Mapping External Functionalities into Autonomic Services

Borbala Katalin Benko
Budapest University of Technology and Economics
Magyar tudosok krt. 2.
H-1117 Budapest, Hungary
+36-1-463-3278
bbenko@hit.bme.hu

Robert Schulcz
Budapest University of Technology and Economics
Magyar tudosok krt. 2.
H-1117 Budapest, Hungary
+36-1-463-3284
schulcz@hit.bme.hu

ABSTRACT
In this paper, we present the functionality mapping model that is used in the CASCADAS ACE model: external code libraries are transformed into services of an autonomic element.

CASACDAS is an EU 6th Framework IST-FET project in the field of situated, autonomic communication; various research topics are organized around a common abstraction called Autonomic Communication Element (ACE). When designing this common abstraction, a big challenge is to define a natural, non-complex but flexible and expressive way for transforming existing code (libraries) into autonomic services of an ACE. In this paper, we introduce, explain, and discuss the developed model, and compare them with existing functionality mapping technologies.

Keywords
Functionality Repository, Autonomic Communication Element, ACE, CASCADAS

1. INTRODUCTION

1.1 The CASCADAS Project
CASCADAS – which is an EU 6th Framework IST-FET project – researches situated, autonomic technologies. Four different viewpoints (aggregation, knowledge management, self-supervision, security) are organized around a common component model. [2][5]

The Autonomic Communication Element (ACE) is the common abstraction used in CASCADAS to model situated, autonomic services; the services of the future are envisioned to be available via ACEs. [4]

In this paper we present the models and concepts applied in the Functionality Repository that is one of the five ACE organs (internal components). The functionality Repository is responsible for mapping the non-ACE codes (libraries) into ACE functionalities.

BIONETICS'07, December 10-13, 2007, Budapest, Hungary.

1.2 ACE Model
The Autonomic Communication Element has been designed on different levels (conceptual model, architectural model, component model) in this paper we are referring to the component model of those.

ACE is a basic building block; the system is formed by communicating and interacting ACEs. Communication is defined as asynchronous message sending.

Internally, an ACE consists of 5 organs[1]: Gateway, Bus, Facilitator, Reasoner, and Functionality Repository. The ACE executes a Plan. The actual Plan defines its reactive and proactive behavior: what to do with the incoming messages, and when to initiate an interaction. The plan is adapted to the actual internal and environmental conditions (ensuring self-awareness, environment-awareness, and adaptivity).

- The Facilitator is the core autonomic component. It creates the initial Plan at startup, then keeps track of the changing internal and external conditions, and modifies or re-generates the Plan according to a self-model.
- The Gateway is the single interface between the ACE and the environment. The Gateway supports multicast and unicast message sending. Received messages are forwarded to the Bus.
- The Bus is in some means a technological necessity: it is the only communication channel between the organs. The Bus provides subscription based message delivery (all subscribed organs will receive the Message that is sent to the Bus), in either synchronous or asynchronous manner. The Bus makes it possible to transparently supervise the ACE (the supervisor needs to access the Bus only in order to monitor/control the whole ACE).
- The Reasoner is the component that executes the Plan. It handles the incoming messages and invokes actions associated to certain steps of the plan.
- The Functionality Repository contains the invokable functionalities of the ACE: common functionalities that are available in all ACEs as well as type-specific functionalities

[1] The word "organ" was chosen (1) to avoid the re-use of the word "component" referring to one complete ACE and (2) to emphasize the intelligent, sometimes biologically or physiologically inspired aspects.

(services that a certain ACE instance is able to provide). Plan actions refer to the functionalities.

In other words, the ACE consists of two intelligent, reflective organs (the Reasoner controls the internal processes, the Facilitator provides constructive criticism and adaptivity); two communicational enablers (the Bus for the internal communication, the Gateway for the external communication), and one functionality container (Functionality Repository).

