using the common functionality `stop_plan_service`. An ACE Plan can also self terminate while performing the transition that executes `stop_plan_service` and pass its own Plan ID to it.

How to pass the entire input message to the functionality?

Developers can pass the entire input message (e.g. `ServiceCallEvent`) as the input parameter to the specific functionality. In order to do this, the functionality must implement a method that accepts the input message as the input parameter. In the Self Model you simply specify `?inputMessage` as the input parameter, and the entire input message will be passed to the called functionality. This might be very useful for debugging purposes.

For example:

```
<transition id="tr4">
  <source>state4</source>
  <destination>state5</destination>
  <priority>1</priority>
  <trigger>cascadas.ace.event.ServiceCallEvent</trigger>
  <guard_condition></guard_condition>
  <action>my_service(msg=?inputMessage)</action>
</transition>
```

The transition will call the `my_service` functionality and will pass the entire `ServiceCallEvent` as the input parameter to it.

3.5 Functionality Repository

The Functionality Repository (or just Repository) is an ACE organ responsible for storing and invoking the specific functionalities of the ACE instance, and for storing and invoking of the common functionalities.

3.5.1 Basic Concept of the Functionality Repository

3.5.1.1 Purpose

The purpose of the Functionality Repository is to enable specific functionalities to *get deployed* into the ACE instance and *get accessed* on request. That's why the Functionality Repository shows a two-faced behaviour:

- It is a storage facility. It keeps track of the deployed functionalities, creates and stores instances of the underlying classes, and maintains call-related variables.
- It provides an invocation interface, handles invocation requests (coming from the Executor) and interprets them as calls to the functionalities.

The primary source of the invocation requests is the Executor. The Executor translates Plan actions into FunctionalityCallEvents, and sends them to the Repository[8].

[8] In the new release (2007 December) we introduced a direct communication channel between the Executor and the repository, instead of the Manager-based channel. This decision has advantages on both the user and the technical level, without limiting the supervisablility of the ACE.

- The Functionality Repository receives the `FunctionalityCallEvents` (FCEs) and maps them into method calls on the deployed functionalities.
- The output of a call is a set of events. The Functionality Repository sends these events to the Manager or to the Gateway (depending on the event type: internal or external).

Besides that, the Functionality Repository also follows the same Event based access model as the other ACE organs do. So it is also ready to receive `FunctionalityCallEvents` from the Manager (mainly for backward compatibility reasons). The handling of an event arriving from the Manager does not differ from the handling of an event arriving from the Executor directly.

3.5.1.2 Connection with other ACE organs and external ACEs

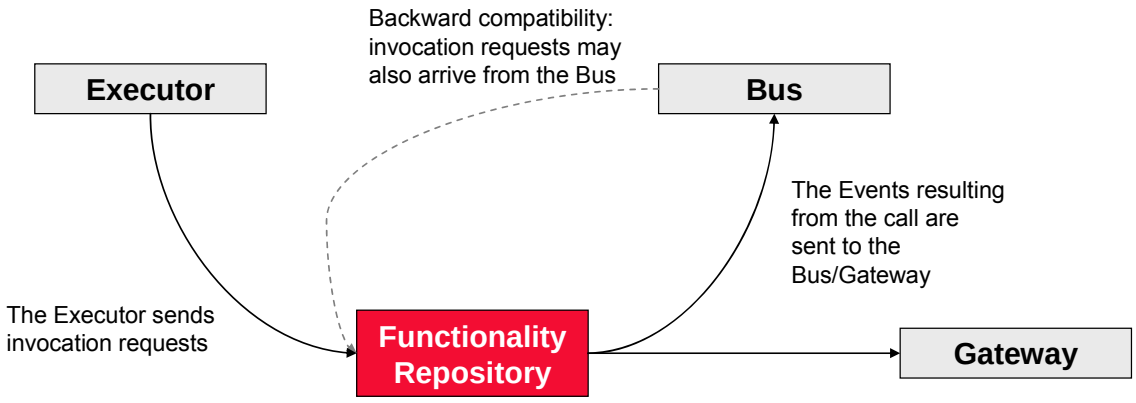


Figure 11: Connections with other organs

The Functionality repository is in connection with 3 ACE organs:

- The Executor is the primary source of the invocation request. Requests are arriving on a direct communication channel (with direct method call) in form of `FunctionalityCallEvents`.
- The Manager (formerly called Bus) is the destination of those result events of the call that result from the call and are ACE-local. For backward compatibility, the Manager is a secondary input source of the `FunctionalityCallEvents`.
- The Gateway is the destination of those result events of the call that are addressed to other ACEs (external, inter-ACE).

3.5.1.3 Design concepts

The main design principle of the Functionality Repository is to support the transformation of existing libraries into the ACE functionalities in a seamless way.

- The porting of existing libraries into an ACE functionality should be possible without the modification of the source code.
- In normal cases, the porting process should not be more complex than providing an XML descriptor about the functionality.
- On the other hand, the creator of the functionality should have complete freedom in customising the process. Custom loader, mapper etc. classes may be used.

Through implementing standard interfaces, the creator gets access to specifically ACE-related resources (e.g. sessions, logs).

The Functionality Repository should be mobile in the meaning of moving from one computer to another.

Based on the fact that the time needed for a functionality call may be long, the Repository should be able to perform the call in a separate thread, in order to prevent the blocking of the ACE.

3.5.1.4 Terms

To avoid the misunderstandings, we differentiate between service and functionality.

- **Service**: the external view of the functionality. Other ACEs see the functionality as a Service, and access it with `ServiceUsageEvents` (or more precisely, with its subclasses, e.g. `ServiceCallEvent` and `ServiceReturnEvent`).
- **Functionality**: the internal view of the functionality. The Plan refers to functionality names (IDs), and the Executor creates `FunctionalityCallEvents` to access them. The `FunctionalityCallEvent` usually contains a wrapped `ServiceUsageEvent`.

3.5.2 The Functionality Model

The functionality model used by the Functionality Repository consists of four parts:

- Unique ID of the functionality (name)
- Event model: the input and output events of the invocation. The input event is always a `FunctionalityCallEvent`. The output events are the events generated by the call.
- Black-box model: input and output parameters (parameter names and types).
- The underlying call sequence. Technical description: class and method names, and their parameter list.

The functionality descriptor is provided in form of an XML file that is placed into a previously specified directory. A separate XML file is created for each functionality.

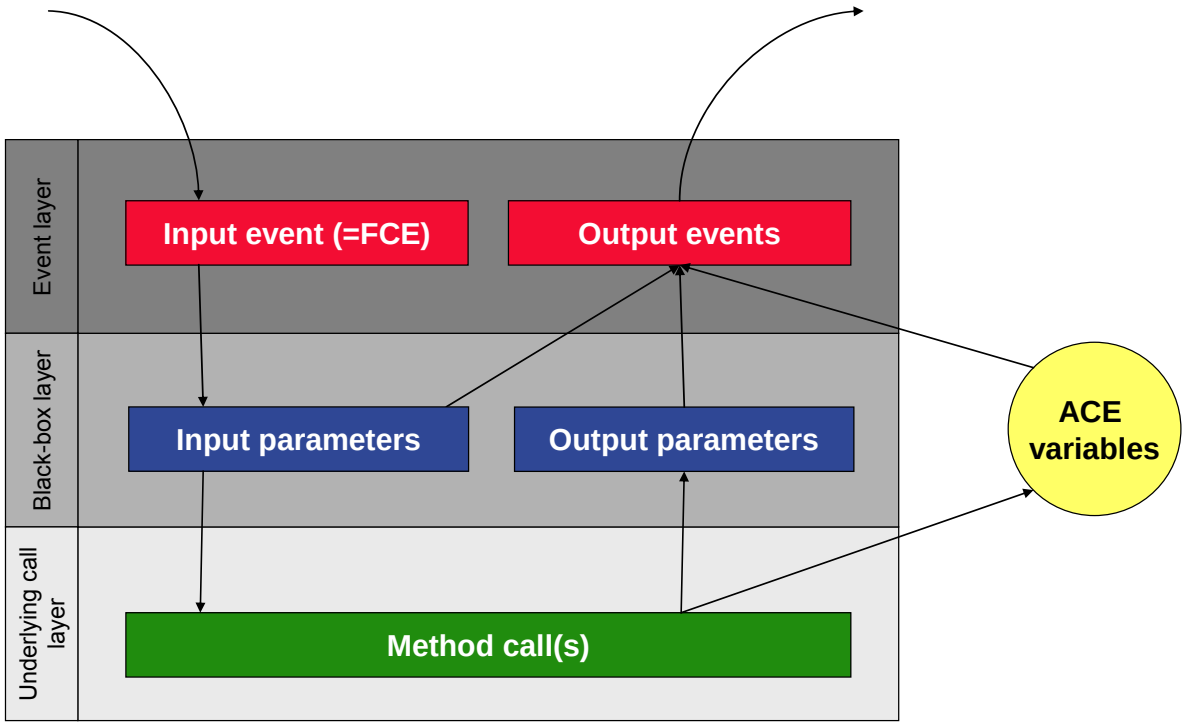


Figure 12: Functionality model

There is a major difference between the classical functionality models and the model we are using in the Toolkit: we abstract the functionality as a set of input and output events. While in classical models, functionalities are abstracted as input and output values. We are using events to describe the functionality. This can also be considered as an additional abstraction layer above the classical model (event layer on top of the black-box layer).

Referring to Figure 12, the two bottom layers (black-box and call layers) describe a classical functionality model. The innovative part is the uppermost layer (the event one). As most readers may be more familiar with the classical models, we give detailed description about the bottom layers before going to the event level.

3.5.2.1 Functionality name

The functionality name must be a string and must be unique. There are no other restrictions, it may consists of a single word or of more than one words, and may contain special characters.

The recommendation is to use lowercase letters, delimit the words by a "_" sign, and to finish the service name with "_service". For example, the followings are well-formed service names:

```
call_counter_service
simple_phonebook_lookup_service
```

3.5.2.2 Black box model

The black box model of the functionality describes the input and output of the underlying call.

Input and output parameters are both optional, as the Functionality Repository supports both no-arg methods and void return values. That's why the following description is also valid.

```
  <black-box-description>
  </black-box-description>
```

The input and output parameters are described with name and type. Type is specified as a string which can be either the fully qualified class name or the name of the primitive type. Own class types are supported as well.

```
  <black-box-description>
    <input>
      <param name="name" type="java.lang.String"/>     // standard class
      <param name="apple" type="mypackage.Apple"/>     // own class
      <param name="i" type="int"/>                     // primitive type
    </input>
    <output name="line" type="java.lang.String"/>
  </black-box-description>
```

3.5.2.3 Underlying call sequence

The underlying call sequence can be a simple method call or a complex series of calls.

In case of a simple call, all the black-box defined input parameters are passed to the method of the described class.

```
<simple-call-details
  class-name="testfunctionality.sessioncontextexample.CallCounter"
  method-name="add"/>
```

In case of a complex call sequence, the input and output parameters of each call are described one by one.

```
  <complex-call-details>
    <call class-name="testfunctionality.MyPhoneBook"
    method-name="loadPhoneBook" >
      <arg ref="phone book file"/>
    </call>
    <call class-name="testfunctionality.MyPhoneBook"
    method-name="lookupPhoneNumber">
      <arg ref="surname"/>
      <arg ref="first name"/>
      <return ref="phone number"/>
    </call>
    <call class-name="testfunctionality.MyCache" method-name="put">
      <return ref="phone number"/>
    </call>
  </complex-call-details>
```

Input and output arguments are optional, just like in the black-box description, to support void and no-arg methods. The referred argument names are either coming from the black-box model or from an output value of a predecessor call.

Functionalities that are especially designed for ACEs may request access to the ACE variables (sessions, logs, ...). Some ACE variables can be used as a permanent storage for data.

3.5.2.4 Event layer

The Event layer describes (the input and) the output events of the call. The input event is a `FunctionalityCallEvent` by definition, so this part is omitted from the descriptor.

The output events can be described in two ways: as an event mapping or as an event mapper. The description also refers to the technical details.

An event mapping simply describes which event type to create and how to fill it with contents. The content may come from the black-box model (e.g. output value of a call) or from an ACE variable.

The following mapping means that two output events will be created, both are `ServiceResponseEvents`, the first will have the "apple" as parameter, and the second the "surname", "first name" and the "phone number". The referred parameter names must be available in from the black box model or from the output of a call.

```
<output-event-mappings>
  <mapping event="cascadas.ace.event.ServiceResponseEvent" role="user">
    <value ref="apple"/>
  </mapping>
  <mapping event="cascadas.ace.event.ServiceResponseEvent" role="user">
    <value ref="surname"/>
    <value ref="first name"/>
    <value ref="phone number"/>
  </mapping>
</output-event-mappings>
```

The "role" attribute of the outgoing event refers to the role of the recipient party, as defined in the contract in which the service has been invoked. So, if the contract defines that ACE1 plays as "provider" and ACE2 plays as "user", the outgoing event addressed to "user" will go to ACE2. Each event can have only one recipient role. If the message should be sent to more than one recipient, separate mappings should be defined for each; or an `OutputEventMapper`.

