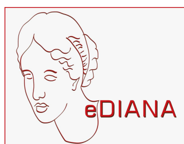


**eDIANA**
Embedded Systems for Energy Efficient Buildings

Grant agreement no.: *100012*

# Framework for adaptation through reconfiguration of components



| **Author(s):** | **X. Elkorobarrutia** | **MU** |
|---|---|---|
| | **J. Agirre** | **MU** |
| | **I. Rosenberg** | **ATOS** |
| | **J. Tillema** | **QUINTOR** |
| | **J. van Beek** | **QUINTOR** |
| | **GJ. van Dijk** | **QUINTOR** |
| | **T. Padilla** | **I&IMS** |
| | **MJ. Mtz. de Lizarduy** | **I&IMS** |



| | |
|---|---|
| **Issue Date** | January 2010 (m12) |
| **Deliverable Number** | D2.2-C |
| **WP Number** | WP2 |
| **Status** | Delivered |

| **Dissemination level** | |
|---|---|
| X | **PU** = Public |
| | **PP** = Restricted to other programme participants (including the JU) |
| | **RE** = Restricted to a group specified by the consortium (including the JU) |
| | **CO** = Confidential, only for members of the consortium (including the JU) |

| **Document history** | | | |
|---|---|---|---|
| ***V*** | ***Date*** | ***Author*** | ***Description*** |
| *1.0* | *2009-06-15* | *I&IMS* | *ToC* |
| *1.2* | *2009-11-13* | *MU* | *A skeleton and ideas to be further detailed* |
| *1.3* | *2009-12-14* | *MU* | *MU contribution* |
| *1.4* | *2009-12-15* | *ATOS* | *ATOS contribution chapter 3* |
| *1.5* | *2009-12-17* | *I&IMS* | *Integration of MU and ATOS contributions.* |
| *1.6* | *2010-01-19* | *MU* | *MU contribution chapter 6* |
| *1.7* | *2010-01-27* | *ATOS* | *ATOS contribution to chapter 4* |
| *2.0* | *2010-02-01* | *I&IMS* | *Release* |

**Disclaimer**

The information in this document is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

The document reflects only the author's views and the Community is not liable for any use that may be made of the information contained therein.

# Summary

*The "Framework of adaptation through reconfiguration of components" is an internal document prepared in the context of WP2, Task 2.2. Middleware and Component Self* with regard to design and development of middleware technologies for eDIANA, in order to ensure the interoperability between devices and sensor involved into eDIANA architecture.*

*This document is about the set of tools that will allow the complex eDIANA platform to be capable to diagnose its own faults and guide an untrained user to repair the system with minimal effort, both from a hardware and software perspective.*

# Contents

**SUMMARY......3**

**ABBREVIATIONS......6**

**TABLE OF FIGURES......7**

**1. INTRODUCTION......8**

**2. SUITABLE SELF* PROPERTIES FOR EDIANA PLATFORM. ...... 11**

2.1 OSGI/JAVA FOCUS ON SELF-CONFIGURATION AND SELF-HEALING......13

**3. RECONFIGURATION AS A MECHANISM FOR ADAPTATION...... 15**

3.1 OSGI – COMPONENT FRAMEWORK. ......15
3.2 SMARTCAM......18
3.3 GENESYS. ......18
3.4 LIRA. ......19
3.5 JMX. ......21
3.6 RUNES......21

**4. MIDDLEWARE SUPPORT FOR SYSTEM RECONFIGURATION...... 23**

4.1 OSGI MIDDLEWARE AND SYSTEMS MANAGEMENT CAPABILITIES. ......23
*4.1.1 Software Component Management......23*
*4.1.2 Service-oriented architectures......23*
*4.1.3 Remote Component Management. ......24*

**5. MODEL-CENTRIC RECONFIGURATION OF COMPONENTS...... 27**

5.1 OVERALL COMPONENT STRUCTURE......28
5.2 EXECUTOR CHANGES. ......29
5.3 HIERARCHICAL STATE-STRUCTURE SPECIFICATION. ......30
5.4 STATE BEHAVIOUR SPECIFICATION......31
5.5 PROGRAMMING MODEL. ......34
5.6 BACKGROUND ELEMENTS. ......36

**6. DETAILED DESCRIPTION OF THE FRAMEWORK FOR COMPONENT RECONFIGURATION...... 38**

6.1 MIDDLEWARE TOOLS. OSGI AND RECONFIGURATION......38
6.2 A FRAMEWORK FOR MODEL-CENTRIC RECONFIGURATION OF *STATECHART* BASED SOFTWARE-COMPONENTS......38
*6.2.1 Programming Model......39*
6.2.1.1 Definition of Hierarchical Structure of States...... 39
6.2.1.2 Definition of the Reactions of the Statechart...... 40
6.2.1.3 Creating a Software-Component Based on a Statechart...... 42
6.2.1.4 Consideration for Guard Evaluation and Action Execution. ...... 45

*6.2.2 Internal Elements for Event Processing.* ........ *46*
*6.2.3 Modifying the Statechart-Model at Run-time.* ........ *49*

**ACKNOWLEDGEMENTS........ 53**

**REFERENCES ........ 54**

# Abbreviations

| | |
|---|---|
| API | Application Programming Interface |
| CAN | Controller Area Network |
| CIM | Common Information Model |
| CMISE | Common Management Information Service Element |
| CORBA | Common Object Request Broker Architecture |
| eDIANA | Embedded Systems for Energy Efficient Buildings |
| GPRS | General Packet Radio Service |
| HTTP | HyperText Transfer Protocol |
| JMX | Java Management eXtensions |
| MDD | Model Driven Development |
| OMA DM | Open Mobile Alliance - Device Management |
| QoS | Quality of Service |
| OSGi | Open Services Gateway Initiative |
| RMI | Remote Method Invocation |
| RUNES | Reconfigurable, Ubiquitous, Networked Embedded Systems |
| SNMP | Simple Network Management Protocol |
| UML | Unified Modelling Language |
| URL | Uniform Resource Location |
| XML | eXtensible Markup Language |

# Table of Figures

FIGURE 3-1. THE RUNES COMPONENT MODEL. 22
FIGURE 5-1. STATECHART BASED COMPONENT. 28
FIGURE 5-2. STATECHART BASED COMPONENT STRUCTURE. 29
FIGURE 5-3. EXTRA CLASSES TO TRAP POSSIBLE EXCEPTION OF THE EXECUTOR PART. 30
FIGURE 5-4. CLASSES FOR THE CONSTRUCTION OF THE STATE-STRUCTURE OF A STATECHART. 31
FIGURE 5-5. CLASSES FOR THE DEFINITION OF THE BEHAVIOUR OF STATES. 32
FIGURE 5-6. STATECHART EXAMPLE. 33
FIGURE 5-7. PREPARATION AND FIRING REACTIONS UPON THE RECEPTION OF AN EVENT. 34
FIGURE 5-8. BASE CLASS FOR ALL STATECHARTS. 34
FIGURE 6-1. STATECHART'S GUARDS AND ACTIONS WILL BE GROUPED IN INSTANCES OF EXECUTOR. 39
FIGURE 6-2. THE API FOR THE DEFINITION OF THE STATE STRUCTURE OF A STATECHART. 40
FIGURE 6-3. CLASSES FOR DE DEFINITION OF THE STATECHART'S REACTIONS. 41
FIGURE 6-4. TRANSITIONS' ENDS HETEROGENEITY. 42
FIGURE 6-5. BASE CLASS FOR CREATION OF STATECHART-BASED COMPONENTS. 42
FIGURA 6-6. A STATECHART EXAMPLE. 43
FIGURE 6-7. METHODS OF A REACTION TO BE LAUNCHED. 47
FIGURE 6-8. INTERFACE FOR EVENT PROCESSING CLASSES 49
FIGURE 6-9. SOFTWARE COMPONENTS WITH BEHAVIOUR MODIFICATION ABILITY. 50
FIGURE 6-10. EACH STATE-MACHINE COULD HAVE VARIOUS BEHAVIOUR MODIFIERS. 51
FIGURA 6-11. MODIFIERS IMPLEMENTED AS ANY ACTIONS ASSOCIATED TO A STATECHART. 51

# 1. Introduction.

System dependability is the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers. Although it seems an intuitive concept, it is difficult to quantify. This is done by means of some set of meta-functional properties like reliability, safety, maintainability, availability and security.

The basic concepts used concerning to the root causes of no proper functioning of systems are defects, errors and faults. The first can be hardware defects or even software design defects. Both those can be the cause of errors: an internal deviation of the internal behaviour of a system and, finally, those erroneous states of a system can lead to system failures: a discrepancy between the external behaviour of a system from its specification.

The techniques to mitigate or to avoid the consequences of system faults can be classified in the next groups:

- prevention,
- error elimination,
- prediction
- tolerance

**Prevention** is related with the using of the appropriate tools and support at design and development-time.

**Error elimination** can be accomplished by verification techniques, at development-time or at run-time, by monitoring the system and maintaining it.

**Prediction** is the capability at run-time to foresee an erroneous state and try to eliminate them or at least mitigate their consequences.

Finally, **fault-tolerance** is a set of techniques that, based on the provision of the necessary redundancy of components, gives a system the capability of correct functioning even in the presence of an erroneous component.

Delving into hardware faults, we refer mostly to computational node's faults and communication infrastructure' faults. This is true in computer intensive systems but having a closer look at control systems: the main source of a system' inability to fulfil its duty is the system's controlled devices and environmental uncertainties. These are also known as environmental failures. In this scenario, the ability of the control

system of coping with those circumstances is more desirable than avoiding or tolerating its' internal faults.

The eDIANA platform fits in this category of systems. It will be more important that the control system is capable of coping with various circumstances like device unavailability, impossibility of communication or variation of working condition of each device than cope with its hardware faults. In few words: *environmental adaptation capability*.

In respect to software faults, those are used to model development-time defects. Validation and verification techniques can help minimizing these faults but the true cause of them is software complexity. Dependability, although being a desired property of systems, increments software complexity.

The development of systems, which environment or even which internal structure can vary at run-time, adds some extra complexity. This is especially true when this adaptation is designed in an *ad hoc* way and not separated from the system's functional aspect.

Self* has addressed this problem and its goal is to manage complexity. It can be resumed as "*technology that manages technology*". Basically, it separates functional aspects of a system from other system management issues. For that to be possible, it is necessary to implement a "control loop" that monitor the system itself, acting on it in order to maintain some desired properties in an acceptable range. This means that software systems need to be instrumented with monitoring capabilities and it has to have some ways of acting on it.

Self* community has made us aware of the need to separate the functional aspects of a system from the management of this functionality to better adapt to varying context or working condition. This awareness permits us to focus on general mechanisms for system adaptation that can be suitable to apply to a wide range of applications.

But first of all, we must define the context variations a system must cope with, the suitable changes in the functional layer that need to be fired at run-time in order to cope with this context variation and, finally, define an infrastructure which will permit lo launch, at run time, those modifications in the functional layer.

In Section 2, the Self* properties addressed by the eDIANA platform will be defined.

Section 3 will argue that reconfiguration is the mean by which a system can be given adaptation capabilities. In particular, there will be taken into account two kinds of reconfiguration: system-wide through architecture reconfiguration and an intra-component through behavioural change.

Section 4 and 5 will delve into each of these two kinds of reconfiguration.

Finally, section 6 will described a more detailed description of the needed platforms and/or framework to use or develop.