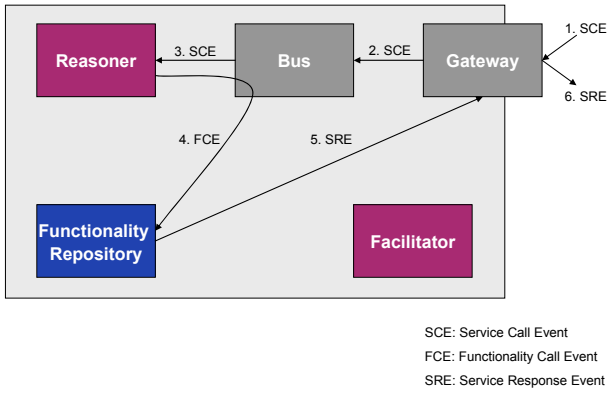

Figure 1. Internal message flow – service invocation

Let us follow an example message flow inside ACE1 when ACE2 (another ACE) invokes one of the services (Figure 1). The service call event arrives at the Gateway, gets forwarded to the Bus, and from there, reaches the Reasoner. The Plan in the Reasoner defines an action for the incoming service call event, meaning, a functionality call event is sent to the Repository (through the Bus). The Repository invokes the requested functionality, creates response events, and sends them to the Bus (internal response) and/or to the Gateway (external response).

This message flow pattern separates the control (Reasoner) and the effective actions (Repository) while enabling self-adaptation (the Facilitator can change the plan at any time). The Reasoner initiates the action, but the Repository is in charge of performing the calls and sending the response events.

2. THE FUNCTIONALITY REPOSITORY

The Functionality Repository is the organ responsible for the functionalities of an ACE. The Repository maintains a registry about the deployed functionalities, performs calls into them when requested, and sends the response events generated by the calls.

The Functionality Repository shows only reactive behavior: it invokes functionalities when requested, but never decides about the invocation itself and never schedules it on its own.

Although, the Repository may not seem to be a crucially innovative organ at the first sigh, it has raised many challenging questions. The most important problem is how to map external libraries into ACE-understandable functionalities (from the remote point of view: into services)[2] in a trouble-free, natural but expressive and flexible way.

2.1 How does the Repository work?

The life of the Functionality repository can be devided into two phases: startup and serving.

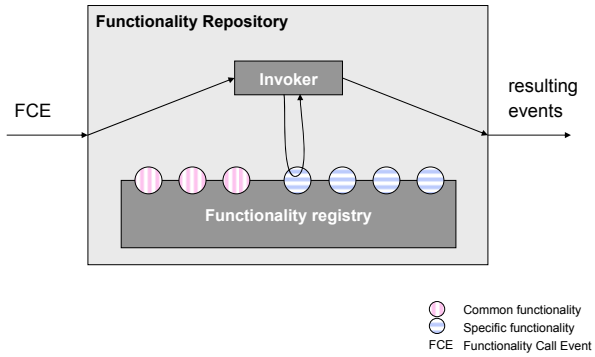

Figure 2. Functionality Repository

At startup, the Functionality Repository (Figure 2) dynamically loads the functionalities from the file system and also performs consistency check. Successfully loaded functionalities will be available throughout the lifetime of the ACE. After startup, the Repository reaches the serving phase, and serves the incoming calls.

The "input" of the Functionality Repository is the functionality call event (FCE) sent by the Reasoner. The FCE contains the functionality name, the call arguments, and the actual environment (sessions). By default, FCEs are handled in a non-blocking manner, in order to prevent long-lasting calls from blocking the whole ACE. Invocations are executed on separate threads.

The "output" of the Functionality Repository is a set of events, generated by the functionality invoked. The Repository sends the resulting internal events to the Bus, and hands the external events to the Gateway.

2.2 Requirements and Consequences

Requirements can be formulated two-folded: in terms of the mapping of external libraries into ACE functionalities; and in terms of the invocation process itself. In this paper we are concentrating on the mapping only.

There are two basic design goals about the functionality mapping.

1. Existing libraries should be easy to ACEify.

   It should be possible to transform an external library into an ACE functionality without source code modification. There

[2] By the terms "functionality" and "service" we mean two views of the same thing: from local point of view, we are speaking about functionalities, while the remote side sees them as services.

should be no mandatory superclass or other code-related restriction[3].