If the functionality is `SessionAware` or `CallWideContextAware` (cf. sections 3.5.6.2 and 3.5.6.3), it may refer to parameter values coming from those contexts with the corresponding prefixes.

```
  <mapping event="cascadas.ace.event.ServiceResponseEvent">
    <value ref="x"/>                          // in/out parameter of a call
    <value ref="globalSession://sum"/>        // from the globalSession
    <value ref="executionSession:///y/>       // from the executionSession
    <value ref="callWideContext://file name"/> // rom the CallWideContext
  </mapping>
```

If the direct event mapping is not enough (e.g. the number of outgoing event is not known when the descriptor XML is created), the programmer may provide a suitable output event mapper descendant (`cascadas.ace.functionality.service.OutputEventMapper`) class that creates the appropriate output events.

```
<output-event-mappings>
  <mapper mapper-class="testfunctionality.MyOutputEventMapper"/>
</output-event-mappings>
```

Mappers are able to create a set of events from the available call parameters (black box and output), and from the session and `CallWideContext`. The specified custom mapper class must be available in the classpath.

```
public interface OutputEventMapper {
    public Set<Event> createOutputEvents(CallParameters params,
    Session     executionSession,     Session     globalSession,     CallWideContext
callWideContext);
}
```

Mappers and mappings can be used independently of each other, also in a mixed way. The following fragment specifies both two mappings and one mapper.

```
  <output-event-mappings>
    <mapping event="cascadas.ace.event.ServiceResponseEvent" role=”user”>
      <value ref="apple"/>
    </mapping>
    <mapping event="cascadas.ace.event.ServiceResponseEvent" role=”supervisor”>
      <value ref="apple"/>
    </mapping>
    <mapper mapper-class="testfunctionality.MyOutputEventMapper"/>
  </output-event-mappings>
```

Examples and more details can be found later.

3.5.3 Accessing a Service from Another ACE

In conformance with the dictionary, we are speaking about calling a service when the functionality is accessed by another ACE.

In order to access the service, the other ACE creates a `ServiceCallEvent` and is going to get back a `ServiceResponseEvent` as a response. Both events are subclasses of the `ServiceUsageEvent`, and both make it possible to pass the parameters (names and values) and other specific data (functionality name, error) to the remote party.

The Executor – when following a transition – extracts the call parameters from the incoming `ServiceUsageEvent` and wraps them into a `FunctionalityCallEvent` and forwards the FCE to the Functionality Repository which will result in the effective call.

The `ServiceCallEvent` should contain all input parameter values that are described in the black box model of the functionality, to avoid faults. For example, if the black-box description of the `phonebook-service` functionality contains the following parameters

```
  <black-box-description>
    <input>
      <param name="surname" type="java.lang.String"/>
      <param name="first name" type="java.lang.String"/>
    </input>
    <output name="phone number" type="java.lang.String"/>
  </ black-box-description>
```

then the following `ServiceCallEvents` are valid because they contains all parameters

```
ServiceCallEvent[first name=”Jack”, surname=”Smith"]
ServiceCallEvent[first name=”Jack”, surname=”Smith",number=”1”]
```

but the next `ServiceCallEvent` will result in an error:

```
ServiceCallEvent[surname=”Smith"]
```

The caller ACE has to prepare the `ServiceCallEvent` (with a Mapping or a Mapper) and send it to the called ACE. The called ACE has got the authority to make a decision whether

or not to perform the call. The decision is made by the Executor, based on the Plan (e.g. guard condition, trigger condition).

3.5.4 Specific Functionalities and Common Functionalities

Some functionalities are available by default on all ACEs while other functionalities are specific to certain ACE type(s). The first group is called "common functionalities" while the second one is called "specific functionalities".

3.5.4.1 Common functionalities

Common functionalities are deployed (loaded) during the ACE start-up by default. For the time being, the following common functionalities are available.

Scope	Functionality name	Description
Service invocation	service_caller_service	Calls a remote service. @deprecated, replaced by generic_service_caller_service
	service_caller_oneparam_service	Calls a remote service with one parameter. @deprecated, replaced by generic_service_caller_service
	generic_service_caller_service	Calls a remote service with arbitrary number of parameters.
	send	Sends an arbitrary inter-ACE event (also fills it with contents).
GN-GA, Contracting	gn_sender_service	Sends a GN.
	gn_answer_service	Answers the GN with a GA.
	contract_establishment_service	Establishes a two-party contract (user-provider), and saves it to the executionSession. @deprecated, replaced by contract_n_establishment_service
	contract_n_establishment_service	Establishes a multi party contract with an arbitrary number of ACEs and roles, and saves it to the executionSession.
	cancel_contract_service	Cancels a contract
Session manipulation	add_to_execution_session	Saves a key-value pair to the execution session. @deprecated, replaced by generic_add_to_execution_session
	add_to_global_session	Saves a key-value pair to the global session. @deprecated, replaced by generic_add_to_global_session

	generic_add_to_execution_session	Saves any number of key-value pairs to the execution session.
	generic_add_to_global_session	Saves any number of key-value pairs to the global session.
	store_contract_in_global_session	Stores the contract in the global session. Also adds entries for all roles of the given contract, referencing to the `AceAddress` of the participant.
Context acquisition	Context_acquisition_service	Initiates the context gathering from the given ACE and variable.
Event scheduling, timing	event_scheduler_service	Schedules an event to happen once in the future. @deprecated, use add_timer_service instead
	ntimes-event-schedule-service.xml	Schedules an event to happen repeatedly n times.
	periodic-event-schedule-service.xml	Schedules an event to happen repeatedly forever.
	wait_service	Waits for a while.
	add_timer_service	Adds a timer that will fire a `TimerExpiredEvent` in the given time.
	cancel_timer_service	Cancels a timer.
Misc.	plan_changer_service	Requests Plan change. @deprecated, removed, use start_plan_service instead
	request_life_cycle_service	Requests a life cycle action.
	start_plan_service	Starts a new parallel Plan.
	stop_plan_service	Stops a Plan.

3.5.4.2 Specific functionalities

Specific functionalities are prepared by the ACE programmer. They consist of the source code (called classes and methods, and optionally output mappers) and the XML descriptor.

By default, the services are deployed to the Functionality Repository at the start-up of the ACE.

The Repository gets a folder name, and loads all functionality descriptor XML files from it. When loading a descriptor XML, the repository checks the availability of the classes and methods, but does not create instances nor performs method calls.

3.5.4.3 Deploying a specific functionality to the Repository

The functionality folder (the folder containing the XML descriptors of the specific functionalities) can be specified in the ACE descriptor file (normally conf/aces/my_type/ace_type.xml).

The steps to deploy your functionality into the ACE are the following.

- Place its descriptor XML into the "repo" folder of your ACE type (conf/aces/my_type/repo).
- Make sure that your classes/jars are available in the classpath.
- Specify the place of the "repo" folder in the ACE descriptor (conf/aces/my_type/ace_type.xml).
- Add your ACE instance to the conf/aces.xml.
- Start the Toolkit and check for error messages originating from the Repository.

Sample descriptor XMLs can be found in the "conf" folder of the Toolkit.

3.5.5 Statelessness

The Repository does not maintain any state information between the calls.

- It is not guaranteed that two subsequent invocations will be executed on the same instance of the called class. (E.g. two "phonebook-lookup-service" calls might be executed on two different MyPhoneBook instances).
- Inside one invocation, in complex call sequence, if two call parts refer to the same class (`MyPhonebook.loadFunctionality()` and `MyPhonebook.lookup()`), it is guaranteed that they will be executed on the same instance of the class.

The following data are lost at the end of the call (after the output events are sent):

- All field values of the called classes.
- All input and output (return) values of the calls.
- Any other side-effect of the call, except for:
  - data that are stored in the global / execution session (see next section)
  - threads that are registered to the central ThreadPool of the ACE

3.5.6 Accessing the Sessions and the Repository-Internal Data Structures

There are ways to store information during/between calls and to make information available for the output event creation: `CallParameters`, `CallWideContext`, `ExecutionSession`, `GlobalSession`. `CallParameters` is a data structure that can be used for output mapping/mappers only.

The word "session" refers to ACE-local variables that store internal state information in a permanent way.

3.5.6.1 CallParameters

`CallParameters` is an auto-created data structure, it contains the input and output parameters of all call parts (which is equal to or more than the black box model). Parameters are key-value pairs (parameter name – parameter value). This is the guaranteed minimum knowledge that is available for the output event creation.

The `CallParameters` is not accessible from inside of a specific functionality class.

For the output event creation, the data stored in the `CallParameters` are directly accessible by name.

```
  <output-event-mappings>
    <mapping event="cascadas.ace.event.ServiceResponseEvent" role="x">
      <value ref="number"/>           // "number" was an in/out param of a call
    </mapping>
  </output-event-mappings>
```

When using `OutputEventMappers`, the `CallParameters` is passed to it as argument.

3.5.6.2 CallWideContext

`CallWideContext` is a data storage facility (contains key-value pairs) that is available throughout the call: from the point when the Repository finds the requested functionality until the point when all output events are sent back.

`CallWideContext` makes it possible to publish (and access) key-value pairs other than the return value of the call parts. It can also be used to store information for the event mapping.

The `CallWideContext` may be available from inside of the called functionality class. In order to get access to it, the class of the specific functionality must implement the interface `CallWideContextAware` (`cascadas.ace.functionality.service.CallWideContextAware`), which makes it possible for the Repository to forward the `CallWideContext` to it.

```
public interface CallWideContextAware {
    void setCallWideContext( CallWideContext callWideContext );
}
```

When event mapping, values stored in the `CallWideContext` should be prefixed with the string `callWideContext://`.

```
    <mapping event="cascadas.ace.event.ServiceResponseEvent" role="x">
      <value ref="callWideContext://file name"/>
    </mapping>
```

The `CallWideContext` is passed to the `OutputEventMappers` as an argument. If the functionality is not `CallWideContextAware`, this is an empty context (not null).

3.5.6.3 Sessions

Sessions contain key-value pairs that are guaranteed to be maintained throughout the session, so even between the calls.

There are two sessions available in each functionality call: the `globalSession` and the `executionSession`. The global session is created at the bootstrap of the ACE and is maintained as long as the ACE is alive (even between migrations). The execution session is created when the current Plan is started, and is valid during the execution of the Plan.

Just like `CallWideContext`, a Session makes it possible to publish (and access) key-value pairs other than the return value of the call parts. It can also be used to store information for the event mapping.

Sessions should be handled with care. As they may keep their contents for long time (maybe even whole lifecycle of the ACE); they should not be abused for storing non inter-call important data.

The Session may be available from inside of the called functionality class. In order to get access to it, the class of the specific functionality must implement the interface `SessionAware` (`cascadas.ace.functionality.service.SessionAware`), which makes it possible for the Repository to forward the Session to it. The two setters will be called automatically by the Repository if the specific functionality implements the interface.

```
public interface SessionAware {
    void setExecutionSession(Session executionSession);
    void setGlobalSession(Session globalSession);
}
```

In case of event mapping, the values stored in the two Sessions should be prefixed with the strings `globalSession://` and `executionSession://`, accordingly.

```
    <mapping event="cascadas.ace.event.ServiceResponseEvent">
      <value ref="globalSession://apple.weight"/>
      <value ref="executionSession://pear.color"/>
    </mapping>
```

The two sessions are also passed to the `OutputEventMappers` as arguments. If the functionality is not `SessionAware`, they are empty (but never null).

3.5.7 Accessing other ACE Resources

3.5.7.1 Logging

The specific functionality is able to gain access to the ACE (instance) logger in case it implements the interface `LogAware`.

3.5.7.2 ThreadPool

The thread pool is a tool to enable the specific functionalities to use own custom threads while ensuring consistency with the ACE life cycle (e.g. ACE shutdown, mobility).

How does the Thread pool work?

Specific functionalities start their custom threads with the help of the pool. The pool maintains a registry about these threads.

On the corresponding life cycle event, the Repository shutdown process also includes the stopping (interrupting) of the threads of the thread pool. This way, the ACE does not leave "zombie" threads in the Java Virtual Machine after shutdown.

Rules:

Specific functionalities must not start threads the usual way (`Thread.start()`) they should use the `ThreadPool` for starting their custom threads.

The thread pool is available for those specific functionalities that implement the interface `cascadas.ace.functionality.service.ThreadPoolAware`.

The `ThreadPool` interface defines a single execute method that executes the specified logic in a new Thread.

```
/** General interface of the ThreadPool towards the specific functionalities. */
public interface ThreadPool {
    /**
     * Starts a new thread with the specified Runnable as target.
     * @param r target
     */
    public void execute( Runnable r );
}
```

3.5.8 What Happens when a Service is Called?

When the Functionality Repository receives a valid `FunctionalityCallEvent`, the following action sequence is performed.[9]

1. The Repository looks up the referred functionality ID.
2. Creates the `CallParams` and the `CallWideContext`.
3. Gets the list of the classes that will be needed (that are referred in the call parts) and creates/gets an instance of each.
4. Checks whether the classes are `CallContextAware`, `SessionAware`, `ThreadPoolAware` or `LogAware`. If yes, forwards them a reference to the concerning resource.
5. Extracts the input parameters from the `FunctionalityCallEvent`.
6. Performs the call parts one by one, and saves their result to the `CallParameters`.
7. Performs the event mappings: creates, fills and sends the output events one by one.
8. Performs the mapper based event mappings: asks the `OutputEventMappers` to create the set of outgoing events, then sends them one by one. The `OutputEventMapper` gets access to the `CallParameters`, to the `CallWideContext`, and to the Sessions.

3.5.9 Technical Expectations, Error Handling

3.5.9.1 Technical expectations

The deployed functionalities must have default (no-arg) constructors.

If an outgoing event is a `ContractEvent`, it must have a constructor that takes the contract as parameter. If the outgoing event is not a `ContractEvent`, it must have a default (no-arg) constructor.

[9] Note: the Session is part of the `FunctionalityCallEvent`.

3.5.9.2 Error handling during the deployment

All deployed functionalities are checked for availability during deployment. Errors are logged, erroneous functionalities are not loaded (but the loading of other functionalities continues).

3.5.9.3 Error handling during the call

By default, the errors are logged but are not sent back to the caller.

In order to support error handling, please include error related fields into the return events. For example, the `ServiceResponseEvent` contains methods for this purpose.

4 Integration Features

This chapter describes how features like aggregation, security, Knowledge Networks and the Supervision System are integrated with and supported by the Toolkit.