# 2. Suitable Self* properties for eDIANA platform.

The main characteristic of a self-governable system is its conscience about itself. The properties that it exhibits are generally among the next four:

- **Self-configuration**: installing, configuring and integrating large systems is challenging and time-consuming. A self-configuring system configures itself according to high level policies.

- **Self-optimization**: A self-optimizing system will continuously seek ways to improve its operation, identifying opportunities to make itself more efficient in performance or cost.

- **Self-protection**: the ability to protect itself from malicious attacks or inadvertent cascading failures.

- **Self-healing**: the ability to detect, diagnose and repair localized problems resulting from bugs or failures in software and hardware.

Those properties are not exclusive, neither independent. For example, a reconfiguration action of the system can be launch in order to heal a system from an erroneous component. Nevertheless, these properties are goals for which there must be procedures, methodologies and supporting tools/frameworks to implement them.

In the scope of the eDIANA platform, self-optimization in not relevant; this property is oriented at optimizing computational resources at run-time. Self-protection will not be considered because the real value of the eDIANA platform is the control it performs on devices and energy consumption and not the control system itself. The correct usage of devices and energy management will be among the functional specifications of the system.

Self-configuring and self-healing will have more priority due to the changing environment in which the eDIANA platform will operate. The availability/unavailability of various services and the presence of different devices can not be anticipated and this implies a dynamic reconfiguration of the system at run-time.

Self-healing would also be necessary in order to cope with malfunctioning software components, defective elements and not desired emergent behaviour arisen from the heterogeneous composition of the system.

Many issues to be accomplished on such platform as eDIANA could be found. In this work package, the ones that affect the dependability of the system must be chosen and prioritized.

The main nature of system malfunction will come from, first, the dynamic and heterogeneous nature of the control system itself and, second, the devices which the platform is on charge. Thus, the main source of the system malfunctioning would come from the changing context in which the system is working on and the complexity that this changing context implies in the system.

Therefore, traditional fault tolerant techniques based on redundancy do not have much sense if we accept that the main source of difficulties would arise from the environment and not the control system itself. Moreover, these kinds of techniques are expensive in terms of development as well as economic costs.

Thus, what would be more suitable is the capability of the control system to have the capability to adapt to new circumstances.

Adaptation means a change of structure or behaviour that makes the system better fulfil its duty. For example, self-configuration can be seen as one kind of adaptation. Thus, from now on, it will be used the term *self-adaptation* and, depending on the obtained benefits, the system will gain *self-configuring* or *self-healing* properties.

In order to give self-adaptation capabilities to a system, this could be done for a specific application problem or could be defined some general approaches that can be applied/adapted in a wide range of application. In a latter approach, reconfiguration can be the mechanism by which the system could be self-adaptive. Thus, there must be defined general reconfiguration mechanisms that, applied in a particular system and in a specific way, gives systems adaptation capabilities.

But for the system to be adaptable, it must be aware of its environment or context because it will change its behaviour according to variations in this environment. For that to be possible, there must be a sensing mechanism that will make the system aware of those working conditions changes. If it would be possible to standardize the elements to be monitored, it will be possible to instrument each component with the necessary support for being monitored or notify information of interest. If not, in order to propose a general mechanism, a notification service (like the one specified in CORBA, for example) can be employed. This service will work in an application agnostic way and each component will need to be tuned to produce/interpret the provided information.

The decision element could be a global component of the system where all adaptation logic is centralized or, also, where each component could have a part of this logic that corresponds to it.

As conclusion, reconfiguration will be the mean by which a system can gain adaptation capabilities. In the eDIANA context, the factors to be taken into account to such adaptation are uncertainties, changes in the environment or the system itself exhibiting malfunctions: devices, available services, human input, communications,

external stimuli range and so on. The types of reconfiguration that will be contemplated for the eDIANA platform will be the topic of next section.

The complete system is made up of devices, which often form a complex network with interdependencies. Each device, seen independently, may either be malfunctioning or working correctly. The events which cause the device to change from one state to the other may depend on the device itself or on some external factors.

A platform which could provide these kinds of capabilities and that is available in the market would be the OSGi platform. This platform has been also evaluated in other working lines of the eDIANA project as are the middleware for device awareness (D2.2A) or the platforms providing service discovery protocols (D2.2B).

## 2.1 OSGi/Java focus on Self-configuration and Self-healing.

The OSGi Service Platform is a platform for developing component based self-adapting and self-configuring software for embedded devices. The self-adapting OSGi software can download, install and run new components on the fly.

Software is omnipresent. Millions of computers surround us, all of them running software. Not only the standard desktop-platforms, servers and main frames come to mind but also PDA's, smart-phones, game consoles, set- top boxes. Even cars and household-devices are or will be driven by (mini-)computers and software and will be network-enabled. All of them need software to run.

Along with the explosion of new devices comes the explosion of software. This calls for a platform independent development model for software, able to run on a variety of platforms. Developing software for multiple platforms however, has never been easy and still isn't trivial when developing for very different platforms, despite the availability of the Java Platform.

An important aspect when developing platform independent software is the availability of the resources a program needs. Although this issue might not be as important when developing for powerful easily upgradable desktop platform, it's something to take into account when developing for small mobile devices with usually rather limited resources. There can also be many differences between these types of devices. Therefore a really platform independent application should adapt itself to the device it is executed on.

- Adaptive software

OSGI enables the developement of adaptive software. Adaptive software, as the name suggests, adapts itself to its environment. In particular it can change its behavior or looks depending on the availability of certain resources like the display,

the input devices, the amount of memory or available network bandwidth. Obviously these changes must occur at runtime because the availability of most resources changes constantly.

Two kinds of adaptivity can be distinguished. Functional adaptivity means that a program exhibits more functionality when the needed resources are available than when they are not available. Another kind is nonfunctional adaptivity. The functionality stays the same, but is degraded to lower performance in the absence of sufficient resources. For example, when there is little memory left, a program could switch to an algorithm that uses less memory, but takes a longer time to execute. OSGi supports both kinds of adaptivity.

- Advantages

There are a number of advantages to adaptive software. First of all, it facilitates the distribution of software to a wide variety of platforms. Software can already be written in a platform independent way, because all platform specific details can be hidden by means of a virtual machine.

However, this does mean you can't take advantage of the platform specific properties such as the size of the available screen. With adaptive software it should be possible to achieve even greater platform independence. Instead of developing a different software distribution for every type of device, it is possible to make one software distribution that can run on several types of devices and, for example, adjusts its user interface to the screen size. On constrained devices hardware resources are typically not available in abundance. As mentioned before, this too is a situation where adaptive software is useful. When the load on a certain resource is too high, the software can detect this and take some action. It may reduce its functionality, but still provide an adequate user experience (graceful degradation) or it may switch to a less resource hungry algorithm or component.

Adaptive software using OSGi may not only provide an overall better user experience, it also simplifies development. Instead of developing multiple parallel versions of a software application for multiple platforms it's easier to maintain one single version running on multiple platforms. This allows you to focus developer resources in one place and deliver better results faster.

# 3. Reconfiguration as a mechanism for adaptation.

Having pointed to reconfiguration as a mean to obtain self-adaptive systems and aiming at defining application-independent strategies, at least, two kinds of approaches can be taken into account: architecture level and inside-component level.

In architecture level approaches, components are seen as the indivisible and atomic building blocks (black boxes). By replacing, replicating, updating components and their association the overall behaviour of the system can be changed/adapted to new environmental circumstances.

Components platforms offer those facilities; rests to define within each application which architectural changes must be launched in response to environmental changes.

## 3.1 OSGi – component framework.

OSGi stands for Open Services Gateway Initiative. The OSGi consortium defines a standardized, component oriented, computing environment for networked services. Adding an OSGi Service Platform to a networked device (embedded as well as servers), adds the capability to manage the lifecycle of the software components in the device from any where in the network. Software components, which are called bundles in the OSGi specifications, can be installed, updated, or removed on the fly without ever having to disrupt the operation of the device. This solves the problem of distributing software to a multitude of different platforms, each having a different procedure to install or uninstall software. The OSGi platform is based on Java, this makes it the ideal platform for the distribution of platform independent software components. Bundles are libraries, applications or services that can dynamically discover and use the libraries and services of other installed bundles. They can be bought off the shelf or developed in-house. The a OSGi Alliance has developed many standard component interfaces that are available for common functions like HTTP servers, configuration , logging, security, user administration, XML, and many more.

- Architecture

The OSGi specification defines an extra layer of abstraction on top of an existing Java Virtual Machine (JVM).

For software developers, consumer device technology provides an increasingly interesting environment in which they can deliver products and services. Consumer devices are much more prolific than personal computers and, perhaps more importantly, are more integrated into everyday life. As a result, software developers benefit from unprecedented levels of access to the user by targeting their products

and services towards consumer devices. The consumer device software market is fueled by four factors:

The increasing power of consumer devices, such as mobile phones or personal digital assistants (PDA's).

- The transformation of "traditional" consumer devices, such as refrigerators and washing machines, into "smart" consumer devices.
- The widespread availability of broadband Internet connectivity.
- The growing prevalence of home-area networks.

The Open Services Gateway Initiative (OSGi) is helping developers realize the potential of the consumer device market; OSGi is an "independent, non-profit corporation working to define and promote open specifications for the delivery of managed broadband services to networks in homes, cars, and other environments." Specifically, the OSGi specification defines a service platform that includes a minimal component-like model and a small framework for managing the components, including a packaging and delivery format.

The OSGi framework facilitates dynamic installation and management of simple components called bundles. These dynamically loadable components interact with each other and form "applications" by providing and using services. A service is a Java interface with defined semantics and potentially multiple implementations. Services are packaged along with their associated resources and deployed via the Internet into the OSGi home services gateway. In this scenario, consumer devices become delivery mechanisms, capable to deliver bundles along with their associated services.

The services gateway acts as the access mechanism or conduit between service providers and the end-user. A typical application scenario is that of a home security company that monitors the end-user's home for security concerns using a combination of hardware (e.g., sensors) and software (e.g., sensor monitors, software control panels for the end-users and the service provider). While OSGi creates a good foundation on which to build such services, it is still fairly low-level; the low abstraction level increases the complexity of offering complex applications on top of the OSGi framework.

At component internals' level, its behaviour can be varied in accordance to different environment or other components needs. This can be obtained at leas by two ways. First, emulating architecture level strategies and changing internal structure of elements (objects, classes, association, etc.) in order to modify the externally visible behaviour of the component. And secondly, having a behavioural model of the

component, modifying it assuming that such models are available at run-time and moreover, needing those model changes to take effect on the running component.

For that last issue, in Section 5 a framework will be specified for developing software components whose behaviour is describable by means of *statechart* in a MDD style of development. But thanks to that framework, the developed components will reflect at run-rime the statechat-model it is derived from. A change in this reflected model will imply automatically a change on the component behaviour.

A component platform is an extension of the typical architecture of a framework, in which core services are available. The core services support the component's interaction with the rest of the platform, and as such express the component behaviour, externally visible at the interfaces. Usually, the core services encompass communications services (for example via message passing), fault tolerance services and timing services. The additional services which can be deployed are called components - there is a common model for their definition, deployment and interconnection. The core services are used as the minimal features upon which to construct the more sophisticated functionalities of the components. This enables the emergence of global application services of the overall system, out of local application services of the constituting components [1]. For example, the built-in message passing interfaces can be extended by adding resilience and control, producing a reliable communication component. In the eDIANA scope, the common model should be versatile enough to be adaptable to components supported by hardware devices, but at the same time complete enough to offer configuration and life-cycle changes at run-time. Another important requirements lies in real-time response: a fire-protection system may define different needs from a light-saving embedded network.