2. Explicitly designed-for-ACE functionalities should have maximal flexibility.

   When the functionality is specifically designed for ACEs, it should have access to as many ACE-related information as possible. Whenever the Repository detects that the functionality is capable of handling a particular information type, it should pass all related (state, execution environment and other) descriptors to it.

The first requirement naturally implies the usage of descriptors for the mapping. We use XML based descriptors, describing the name of the functionality, the call details and the output. In simple cases, transforming an existing library into an ACE functionality is not more complex than providing an XML descriptor for the mapping.

The second requirement refers the code level, so it is quite straightforward to mark the ability of dealing with ACE data on the code level as well. The implementation of a given super-interface guarantees that the functionality is ready to handle the variable. A separate super-interface is defined for each ACE-related variable; so that functionalities can express their very exact interest[4].

It is theoretically possible to differentiate between normal functionalities and those functionalities specifically designed for ACEs, but the difference only affects the "input-output range" of the functionality; in all other aspects the invocation process is the same.

3. FUNCTIONALITY MODEL

Functionality model describes the functionality on an abstract level; and besides of that, also depicts our understanding of the term "functionality". In the ACE, the model of a functionality is given in its XML descriptor (Functionality descriptor).

3.1 Three-layered Functionality Model

We defined a three-layered functionality model (Figure 3).

- Underlying call layer. The bottommost layer describes the underlying call sequence. It describes which methods of which classes to invoke as well as the calling order and the parameter list.
- Black-box layer. The second layer defines the black-box model of the complete call sequence: the input parameters (if any) and the output parameters of the whole calling process (in an even less mandatory way).
- Event layer. The third layer defines the input and the output of the call in means of events. The input event is always an FCE. The output is an arbitrary set of events (internal or external).

[3] Except for a mandatory default (no-arg) constructor.

[4] For example: if a functionality wants to access the ACE variable "x", it has to implement the XAware interface. Similarly, a functionality interested in both "x" and "y" should implement XAware and YAware.

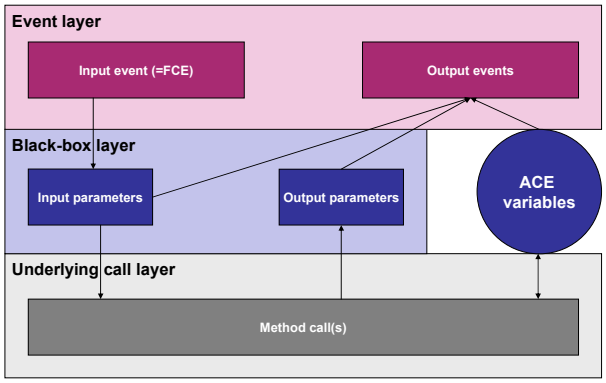

Figure 3. The three-layered functionality model

This three-layered model helps separating different abstraction levels of the call. From the caller's point of view, it is enough to know the Black-box layer (what the mandatory input parameters are) and the Event layer (what output events will be generated). From the Repository's point of view, the Underlying call layer is also essential.

For those functionalities specifically designed for ACEs, ACE-related variables are also available. These variables can be used inside of the Underlying call and on the Event layers.

Functionalities are labeled with unique names.

The above model significantly differs from the classical method-based approach as we use events on the highest abstraction level, instead of methods.

3.2 Underlying Call Layer

The Underlying call layer describes the method calls that technically mean the functionality. We differentiate between a single method call (simple call) and a sequence of method calls (complex call).

3.2.1 Simple call

Simple call means a single method invocation; where the input and output parameters of the black-box model are directly used as input and output arguments. The referred class is instantiated with its default constructor and the specified method is called.

The call description in Figure 4 creates a new instance of the `example.PhoneBook` class, to invoke its `search` method with the input parameters defined in the black-box model, and the return value will be saved under the output name of the black-box model.

```
<simple-call-details
  class-name="example.PhoneBook"
  method-name="search"/>
```

Figure 4. Simple call

3.2.2 Complex call

A complex call consists of a sequence of simple calls with explicitly marked input and output parameters. The origin of the input arguments of any simple call item may be either the black-box model or the output of a former call item in the series. When

performing a complex call, first all referred classes are instantiated[5], then the methods are invoked in the defined order with the specified parameters.