4.1 Knowledge Network Support

Knowledge Networks as they are developed by WP5 are a mechanism to structure and organise knowledge. Their key elements are storage components, a querying interface, and a verification mechanism. Information can be stored in Knowledge Atoms and Atom Repositories where Atoms comprise one portion of information and Atom Repositories summarise a set of Atoms. Knowledge Containers are elements that organise other, related components in higher-level structures. The components to be organised can again be Atoms, Atom Repositories or Containers themselves. The querying interface can be used by entities, which do not necessarily need to be part of the Knowledge Network, in order to search for information related to a topic, i.e. a concept of interest. The answer to a placed query is the data found in any components of the Knowledge Network and related to the requested concept. Organisation and linking of data is internal to the network and takes place autonomically. In order to maximise the validity of information a Context Verification mechanism can be utilised to check the correctness of data with respect to former experiences and other related knowledge available within the network. This Knowledge Network paradigm enriches an environment with well organised and thoroughly maintained data, i.e. information. Services in an environment that provides a Knowledge Network can access and benefit from this information. Figure 13 shows the interaction between requesters and the Knowledge Network where all entities are realised as ACEs. Detailed explanations of that topic are available in [D5.3].

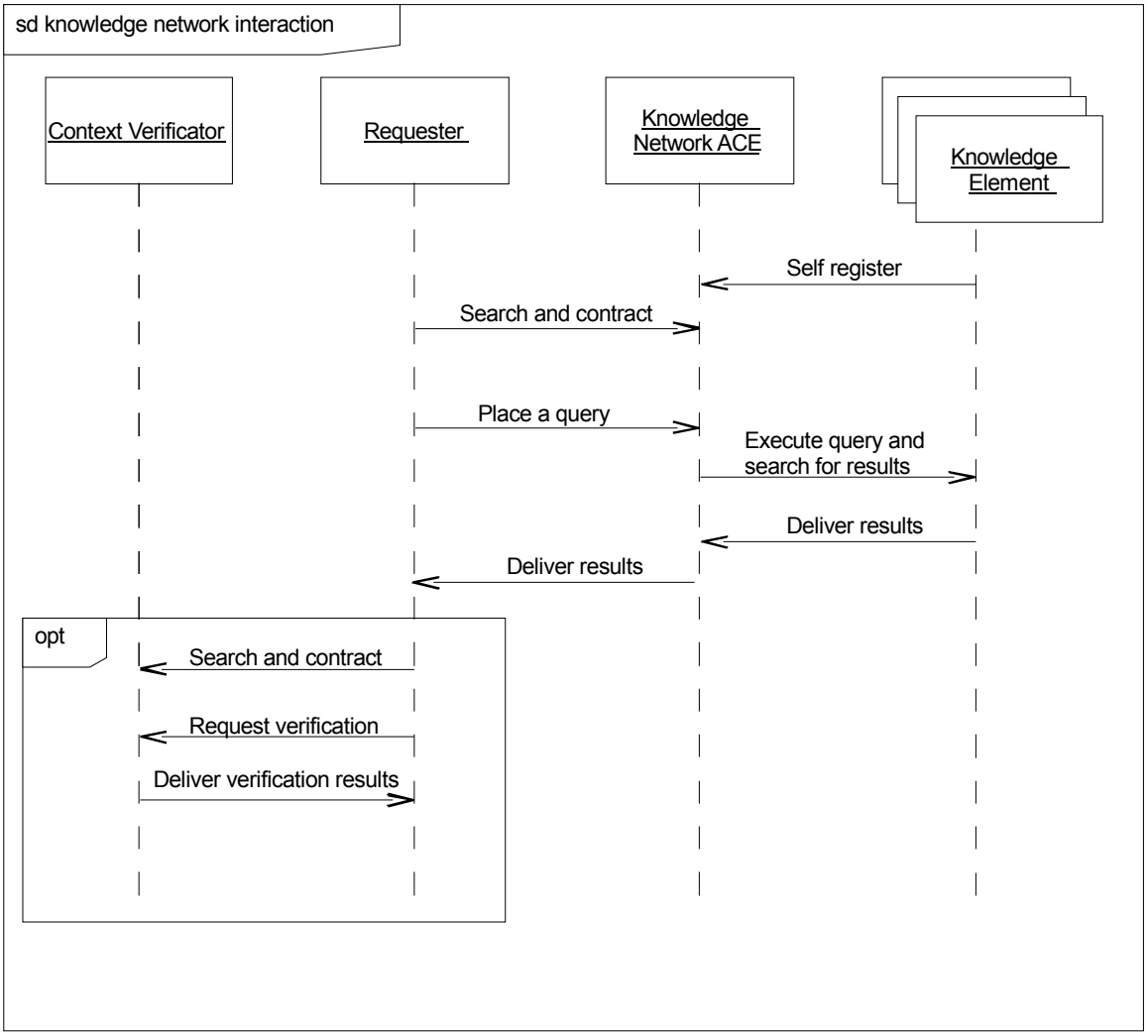


Figure 13: Interaction between requesters and the Knowledge Network

In order to enable the usage of Knowledge Networks for all applications within the scope of the CASCADAS project, their design was aligned with the development of the WP1 Toolkit. All Knowledge Network components can exist as separate entities exhibiting and utilising services and interfaces of others. During the last research period, these entities were ported to the Toolkit concept which means that they were realised as ACEs, as envisaged by WP1. Due to that, Knowledge Networks can now be integrated in any other ACE based application environment. Along with the porting to ACEs, further requirements arose and still arise which demand extension and adaptation of the Toolkit to the needs of WP5.

Until now, the integration process made very large progress but is not yet entirely completed. Though the implementation is already done in large parts, more analyses and adaptation work have to take place. In order to demonstrate the results which have been achieved by now, the Context Verification system as one part of Knowledge Networks was integrated in the WP1 application example. The application itself is described in chapter 5. In the remainder of this section we will only explain how Context Verification was integrated to exemplarily show that the concept of Knowledge Networks is supported by the CASCADAS Toolkit.

The Context Verificator is integrated into the example application (cf. chapter 5) as one dedicated ACE. In order to verify all data the displaying ACE uses to select an advert, it searches for a component which offers a Context Verification service by sending out an according Goal Needed event. Subsequently, the Verificator ACE will answer with a Goal Achievable which comprises all information that is needed to call the verification service. This includes the service name and the parameters which have to be passed. When receiving the Goal Achievable, the Display creates a contract to the Verificator and is, from now on, able to call its service whenever this is required. In the flow of the application, the Display periodically collects data about the surrounding people and hands them to the Verificator ACE before selecting an advert accordingly. The Context Verificator then searches for formerly made experiences which are related to the data it currently has to validate. Based on the search results, it evaluates all available information and decides whether to classify the current contextual data as valid (true) or suspicious (false). An advert will only be displayed if the current context is regarded to be correct. The implementation of the Context Verificator and the performed algorithm are described in detail in [D5.3].

4.2 Supervision Support

The capabilities of being supervised are inherent to all ACEs as they are specified in a dedicated organ. Its implementation can be found in the `cascadas.ace.supervision` package.

Supervision works by adding so-called *checker* objects to critical monitoring points in the ACE. Currently these are the outgoing queues of the Manager (enabling supervision of internally dispatched events) and the Gateway (enabling supervision of externally dispatched events). A *checker* is consulted every time a monitoring event occurs and the event is either accepted or denied, which may lead to an exception in the supervised organ (e.g. the Manager). It is planned to also integrate supervision checkers at other points within an ACE, for example during execution of a transition of the Self Model.

Figure 14 gives an overview of the employed interfaces for the integration of supervision checkers with an organ. The supervised organ is supposed to implement the `Supervisable` interface. Once supervision starts, a call to the `startBeingSupervised` method passes a `SupervisionChecker` interface object to the supervisable. It is now expected that every monitoring event is handed to the `check` method of the interface and the returned `Result` followed. Supervision stops for the supervisable with a call to the `stopBeingSupervised` method, which anticipates that the corresponding checker object is returned. During supervision operation, a supervisable must be ready to integrate additional monitoring events in its operations that are passed via the `insertEventBySupervisor` method.

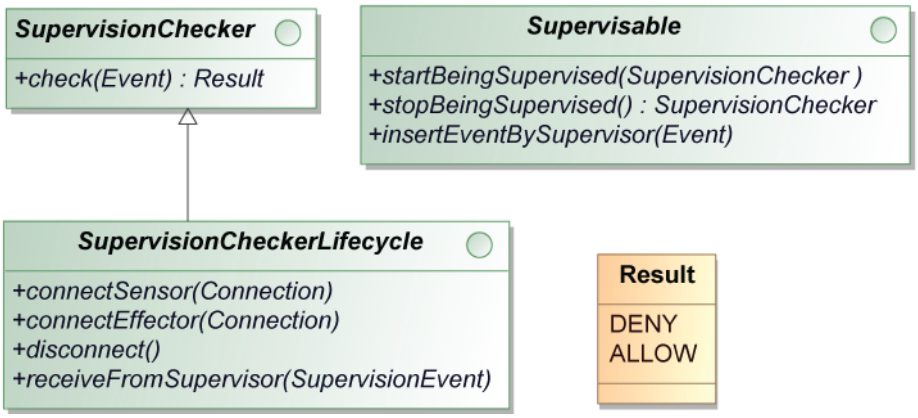


Figure 14: Interfaces of the supervision sub-system

The `SupervisionCheckerLifecycle` interface is used by a Supervision organ in the setup of a supervision checker. As supervision checker objects are created independently of the supervised ACE, they are created by a supervision ACE and then passed to the supervised ACE, where the Supervision organ puts them into place.

When a checker object is migrated to the Supervision organ of the supervised ACE, two independent DIET connections are automatically established: One connection leads to the supervision sensor responsible for the supervised ACE (a monitoring component), the other connection ends at an effector (a controlling component). Both connections are handed to the checker object by the corresponding methods `connectSensor` and `connectEffector`.

Any supervision events that are received by the supervised ACE are reported via a call to the `receiveFromSupervisor` method. At the end of the supervision operation, a call to `disconnect` informs the checker that these connections are about to be removed. Afterwards the checker objects are considered to be unusable.

Apart from the local setup of the checker objects, supervision is controlled by events. Figure 15 shows the basic events together with an abbreviated version of the Supervision organ's signature, more specific the event handling routines of the central `cascadas.ace.supervision.SupervisionJob` class. Two small event hierarchies can be differentiated: Local events descending from the `SupervisionEvent` type (namely `BusSupervisionEvent`, and `GatewaySupervision`) and events that are being sent to other ACEs. These event types are descending from `ServiceUsageEvent` and are used to remotely control the supervision operation lifecycle of an ACE: `ConfigurationEvent`, `StartSupervisionEvent`, and `StopSupervisionEvent`.

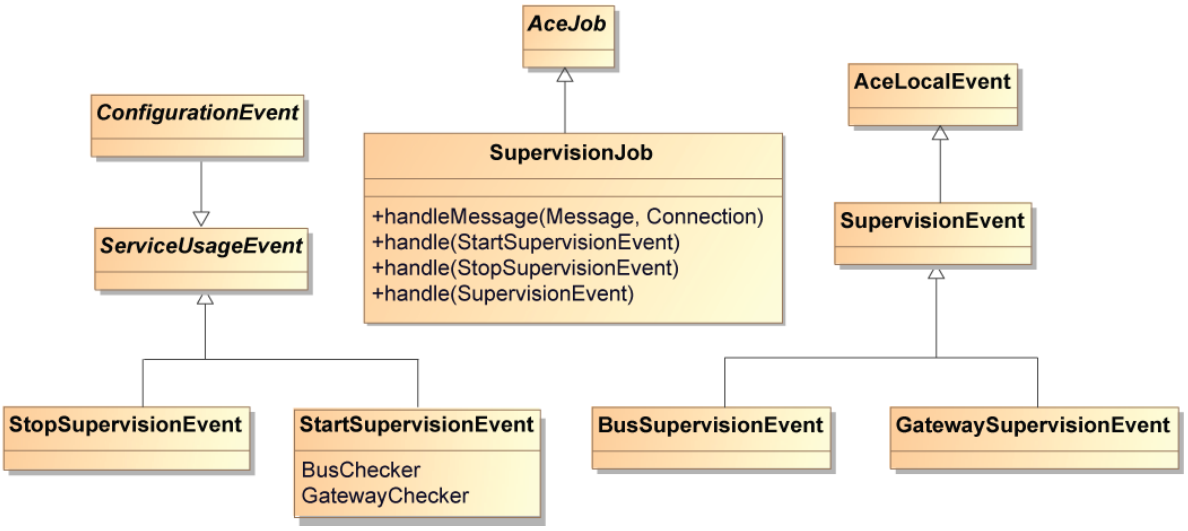


Figure 15: Events of the supervision sub-system

Before a supervision system (a construct of several ACEs, including effectors and sensors) will begin with a supervision task, it is contracted by a system that is looking to be supervised (via GN-GA protocol and subsequent contracting). After the control contract[10] is in place, certain addresses are configured within the supervision system using messages of type `ConfigurationEvent`. When the supervision system is set up properly, it will send a `StartSupervisionEvent` containing instantiated checker objects to the system under supervision, triggering a local dispatch and connection of the transported checkers.

At this point in time the supervision operation is fully working: checker objects transmit `SupervisionEvents` to the supervisor. A supervisor ACE (e.g. the effector) in turn influences the supervised ACE by using messages of type `BusSupervisionEvent` and `GatewaySupervisionEvent`. A standard communication pattern (e.g. Request-Response) is not prescribed. This is freely designable and depends on the supervision system's Self Model and the nature of the checker objects. At the end of a supervision session, the supervisor sends a `StopSupervisionEvent` using the control contract.