A desirable quality of the framework is the capacity of hot-swapping: replacing a component without having to interfere in the lifecycle of the other components of the framework, in particular those which are connected to the component to replace. As the interconnection of the components is dependent on the framework's core services, the replacement of components must rely on functionality offered by the core services. One typical situation that must be addressed concerns message buffering: during the replacement of a component, the messages which are sent to the component must stop reaching the old component, and start reaching the new one. When the component replacement is not an instantaneous operation, the platform must retain the messages that otherwise would be lost, and must be capable of replaying them to the new component once it is fully deployed.

Another point concerns heterogeneity. As P. Costa et al. [2] describe it, "*Next generation embedded systems will be composed of large numbers of heterogeneous devices. These will typically be resource-constrained (such as sensor motes), will use different operating systems, and will be connected through different types of*

*network interfaces. Additionally, they may be mobile and/or form ad-hoc networks with their peers, and will need to be adaptive to changing conditions based on context-awareness".*

The platform that federates them must be capable of supporting such changeability, in a consistent way. To be able to offer this, the platform must have a rigorous interface specification, which describes precisely how components must integrate with the framework, and the expected behaviour of the different elements upon the possible stimuli of the environment (which produce component response, including failure). When this is offered, the replacement component may be a totally different implementation, as long as the interfaces presented remain the same. This allows for an increased range of acceptable components, making the platform more versatile. As a side note, it must be taken into account that the complexity of the platform, and possible design faults, are of particular importance for safety-critical applications needing to be certified – the over-specific specification of the interfaces is then seen as a requirement.

In the following paragraphs, we propose several examples of such component frameworks.

## 3.2 SmartCAM.

The SmartCAM platform [3], further refined as the DSC-Agents platform [4], is a light-weight middleware designed for smart cameras, offering flexible algorithm communication and dynamic component composition.

It defines publication mechanisms which enables rich communication behaviours. For example, components can be advertised, including the description of the behaviour (business logic).

The algorithms implemented within the components can be replaced by sending new versions over the communication channel, using dynamic loading and linking through the Texas Instrument software framework Dynamic Loader. The replaceable algorithms are seen as another component within the framework, and offer specific interfaces and descriptions. Moreover, the framework constantly monitors QoS, making sure that the components are acting within their limits, degrading performance gracefully or replacing components when violations are reported.

## 3.3 GENESYS.

The GENESYS Architecture proposes a "*conceptual model and terminology for component-based development of distributed real-time systems*".

The core platform services provide elementary capabilities for the interaction of components, such as message-based communication between components or a

global time base. In addition, the specification of a component's interface builds upon the concepts and operations of the core platform services.

The component interface specification constrains the use of these operations and assigns contextual information (e.g. semantics in relation to the component environment) and significant properties (e.g. reliability requirements, energy constraints).

The core platform services are a key aspect in the interaction between integrator and component developer. Different types of services can be distinguished: global application services of the overall system, local application services of components, and platform services. The critical concept of component interface is introduced through the Linking Interface (borderline between component and platform) and it represents the unique point of transfer of information between the component and the rest of the platform – it must define the syntax of the information exchange, the different timing events and possibly the relationship with the environment [5]-.

The GENESYS architecture [6] provides deep insight into which concepts must be addressed by component frameworks. For example, it makes specific distinction between integration levels, ranging from system, device, to chip. It also presents the needed abstraction levels required to analyse, design and introspect the platform. It recommends the maximum simplification of the platform and systems to increase the various qualities included within dependability, this can be reached through abstraction, partitioning or segmentation. Other important themes which are addressed include network connectivity, resource management, and the typical robustness and security features.

A reconfiguration system is provided within the GENESYS architecture [6], and centres on providing a resource reallocation scheme. New resource allocations can be calculated by a general purpose micro component at run-time or can be determined offline before deployment and fetched at run-time.

The reconfiguration core service investigates new resource allocations by checking the compliance with constraints in the time-triggered communication schedule, and also tracks potential collisions, and finally approves it for distribution.

The exploitation of the GENESYS architecture is already under way through the INDEXYS project [7] based on use cases in the automotive, aerospace and railway sectors.

## 3.4 LIRA.

Other platform available for reconfiguration is LIRA. It is a lightweight infrastructure for managing dynamic reconfiguration that applies and extends the concepts of network management to component-based, distributed software systems.

LIRA is designed to perform both component-level reconfigurations and scalable application-level reconfigurations, the former through agents associated with individual components and the latter through a hierarchy of managers.

Agents are programmed on a component-by-component basis to respond to reconfiguration requests appropriate for that component. Managers embody the logic for monitoring the state of one or more components, and for determining when and how to execute reconfiguration activities. A simple protocol based on SNMP is used for communication among managers and agents.

LIRA offers a decision maker for selecting suitable new configurations and allows for monitoring and reconfiguration at components and applications level, while decisions are taken following the feedbacks provided by the evaluation of statistical Petri net models [8]. This advanced design in the platform logic allows complicated behaviours to emerge.

LIRA originally stemmed from a heavyweight platform, which after inspection and intensive diet was thinned down to a lightweight platform. The initial platform depended on a software deployment system called *Software Dock*, which represented component dependencies and constraints arising from heterogeneous deployment environments.

However, it did not provide explicit support for dynamic reconfiguration although its infrastructure was designed to accommodate the future introduction of such a capability. Severe demands were imposed on component and application developers: extra functionality had to be deployed on each host to support the deployment activities and communications between components. It also required the use of a special deployment language. The *Software Dock* leads to a "heavyweight" solution to the problem of dynamic reconfiguration, a characteristic shared by many other reconfiguration systems (e.g., DRS [9], Lua [10] and PRISMA [11]).

LIRA has its original design based on the concepts of SNMP (Simple Network Management Protocol): agents, mangers, a protocol and management information. Reconfiguration agents, in charge of lifecycle management, are associated with execution components, while a host agent is associated with a device in the network and is responsible for installing and activating components on that host. The architecture is hierarchical to support scalable management.

Maintaining consistency during and after a reconfiguration is problematic. Usually, the consistency properties of the system are expressed through logical constraints that should be respected, either *a posteriori* or *a priori*.

If the constraints are seen as post-conditions [12], the reconfiguration must be undone if a constraint is violated. If the constraints are seen as preconditions [13], the reconfiguration can be done only if the constraints are satisfied.

LIRA approaches the consistency problem by delegating responsibility to agents to do what is necessary to guarantee consistency and state integrity. This is in line with the idea of "self-organizing software architectures", where the goal is to minimize the amount of explicit management and reduce protocol communication. It is also in line with the idea of a lightweight service, where we assume that the component developer has the proper insight about how best to maintain consistency. This is in contrast to having the reconfiguration infrastructure impose some sort of consistency-maintenance scheme of its own.

## 3.5 JMX.

Java Management eXtensions (JMX) is a specification that defines an architecture, design pattern, APIs, and services for application and network management in the Java programming language. Under JMX, each managed resource and its reconfiguration services are captured as a so-called MBean.

The MBean is registered with an MBean server inside a JMX agent. The JMX agent controls the registered resources and makes them available to remote management applications. The reconfiguration logic of JMX agents can be dynamically extended by registering MBeans. Finally, the JMX specification allows communication among different kinds of managers through connectors and protocol adaptors that provide integration with HTTP, RMI, and even the SNMP protocols.

While JMX is clearly a powerful reconfiguration framework, it is also a heavyweight mechanism and one that is strongly tied to one specific platform, namely Java.

## 3.6 RUNES.

The range of devices involved in networked embedded environments inevitably leads to significant complexity in appropriately configuring, deploying, and dynamically reconfiguring the software.

There is therefore a need for dedicated middleware platforms addressing networked embedded systems, with abstractions that can span the full range of heterogeneous systems (encompasses dedicated radio layers, networks, middleware, and specialised simulation and verification tools) and which also offer consistent mechanisms with which to configure, deploy, and dynamically reconfigure both system and application level software.

The EU-funded RUNES (Reconfigurable, Ubiquitous, Networked Embedded Systems) approach is based on a small and efficient *'middleware kernel'* which supports highly modularised and customisable component-based middleware services that can be tailored for specific embedded environments, and are runtime reconfigurable to support adaptability. These services are primarily communications-related but also address a range of other concerns including service discovery and logical mobility.

The middleware provision comprises two distinct and orthogonal parts:

- first, a foundation layer – called the middleware kernel – which is the runtime realisation of a simple but well-defined software component model;
- second, on top of the middleware kernel, a layer of component frameworks that offer a configurable and extensible set of middleware and application services.

In this two-layer architecture, the software infrastructure for a specific heterogeneous, embedded networked system is achieved by providing an appropriate implementation or implementations of the middleware kernel, and of the required configuration of middleware (or application) services running on top of it.

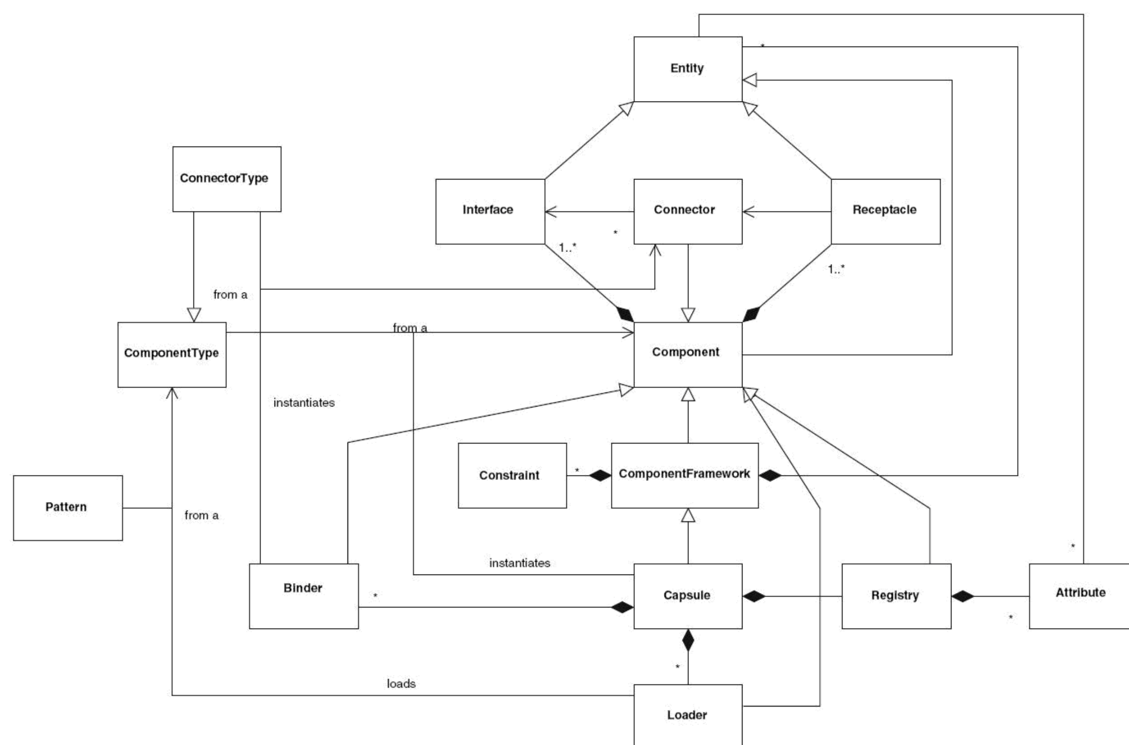


*Figure 3-1. The RUNES component model.*

# 4. Middleware support for system reconfiguration.

## 4.1 OSGi middleware and systems management capabilities.

OSGi provides a flexible remote management architecture that allows platform operators (the organization that manages the platform) and enterprises to manage thousands or even millions of Service Platforms from a single management domain. The OSGi Service Platform architecture allows the operator to control a platform in fine detail by using a model where the operators can ensure that their required policies are implemented, owever secure or flexible those policies are desired to be: from a walled garden to the anarchy of a PC.

