MOIRAE – An Innovative Component Architecture with Distributed Control Features

Katia Leal[1], José Herrera[1], José M. Peña[1]*, and Ernestina Menasalvas[2]

[1] DATSI, Universidad Politécnica de Madrid, Spain
[2] DLSIIS, Universidad Politécnica de Madrid, Spain

Abstract. Today's distributed systems are very complex applications. Taming this complexity, providing both adaptabity and performance is still an open issue for distributed architectures. Component-based development is a key technique to design plugable elements for a distributed system. This article describes a new architecture for distributed component systems. This architecture defines each of its components as two different planes (i) operational actions and (ii) control policies. New issues in terms of adaptability and flexibility are available by this new architecture, called MOIRAE (Management of Operational Interconnected Robots using Agent Engines).

1 Introduction

Different component architectures has been proposed [1,6,5] in order to build complex distributed systems. MOIRAE architecture is an inovative approach for distributed environments with a strong emphasis on what has been called control and behaviour management. A MOIRAE application is designed under a tightly-coupled model called Policy/Mechanism (P/M) model. Using this architecture high flexible and adaptative systems can be designed to tackle performance-restrictive problems like distributed data mining scenarios, as we have presented before [4,3].

This article introduces a new development environment based on the MOIRAE architecture, MOIRAEToolKit. This environment allows developers to design and implement MOIRAE components in a easy manner.

The rest of the article is organized as follows. Section 2 presents the P/M model and its advantages in adaptative environments. In section 3 the MOIRAE architecture is reviewed. Section 4 introduces the new development environment called MOIRAEToolKit. The most important module of the MOIRAEToolKit, the Code Generator, is presented in the section 5. The article ends with a simple example, presented in section 6 followed by the conclusions and future lines shown in section 7.

* This work has been partially supported by UPM project #14494.

2 Policy/Mechanism Model

Complex applications like operating systems (OS) and database management systems (DBMS) have to provide both efficient resource management (performance) and extensibility with new requirements, services and tasks (flexibility). These applications are designed following a clear distintion between (i) operations and functions provided by the system and (ii) the rules under which these functions are managed. These rules decide when and what functions are used in order to perform user tasks, as well as which parameters and options configure them. These functions are called mechanisms and they are implemented for each of the simple operations the system provides. The policies are the separate descriptions of those management and configuration rules.

The Policy/Management model (P/M model) is a well-known technique in the design of complex systems. P/M models is the foundation of the MOIRAE reference architecture. This architecture extends how this technique has been used, making it applicable beyond the design phase. A MOIRAE component is an element that implements one or more mechanisms that can be managed by dynamic runtime-configurable policies.

3 MOIRAE Architecture

MOIRAE [2] is a reference architecture described by 4 different models:

① **Component Model**: Describes the atomic elements of the architecture. These elements are called *components*. This model defines each of the modules which the elements are divided into. Some of these parts have specific functionalities others depend on the operational function provided by the component.

② **Relationship Model**: Specifies the different communication methods among the components. This model defines relationship schemas depending on the cardinality of the members (unicast vs multicast), the scope of the relationship (public vs private) and the addressing methods (public vs anonymous).

③ **Architecture Model**: Combines Relationship and Component Models to describe how the architecture solves execution problems. This model provides a description of the cooperative work performed by the architecture.

④ **Control Model**: Defines how control decisions are performed. This model shows the mechanism used to solve control conflicts when they arise. Control Model is based on the concept of *control policies*.

3.1 Component Model

This Model describes the structure that every element takes as a skeleton. MOIRAE architecture provides control features based on the functionalities provided by each of the elements. These atomic elements are called *components*.

Following the M/P paradigm each component defines two kinds of functions, operational functions and control functions. Each of these groups of functions is implemented in a *plane*. The *Operational Plane* includes all the modules that provide the operational functions. The *Control Plane* provides the features necessary to control the other *plane* as well as to interact with other *Control Planes*.

The *Control Plane* of a MOIRAE component is a control agent itself. The elements inside of this plane are:

① *Event Sensor*: This element detects the abnormal situations that occurs inside the operational part of the component. These sensors are activated as asynchronous messages.

② *Actors*: They are a group of functions executed by the control plane to modify the operational plane. Using these functions the control part can execute actions, modify parameters or enable/disable operational functionalities.