The `handleMessage` and `handle(SupervisionEvent)` methods of the `SupervisionJob` class expose different behaviour, depending on the role of the ACE the Supervision organ belongs to. If they belong to a supervised ACE, `handleMessage` forwards the transported message to the supervision checkers and `handle(SupervisionEvent)` does nothing. If they belong to a supervising ACE, `handleMessage` distributes messages to the Bus (serving as triggers for the supervisor Self Model) and `handle(SupervisionEvent)` will forward the events to the supervised ACE.

Requesting supervision is a much more trivial task. Any ACE may request supervision by following three steps:

1) Discovering an available supervision system. This is done by sending a GN for "supervision" and by analysing the GA responses.

[10] We refer to this as the **main** or **control** supervision contract

2) Contracting the supervision system (e.g. using the addresses returned by GA responses in step 1).
3) Sending configuration event(s) to the supervision system that contain addresses of all ACEs that are part of the supervision. Currently this is a single event of type `cascadas.supervision.interaction.protocol.SupervisionConfigEvent` (part of WP2 toolkit extension), but this is likely to be replaced by a more flexible configuration mechanism in the near future.

Once all addresses are transmitted to the supervision system, a supervision session is set up automatically.

4.3 Aggregation Support

In the Gateway section of the [D1.2] deliverable it has been shown that the GN-GA protocol is executed using a publish-subscribe middleware based on REDS. The Gateway component publishes the goals the ACE can achieve or needs and subscribes to the goals it is interested to.

The main characteristics of this model are:

- Broadcast communication of the goals using session-less event-based communication in REDS;
- One-to-one session-capable communication managed by DIET contracts.

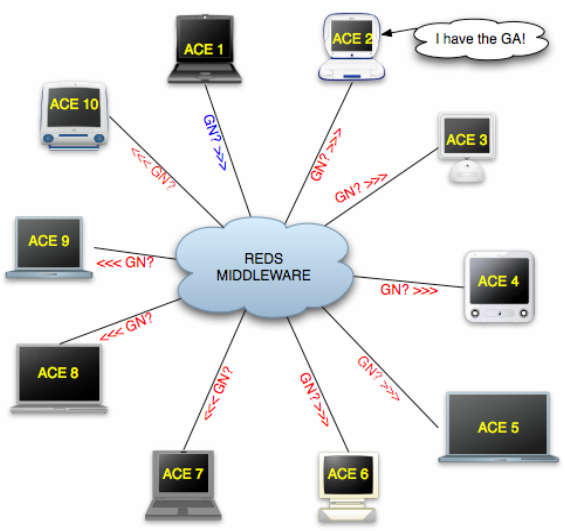


Figure 16: A GN is sent to all the ACEs when there is only broadcast communication available.

This mechanism is very powerful, but it is not suitable in situations when an ACE has limited computational or communicational resources. A possible limitation in its computational power can make it difficult for a node to keep up with processing all the goals that are produced in a network where there are thousands of nodes. For the same reason if we have limited communicational resources the broadcast solution does not scale well and can be an issue when the links have a low capacity.

A possible idea to solve this problem is the introduction of the concept of *group* (or *cluster*). A group can be seen as a way to reduce the scope in which other ACEs are reachable using GN-GA events. This way all the events are only sent to the group the ACE belongs to or, in different words, each ACE would have a list of other ACEs that are reachable directly.

To apply this simple idea, the components of the group should be dynamically modifiable in order to put matching GAs/GNs in the same group, and therefore to establish one-to-one DIET contracts like in the broadcast approach.

The solution to this problem is to integrate the self-aggregation algorithms that have been proposed and simulated in the [D3.1] and [D3.3] deliverables in the communication model of the ACE. The following paragraphs show how it is possible to create the concept of group communication and integrate the self-organisation algorithms in the Gateway component of the ACE Toolkit.

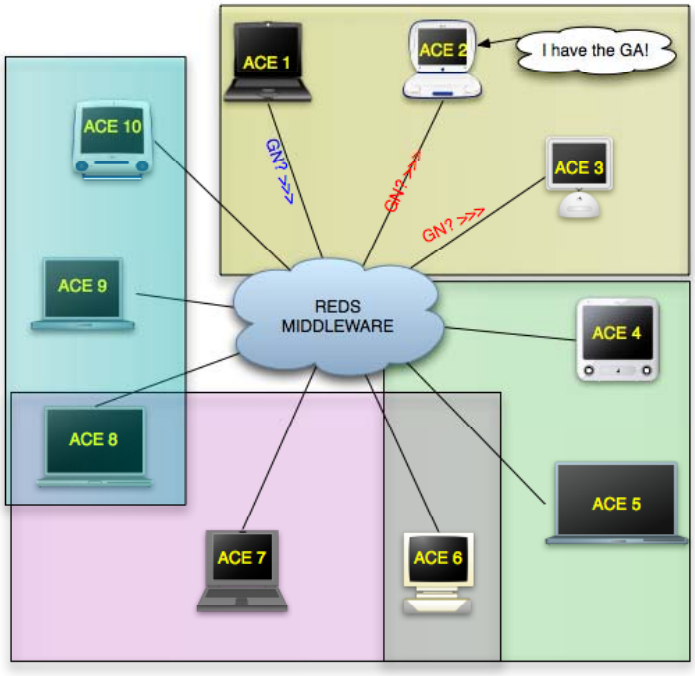


Figure 17: If group communication is possible, there is no need to send the GN to all the ACEs of the network. In the figure the groups are represented by rectangles. Groups may also overlap.

4.3.1 Group Communication

As anticipated in the previous paragraph, the group communication requires that each ACE is capable of managing a list of ACEs called Neighbour List. Basic operations that can be done on this list are simply adding and removing known ACEs. Our goal is to limit direct communication only among ACEs and their neighbours. The criteria in which a group is created will be the equality of node goals, this way it is possible to aggregate nodes that offer/are interested to the same goals.

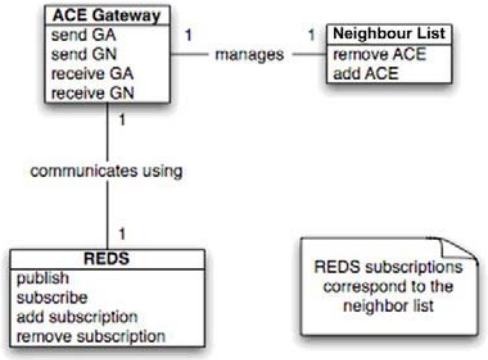


Figure 18: Diagram that shows the relations between Gateway, Neighbour List and the REDS middleware.

4.3.2 Initialisation

Each ACE should initialise its Neighbour List in order to create an initial overlay network among nodes (this will be built on top of the REDS middleware). This initialisation can be performed by providing a list of known nodes during ACE start-up or, alternatively, by using any other mechanism to find the initial neighbours. If broadcast communication is still an option, it can be used only for initialisation purposes by sending a "discover" message to all the other nodes.

4.3.3 Group Communication using REDS

After initialisation is completed each ACE connects to one of the brokers of the REDS overlay network. The difference with respect to the previous approach is that the initial subscriptions are more specific: each ACE subscribes only to events that are sent by a neighbour node. As soon as the Neighbour List changes due to the appearance/disappearance of a new node, or overlay network rewiring, the list of subscriptions to the REDS middleware changes accordingly. Broadcast communication, if available, can be always available to the ACE when it is deployed on small networks or networks where the saturation of communication channels is not an issue.

4.3.4 Integration of Self-Aggregation in the Gateway Component

The logic that is behind the group communication can be integrated into the Gateway component. This component can make group communication transparent to the rest of the ACE in such a way that other components are unaware of what is happening on the logical overlay level.

4.3.5 Updating the Goals on the Neighbour List

The first modification that is required on the Gateway is the storage of the Neighbour List. Each element of this list contains information about the identification of the neighbour ACE and its goals. It is also necessary that the Gateway component subscribes to the events that are sent to the Bus of the ACE when a new goal (GA or GN) is added or removed.

Each time a goal is changed, the Gateway component upgrades its state and sends an update message to the ACEs that are on the Neighbour List.

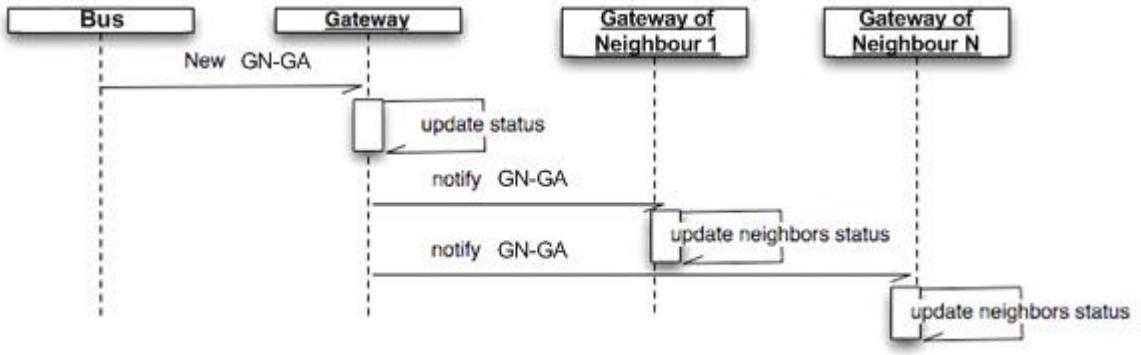


Figure 19: Each GN-GA event that appears in the Event Bus updates the status of the local Gateway component and sends (using REDS) an update messages to all the neighbours of the ACE.

4.3.6 Using Self-Aggregation Algorithm to Update the Neighbour List

When the broadcast communication is no longer possible, the only possible way to establish a contract between two ACEs is that the ACE that has a GN of a particular type is a neighbour of the ACE that has the matching GA. Having this means that the overlay network should evolve in order to guarantee that all GAs will be matched with the GNs. The mechanisms that we are going to use are the self-aggregation algorithms that can be found in deliverables [D3.1] and [D3.3]. The purpose of these algorithms is to rewire the topology of a network in order to cluster nodes that share the same property (in this case the property is the goal of the node). This algorithm will be executed on a new thread that becomes completely independent from the other components of the ACE. The only shared resource will be the access to the Neighbour List, and to the REDS middleware to exchange the messages required by the clustering protocol.

The initialisation of the algorithm is distributed and can be performed in three ways:

1. During the ACE initialisation;
2. When a new GN-GA event has been generated locally;
3. When a neighbour node generates a new GN-GA event.

Using the first policy we have an overlay network that evolves continuously, while with the other two approaches the algorithm is started only when it is needed. In the third approach a goal that has been received remotely should be forwarded to the neighbour nodes in order to activate the algorithm on all ACEs of the network.

The termination problem can be addressed in the following two ways:

1. Algorithm termination occurs when the ACE terminates;
2. Termination occurs when the Neighbour List does not change for a certain amount of time.

The first termination method simply means that the algorithm never stops when the ACE is alive. This can give more time to the algorithm to reach a good degree of convergence,

but requires more messages. Alternatively, the Gateway can monitor the activities on the Neighbour List and suspend the algorithm when there are no insertions/removals from the list for a certain amount of time. This amount of time can be chosen to obtain the right trade off between algorithm convergence and the network load.

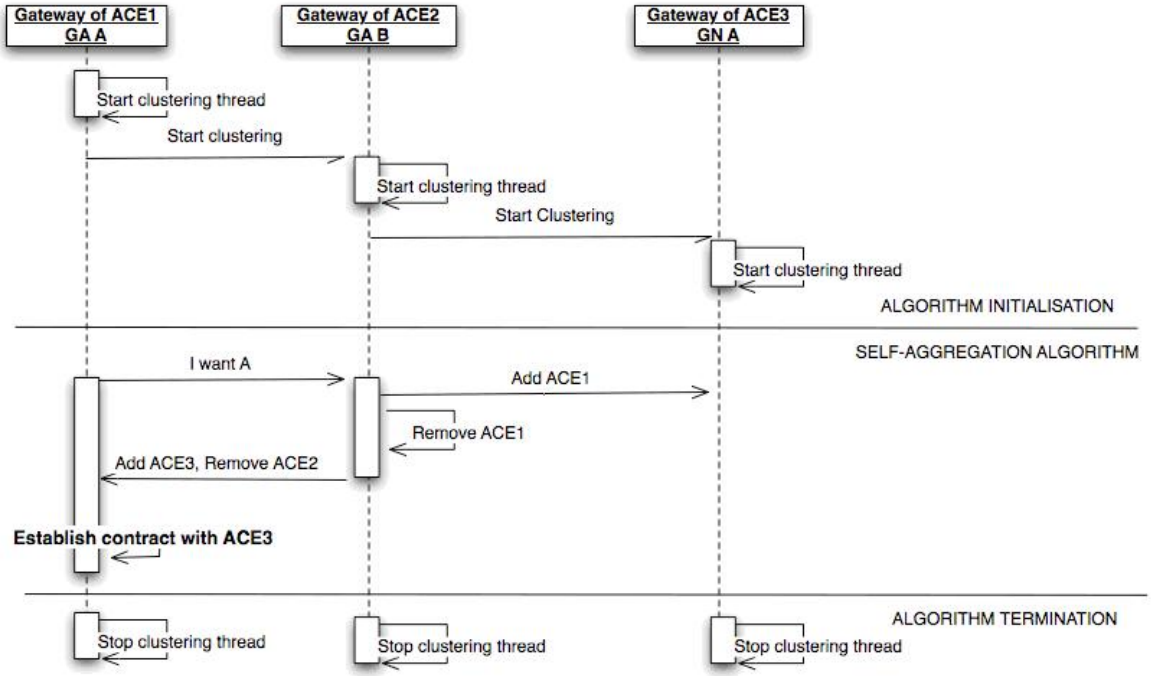


Figure 20: UML Sequence diagram with algorithm initialisation, self-aggregation, and termination. At start-up ACE2 is a neighbour of ACE1 and ACE3, then the connection between ACE1 and ACE2 is replaced with the new connection between ACE1 and ACE3.