### 4.1.1 Software Component Management.

the OSGi Service Platform represents an excellent way to deploy composite applications. The platform helps during the development of the components and with the assembly and configuration of them at runtime. Additionally, the OSGi Service Platform provides comprehensive life cycle management capabilities for these components so that each component can be controlled individually. the OSGi Service Platform represents an excellent way to deploy composite applications. The platform helps during the development of the components and with the assembly and configuration of them at runtime. Additionally, the OSGi Service Platform provides comprehensive life cycle management capabilities for these components so that each component can be controlled individually.

### 4.1.2 Service-oriented architectures.

On top of the life cycle management capabilities, the OSGi Service Platform introduces the concepts of Service-Oriented Architectures (SOA). Whereas SOA is normally associated with services communicating over networks across machine boundaries, the OSGi Service Platform applies these concepts to the local virtual machine only. Each component can add it’s own services or use services from other components to participate in this SOA infrastructure. As described, the OSGi Service Platform controls the life cycle of each component. In addition, the creation of services and the wiring between the components gets tracked.

The functions provided by the framework are based around the following life cycle:

- **Packaging**: The packaging of bundles is based on the Java Archive (JAR) File Format. The framework specifics how to supply addition metadata to e. g. distinguish between different version.
- **Install**: The framework validates the bundle and ensures that all requirements are met.
- **Start/stop**: Installed bundles can be started and stopped.
- **Update**: The framework allows to update installed bundles. It ensures that all previous loaded code gets replaced and that depending bundles are supplied with the new code.
- **Uninstall**: Finally, a bundle can b e removed from the container.

### 4.1.3 Remote Component Management.

The OSGi Service Platform is specifically designed for devices that can operate unattended or under control of a platform operator. These are the devices that need remote management. Managing devices remotely requires a protocol. Selecting an appropriate protocol is difficult because there are so many choices: SNMP, CMISE, CIM, OMA DM, and more.

The OSGi Alliance decided that no management protocol can be preferred over others because no protocol is suitable for all cases. The OSGi Alliance therefore chose an architecture that provides a management API to be used by an authorized bundle. This authorized bundle can then map a protocol to API calls. This is a very powerful concept that offers the same interoperability as a standard protocol. However, the benefits of this concept are not always immediately obvious. The benefit of a standard protocol is that any device can be managed by any management system. With the OSGi remote management architecture, the same advantage is achieved with an API and bundles which provide a higher level of abstraction with respect to a protocol. As long as a management system can provide a standardized OSGi protocol bundle, it can operate with any manufacturer’s device, regardless of the underlying protocol.

By not specifying a protocol, the management system can optimize for the carrier characteristics. For example, a mobile phone with GPRS has high latency, expensive bandwidth, and is not always online. A residential gateway is always online, has low latency, and near zero cost bandwidth. Obviously, a protocol suitable for a mobile phone might not leverage the more privileged position of the gateway. Vice versa, a protocol that works well for a residential gateway does not work for a mobile phone.

The OSGi management model therefore allows both industries to use their optimal management protocol without wreaking havoc on the interoperability.

As the OSGi framework is a module system for Java that implements a complete and dynamic component model, it is a perfect fit for the model presented for the eDIANA architecture. The remote control and management allows building up extremely dynamic systems, which can be instrumented and configured to be highly self-managed. This possibility can be leveraged to apply to heterogeneous sets of devices, where the proper abstractions allow keeping the management burden to an affordable level.

The core of the OSGi framework is composed of many services. These services can be implemented by bundles, and also register within the Registry. The core services can be divided in four main categories: framework services, system services, protocol services, and lastly miscellaneous services.

The framework services include *Permission Admin* (to segment interactions between bundles), *Package Admin* (information about the actual package sharing state of the system, specifically useful for shared packages), *Start Level* (used at framework initialization), *URL Handler* (to manage service and resource lookup).

The system services include *Logging* (dispatching its input to other bundles subscribed to it), *Configuration Admin* (to set up the configuration information of deployed bundles), *Device Access* (to coordinate the automatic inclusion of existing devices), *User Admin* (for authentication and authorization purposes), *IO Connector* (implementing the CDC/CLDC package as a service, for new and alternative protocol schemes), *Preferences* (to store preferences to be used by other bundles), *Component Runtime* (offering simpler development and deployment through XML based declaration of dependencies), *Deployment Admin*, *Event Admin* (communications based on a publish-and-subscribe model), *Application Admin* (management of an environment).

The protocol services include HTTP Service, UPnP Device Service (for Universal Plug and Play devices). DMT Admin (following the Open Mobile Alliance device management specifications).

The Miscellaneous Services include Wire Admin (connection between Producer and Consumer services), XML Parser (for a JAXP compatible parser), Measurement and State (to provide instrumentation, through simplification of the management of measurements).

The OSGi framework is widely accepted. Since its foundation in 1999, it has brought together the main creators of mobile technologies, as well as many big software companies. The OSGi specifications are now used in applications ranging from mobile phones to the open source Eclipse IDE. Other application areas include automobiles,

industrial automation, building automation, PDAs, grid computing, entertainment, fleet management and application servers. The latest release of the specification is Version 4.2 of September 2009. The releases (October 2005) 4.0 and (May 2007) 4.1 are very widely used currently.

# 5. Model-centric reconfiguration of components.

In this section, a mechanism for giving a software component the ability to change its behaviour at run-rime will be described. This mechanism is intended not to be hardwired with the rest of the functionality and, thus, it will become a generic approach. We will assume that the behaviour of the components is described by means of *statechart*s.

As noticed in [14], the adaptation mechanism implemented in embedded systems has evolved and will probably evolve in the following way. Indeed, in a first stage, adaptation was not considered. At a second stage, it had not been separated from the rest of the functionalities. At a third stage, where we probably are nowadays, we consciously consider it but there is no proven engineering practice to cope with it in a systematic way. And at a final stage, this engineering practice will arise. Moreover, such adaptation mechanisms could be a complementary technique to traditional redundancy-based fault-tolerance approaches, even a cheaper one.

The selected approach goes towards an adaptation-engineering. Many times, depending of the surrounding conditions, a software component must behave in different ways. Each stable set of those conditions will be referred as *"working mode"*. In developing such components it is usual to hardwire the code related to every working mode and the code in charge of the behaviour change. This last one is not separated from the rest of the functionality.

The following approach is illustrated in figure 5-1 where each *statechart* based component is provided with a basic behaviour plus some *"behaviour modifiers"*. Whenever a component needs to change its working mode, it will launch one of those modifiers as a reconfiguration of its behaviour. This way, once the core functionality has been provided, the code related to each working mode can be developed in an independent way and a framework will assist at run-rime in changing the behaviour of the component.

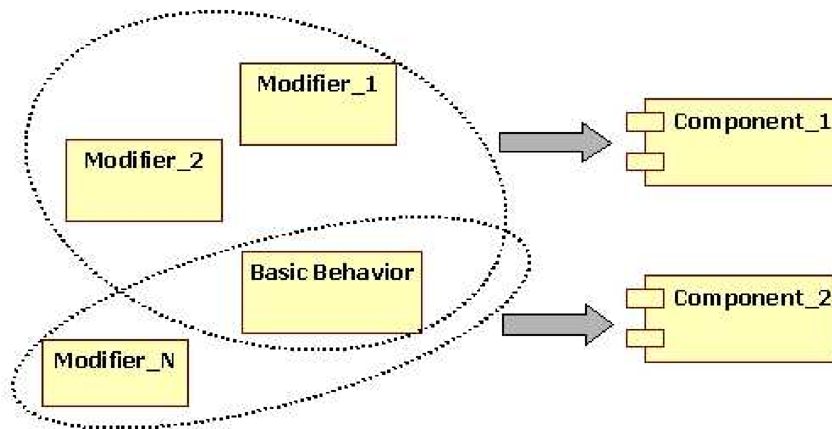


*Figure 5-1. Statechart based component.*

## 5.1 Overall Component Structure.

It will be proposed a model centric approach [15]. This is, the *statechart*-model of the component will not be used only as a design artefact, but it will persist at run–time reflecting its behaviour. Any changes to the component's behaviour will be accomplished changing this model which will immediately affect the component's behaviour.

The next concepts need to be distinguished: the set of events that the component will be aware of and upon whose reception it will potentially fire some reactions; the set of actions that will define those reactions and to which we will refer as functionality; and finally, the control or the part of the component that determines the actions that must be launched as a reaction to each particular event in each state of the component. This last one is the core of a *statechart*.

Figure 5-2 illustrates the overall structure of the proposed software component. Some of its parts will be provided by a framework and the specific application will need to be defined by the developer. The roles of each part are:

- **Buffer**: The events that the component will react to will be materialized as messages that will be stored in this buffer until the component processes them.
- **Timer**: a timing facility provided by the framework.
- **Executor**: a part that defines all the actions that can be fired by the component and all the guards needed to be evaluated in order to take the decision to fire a reaction.
- **StatechartDefinition**: it describes the *statechart* model of the component: states, hierarchy, region, references to guards and actions, etc.

- **GlobalRepository**: a data structure that maintains a snapshot of the component: the set of the states that are active and the message the component is processing in each moment.
- **Dispatcher**: a component provided by the framework that upon the reception of a message, interrogates the *StatechartDefinition* part to determine the action to be fired and

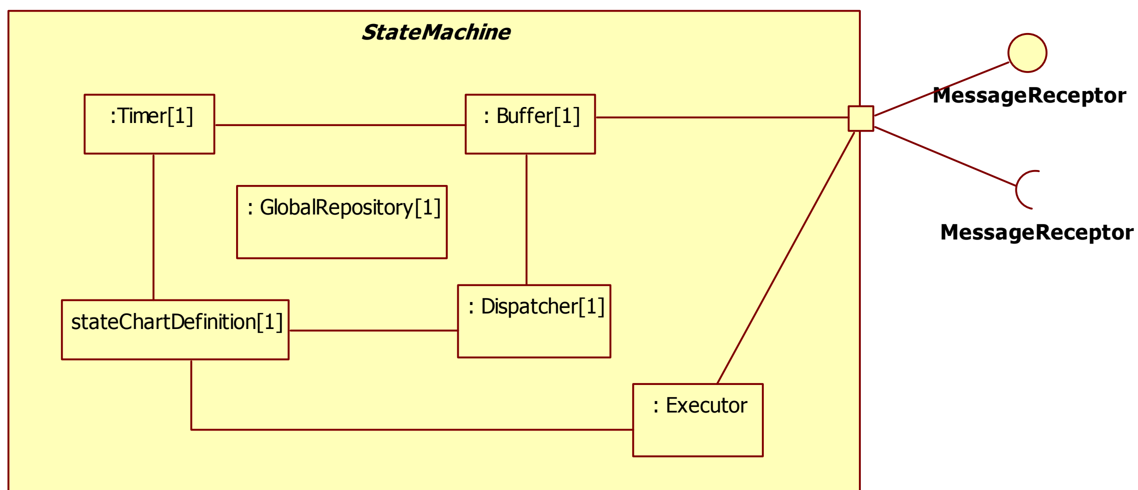


*Figure 5-2. Statechart based component structure.*

In order to more precisely define the kinds of reconfiguration that can be launched at run-time, three elements can be considered: The set of events that the component reacts to, the *statechart* definition and the executor parts. The first one is a system-wide concern; thus, it will not be considered.