③ *Sensors*: These elements are used when the control plane wants to scan the situations of the operational part. As a difference with the *Event Sersors*, these are synchronous information-gathering mechanisms.

④ *Control Interface*: This module allows the control plane to interact with the control planes of other components.

⑤ *Policy Engine*: It is the central module of the control part. This element manages the decisions taken by the control plane. When the control plane is summoned, either by a conflict (via *event sensor*) or by another component (via *control interface*), the *policy engine* is activated. This module may use the *sensors* to evaluate the status of the operational plane and also interact with other control components to decide the actions to be performed. These actions are either executed locally by the *actors* or submitted to another component using the *control interface*.

3.2 Relationship and Architecture Models

These two models describe the possible organization schemas of the components in the system. The *Relationship Model* defines the cardinality, scope and visibility of component relations. The *organization graphs*, interaction topologies of the components, are classified by the *Architecture Model*. Both Models are very important parts of the MOIRAE architecture, but they are not so related with the development environment MOIRAEToolKit. For a detailed description of this model see [2].

3.3 Control Model

The last Model of the architecture is the Control Model. This Model shows how control decisions are taken either locally or as a contribution of different control planes. When the control plane is activated, for example when a conflict is detected by the *event sensor*, the *policy engine* evaluates the alternatives to solve the problem. As a result, the control plane returns a sequence of actions to be performed to solve the conflict. For complex problems the control plane would be unable to achieve an appropriate solution by itself. Control Model specifies three different control actions that rule the cooperative solution of complex problems:

❒ **Control Propagation**: When a control plane is unable to solve a problem it submits the problem description (e.g.: the conflict) and any additional information to the control plane immediately superior in the hierarchy.

❒ **Control Delegation**: After receiving a control propagation from a lower element, the upper element may take three different alternatives:

① If it is also unable to solve the problem it propagates up the conflict as well.

② If it can solve the problem, it may reply to the original component with the sequence of actions necessary to solve the problem. This original component executes these actions.

③ In the last situation it is also possible that the component, instead of replying with the sequence of actions the component may provide the p-DB information necessary to solve the problem in the lower component. This information could be used also in any future situation. This alternative is called *Control Delegation*.

❒ **Control Revoke**: This action is the opposite to the *control delegation* one. Using this control action any upper component in the hierarchy may delete information from the p-DB of any lower element. This action may be executed anytime and not only as a response of a *control propagation*.

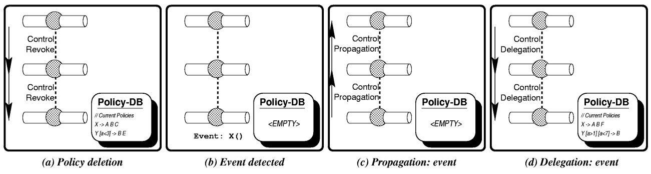

Fig. 1. Policy management actions

4 MOIRAE ToolKit

MOIRAEToolKit is a collection of tools (as introduced by figure 2) to develop the components of the MOIRAE control architecture presented before. The figure shows the relationship among the different elements that are included in MOIRAE environment. These tools are:

❒ Analyser (*MOIRAE Parser*)
❒ Component Compiler
❒ Policy Verifier
❒ Document Generator

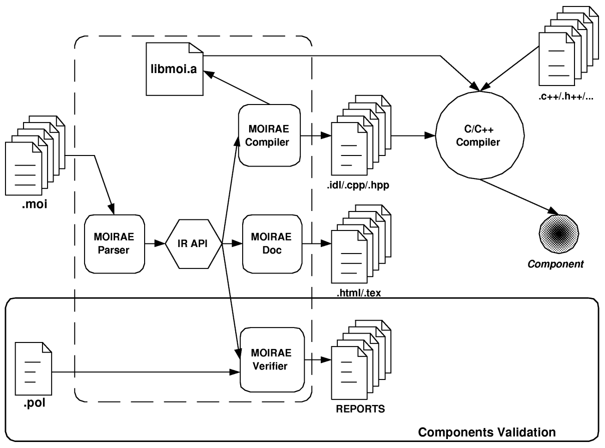

Fig. 2. MOIRAE tools.