4.3.7 Self-Aggregation Issues with Multiple Different Goals

The aggregation algorithms have been extensively studied and tested in networks where each node has only a single type that can be equal or different from another one (see [D3.1] and [D3.3]). To apply aggregation algorithms to ACEs we proposed to map nodes to ACEs (i.e., a node is an ACE) and node types to ACE goals (i.e., a type is a goal). The original aggregation algorithms are studied to work with only a single node type, while instead ACEs can have more than one goal. Accordingly, this situation is not compatible with the original version of the algorithms and will be studied in the next months.

4.3.8 Load-balancing Example

Until now we have seen the self-aggregation as a possible way to permit the creation of contracts between ACEs without using broadcast communication. The self-aggregation can be exploited also in different ways: if, for example, we consider an ACE that has multiple GNs of the same type, and it has in its Neighbour List more than one neighbour with the matching GA, then it can establish contracts with different ACEs without doing further rewiring on the overlay network. This means that when the network is clustered, load

balancing will be a natural consequence of the rewiring that is performed by the self-aggregation algorithms.

4.3.9 Discussion

In this section we have seen that the broadcast communication mechanism does not scale well with network size, and therefore a different communication mechanism can be added to the ACE. A new mechanism, called group communication, has been proposed: it considers a network in which each ACE starts with a limited knowledge on the other ACEs, after start-up then all ACEs start a self-aggregation algorithm to reorganise their connections and finally to form clusters of ACEs that share the same goals. These groups have the property of putting together nodes with the same GA and nodes with the same GN.

After the self-aggregation has been executed the system can benefit from the following situations:

- Cluster of ACEs with the same GAs: this is positive because when some GN enters the cluster, its workload can be load balanced among all the others ACEs;
- Cluster of ACEs with the same GNs: when one of the ACEs with the corresponding GA enters the cluster, it can then easily passed to all the ACEs of the cluster;
- Cluster of ACEs with corresponding GAs/GNs: a contract between each couple of ACEs is established, and the GNs are progressively cancelled.

4.4 Security Support

Security is integrated to the Toolkit on the service level: certain ACEs do have security capabilities, and provide security related services to the benefit of other ACEs. This way, security becomes accessible but not mandatory for all members of the ACE population.

Security related services integrated to the ACE Toolkit are going to cover all basic security principles applicable for a distributed system: authentication, confidentiality, and integrity.

- Authentication enables the node to ensure the identity of the other node it communicates with. Without authentication, the attacker could masquerade the node thus gaining unauthorised access to resources. Typical tools for authentication are digital signature, and shared common secret (e.g. password).
- Confidentiality ensures that certain information is never disclosed to unauthorised parties. Transmission of sensitive information always requires confidentiality. Leakage of such information to an eavesdropper could result in severe consequences. The typical tool for confidentiality is the message encryption which can be achieved with numerous techniques (e.g. symmetric, asymmetric encryption).
- Integration guarantees that the transferred message is not corrupted – corruption may occur due to failures or because of malicious attacks. A typical tool for integrity is to append an un-forgeable digest to the message, using hash functions.

To comply with the lightweight and distributed ACE model, the automatic formation of security domains is envisioned. Members of the same domain are under the same administrative control.

The integration of security features addresses the specific part of the ACE: the Functionality Repository and the Self Model.

In some cases, the cooperation of several security ACEs is required to provide the intended aggregated service (e.g. key provider, encoder, and decoder).

5 Sample Application Description

The advertisement demo application was developed in order to test and prove WP1 Toolkit features during their development. It provides other WPs with programming examples that show how to use the Toolkit properly.

To highlight the new aspects of the ACE Toolkit we extended the advertisement demo application. The enhancements which were added have the purpose to show that the Toolkit supports the integration of the Supervision System and the Knowledge Network. So, the improved example application demonstrates that it is possible to develop a Supervision System as intended by WP2 and a Knowledge Network structure as WP5 envisions it to prove that the work of WP1 fits to the work of the other WPs.

To show that integration of the WP1 and WP5 results is possible, a Context Verification system was implemented and the WP1 advertisement application was adapted accordingly. Therefore, one additional Context Verifier ACE is needed. The application itself is only affected in so far that the display ACE will, before displaying adverts, send the gathered context information to the Verifier and only displays the associated advert if the context is correct. Otherwise an error message will be shown. The GUI was extended in the way that enables entering incorrect context information. What also was necessary is the adaptation of the context information to the scenario described in this chapter.

5.1 Structure of the Application

In order to adapt to the CASCADAS advertisement auction scenario, we use the display ACE to show adverts according to the interest of people which are supposed to be in front of the screen. Sensory data about present people can be of the categories *gender, dress code*, and *hobbies*. As already implemented in the last advertisement example (cf. [D1.2]), one ACE per category will provide the according data, gathered from all present persons. Table 2 shows, which sensory data is associated with which advert.

Gender	Dress code	Hobbies	Advert
Male	Business	Cars	Limousine
Male	Business	Clothes	Business suit
Male	Leisure time	Cars	Formula one
Male	Leisure time	Clothes	Jeans
Male	Swim suit	Cars	Invalid context
Male	Swim suit	Clothes	Invalid context

Female	Business	Cars	Limousine
Female	Business	Clothes	Woman's suit
Female	Leisure time	Cars	City car
Female	Leisure time	Clothes	Jeans
Female	Swim suit	Cars	Invalid context
Female	Swim suit	Clothes	Invalid context

Table 2: adverts associated to sensory data

The ACEs presented in Figure 21 are necessary for realising the adapted advertisement example.

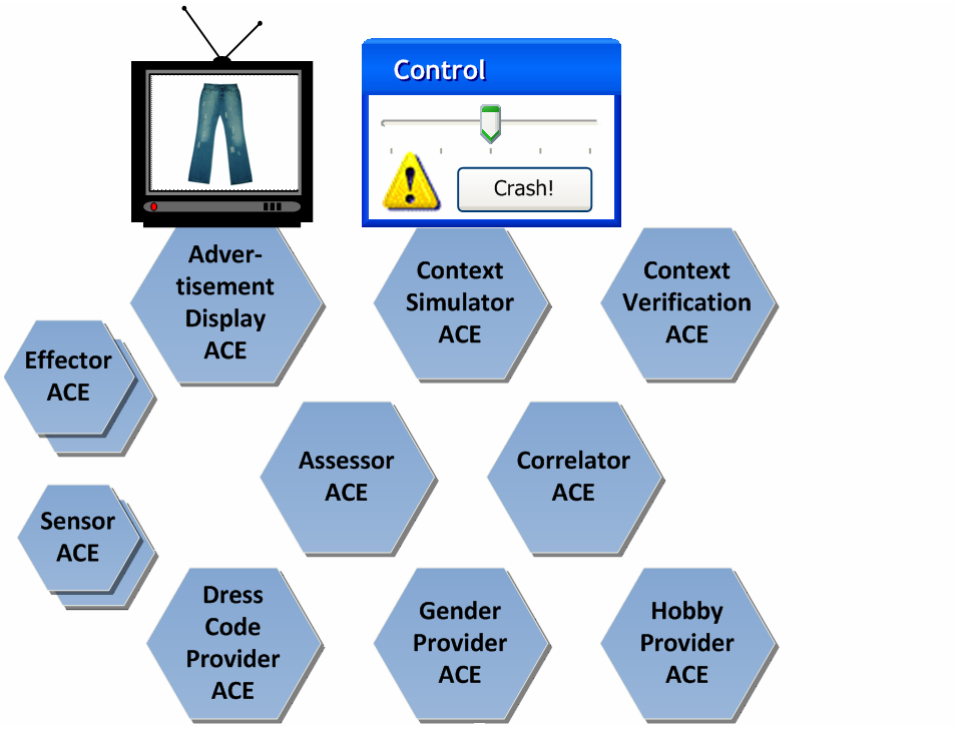


Figure 21: ACEs in the advertisement example application

The **advertisement display ACE** shows adverts based on the information it gets from the ACEs around. It uses the dress code provider ACE and the hobby provider ACE as context providers and the gender provider ACE for delivering information as an answer to a service call. In case the context data seems to be suspicious or incorrect, the display ACE shows an according message instead of any advert.

The **context simulator ACE** shows a GUI on which the values of all sensors can be set by the application user. To demonstrate the two concepts of data acquisition, i.e. context gathering and acquiring data via service request, it is clearly labelled which sensor is which kind of data source. The adjusted sensor values are handed over to the corresponding provider ACEs.

The **context verification ACE** offers the service to verify a context pattern, i.e. a set of sensor values, and to return whether this pattern is regarded as valid or not. This service is requested each time the advertisement display ACE selects a new advert. Before presenting any information on the screen, the recently acquired sensor values are passed to the context verification ACE in order to be tested. Only if a positive answer is returned, the advert is displayed. Otherwise an error message is printed on the screen.

The **gender provider ACE** offers the service to deliver the information if the majority of the people in front of the advertisement display are female or male. The gender provider ACE gets this information from the context simulator ACE and only answers once, not periodically, per service request.

The **dress code provider ACE** delivers context information about the dress code of the majority of the people standing in front of the advertisement display. It is only requested by the advertisement display once and then periodically updates the dress code information. The dress code provider ACE gets its data from the context simulator ACE.

The **hobby provider ACE** delivers, similar to the dress code provider ACE, context information about the hobbies of the majority of the people standing in front of the advertisement display. It is as well requested only once, updates the hobby status periodically, and get its data from the context simulator ACE.

The **effector** and **sensor ACE**s are part of mechanisms for realising supervision functionality. They instrument the supervised ACEs (advertisement display ACE and dress code provider ACE) with sensor and motor channels to be used by the correlator and assessor ACE.

The **correlator ACE** is responsible for interacting with the sensor ACEs, monitoring the operation of the advertisement display and dress code provider ACE, and alerting the assessor ACE if an abnormal situation arises. In the example such a situation is characterised by triggering a "crash" of the advertisement display ACE Gateway using the context simulator. The crash is simulated by having the display ACE stop receiving external events with updated information.

The **assessor ACE** analyses input alerts from the correlator ACE and decides if an intervention by the Supervision System is necessary. In the case of crash of the display ACE it employs the respective effectors to influence the instrumented ACEs: The advertisement display ACE's Gateway is restarted and the provider ACE internal state is corrected.

Figure 22 shows a screenshot of the application showing a GUI to simulate contextual data and providing the possibility to simulate a crash. The advertisement display and a set of log messages listed with the Chainsaw logging tool can be seen as well.

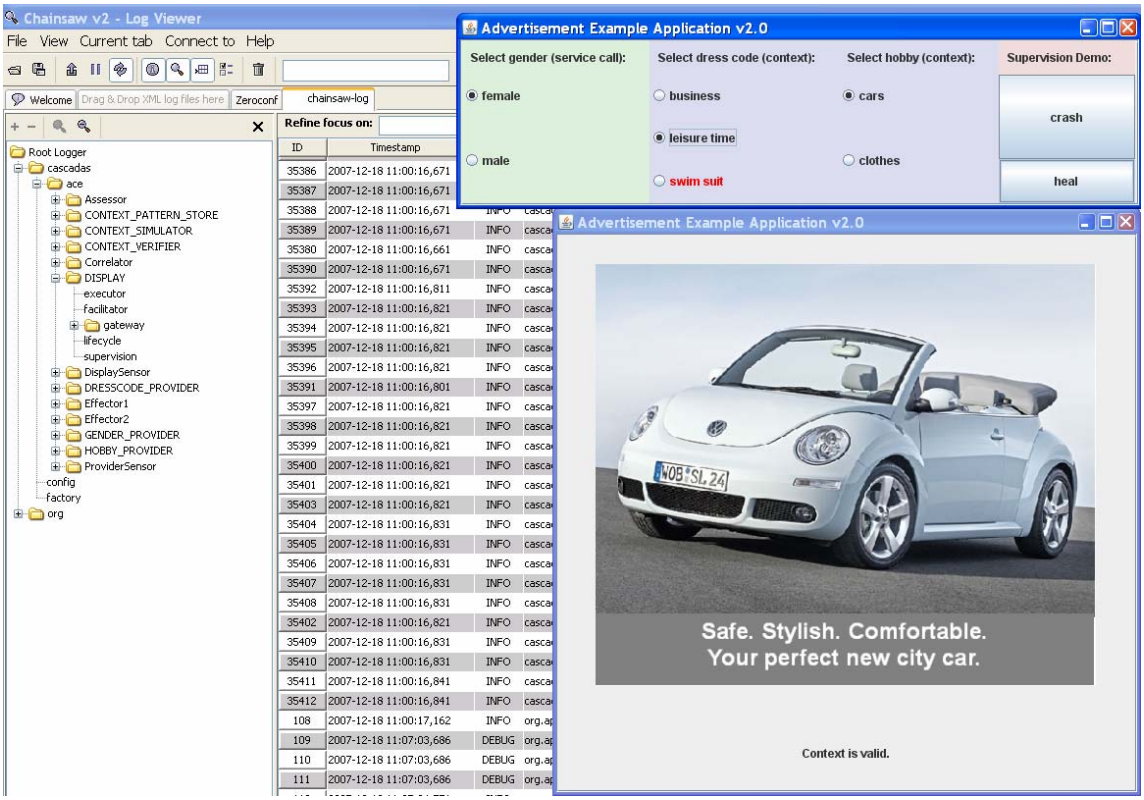


Figure 22: screenshot of the advertisement application

5.2 Supervision in the Advertisement Example

Figure 23 shows the integration of the supervision ACEs with the example logic ACEs in greater detail.