The only thing to be defined about events is the form in which they will be materialized in our component. Every event will be notified trough a message. There is an abstract base class called "Event" form all the rest of messages will derive. An infrastructure must be supplied in order to receive those events or notify to the rest of the system.

## 5.2 Executor Changes.

The duty of the executor part is to provide the bodies of all the guards that potentially must be evaluated in order to determine the reactions to be fired and, also, the bodies of the actions that can be fired as a reaction to an event; for

instance, the implementation of the actions and guards that have been associated to all transition (normal or internal) of the *statechart*.

One motivation for changing the executor part at run-time is the toleration of software faults. Upon the detection of those faults, the framework can replace the current one with another one. This way, in an easy form, there can be applied Backward-Recovery [16] algorithm to this particular part.

In a simple form, it can be a class having the mentioned elements plus some pseudo-state variables. For enabling the above mentioned run-time change in a generic manner, figure 5-3 illustrates one way to accomplish it. For that issue, an extra indirection is inserted that whenever a guard must be evaluated or an action must be executed, traps any exception launched from the current executor and, provided the necessary redundancy it an redirect the actual invocation to another version.

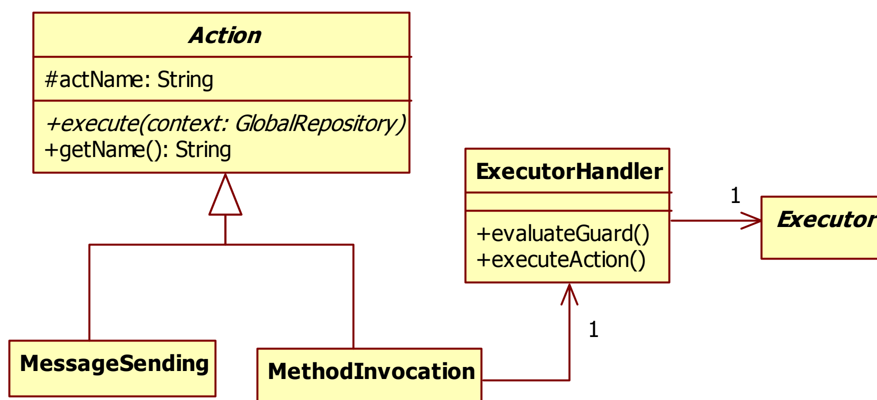


*Figure 5-3. Extra classes to trap possible exception of the executor part.*

## 5.3 Hierarchical State-Structure Specification.

Returning to the *StatechartDefinition* part, this is the element where it is possible to implement behaviour changes in a coarse grained way. As said before, it constitutes the specification of the *statechart*. But instead of serving the *statechart*-model as a design artefact, it is used at run time for determining the pertinent reaction or sequence of actions to be fired.

The dispatcher is an element provided by the framework that implements the UML [17] *statechart* execution semantic and acts as the "motor" for any *statechart*. For that issue, the *statechart* model should be described in some manner.

The selected one is to construct in a programmatic way a composite object that reflects this *statechart* model.

The figure 5-4 shows main classes of the proposed framework in order to describe the desired *statechart*. It focuses mainly in the principal elements for defining the structure of a hierarchical state-machine [18].

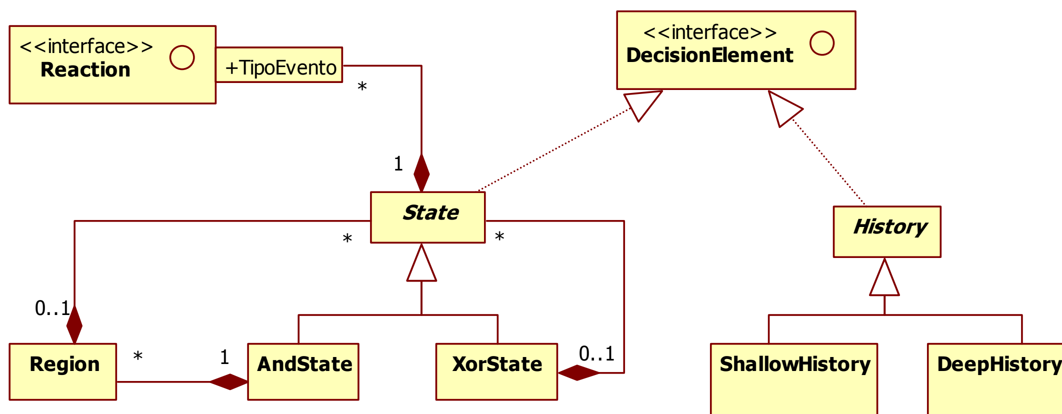


*Figure 5-4. Classes for the construction of the state-structure of a statechart.*

## 5.4 State Behaviour Specification.

Once determined the *statechart* structure, there must be defined the reactions that must be fired in each state: transitions and internal transitions.

The classes that permit us to define those reactions are illustrated in figure 5-5.

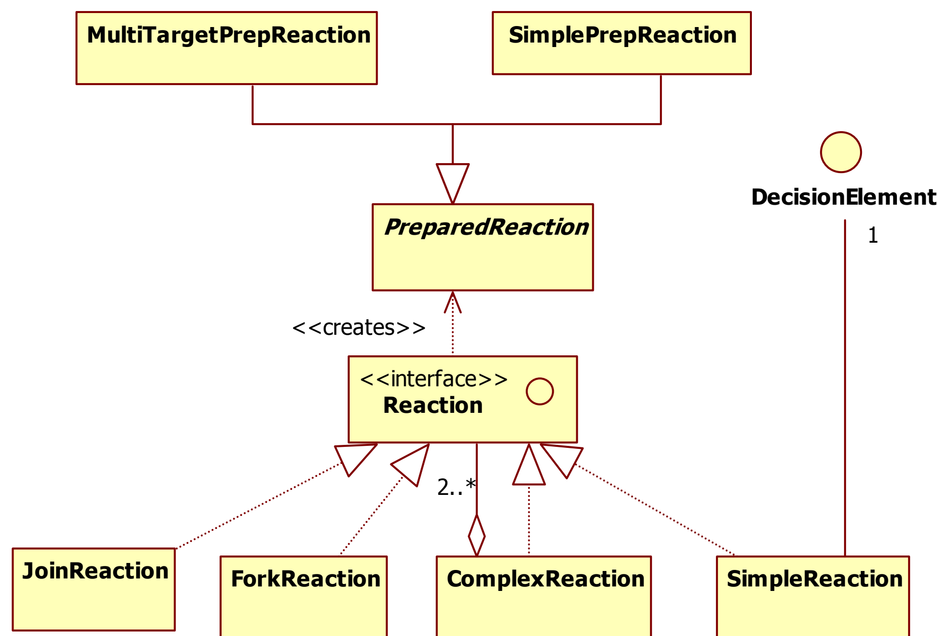


*Figure 5-5. Classes for the definition of the behaviour of states.*

A reaction basically is a transition from one state to other. Depending on the type of transitions it can have multiple sources (join), multiple targets (fork) or even no target (internal transition). Due to implementation issues those concepts are reflected in different classes.

The classes derived from `PreparedReaction` deserve additional explanation. The reason for them is that a transition in a hierarchical state machine actually is a family of transitions in their counterpart plane statemachines.

To illustrate this concept lets take the example of the figure 5-6 and consider the transition labelled as `ev4`. Its source at design time is *state A* but its meaning for run-time is that its real source will be *state A* or *B* depending on which of them is the active one. This means that the possible reactions that would be launched could be different. At least, they can differ in the fact that one mush executes the exit action associated with the *state A* and the other, the one associated with *state B*.

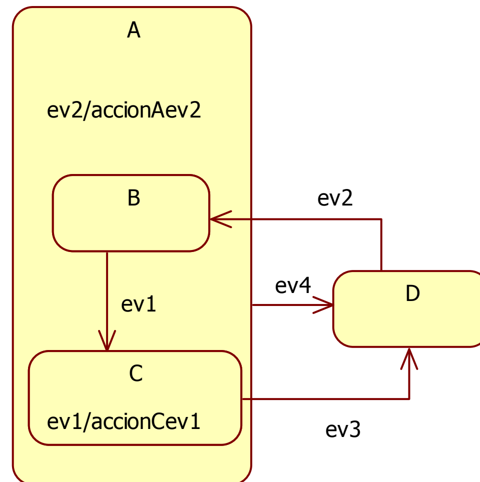


*Figure 5-6. Statechart example.*

Then, the classes that derive from `PreparedReaction` are the exact reactions that must be fired and, for that, they must be created at run-time depending of the *statechart* definition itself but, also, the current active state set. Figure 5-7 illustrates how are created those `PreparedReactions`.

The classes enumerated until now conform the interface of the API that the framework must offer to the developer. Resuming, first we need to define the *statechart* structure, then, based on this structure, we must define the reactions associated with each state and, finally, an executor must be provided (or more, if we want to implement some kind of software fault tolerance).

What the framework must additionally offer is the necessary infrastructure for *statechart* execution, message reception and sending and timeout facility.

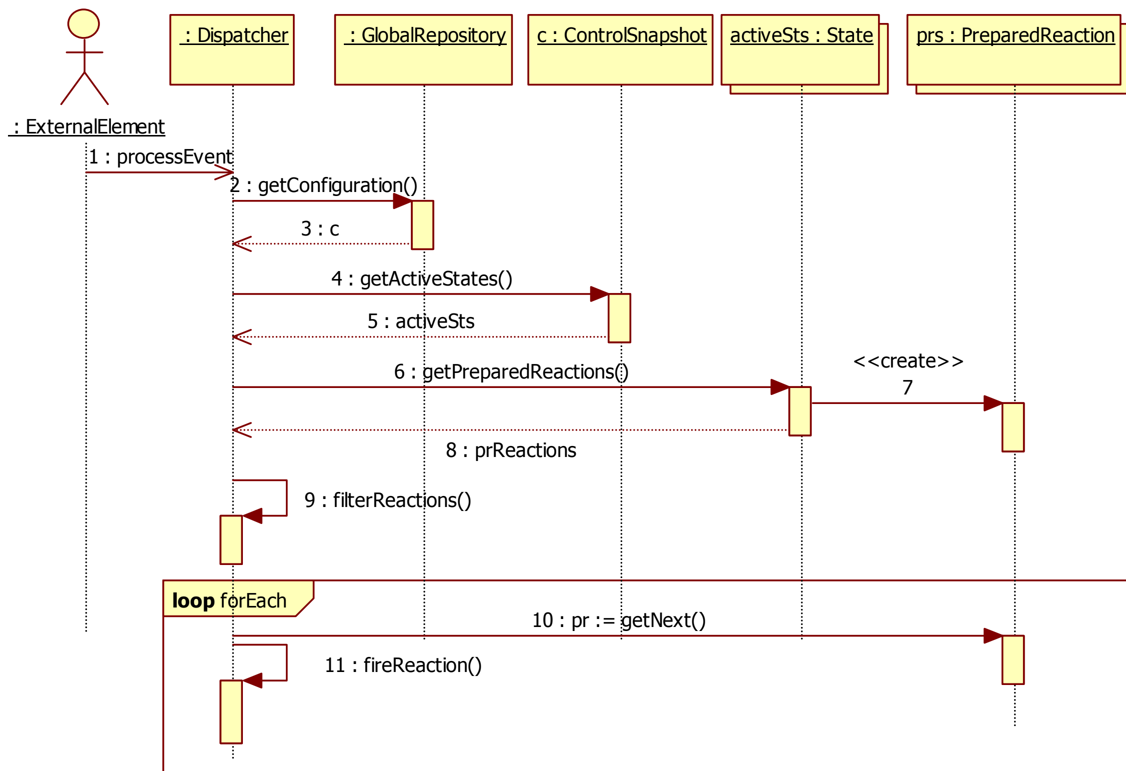


*Figure 5-7. Preparation and firing reactions upon the reception of an event.*

## 5.5 Programming Model.

Until now emphasis has been placed on classes that form the API of the framework. This API will enable a Model Driven style of development but, in a transparent way for the programmer, it will create a reflective architecture of software components. The abstract class of figure 5-8 will be provided, from which all *statechart* -based components will be derived.