According to the example in Figure 5 two classes are instantiated[6] (the `example.PhoneBook` and the `example.Cache`), then 3 method calls are to be performed. First, the `phoneBook.load` method is invoked with the value of the `phone book file` parameter (coming from the black-box model). The return value of the method call is ignored (if any). Then, the `phoneBook.search` method is invoked with the `surname` and `first name` as parameters, and the result is saved under the name `phone number`. In the last call, the value-triplet [`surname, first name, phone number`] is put to the cache. Two of the values come from the black box model; while one value (`phone number`) comes from the output of a previous call item.

```
<complex-call-details>
  <call
    class-name="example.PhoneBook"
    method-name="load" >
      <arg ref="phone book file"/>
    </call>
  <call
    class-name="example.PhoneBook"
    method-name="search">
      <arg ref="surname"/>
      <arg ref="first name"/>
      <return ref="phone number"/>
    </call>
  <call
    class-name="example.Cache"
    method-name="put">
      <arg ref="surname"/>
      <arg ref="first name"/>
      <arg ref="phone number"/>
    </call>
</complex-call-details>
```

Figure 5. Complex call

For input and output arguments we use reference names, the actual value of the referred variable changes dynamically, depending on the call (when the `surname` parameter of the incoming call has the value "`John`" then `surname` is going to have the value `John` and John's number will be looked up). The Repository is in charge of resolving the reference names into their actual values. Referenced values must be properly typed; otherwise the method invocation will fail technically.

3.3 Black-box Layer

The Black box layer defines the input and output parameters of the Underlying call layer. It provides a higher-level abstraction of the call: only cares about the mandatory input and output parameters and hides the underlying process.

The input section specifies all external input parameters that are required during the call. Input parameters are defined with name and type; their actual value is extracted from the input event. The type of the parameter can be any valid primitive or complex Java type; complex types are given as qualified names.

[5] or an existing instance is re-used

[6] or an existing instance is re-used

The output is a single variable, similarly defined with name and type. In simple calls, the output of the method is saved under the reference name defined as output value. For complex calls, the output value is not important, it can be the output of any of the calls (their reference names must match).

Neither the input nor the output values are mandatory in all cases; the only restriction is that the description has to contain all technically important elements.

```
<black-box-description>
  <input>
    <param name="name" type="java.lang.String"/>
    <param name="apple" type="example.Apple"/>
    <param name="item counter" type="int"/>
  </input>
  <output name="summary" type="java.lang.String"/>
</black-box-description>
```

Figure 6. Black box model

Figure 6 contains an example for a fully filled black-box model. There are three input values and a single output. The first input value is called `name` and its type is a standard java class (`String`); the second input parameter – `apple` – is of a custom class (custom classes must be available in the classpath[7]); while the last input argument is of a primitive type (integer).

3.4 Event Layer

The Event layer describes the event model of the functionality. As the input event is fixed (FunctionalityCallEvent), the description contains information about the output events only.

There are two ways to define the output events: mapping and mapper. It is possible to use more than one mappings and/or mappers in the output section, even in a mixed way (e.g. three mappings and two mappers).

3.4.1 Mapping

Mapping means a direct mapping of the some parameters or ACE variables to an event and its attributes. First the event itself is declared, and then the parameter mappings follow. Figure 7 shows an example where three arguments are mapped to a service response event. First the SRE is instantiated and gets addressed to the remote ACE that plays the `user` role[8]; then the four attributes are added to the message. The first three attributes (`surname, first name, phone number`) are refer to the call parameters (input and output), while the last one is comes from one of the ACE variables (from the `globalSession`, in this case). The SRE is sent out through the Gateway by the Repository.

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

[7] Where the Java Virtula Machine looks up the classes for loading.