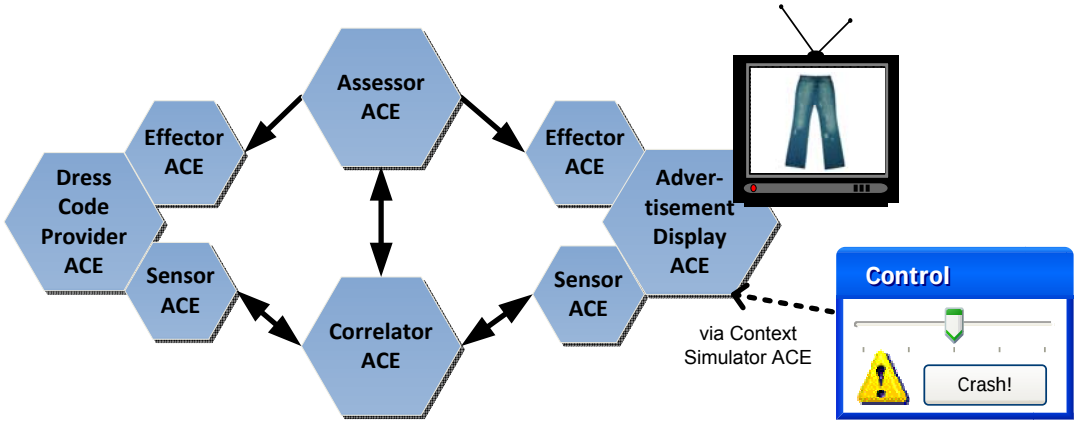


Figure 23: Integration of Supervision

Supervision capabilities are demonstrated by issuing a "crash" command using the context simulator. The command is triggering the advertisement display ACE to pretend a crashed state by stopping to receive information updates from the provider ACEs. This event will be reported to the correlator ACE by a sensor ACE that is constantly monitoring the dress

code provider ACE's operational behaviour. The correlator ACE recognises the critical event and instructs the assessor ACEs to take countermeasures by restarting the advertisement display ACE's Gateway and changing the state of the dress code provider to successfully cope with the changed situation.

5.3 Context Verification in the Advertisement Example

In our example application, context verification is a service that is requested by the advertisement display ACE. The data to be verified are a set of sensor values that is, based on reference pattern analyses, classified as valid or invalid.

The reference context patterns (sets of sensor values) needed for deciding whether a context is valid or not are stored in the context verificator ACE at the moment. This is subject to change. In the future, reference data will be stored in Knowledge Atoms or Atom Repository as they are provided by WP5. Context verification is described in detail in [D5.3].

Including context verification in the way described above, we integrate the context verification principle performed by active components of the Knowledge Network. This demonstrates the applicability of the CASCADAS Toolkit to the Knowledge Network principles on one specific application example.

What is not included in our example so far are enhanced knowledge organisation mechanisms. Those could be an extension to our advertisement application. In detail, knowledge organisation could here mean that complex context patterns are analysed and parts of them are associated with corresponding context storage ACEs. Let us for example regard the context pattern

"gender:female, dress code:business, hobby:cars".

This is not only a member of the context group "gender, dress code, hobby" but parts of this pattern as well belong to the groups "gender, dress code", "gender, hobby", and "dress code, hobby". An advanced knowledge organisation mechanism could figure out these additional relations and could aggregate the data in a suitable Knowledge Container.

6 Tutorial: Programming and Executing ACEs

The concept of programming ACEs using the CASCADAS Toolkit has not changed since deliverable D1.2. An ACE programmer has to create Self Models, specific functionalities, as well as descriptor and configuration files. What massively changed are the syntax (both, Self Model and functionality description syntax) and the set of available features in terms of architecture and common functionalities. This is the reason for significant differences to what was described in [D1.2]. The following section presents a tutorial on how to program ACE based applications using the current version of the CASCADAS Toolkit.

When programming ACEs developers have to keep in mind the following:

- An ACE is an Autonomic Communication Element that provides services to other ACEs or uses services from other ACEs or both.
- A Service might be provided by a group of ACEs where each ACE contributes only a part (sub-service) of the overall service provided by the group.

- An ACE application usually consists of several ACEs which might reside in multiple, distributed and remote execution environments.
- ACEs use the GN-GA protocol for service discovery.
- Inter ACE communication is handled directly between two ACEs using contracts.
- The entire ACE communication, both GN-GA and contract based, is asynchronous.
- An ACE communicates using events. (The entire communication is event based.)
- An ACE Plan is a state machine.
- A Self Model defines how to create and modify ACE Plans. It defines different ACE behaviours.
- Each ACE provides a set of common functionalities which can be used.
- An ACE action is defined as calling a local functionality.

These are a few main statements about ACEs developers should always keep in mind. In more general view, an ACE can be seen as a container that comprises all basic functionalities like life cycle functionalities, communication functionalities and a set of common functionalities.

Developers do not need to care about how the communication between ACEs is performed. Both ACE discovery process using GN-GA as well as contracting which provides direct communication between ACEs are provided and handled by the ACE itself. The same applies for the ACE life cycle. The entire life cycle process is implemented and executed by the ACE.

All developers need to do in order to create a new ACE is to implement its specific functionalities and to specify the ACE operation logic within the Self Model of the ACE (see yellow parts in the Figure 1).

6.1 Developing ACE Specific Functionalities

The goal of every ACE is to provide certain services to other ACEs. A service can be any functionality or information an ACE can provide. Controlling a power switch in the living room, providing the current value of an sensor or providing the telephone number for a user name are only a few examples of how an ACE service might look like.

A service can be considered as functionality or a set of functionalities an ACE implements. All ACE functionalities are implemented as Java classes. In addition to the Java implementation, a functionality description file is required (see chapter 3.5.2) where the access to the specific functionality is specified.

6.1.1 Functionality Java Classes

The ACE functionality implementation is a regular Java class. It must have a default (no-arg) constructor and a public method that is basically the interface to the functionality. It can use any other Java classes and features that are required.

For example, an ACE functionality that provides a user's phone number as String could look like this:

```
public class TelephoneProvider {
  /** Default no-args Constructor */
  public TelephoneProvider(){
    //Default Constructor
  }

  /** Method that provides the functionality */
  public String getPhoneNumber(String userName){
    String phoneNumber;
    //get phone number from the DB ....
    return phoneNumber;
  }
}
```

In the example above, the method `getPhoneNumber(String userName)` is the interface to the functionality. Before the Executor can call it, the developer needs to specify it within the service description file. This will be explained in the next section.

How to access the session objects?

Every ACE maintains the global and execution session objects (cf. chapter 3.5.6). The global session object remains active during the ACE life time whereas the execution session remains for the time of the Plan execution. There is one execution session object per ACE Plan.

In order to access the global or execution session objects within the specific functionality, the functionality must implement the `cascadas.ace.functionality.service.SessionAware` interface, which makes it possible for the Repository to forward the session objects to it.

```
public class TelephoneProvider implements SessionAware{
  Session executionSession;
  Session globalSession;

  /**This method sets a local reference to the executionSession*/
  public void setExecutionSession(Session executionSession) {
    this.executionSession = executionSession;
  }

  /**This method sets a local reference to the globalSession*/
  public void setGlobalSession(Session globalSession) {
    this.globalSession = globalSession;
  }

  /** Method that provides the functionality */
  public String getPhoneNumber(String userName){
    String phoneNumber;
    //get phone number from the DB ....
    /*Put phone number into execution session*/
    executionSession.put("users_phone",phoneNumber);
    return phoneNumber;
  }
}
```

How to access the CallWideContext within the specific functionality?

In addition to session objects, every ACE maintains the `CallWideContext` (cf. chapter 3.5.6.2). `CallWideContext` is a data storage facility (contains key-value pairs) that remains active during the functionality call: from the point when the Repository finds the requested functionality until the point when all output events are sent back.

`CallWideContext` can be used in order to pass the data to the event mapper for event creation or to the event mapping in order to initialise the output event with the desired parameters (cf. chapter 3.5.2.4).

In order to access the `CallWideContext` within the specific functionality, the functionality must implement the `cascadas.ace.functionality.service.CallWideContextAware` interface, which makes it possible for the Repository to forward the `CallWideContext` to it.

```
public class TelephoneProvider implements CallWideContextAware{
  CallWideContext callWideContext;

  /**This method sets a local reference to the CallWideContext*/
  public void setCallWideContext(CallWideContext callWideContext) {
    this.callWideContext = callWideContext;
  }

  /** Method that provides the functionality */
  public void getPhoneNumber(String userName){
    String phoneNumber;
    String altPhoneNumber;
    //get phone number from the DB ....
    /*Put phone number into execution session*/
    callWideContext.put("phone_number",phoneNumber);
    callWideContext.put("alternative_phone_number",altPhoneNumber);
  }
}
```

How to create a custom event mapper?

Calling the functionality usually results in sending events. For example in order to send the return values to the requester the functionality descriptor has to specify a `ServiceResponseEvent` to be sent using event mapping and initialise it with the required parameters (cf. chapter 6.1.2).

If the direct event mapping is not enough (e.g. the number of outgoing events is not known when the descriptor XML is created), a custom event mapper has to be created. Such a Custom event mapper allows you to create events from scratch, initialise them with additional parameters or set contracts.

An event mapper java class must implement the `cascadas.ace.functionality.service.OutputEventMapper` interface, which makes it possible for the Repository to access all output events which have been created by the mapper and send them to the bus.

```
public class PhoneNumberEventMapper implements OutputEventMapper {

  public PhoneNumberEventMapper() {
    //Default Constructor
  }

  /** Method will be called by the Repository in order to produce
   *  the output events of the functionality.
   */
  public Set<Event> createOutputEvents(CallParameters params,
     Session executionSession, Session globalSession,
     CallWideContext callWideContext) {
    Set<Event> ret = new HashSet<Event>();
    Contract c = (Contract) params.get("phoneContract");
```

```
    ServiceResponseEvent evt;
    evt = new ServiceResponseEvent(c);
    evt.addParam("primary_phone_number", callWideContext.get("phone_number"));
    evt.addParam("second_phone_numebr",
callWideContext.get("alternative_phone_number"));

    ret.add(evt); //Add ServiceResponseEvent to the set of output events.
    return ret;
  }
}
```

In the example above, the `PhoneNumberEventMapper` will create a `ServiceResponseEvent` which contains two parameters `primary_phone_number` and `second_phone_number`. While invoking the event mapper's `createOutputEvents()` method, the Repository will send the event out.

In the service call sequence, the Repository first calls the ACE specific functionality and afterwards calls the specified mapper. All parameters which are set to the `CallWideContext` within the specific functionality will be passed to the event mapper.

These are the parameters which will be passed to the mapper's `createOutputEvents()` method:

- `CallParameters params` (Output parameter as defined in the functionality description.)
- `Session executionSession` (The Plan execution session.)
- `Session globalSession` (The ACE global session.)
- `CallWideContext callWideContext` (The `CallWideContext` including all parameters which have been set within the specific functionality.)

How to create custom events?

An ACE distinguishes between internal and external events. Internal events are handled by the ACE itself. A functionality could for example send an internal custom event in order to trigger modification of the ACE behaviour. The external events will be delivered to the remote ACE.

Internal events derive from the `cascadas.ace.event.AceLocalEvent` class whereas the external events derive from the `cascadas.ace.event.Envelope` class. Developers are free to use the standard ACE events or to create custom events. Any event can be sent from the specific functionality using the event mapper or while specifying its mapping in the functionality description file (cf. next chapter).

How to write log messages from the specific functionality?

Logging facility might be very helpful in order to understand what happens within the ACE. The ACE Toolkit provides a powerful and easy to use logging facility which is capable of writing log messages to multiple outputs (cf. chapter 2.1 and chapter 6.3.2).

In order to write log messages to the standard ACE logger the specific functionality class must implement the `cascadas.ace.functionality.service.LogAware` interface.

```
public class TelephoneProvider implements LogAware {
  Logger logger;

  public TelephoneProvider() {
    //Default Constructor
```

```
  }

  /**This method sets a local reference for the logger*/
  public void setLogger(Logger logger){
    this.logger = logger;
  }

  /** Method that provides the functionality */
  public void getPhoneNumber(String userName){
    //get phone number from the DB ....
    logger.severe("Unable to connect to database");
  }
}
```

ACE logging uses `java.util.logging` facilities. Log levels used by the ACE loggers are the same as defined by the `java.util.logging`.

How to handle custom threads within the specific functionality?

Sometimes a specific functionality requires additional threads for fulfilling a desired task. In order to ensure the consistency with the ACE life cycle (e.g. stop or move) an ACE provides developers the ability to use a thread pool. The ACE autonomously manages the thread pool and decides what to do with threads when a life cycle action is performed. This way, the ACE does not leave zombie threads in the Java Virtual Machine after being terminated.

Specific functionalities should use the thread pool for starting their custom threads. In order to access the thread pool the specific functionality must implement the `cascadas.ace.functionality.service.ThreadPoolAware` interface.

```
//======= Specific functionality class that uses multithreading
public class TelephoneProvider implements ThreadPoolAware {
  ThreadPool threadPool;

  public TelephoneProvider() {
    //Default Constructor
  }

  /**This method sets a local reference for the Thread Pool*/
  public void setThreadPool(ThreadPool pool){
    this.threadPool = pool;
  }

  /** Method that starts a thread in the ACE thread pool */
  public void myMethod(){
    Runnable myThread = new BasicThread1();
    threadPool.execute(myThread);
  }
}

//======= A Basic thread class
class BasicThread1 implements Runnable {
  public void run() {
    //Do something
  }
}
```

6.1.2 Functionality Descriptor

In order to call a specific functionality, developers must provide its functionality description. The functionality description specifies parameters like functionality id, input and output parameters, class and method to be called as well as event mappers if used. More details on the functionality description model can be found in chapter 3.5.2.

The ACE functionality descriptor is an XML file that specifies how to call the functionality and is available in the ACE configuration folder (subfolder /repo). Each ACE functionality is specified in a separate XML file. The file names can be freely chosen. The Repository will read all XML files that are available in the specified /repo subfolder and will initialise the parameters accordingly. The functionality description syntax is specified in `conf/dtd/functionality.dtd` file.