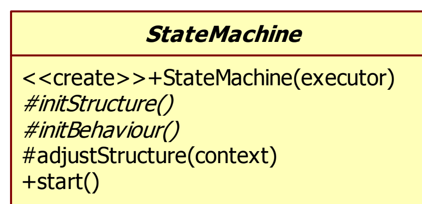


*Figure 5-8. Base class for all statecharts.*

`initStructure`() and `initBehaviour`() are abstract methods that the developer must override. The first will serve to define the state hierarchy and the second, to define the behaviour of each those states. `start`() is a final method that makes the component run.

The following Java code snippet shows how the *statechart* is defined at initialization time by overloading the pertinent methods.

```
class NewStatechart extends StateMachine
{
  ….
  protected void initStructure ()
  {
    sRoot =new XorState ("sRoot");
    rootState = sRoot ;
    sIdle =new XorState ("Idle");
    sRoot . addInitialState (sIdle);
    sControlling =new XorState ("Controlling");
    sRoot . addState (sControlling);
    sStandBy =new XorState ("StandBy");
    sStandBy . setTimer (60*1000); //1min
    ........
  }
  protected void initBehaviour ()
  {
    storeTask =new MethodInvocation (" storeTask ");
    rStoreTask =new SimpleReaction(sControlling ,null ,
                               storeTask ," StoreTask ");
    sRoot. addReaction(EvSetTemperature .class , rStoreTask );
   .....
```

```
    }
  }
```

If the *statechart* control part can be defined at initialization time, this means that can be done at any moment at run time. Then, the behaviour modifications that have been mentioned before consist of such method calls. Apart from reaction and state addition methods, there will also methods for subtracting them.

## 5.6 Background Elements.

The above mentioned approach, to be effective, must be supplied with something called dispatcher, a timeout facility and some messaging infrastructure.

The dispatcher is the core of the *statechart* based component. Having the description provided by the developer, every time a message/event arrives, the dispatcher must determine the reactions to be fired. Although in plane state-machines could be a trivial task, it is not in hierarchical state-machines.

First, the exact transition must be determined. As said before, those transition's instances can be different depending on the current set of active states. Besides that, the proper order of exit and entry actions must be determined and, finally, multiple reactions can be activated if the *statechart* contains multiple regions. For that issue, it is proposed to follow the algorithm described in [19].

Taking into account timing considerations, most model driven approaches take a platform specific way. Even the UML standard, although having defined the "`TimeEvent`" concept, does not define its use in a standard way. If it is aimed to change the *statechart* model at run-time, we must be able to do so with time related elements.

A feasible approach could be to give to every state the ability to launch a time event, if some amount of time has been elapsed since its activation. If it's applied only this mechanism, this would force the developer to employ more states than employing platform specific solutions, but the obtained benefit is that we can work with timeout events at model level. Only the virtual timer of the state of interest has to be specified and activated and, thus, it is possible to modify them following a generic approach.

Finally, a messaging infrastructure is needed if those software components will be aware of the events of interest in their context and notify this environment of relevant circumstances inside it. The content of the messages will be application-specific, but each component has to have some element that puts incoming messages in a buffer for further processing it and sends messages. There could be

multiple implementations because it will depend on the particular communication technology: TCP/IP, CORBA, CAN, CANopen, ZigBee, etc.

# 6. Detailed description of the framework for component reconfiguration.

This chapter is distributed in two sections: one describing middleware tools as is OSGi platform and the other one to describe frameworks using model-centric mechanism for components reconfiguration.

## 6.1 Middleware tools. OSGi and reconfiguration.

Using OSGi, software components can be installed, updated, or removed on the fly without ever having to disrupt the operation of the device. The OSGi Service Platform brings standardized reconfiguration technology to small embedded devices, desktops, and servers enabling these devices to run continuously, even when their software is configured, updated, or augmented.

When updating a bundle at reconfiguration the OSGi Framework first stops existing applications after which their resources are cleaned up. Code is unloaded and replaced with the updated code. After the code is updated, the bundle is restarted. This all happens without restarting the JVM.

For the management of OSGi bundles OSGi uses an OSGi repository, new software (called bundles in OSGi) can be downloaded from repository. Many different kind of OSGi repository's are available for various kinds of environments.

## 6.2 A framework for model-centric reconfiguration of *statechart* based software-components.

In this section, the proposal of section 5 will be detailed.

The targets of the proposed framework are those software components whose behaviour is specified by means of *statechart*s and it will be useful when an adaptation of the component could be specified as a behaviour-change or, more precisely, as a *statechart*-model change. The developer will be supplied with facilities that permit to launch those behaviour changes at run-time.

In section 6.2.1, the programming model will be presented for software-component development based on *statechart*-specification. The presented API is intended to help the programmer at defining such a component in a Model-Driven style of programming and to encapsulate the needed patterns and algorithms that implements UML *statechart* execution semantic.

Section 6.2.2 will add some requisites to the implementers of framework and, finally, section 6.2.3 will show the how run-time *statechart* modification can be effected at-run time.

### 6.2.1 Programming Model.

#### 6.2.1.1 Definition of Hierarchical Structure of States.

First of all, the framework must support a Model-Driven style of development.

The developer of a software component under consideration must first supply a class which methods are the implementation of each guard and action specified in the *statechart*. As figure 6-1 shows, all these guards and actions are materialized as methods of a class.

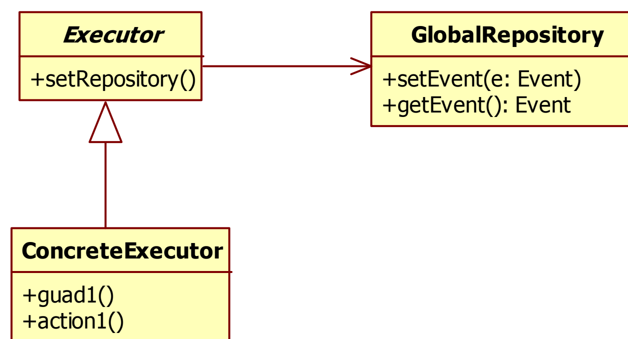


*Figure 6-1. Statechart's guards and actions will be grouped in instances of Executor.*

One problem that arises in event processing is parameter passing. Upon the reception of an event, many guards can be needed to be evaluated and many actions fired. The event types and the parameters that they can carry are application dependent. Thus, in order to not impose application dependent issues to the framework, the `globalRepository` element depicted, in figure 5-2, maintains the event instance that the component is processing in each instant. Therefore, if in a guard or action implementation the developer need to access those parameters, this global repository must be accessed. This ability is inherited from Executor abstract class and is what this framework imposes to the developer.

To develop a hierarchical state machine, the developer is supplied with some classes that permit to define it in a programmatically way.

Figure 6-2 illustrates them showing only the relevant methods (figure 5-5 shows a more general view and their relations with other concerns). The definition of the structure of the *statechart* consists on calling a subset of the methods depicted in figure 6-2, basically, `addRegion`, `addState`, `setHistory`, `setDeepHistory` and `addFinalState`.

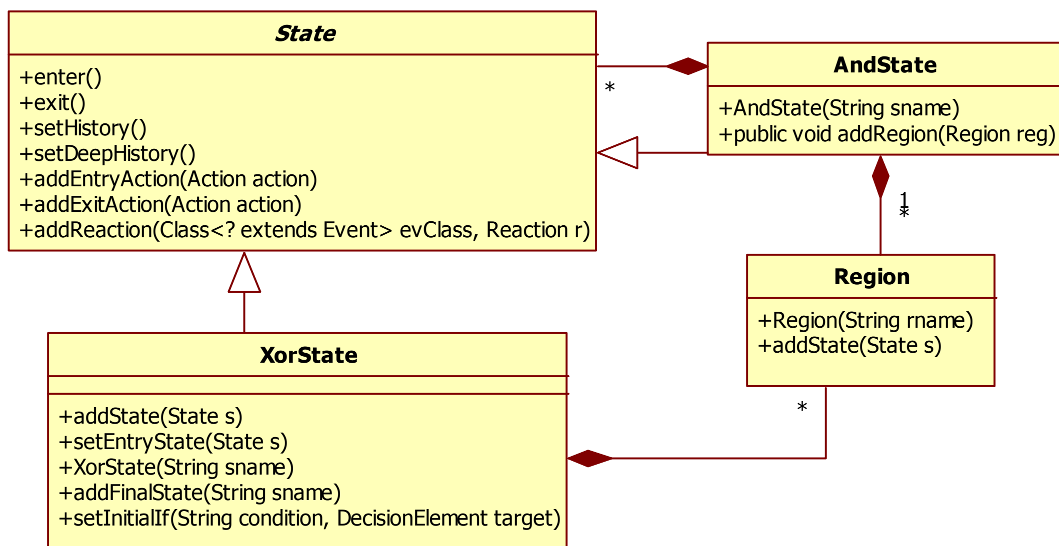


*Figure 6-2. The API for the definition of the state structure of a statechart.*

### 6.2.1.2 Definition of the Reactions of the Statechart.

Once the *statechart*'s structure can be defined, there is needed an API for the definition of the component's reactions. Generally speaking, a reaction is basically a transition among states that can be guarded by some conditions, can have multiple targets or sources and, in the degenerated case, there is no change of state (those last ones are called internal transitions).

Figure 6-3 details the classes oriented for that issue showing only the methods intended for the developer.

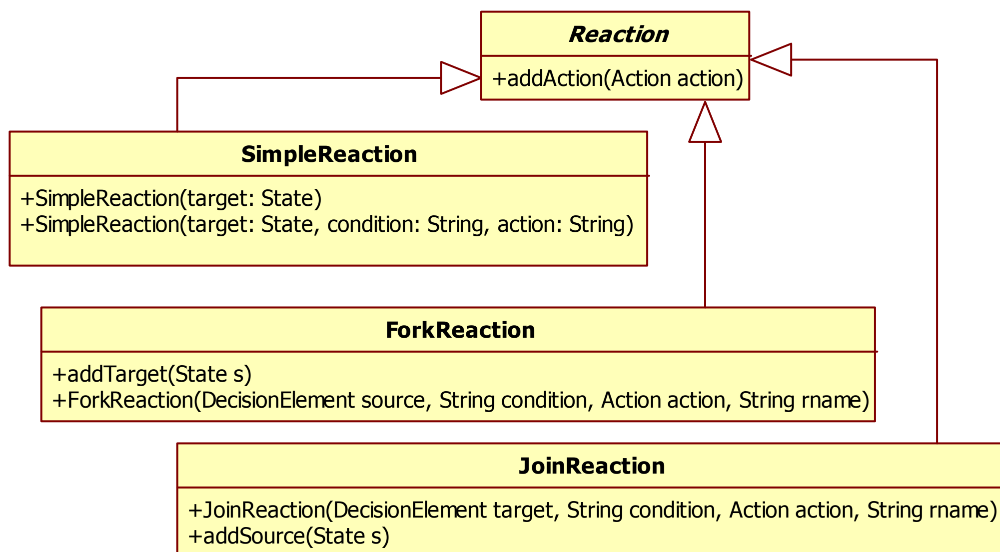


*Figure 6-3. Classes for de definition of the statechart's reactions.*

A reaction is a property of a "particular" state. As presented before, all state classes have the `addReaction` method. In specifying a reaction, we must specify the target (null if it is an internal transition), a guard identifier (its method name in the executor class) and possibly a set of actions. The API for those issues is obvious from figure 6-3.