[8] Service call and service response events are sent over contracted connections where the parties are playing previously agreed roles. The details of the contract concept are outside of the scope of this paper.

```
    <value ref="phone number"/>
    <value ref="globalSession://signature">
  </mapping>
</output-event-mappings>
```

Figure 7. Mapping

The biggest advantage of the mapping concept is that it does not require program (code) writing. The person who creates the Functionality descriptor does not have to care about the details of the ACE-defined events, nor to modify the source code or to include additional classes. The Repository hides several technical details (handles the contracts, and fills the obligatory fields of the message).

On the other hand, the mapping is not flexible enough in some cases. We have to know the exact number of outgoing messages when the descriptor is created, so with this syntax, it is not possible to send output events depending on the result (e.g. separate output message per phone number found; or a warning to the supervisor about suspicious calls). The attribute list isn't dynamic either: it is impossible to describe that the passed attributes should depend on a condition. In cases when the Mapping is not enough to describe the intended operation, Mappers should be used.

3.4.2 Mapper

Mappers are special units able to dynamically generate output events from the actual after-call environment (input and output values, ACE variables). Mappers may dynamically define the number, type and content of the output events depending on the current circumstances; resulting in a way of flexibility that is missing from the Mapping concept. Technically, Mappers are required to implement a predefined interface. Figure 8 shows an example where all the output events are generated by the class `example.MyMapper`.

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

Figure 8. Mapper

Mappers have unlimited control over the number and type of the outgoing events; which also means that Mappers should be designed with care, as, for example, no other component is going to fill the mandatory fields of a message.

However, the Repository does not differentiate between events resulting from Mappings and those generated by Mappers. In both cases, internal events are sent to the Bus, external events are sent to the Gateway.

3.5 Example: a simple ACEification

Let us see a simple example about transforming an existing library (program code) into an ACE functionality. We have chosen a simple string concatenation service, with the following program code.

```
package example;
public class StringProcessor {
  // default constructor
  public StringProcessor() { }
  // functionality
  public String concat(String a, String b) {
    return a+b;
  }
}
```

Figure 9. Source code of a service

The only required step to turn the above class into an ACE functionality is to create a suitable Functionality descriptor. The descriptor consists of an event layer descriptor, a black-box model and the underlying call sequence.

```
<?xml version="1.0" encoding="UTF-8"?>
<functionality id="string_concat_service">
 <black-box-description>
  <input>
    <param name="first string"
           type="java.lang.String"/>
    <param name="second string"
           type="java.lang.String"/>
  </input>
  <output name="concatenation"
          type="java.lang.String"/>
 </black-box-description>
 <simple-call-details
    class-name="example.StringProcessor"
    method-name="concat"/>
 <output-event-mappings>
   <mapping
   event="cascadas.ace.event.ServiceResponseEvent"
   target-role="user">
     <value ref="concatenation"/>
  </mapping>
 </output-event-mappings>
</functionality>
```

Figure 10. Functionality descriptor

The descriptor in Figure 10 assigns the name `string_concat_service` to the functionality, and defines the three layers as follows. The Black-box layer describes two input strings and one output string. The Call layer specifies the class to be instantiated (`example.String-Processor`) and the method to be invoked (`concat`). The Event layer specifies that the output event is an SRE, which is addressed to the `user` of the service; and the event contains the concatenation of the two input strings.

4. ACE VARIABLES

Custom services may need to access more data just some input and a single output parameter. If a functionality expresses awareness of an ACE-related variable –in way of implementing an interface – it gets full access to that variable.

There are two general ACE variables (globalSession and executionSession) and one Repository-defined variable (callWideContext). Functionalities are able to express exact interest in one or more of the variables.

All three ACE variables are rather containers than simple variables. They are able to store name-value pairs, without restrictions on the value type. So, it is possible to save variable to any of the container following a `name=value` pattern (e.g. `globalSession.put("temperature",22)`).

4.1 CallWideContext

The call-wide context is a container available throughout the call. It is designed to store key-value pairs that are important during the call, but are not direct input or output parameters of any call segment.