Functionality description contains:

Functionality ID	Functionality ID that is unique for all functionalities available within the ACE. The functionality ID is used within the Self Model in order to invoke the functionality.
Black Box Description	Specifies all input and output parameters which are required for calling the functionality. The parameters must be in sync with the input parameters of the java method that will be called. The parameter number and type can be also variable.
Simple call details	Specifies the name of the Java Class that should be instantiated and the name of the method that will be called.
Output Event Mappings	Output event mappings specification is only required if specific functionality needs to send an event. It specifies which event to send and parameters it should be initialised with.

```
<functionality id="hello_world_service">
  <black-box-description>
    <input>
      <param name="userName" type="java.lang.String"/>
    </input>
  </black-box-description>
  <simple-call-details class-name="example.HelloWorld" method-name="sayHello"/>
  <output-event-mappings/>
</functionality>
```

The example above shows a very simple `hello_world_service` which has been implemented in the `example.HelloWorld` class in the `sayHello(String userName)` method. Neither mappers nor mappings are specified which means the service call will not send any events. Using this service description the `hello_world_service` can be called from the Self Model.

Black Box Description Input Parameters

The input parameters have to be specified within the `<input>` section of the functionality black box description. Either a predefined set of parameters with the parameter name and type `<param name="" type=""/>` can be used, or the amount and types of parameters can be kept variable.

If a *predefined set of input parameters* is used, every input parameter must be specified as it is required by the java method call.

```
<functionality id="get_phone_number">
  <black-box-description>
    <input>
      <param name="firstName" type="java.lang.String"/>
      <param name="surName" type="java.lang.String"/>
    </input>
  </black-box-description>
  <simple-call-details class-name="example.PhoneBook" method-name="getTelNumber"/>
  <output-event-mappings/>
</functionality>
```

In the example the `get_phone_number` service requires two input parameters `firstName` and `surName`.

If the *amount of parameters needs to be kept variable*, then the `<varargs/>` element must be specified within the `<input>` section of the functionality description file.

```
<functionality id="get_phone_number2">
  <black-box-description>
    <input>
      <varargs/>
    </input>
  </black-box-description>
  <simple-call-details class-name="example.TelephoneProvider"
method-name="getTelNumberVararg"/>
  <output-event-mappings/>
</functionality>
```

In the example above the `get_phone_number2` service can be called with a different amount of arguments. If the input parameters specifies `<varargs/>` the Java method must accept an Array of `cascadas.ace.functionality.service.Argument` objects. A Java implementation for the example above could look like this:

```
import cascadas.ace.functionality.service.Argument;

public class TelephoneProvider {

  public void getTelNumberVararg(Argument[] args) {
    for (Argument a : args) {
      String paramName = a.getName();
      String paramvalue = a.getValue();
      //Do something
    }
  }
}
```

The parameter specified within the `<output>` section defines the name of the parameter where the method's return value (if any) will be stored. The input and output parameters can be any Java objects.

Output Event Mappers and Mappings

In case specific functionality needs to send events (for example if it returns a value, a `ServiceResponseEvent` has to be sent), output event mappings must be specified within the service description. There are two possibilities when specifying output events. A functionality can send either predefined output events like `ServiceResponseEvent` for

example and initialise them with the desired parameters, or it can call a custom event mapper that will create events from scratch and send them.

How to send return value using a predefined event?

Within the <mapping> section, developers can specify the predefined event to be sent when the functionality is invoked. The <value> elements specify the event initialisation parameters.

```
<functionality id="get_phone_number_service">
  <black-box-description>
    <input>
      <param name="userName" type="java.lang.String"/>
    </input>
    <output name="phone_numer" type="java.lang.String"/>
  </black-box-description>
  <simple-call-details class-name="example.TelephoneProvider"
method-name="getPhoneNumber"/>
  <output-event-mappings>
    <mapping                                              target-role="provider"
event="cascadas.ace.event.ServiceResponseEvent">
      <value ref="phone_numer"/>
    </mapping>
  </output-event-mappings>
</functionality>
```

In the example the `get_phone_number_service` corresponds to the `public String getPhoneNumber(String userName)` method. The return value of the method is stored as the `phone_number` output parameter. When invoking the `get_phone_number_service` the `ServiceResponseEvent` initialised with the `phone_number` output parameter will be sent to the requester.

In addition to the return value as presented in the example above, the output event can be initialised with values from:

- Call Wide Context using `?callWideContext://param_name`
- Execution Session using `?executionSession://param_name`
- Global Session using `?globalSession://param_name`

```
<output-event-mappings>
  <mapping target-role="" event="cascadas.ace.event.ServiceResponseEvent">
    <value ref="?callWideContext://param_name"/>
    <value ref="?executionSession://param_name"/>
    <value ref="?globalSession://param_name"/>
  </mapping>
</output-event-mappings>
```

Please look at chapter 3.5.2 for more details.

How to send multiple predefined events when calling the functionality?

In case the functionality call needs to send multiple events, multiple <mapping> sections have to be specified within the <output-event-mappings> section.

```
<output-event-mappings>
  <mapping target-role="" event="cascadas.ace.event.ServiceResponseEvent">
    <value ref="?callWideContext://param_name"/>
  </mapping>
  <mapping target-role="" event="example.SampleEvent">
  </mapping>
```

```
</output-event-mappings>
```

In the example above the service call will result in two events, the `ServiceResponseEvent` and the `SampleEvent`.

How to use a custom event mapper?

As explained in the previous chapter, developers can also write a customer event mapper that sends events. In order to use a custom mapper, the developer needs to specify it within the `<mapper>` section in the `<output-event-mappings>`.

```
<output-event-mappings>
  <mapper mapper-class="example.PhoneNumberEventMapper"/>
</output-event-mappings>
```

The Repository will instantiate the mapper class and will call its `createOutputEvents()` method. All events returned by the method will be sent out.

6.2 Creating the ACE Self Model

Chapter 6.1 explained how to create specific ACE functionalities and how to create their functionality description in order to use them. This chapter explains how to create the ACE Self Model which basically describes the ACE behaviour. Based on the Self Model definitions, ACE Plans will be created that use (invoke) functionalities, both specific and common functionalities, which are available within the ACE.

A Self Model is an XML file (usually `/model/selfmodel.xml` is used) that defines the ACE behaviour. In the ACE Self Model developers have to specify when and how to create and modify different ACE Plans. Each ACE Plan performs certain tasks and all ACE Plans together form the ACE behaviour.

Here are some statements developers should keep in mind when creating Self Models:

- A Self Model contains all possible Plans that might be created by an ACE.
- A Self Model must have one default Plan.
- The default ACE Plan is the starting point of the ACE execution process.
- A Plan can initiate the creation of further Plans.
- An ACE Plan is a state machine.
- Every Plan in the Self Model contains: full set of states, full set of transitions as well as creation and modification rules which define how to combine the states and transitions into a state machine.
- States define different states the ACE Plan execution has to go through.
- Transitions specify the actions which need to be performed as well as conditions under which they should be triggered.
- Plan creation and modification rules are written in standard RuleML language.

The Self Model syntax is described in chapter 3.4.2. It s required to read it first before continuing with this chapter.

For demonstration purposes a sample Self Model of an ACE that contains only one Plan which provides the phone book service is used.

```
<!DOCTYPE selfModel SYSTEM "../../../dtd/selfmodel.dtd">
<selfModel>
```

```
<plan id="Plan1" default="true">
  <description>Self Model of the Phone Book ACE</description>
  <states>
    <state id="state1" >
      <friendly_name>InitializationState</friendly_name>
      <desirability_level>1</desirability_level>
    </state>
    <state id="state2" >
      <friendly_name>Ready</friendly_name>
      <desirability_level>1</desirability_level>
    </state>
    <state id="state3" >
      <friendly_name>PhoneNumberProvided</friendly_name>
      <desirability_level>1</desirability_level>
    </state>
  </states>
  <transitions>
    <transition id="tr1">
      <description>Initialize the DB connection</description>
      <source>state1</source>
      <destination>state2</destination>
      <priority/>
      <trigger>@auto</trigger>
      <guard_condition/>
      <action>init_db_connection</action>
    </transition>
    <transition id="tr2">
      <description>Answer GN=phone_number</description>
      <source>state2</source>
      <destination>state2</destination>
      <priority/>
      <trigger>cascadas.ace.event.GoalNeededEvent</trigger>
      <guard_condition>
      EQUALS(?inputMessage://goalName,phone_number)</guard_condition>
      <action>gn_answer_service(goal=?inputMessage://goal,
        serviceName=get_phone_number_service,
        myAddress=?globalSession://aceAddress)
      </action>
    </transition>
    <transition id="tr2">
      <description>Answer GN=phone_number</description>
      <source>state2</source>
      <destination>state3</destination>
      <priority/>
      <trigger>cascadas.ace.event.ServiceCallEvent</trigger>
      <guard_condition>
        EQUALS(?inputMessage://serviceName,get_phone_number_service)
      </guard_condition>
      <action>
        get_phone_number_service(userName=?inputMessage://user_name)
      </action>
    </transition>
  </transitions>
  <creationRuleML>
    <Assert>
      <And>
        <Atom closure="universal" >
          <Rel>createState</Rel>
          <Ind>state1</Ind>
          <Ind>state2</Ind>
          <Ind>state3</Ind>
        </Atom>
        <Atom closure="universal" >
```

```
            <Rel>initState</Rel>
            <Ind>state1</Ind>
          </Atom>
          <Atom closure="universal" >
            <Rel>createTransition</Rel>
            <Ind>tr1</Ind>
            <Ind>tr2</Ind>
            <Ind>tr3</Ind>
          </Atom>
        </And>
      </Assert>
    </creationRuleML>
    <modificationRuleML>
    </modificationRuleML>
  </plan>
</selfModel>
```

The Self Model shown in the example above will create one ACE Plan with 3 states and 3 transitions. The initial state is state1 which is the starting point of the Plan execution. The ACE model describes a service provider ACE. This service provider ACE waits for a Goal Needed to arrive and responds with a Goal Achievable if the goal can be fulfilled.

Action in terms of the Self Model means invoking a local functionality and is defined within the transition. An action has to be specified within the `<action>` element. It has to contain the functionality ID as specified within the functionality description with all required parameters including initialisation values within the brackets. For example: `<action>get_phone_number_service(userName=?inputMessage://user_name )</action>` calls the specific functionality `get_phone_number_service` and initialises its input value `username` with the `user_name` parameter from the input message.

Beside the specific functionalities, there is also a set of predefined functionalities called "common functionalities" (cf. chapter 3.5.4.1) which can be used within the Self Model of every ACE. In the example above `gn_answer_service` is the common functionality that allows sending the `GoalAchievableEvent` as the response to the `GoalNeededEvent`.

The event type specified within the `<trigger>` element specifies the event type that might trigger the transition. The transition will be evaluated only if this event type arrives. Within `<guard_condition>` developers can define the condition that needs to be fulfilled in order to run the action. After the action has been successfully executed, the Plan execution will move to the state defined within the transition's `<destination>` element.

Finally, you have to define the Plan creation and modification rules which will create or modify Plans. Please look at the chapter 3.4.2.2 for more details how to create the RuleML.

It is important to realise that in order to program an ACE you do not write a main application.

You have to write a selfmodel.xml file specifying the global behaviour (i.e. business logic) of your ACE.

Then you write separate methods (i.e. functionalities) specifying the basic individual actions the ACE has to undertake (you can also have a general library of these functionalities).

Then you hook-up such functionalities in the Self Model, so that the functionalities will be called when the Self Model is executed.

All the bindings and all the machinery required to run the Self Model are the core of the ACE architecture and a developer should not care about them.

6.3 Executing the ACE Application

6.3.1 Setting Up the ACE Application Environment

All application configuration files are located in the `/conf` folder. The subfolder structure is used as follows:

```
conf/aces/
    /commonfunctionalities/
    /dtd/
    /aces.xml
    /setting.properties
```

The `settings.properties` and `aces.xml` contain the application configuration parameters. The `settings.properties` specifies the application environment settings like the location of the common functionalities folder and location of the `aces.xml` file. Parameters for ACE logging as well as DIET and REDS parameters have to be defined here. Please look at the comments of a sample `settings.properties` file for more details.

The `aces.xml` file specifies all ACEs which will be created when starting the application. ACEs will be initialised and started in the same sequence as specified within the `aces.xml` file. Each ACE has to be specified within the <ace> element with its name and path to its ace type description file.

```
<aces>
  <ace name="display" type="conf/aces/advertizement_display/ace-type.xml"/>
  <ace name="age_provider" type="conf/aces/age_provider/ace-type.xml"/>
  <ace name="weather_provider" type="conf/aces/weather_provider/ace-type.xml"/>
</aces>
```

This basically states that our application will be composed of three ACEs. Each ACE is characterised by a name and a type that is a reference to another XML configuration file.

Each ACE must have its "type description file" where the path to the Self Model file and the path to the specific functionality folder with the functionality description files are defined. Here is the example `ace-type.xml` from the `weather_provider` ACE.

```
<properties>
<comment></comment>
<entry key="model">./conf/aces/weather_provider/model/selfmodel.xml</entry>
<entry key="functionalDescription">./conf/aces/weather_provider/repo/</entry>
</properties>
```

The two most important elements of this file are the <entry> tags. They indicate a reference to the two fundamental building blocks of an ACE: the Self Model and the Functionality Repository.

Specifically, `<entry key="model">` specifies the location of the `selfmodel.xml` describing how the ACE Plans will be executed.

Similarly, `<entry key="functionalDescription">` specifies the directory that contains the description of the ACE functionalities.