Every reaction can have an associated action. This holds for state's entry and exit actions. The way to specify will be trough action subclasses and, in particular, `MethodInvocation`'s instances. These constitute an indirection to a method of the executor object. Then, the only thing that the programmer must specify is an identifier or name of the intended method. This is achieved when creating a reaction or through the method `addAction`, figure 5-3.

In some methods of the presented API, it appears the class named `DecisionElement`. The reason of its existence can be illustrated in figure 6-4.

The reaction of this *statechart* in state `s1` upon the reception of the event `AnEvent` is very variable. The target state depends on the run-time computation of `guard1` and `gurad2` and also of the last sub-state of `s3` we have been in. Apart of that, it can be appreciated that even if the ultimate target must be a state, at design time, the end of the transition can be very homogeneous: an state, a shallow history element, a deep-history, a fork synchronization, a choice point, etc.

Thus, for the sake of uniformly coping with all of them, they must implement the interface `DecisionElement` that serves for recursively deciding where does end a

particular instance of a transition but, anyway, the developer shall only know this fact but not use it. This is aimed for the internal working of the framework.

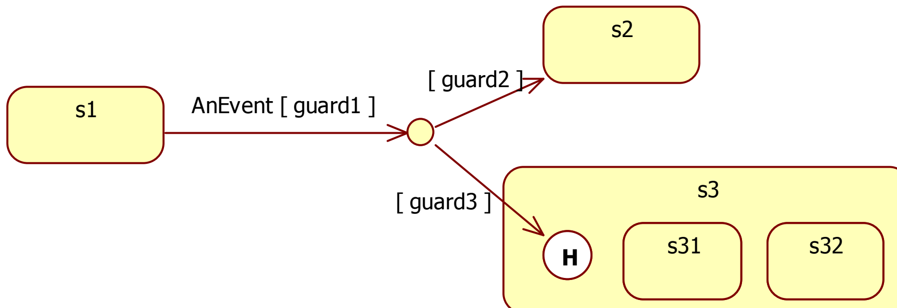


*Figure 6-4. Transitions' ends heterogeneity.*

#### 6.2.1.3 Creating a Software-Component Based on a Statechart.

The part of the API presented before must be used in a precise order. Figure 6-5 shows from where all *statecharts* inherit.

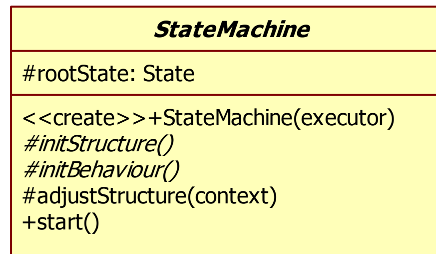


*Figure 6-5. Base class for creation of statechart-based components.*

- *initStructure()* is an abstract method that the developer must define in a subclass of *StateMachine* in order to define the, possibly, hierarchical structure of states. The only thing that is derived from class is a reference to the root state.
- *initBehaviour()* is an abstract method where the developer can define the reactions that may be fired in each state.
- *adjustStructure()* is a method aimed to be used only by the framework. Here, depending of the framework need, some computation can be done after the developer has defined the *statechart*.

- *start()* is the method that must be called in order to make the *statechart* begin processing events.

Obviously, at state-machine creation, an executor object must be provided.

To illustrate with an example let's assume that we would like to implement the *statechart* of figure. For the sake of simplicity, each transition is labelled with a letter and even not showed; we will assume that it's launched after the reception of an event of type `Ev<letter>` and it has associated an action named with the name of the event and numbers of the source and target states.

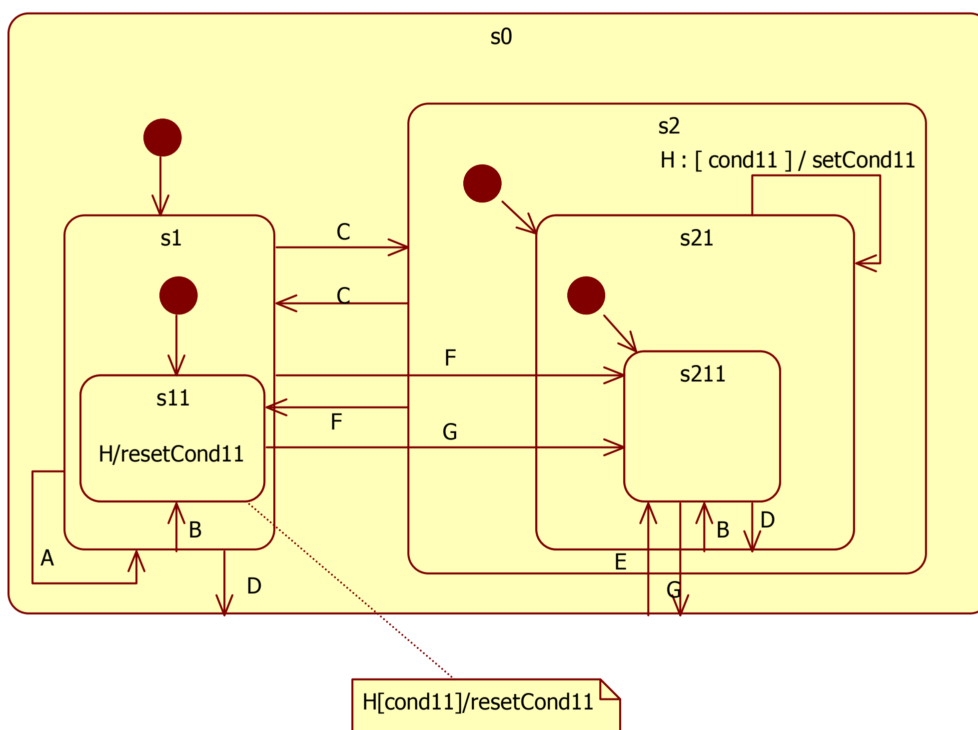


*Figura 6-6. A statechart example.*

Below is shown the code excerpt that, using the proposed framework, implements the *statechart* of figure 6-6 and puts running one instance of it. Of course, the supplied executor must implement the methods in the *statechart* definition.

```
public class ExampleStateMachine extends StateMachine

{
```

```
protected XorState s1,s2,s11,s21,s211;


public SmQuantum(Executor executor) throws Exception
{
  super(executor);
}


protected void initStructure()
{
  XorState root;


  root=new XorState("root");
  rootState=root;
  s1=new XorState("s1");
  root.addState(s1);
  ..........
}




pprotected void initBehaviour()
{
  Action actE0_211=new MethodInvocation("evE0_211");
  Reaction rEvE0_211=
    new SimpleReaction(s211,null,actE0_211,"EvE_211");
  rEvE0_211.addAction(new MethodInvocation("evO"));
  rootState.addReaction(EvE.class,rEvE0_211);
```

```
    Action actEvC1_2=new MethodInvocation("evC1_2");
    Reaction rEvC1_2=new SimpleReaction(s2,null,actEvC1_2,"EvC1_2");
    s1.addReaction(EvC.class,rEvC1_2);
    ........
  }

  public static void main(String[] args)throws Exception
  {
    SmQuantum sm=new SmQuantum(new ExampleExecutor());
    sm.start();
  }
}
```

#### 6.2.1.4 Consideration for Guard Evaluation and Action Execution.

Until now, it has been shown the API of a framework for defining a *statechart* based software component. Once defined it and put it running, the framework will be responsible of invoking, when needed, the proper methods of the provided Executor instance for guard evaluation and action execution.

Many generic approaches can be selected for that issue, depending of the selected platform and programming language. Remember that the parameter passing problem has been resolved through the `globalRepository` element. It temporally stores the event currently being processed. Then, it rests to determine how to instrument the invocation of those guard evaluations and action executions. What we need to take into account is the fact that whatever mechanism being chosen, it cannot be hardwired in code in order to be possible to change it at run-time.

The most generic approach would be to assign a identifier to each of these methods and have an entry point to all of them, specifying which of them need to be executed (similar to operating system calls) but, for the sake of simplicity, assuming that we have a reflective platform like Java or .NET, we will advocate for using method names instead.

Using the reflective facilities of those platforms, only a string must be specified and that is why in the examples and API part presented before, in all reaction specification, the names of the guards and other methods must be specified.

### 6.2.2 Internal Elements for Event Processing.

In the previous part we have seen an API for developing *statechart*-based components and although it must be done in a programmatic way, it serves as a description of the *statechart*-model.

Now, it must be provided an internal infrastructure for event-processing following the semantic defined in UML. The next algorithm is suitable for this purpose:

1. Wait a message (event).
2. Interrogate all the active states, if some reactions need to be fired.
3. Once collected, launch them in any order (no priority implemented).

A transition, as defined in the UML standard for *statechart*s, corresponds to a family of transitions in their flat state-machines. But at run-time, only one transition of this family must be launched and, thus, the pertinent one must be selected.

Then, the next algorithm can be suitable for such purpose upon the reception of event ev:

```
for each active state s
{
  Rection r;

  r=ask s if it has some reaction defined for event ev
    //asking to state s involucrates also its superstates
  if(r!=null)
  {
    ask its guards if currently are true
    if so, add this reaction to the set of reaction to be  launched
      // this is what is called a PreparedReaction
```

```
        }
    }
```

In hierarchical state machines, launching a reaction is also undefined until run-time. For example, if the target of a transition has been divided into substates and so on, it must be determined which one of them must be entered. This circumstance aggravates if we take into account that initial states can be guarded and that the target state can have a shallow or deep history substate.

For that issue, the class of figure 6-7 serves for holding the information of each concrete reaction and has the method `markEntryPath()` that must be executed before its launching. It marks all the states that must be entered when the reaction will be fired.

| ***PreparedReaction*** |
| --- |
| #stateDefinedIn: State<br>#startingState: State<br>#action: String |
| +PreparedReaction(stateDefinedIn: State, startingState: State, action: String)<br>+getStateDefinedIn(): State<br>+getStartingState(): State<br>*+isSelfTransition(): boolean*<br>*+markEntryPath(): State*<br>+executeAction(executor) |

*Figure 6-7. Methods of a Reaction to be launched.*

And finally, the algorithm to fire a particular reaction (`PreparedReaction`) must be the next (it's expressed in Java but is intended to serve as only an algorithm).

```
State sTop, s;

sTop=pr.markEntryPath();

if(sTop==null)//enters if pr is an internal transition
{
```

```
      if(pr.isSelfTransition()) pr.getStateDefinedIn().exit();
      pr.fireAction(context);
      if(pr.isSelfTransition()) pr.getStateDefinedIn().enter();
    }
    else
    {
      s=pr.getStartingState();
      if(sTop==pr.getStateDefinedIn()) sTop.exitTillTop();
      else s.exitTill(sTop);
      pr.fireAction(context);
      if(sTop==pr.getStateDefinedIn()) sTop.enterWithoutAction();
      else sTop.goMarked();
    }
  }
```

Some comments must be done about this algorithm:

- It distinguishes between internal, self and the rest of transitions.
- When marking the entry path, the state until which must be exited is determined.
- Following the UML execution semantic, first exit actions must be fired beginning from the most nested state upward. Then, the actions associated with the Reaction must be fired and, finally, entry actions must be executed in all the marked states beginning from less nested state downward.