With help of the call-wide context, the functionality can produce more than one "output" values. The model used in the Underlying call layer only allows one output value per method call, which is natural in a programming language like Java, but may bind the hands of the programmer when more than one value should be published (e.g. two fields should be added to the output event). In this case, the functionality can use the call-wide context as a temporal storage for the second value. The call-wide context can also be referenced during the output event generation (Figure 11).

```
<mapping
  event="cascadas.ace.event.ServiceResponseEvent"
  target-role="user">
    <value ref="the normal return value"/>
    <value ref="callWideContext://plum"/>
</mapping>
```

Figure 11. Referencing to the call-wide context in the output event mapping

Another way to use the call-wide context is to pass arguments between different classes. Let us suppose that a call sequence consists of calls to classes A, B and C: A reads in people's names and addresses from a list; B looks up the weight and height of each person in a medical database; finally C selects the thinnest person which will be the return value of the service. Although it is possible to pass the parameters between the classes as complex return types; that'll results in a complicated code. If A, B and C all declare themselves to be aware of the CallWideContext, they can use it as a convenient storage for their specific temporal data.

The call-wide context is created when the input event arrives, and is available until the last output event is sent.

4.2 Execution Session and Global Session

The execution session and the global session are two deeply ACE-related containers; they are created and maintained by the Reasoner, and they hold all state-specific ACE data.

The global session gets created at the initiation of the ACE, and is maintained as long as the ACE is alive (event survives movements).

The execution session is valid for a plan; and is lost when the plan is replaced.

With the help of the two sessions, the programmer is able to save and access state information between calls. By state information we mean both ACE-related and functionality-related information. (It might be necessary to store functionality-related information between calls because it is not guaranteed that the next call to the functionality will be executed on the same class instances, meaning that the values of the fields may get lost.)

5. FURTHER PROPERTIES

5.1 Multithreading

The functionality model of the Repository guarantees the thread-safe execution unless the functionality itself makes it technically impossible (e.g. non-thread-safe singletons in the background). In normal cases – where there are no static methods or variables – the functionality itself does not need to care about the thread safety; the Repository guarantees that no parallel/overlapping calls will be executed on the same instance of the class.

The Repository itself does not put restriction on the number of parallel functionality call requests. This means that the Reasoner may invoke a new – or the same – functionality before the execution of the actual one is finished; and threading problems must be handled by the Repository.

The Repository contains a built-in load balancing logic to decide when to create a new instance of the called class, and when to reuse existing (old) instances.

5.2 Statefulness

The state of a functionality is preserved during the call (from the first method invocation to the last output event); but is not guaranteed to be preserved between calls. This implies that stateful functionalities should store/restore their state variables to/from one of the sessions. Saving/loading variables from/to a session is the responsibility of the functionality.

5.3 Instantiation

The classes referenced in the Functionality descriptor are automatically instantiated by the Repository through their default constructor exactly when the request (FCE) arrives. If no request arrives, then no instance is created.

If the instantiation policy is not suitable in a special case, the programmer of the functionality should create a wrapper for the functionality. The wrapper may hide the non-default-constructor, or may refer to singletons in the background.

5.4 Chaining and Code Fragmentation

Functionalities cannot invoke each other directly but their output event may request the invocation of another functionality (or even itself). Thus, cross-invocation is possible in means of chaining, but not in means of invoking a functionality in the middle of another.

The above restriction may be resolved by code fragmentation. Let us suppose that service SA want to invoke SB in the middle of the code. By fragmenting the code of SA into two sub-functionalities (SA1, SA2) so that SA1 is the code before the invocation and SA2 is the code after the invocation; the problem can simply be solved (state information can be saved in the session).

5.5 Autonomic Aspects

Our goal with the functionality model was to map "normal program code" into functionalities of a self-aware system.

With the three-level description, we can guarantee that the autonomic components (Facilitator, Reasoner) are able to model

the functionalities properly, in order to achieve self-awareness and self-reflection.

We believe that our choice of modeling the functionalities as input and output events was a right and straightforward one, beating several burdens of the classical "method" approach.