If the application requires multiple instances of the same ACE type (for example multiple weather provider ACEs), developers can specify this within the `aces.xml` file. Please note that the ACE names in the `aces.xml` file must be unique in order to distinguish between them.

```
<aces>
  <ace name="display" type="conf/aces/advertizement_display/ace-type.xml"/>
  <ace name="age_provider" type="conf/aces/age_provider/ace-type.xml"/>
  <ace name="weather_provider1" type="conf/aces/weather_provider/ace-type.xml"/>
  <ace name="weather_provider2" type="conf/aces/weather_provider/ace-type.xml"/>
  <ace name="weather_provider3" type="conf/aces/weather_provider/ace-type.xml"/>
  <ace name="weather_provider4" type="conf/aces/weather_provider/ace-type.xml"/>
</aces>
```

The example above will create four weather provider ACEs of the same type.

6.3.2 Running the ACE Application

ACEs are created through an instance of the class `cascadas.ace.AceFactory` by analysis of application configuration files described in the previous chapter. The `AceFactory` contains the the `AceFactory.main()` method. You can pass the path to your custom `settings.properties` configuration file. When no argument is passed, the factory will use the `./conf/settings.properties` by default.

With respect to the settings from the `settings.properties` file, `AceFactory` will create specified ACEs by resolving their model definitions and specific functionality mappings as defined in the corresponding XML configuration files (cf. chapter 6.3.1).

Can I see the ACEs?

In order to visualise your ACE application and see the running ACEs you should either use the Elvis visualisation tool or the VISA (Visualising ACEs) tool.

Elvis is a DIET agent visualisation tool which has been provided together with the DIET Agents platform. It shows you the ACEs in a DIET specific manner including all supporting components like mirror agents and connections for example (see Figure 24).

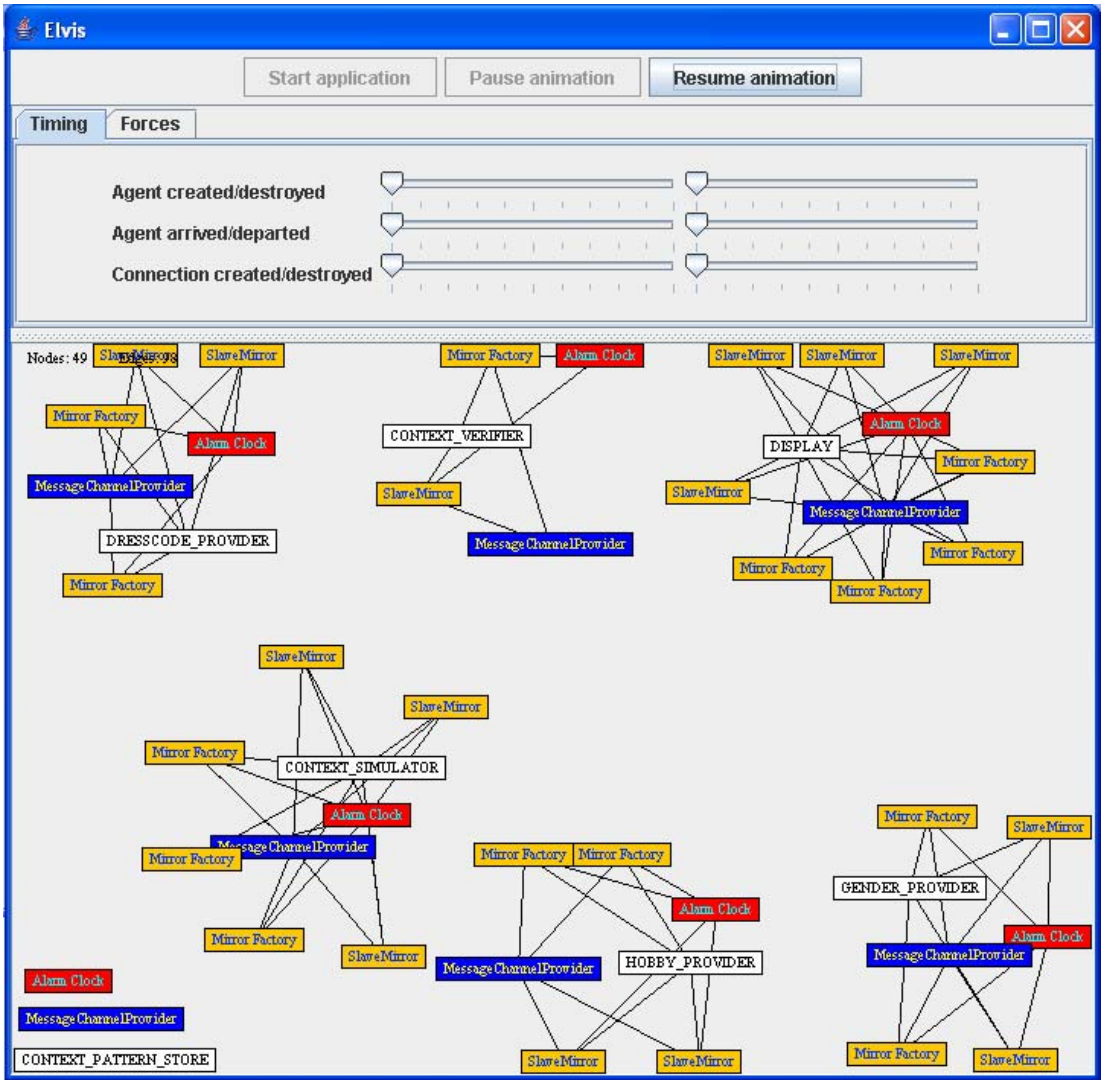


Figure 24: Visualising ACEs using Elvis

In order to visualise the application using Elvis, you have to start the Elvis tool passing the reference to the `AceFactory` as parameter to it: `com.btexact.diet.elvis.Elvis cascadas.ace.AceFactory`.

VISA (Visualising ACEs) was developed in order to visualise ACEs in an ACE-specific way, without additional "DIET specific things" like Elvis does. The VISA tool is a java applet that can be started using a web browser or applet viewer. It uses application log messages and displays ACEs based on log messages.

In order to use VISA the log level must be set to at least INFO, and an additional socket log output that sends log messages in XML format to port 4001 must be specified within the `settings.properties` file.

```
logging.outputs = socket,localhost:4001,xml
```

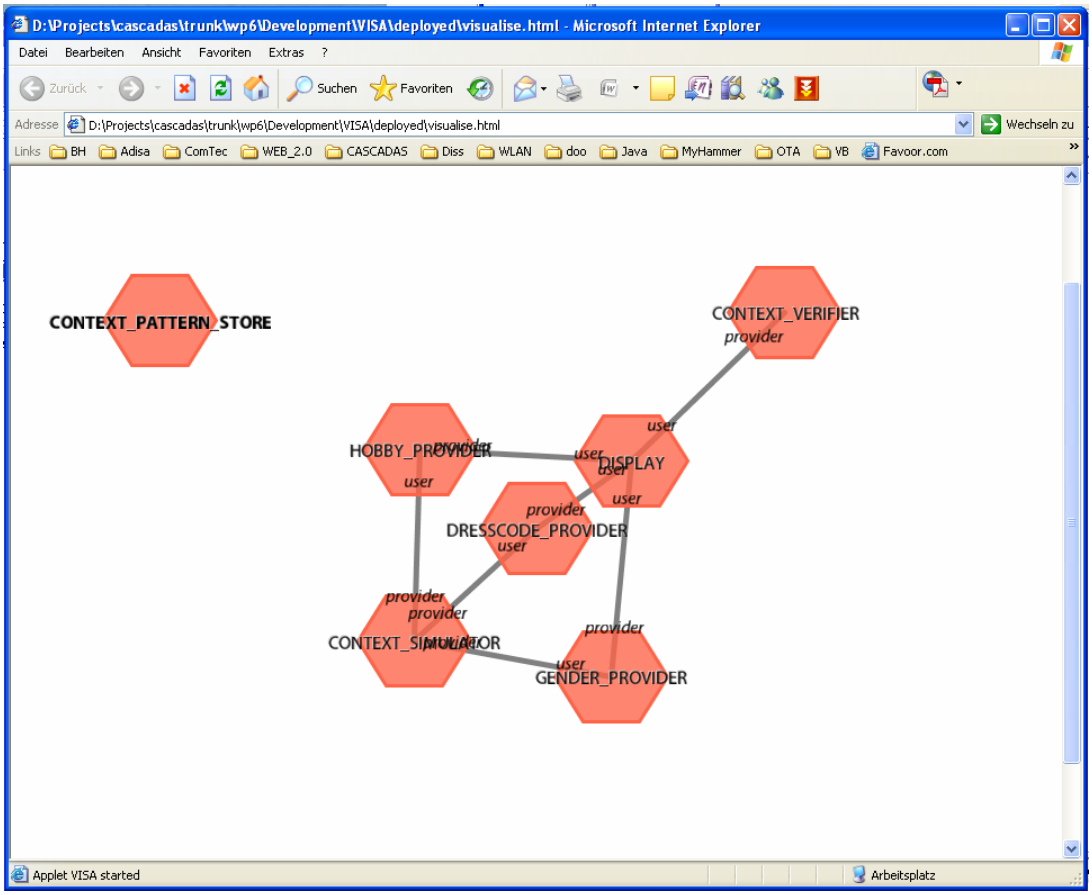


Figure 25: Visualising ACEs using VISA

VISA shows all ACEs that are involved in the ACE application independent of their location.

How do I view the log messages?

The new ACE Toolkit provides an extensive logging facility that can produce various log outputs like for example logging to the console or to a socket. The best way to view the log messages is to use the Chainsaw log viewer [Chsw]. It is an open source toolkit developed by the apache software foundation.

In order to use the Chainsaw log viewer, you have to initialise it with the `misc/chainsaw-config.xml` file. The log viewer will create a socket receiver to port 4000 and will wait for log messages to be displayed. In the `settings.properties` file the corresponding log output must be specified.

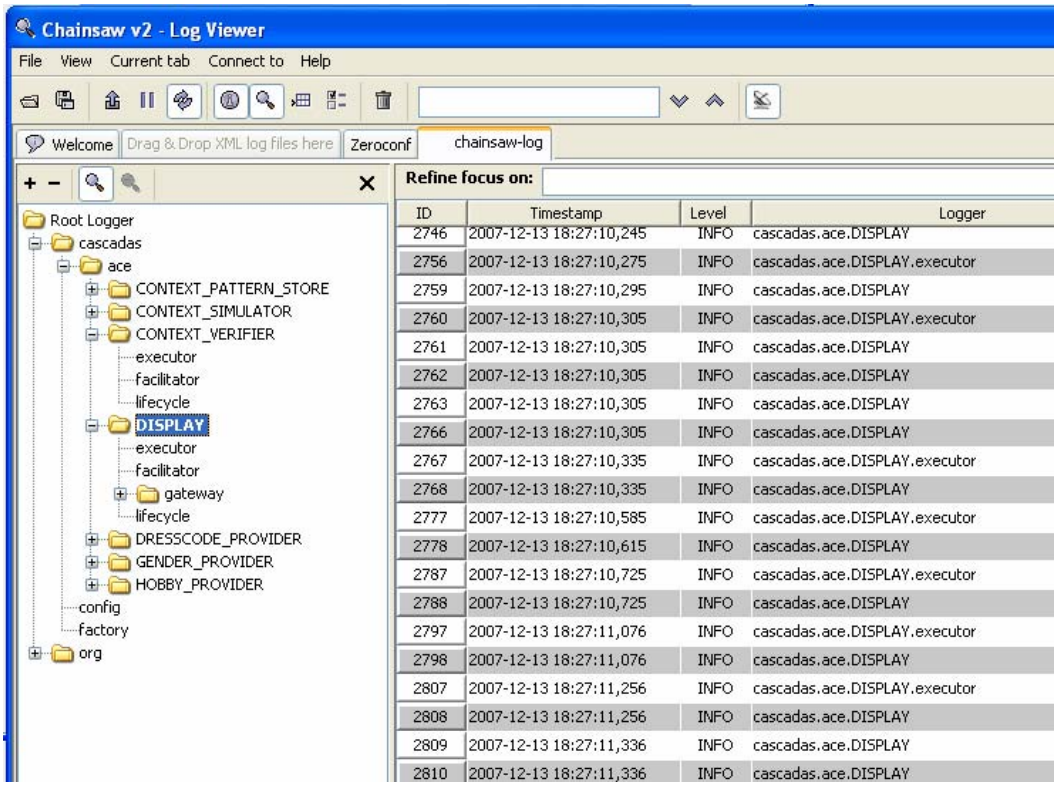


Figure 26: Viewing ACE log messages using Chainsaw

Chainsaw will show a folder tree with log messages placed within different ACE folders. As presented in the figure above, this tool allows you to filter and view log messages per ACE name. The ACE internal messages can be viewed on ACE organ basis. This allows you to analyse the ACE execution in a very convenient way.

7 Conclusion and Outlook

The document at hand is part of the CASCADAS deliverable D1.3, first prototype integration. It constitutes the documentation accompanying the source code of the integrated CASCADAS Toolkit. In this document we have described in detail the ACE component model and the features and mechanisms of each of the organs an ACE is composed of. Furthermore, the Toolkit's mechanisms to support security, supervision, aggregation, and Knowledge Networks have been described. The document also includes a description of the example application included in the Toolkit as well as a tutorial on how to program ACEs using the Toolkit.

Apart from providing a stable basic run-time environment for the creation and execution of ACEs the CASCADAS Toolkit also possesses a variety of advanced features as described in this document. It is also integrated with the concepts and tools provided by other work packages. The example application included in the Toolkit shows some of the Toolkit's potential in a practical manner for the project's pervasive advertisement scenario.

Therefore, in summary, the current release of the CASCADAS Toolkit is fully in line with what was expected to be available at this point in time. For the future, it is planned that the Toolkit will – besides continuously being adapted to the requirements from other work

packages – be enhanced by additional features for contract negotiation, advanced lifecycle management including deployment decision making, ACEs' self-reflection capabilities by means of semantics as well as further tools to facilitate ACE development.