This set of algorithms can be implemented in a materialization of the interface of figure 6-8. A requisite for it is that it must be provided the reference of the `globalRepository` element (for example at construction time).

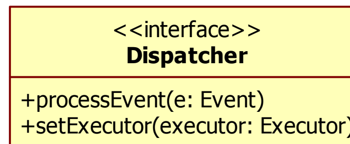


*Figure 6-8. Interface for event Processing Classes*

The final element of this framework is an event generator. This is the most open issue, due the differences in nature that can be in events' sources.

In order to define a generic approach, the *statecart* will be notified about the occurrence of the events of interest by means of messages. The framework will provide a message bugger for a *statechart* and also a thread for event processing. How those messages are put in the buffer is an application independent issue and, thus, left open.

### 6.2.3 Modifying the *Statechart*-Model at Run-time.

Now that an API has been shown that, thanks to framework, makes possible to change at run-time the *statechart*-model that governs the behaviour of a software component, it rest to define the mechanism that permits to accomplish it.

First of al, it must be highlighted the ability that this framework gives software components to modify its behaviour at run-rime but, at least, there are the following alternatives: to have an externally accessible port through which another element can give instruction to change its behaviour; another variant of this is to externally load an element capable of changing the behaviour.

But because of practical reasons, the selected approach is the one illustrated in figure 6-9.

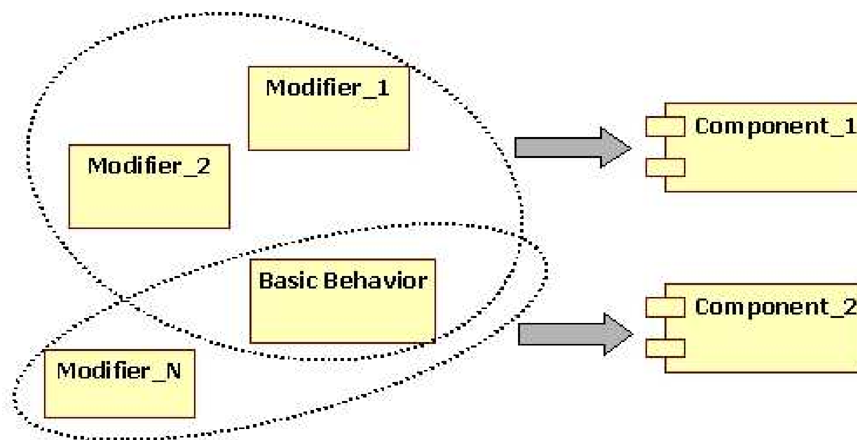


*Figure 6-9. Software components with behaviour modification ability.*

Each *statechart*-based software-component will be supplied with a basic behaviour and possibly various behaviour modifiers: i.e., having at initialization-time a basic *statechart* that governs the component's behaviour at any moment at run-time, it will be possible to launch a behaviour modifier. This will change the *statechart* that governs the software-component behaviour.

The intended mechanism aims allowing to program *statechart* modification if, holding some condition, a particular event is received. The condition will be a guard and the fact of being in some particular state. The event that launches such modification is intended to carry the notification of some circumstance variations in the environment and, thus, the component adapts its behaviour in response to this environmental change. Note that all different modification is conceived at design time.

What this infrastructure support is to design each different behaviour or working mode in a separate way: i.e., it offers an adaptation engineering mechanism.

Next, the selected approach to implement this mechanism will be described.

Conceptually, as illustrated in figure 6-10, each state-machine could have various behaviour-modifiers. Those will be implemented as state-machines' methods. We have seen before the ability that this framework offers to define a *statechart* at initialization time in a programmatic manner. Thus, it is also possible to effect *statechart* modification at any instant at run-time.

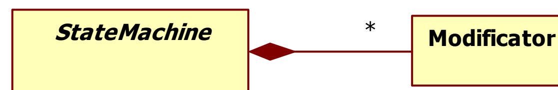


*Figure 6-10. Each state-machine could have various behaviour modifiers.*

Now, it rest to define when those modifiers will be launched. Extending the class diagram of figure 5-3, as shown in figure 6-11, a modifier can be implemented as an action associated to any reaction. For that issue, it will need the modifier's identifier.

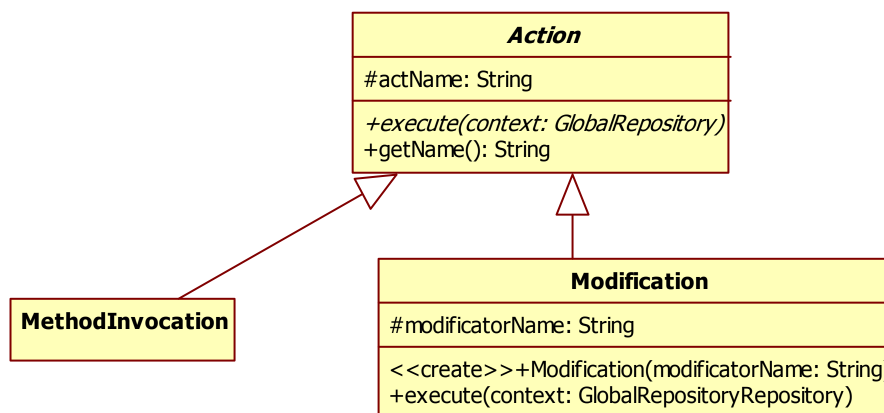


*Figura 6-11. Modifiers implemented as any actions associated to a statechart.*

To illustrate with a simple example, let's assume an almost empty *statechart* that upon the reception of a particular event must become the *statechart* of figure 6-6. The next code shows only the methods on charge of defining the *statechart* at the initialization time. As it can be seen, the only behaviour implemented is to execute the method `modifier01`() upon the reception of event EvA.

```
public class ExampleStateMachine extends StateMachine
{
  public void initStructure()
  {
    XorState root;
```

```
    root=new XorState("root");

    rootState=root;

  }


  public void initBehaviour()

  {

    Action m1=new Modification("modifier01");

    Reaction r=new SimpleReaction(null,null,m1,"mod01");

    rootState.addReaction(EvA.class, r);

  }

  public void modifier01() {.....}

}
```

The method `modifier01`() is in charge of modifying the *statechart* that governs the behaviour of the component. Its implementation is almost identical to the code shown in relation to the *statechart* of figure 6-6.

The framework presented until now offers facilities to add elements to a *statechart*: sub-states, regions, reactions, etc., but in order to apply modifiers, it is also necessary to be able to eliminate those elements.

An immediate implementation of that is to offer an counterpart `removeXX`() method for each `addXX`() method in the state classes and its subclasses, `region`, `reaction` and its subclasses.

# Acknowledgements

The eDIANA Consortium would like to acknowledge the financial support of the European Commission and National Public Authorities from Spain, Netherlands, Germany, Finland and Italy under the ARTEMIS Joint Technology Initiative.

# References

[1] Roman Obermaisser, Bernhard Huber: The GENESYS Architecture: A Conceptual Model for Component-Based Distributed Real-Time Systems. Lecture Notes in Computer Science, Springer Berlin / Heidelberg, , Volume 5860/2009, in Software Technologies for Embedded and Ubiquitous Systems, p296-307.

[2] Paolo Costa, Geoff Coulson, Cecilia Mascolo, Luca Mottola, Gian Picco, Stefanos Zachariadis: Reconfigurable Component-based Middleware for Networked Embedded Systems, in International Journal of Wireless Information Networks, Volume 14, Number 2, June 2007, pp. 149-162(14).

[3] Andreas Doblander, Bernhard Rinner, Norbert Trenkwalder, Andreas Zoufal: A Middleware Framework for Dynamic Reconfiguration and Component Composition in Embedded Smart Cameras, in WSEAS Transactions on Computers, 5(3) pages 574-581, WSEAS Press. March 2006.

[4] Bernhard Rinner, Markus Quaritsch: Embedded Middleware for Smart Camera Networks and Sensor Fusion, in Multi-Camera Networks: Principles and Applications (July 2009).

[5] Roman Obermaisser, Bernhard Huber: The GENESYS Architecture: A Conceptual Model for Component-Based Distributed Real-Time Systems. Lecture Notes in Computer Science, Springer Berlin / Heidelberg, , Volume 5860/2009, in Software Technologies for Embedded and Ubiquitous Systems, p296-307.

[6] R. Obermaisser, H.Kopetz (editors): GENESYS: A Candidate for an ARTEMIS Cross-Domain Reference Architecture for Embedded Systems, SpringerVerlag, 2009, ISBN 978-3-8381-1040-0

[7] The INDEXYS project, funded under the ARTEMIS Programme (Topic "SP5 Computing Environments for Embedded Systems"), and coordinated by TTTech Computertechnik AG, Austria, http://www.indexys.eu/

[8] Marco Castaldi, Antonio Carzaniga, Paola Inverardi, and Alexander L. Wolf, A Lightweight Infrastructure for Reconfiguring Applications, in Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2649/2003, Software Configuration Management, 2003, p201-205

[9] J.P.A. Almeida, M. Wegdam, M. van Sinderen, and L. Nieuwenhuis. Transparent Dynamic Reconfiguration for CORBA. In Proceedings of the 3rd International Symposium on Distributed Objects and Applications, pages 197–207. IEEE Computer Society, September 2001.

[10] T. Batista and N. Rodriguez. Dynamic Reconfiguration of Component-Based Applications. In Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems, pages 32–39. IEEE Computer Society, June 2000.

[11] J. Berghoff, O. Drobnik, A. Lingnau, and C. Monch. Agent-Based Configuration Management of Distributed Applications. In Proceedings of the 3rd International

Conference on Configurable Distributed Systems, pages 52–59. IEEE Computer Society, May 1996.
[12] A. Young and J. Magee. A Flexible Approach to Evolution of Reconfigurable Systems. In Proceedings of the IEE/IFIP International Workshop on Configurable Distributed Systems, pages 152–163, March 1992.
[13] M. Endler. A Language for Implementing Generic Dynamic Reconfigurations of Distributed Programs. In Proceedings of 12th Brazilian Symposium on Computer Networks, pages 175–187, 1994.
[14] Adler, Rasmus, Schneider, Daniel Trapp, Mario:" Development of Save and Reliable Embedded Systems Using Dynamic Adaptation". In: M-ADAPT Model Driven Software Adaptation, Berlin (2007)
[15] Oscar Nierstrasz, Marcus Denker, and Lucas Reggli. "Model-Centric, Context-Aware Software Adaptation". In Software Engineering for Self-Adaptive Systems, Springer-Verlang 2009.
[16] Pankaj Jalote. "Fault tolerance in distributed systems". Prentice Hall, 1994.
[17] OMG Unified Modeling Language (UML), Superstructure V2.2, chapter 15, State Machines, OMG, February 2009
[18] David Harel. Statecharts: "A visual formalism for complex systems". Science of Computer Programming, pages 231–274, 1987.
[19] Miro Samek. "Practical statecharts in C/C++": An introduction to quantum programming. CMP Books, 2002.