6. COMPARISION WITH EXISTING TECHNOLOGIES

The presented functionality model has been motivated by and shows similarities with several of today's leading technologies. In this chapter we discuss the relationship with Web Services (WS) [3], Enterprise Java Beans (EJB) [1], and classical execution frameworks.

6.1 Relationship with Web Services

The Web Services technology has defined one of the most complete and best-detailed functionality models of today.

Although WS has introduced many basic abstractions; their goals differ from the goals of CASCADAS. WS provides a standard interface to the native code, with well-defined input and output mappings, client-instance assignment, and statefulness definition; but this standard interface is still the classical "method" format. In CASCADAS, we map the program code to a "service" which significantly differs from a classical "method" concept: the input and the outputs are events, and output events are addressed to contracted roles.

WS defines at least three client-service assignment models (scopes): one service instance per call, or per client, or for everybody; the last two ones have importance in case of stateful services. In CASCADAS, currently, we are using a per call model only.

Just like WS (and EJB), we use descriptors for our services. A big advantage of this is that the descriptor itself provides a clear abstract picture about the functionality (on different levels), making the work easier both on the self-model creator and on the caller side. We use three-layered descriptors, while WS descriptors are one-tier only.

6.2 Relationship with Enterprise Java Beans

Many details have been borrowed from the statefulness and persistence concepts of the Enterprise Java Beans – but in a rather simplified way.

EJB differentiates between stateful and stateless beans, stateful ones are guaranteed to preserve their state during the session. In our model, functionalities are stateless by default, but they are allowed to save/restore their state information.

In EJB, the bean can have metadata assigned describing how to save/restore the state information between calls (e.g. into database); and the bean container is in charge of ensuring the consistency. In our model, the save/restore code is placed (left) in the functionality itself. We believe that in simple cases[9] it is more natural to leave this task for the programmer than to provide meta-descriptors, based on which the Repository can symbolically invoke loader/saver code. In later versions of the Repository this may change.

6.3 Comparison with Classical Frameworks

We categorically do not follow the approach of many classical frameworks where deployed services must extend a fixed superclass. In CASCADAS, services can be introduced without source-code level restrictions[10] which – we believe – makes the system more open and flexible. The descriptor approach helps in understanding the functionalities in a clear and abstract way, without the need for checking the documentation.

7. SUMMARY

In this paper we have presented the functionality mapping model used in CASCADAS ACEs. We have chosen not to follow the classical method (input values, one output value) approach; but to define the functionality as a set of input and output events on the top level. We introduced a three-layer functionality model ensuring the system to be self-aware. Several other aspects (threading, statefulness) were also presented in details. The comparison of our model and existing technologies revealed several similarities and differences as well.

8. ACKNOWLEDGEMENTS

Special thanks to WP1 of the CASCADAS project and to Cefn Hoile from BT for the fruitful discussions. Thanks to the WP6 people for allowing me to better understand what they need from a Repository.

9. REFERENCES

[1] B. Burke, R. Monson-Haefel. Enterprise JavaBeans 3.0, O'Reilly Media, Inc, 2006., ISBN 059600978X

[2] CASCADAS project website: http://www.cascadas-project.org

[3] F. Curbera, W.A. Nagy, and S. Weerawarana. Web services: Why and how. In OOPSLA 2001 Workshop on Object-Oriented Web Services. ACM, 2001. 34

[4] E. Hoefig, B. Wuest, B. K. Benko, A. Mannella, M. Mamei, E. Di Nitto: On Concepts for Autonomic Communication Elements, in Proceedings of *IEEE International Workshop on Modelling Autonomic Communications Environments 2006*, Dublin, 2006

[5] A. Manzalini, F. Zambonelli: Towards Autonomic and Situation-Aware Communication Services: the CASCADAS Vision, in *Proceedings of IEEE Workshop on Distributed Intelligent Systems 2006*, Prague, 2006

[9] In CASCADAS the support for very high load or achieving extreme robustness are not as essential as in EJB/WS.

[10] The only restriction is to have a default constructor.