4.1 Analyser (*MOIRAE Parser*)

This first module deals with lexical, syntactic and semantic (L/S/S) analysis of component description and implementation files (*.moi*). This element is the single frontend of most of the modules as its output is consumed by the different tools of this suit. This result is an internal structure called **IR** (*Internal Representation*). This structure is an abstract representation of the useful information provided by user through *.moi* file. This representation is a L/S/S-proven description of the defined component itself as well as the rest of the components it is related with. This module it is not actually a tool but it is linked with all the other modules of the systems in order to perform this common verification and analysis tasks.

4.2 Components Compiler

This module is basically a code generator tool. It takes the descriptive information represented by the IR and, depending on the library structure, it generates several high level language files, specifically in C++. Some of the classes generated by this tool, inherit from library classes. The others files depend directly on the options specified by user, such as the kind of middleware or the type of policy engine. This module is a key element of the MOIRAEToolKit because it provides part of the final code that will be compiled and linked when the actual component is built.

4.3 Policy Verifier

Policy Verifier is the second tool of the development environment. This module takes one or more component descriptions (IR structures) and a list of policy files. This module verifies whether the set of rules and procedures defined by the policies are able to solve all the possible conflicts generated by this group of components. As a result this module reports if the components can be properly controlled by the provided policies.

The verifications performed by this tool are approached in the following ways:

① Component local verification: implementation of all the functions declared.
② Component global verification: references to other components functions.
③ All events are covered by policies rules.
④ Conflicts and infinite loops detection.

4.4 Document Generator

The last tool of MOIRAE environment generates documentation based on HTML and LaTeX files, that describes services, commands and other component elements as well as it's implementation to the detail level defined in the *.moi* files.

This module reports all the information related to components in a more bright and compact way. In a future, this mechanism could be extended to generate component description tags to be processed by other development environment or registering services.

5 Component Code Generation

The source file of a component description is the *.moi* file. Using this file the Code Generator creates several C++ files that includes subclasses inherited from the MOIRAE component library classes. All the code generated deals with communication, control and registration operations. The actual operations (the tasks performed by the component) should be programmed as different files that will be compiled when the executable file is linked. A set of auxiliar files required for the compilation and execution of the component are also generated by this mechanism. These files are a default Makefile and several scripts to start/stop component execution. The *.moi* file has a source code according to the MOIRAE component description syntax. This description represents a component definition and it has to be provided by the developer.

The Code Generator module (presented above), uses the IR (Internal Representation) structure generated by the MOIRAE Parser. Figure 3 shows the complete sequence of the code generation procedure. In this figure, it can be seen that Code Generator uses IR class to generate the result set of files. Code Generator obtains component description from IR structure and generates subclasses that inherit from de MOIRAE library.

5.1 MOIRAE Library

MOIRAE library is a set of C++ classes each of which represents part of the structure and functionalities of a generic MOIRAE component. For example,

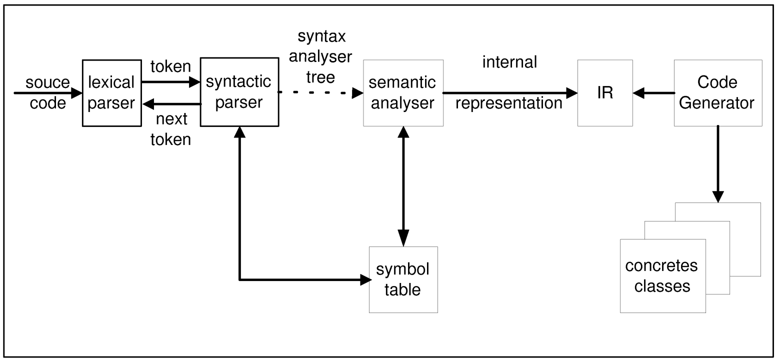

Fig. 3. Relationships between Code Generator, IR, and MOIRAE Parser

there is a class that represents the 'Operational Interface' of a MOIRAE component, as well as the 'Control Plane', 'Event Sensors' and so on. Using the attributes and methods contained in library classes, the features provided by a generic MOIRAE component can be obtained.

Code Generator creates a set of classes inherited from library classes, these new classes are intended to contain the specific functionalities of each of the components user specifies.

MOIRAE library includes the following classes:

❒ The different ways of communication between two agents (sockets or KQML), that is, between different control planes of several components.
❒ The different kinds of information that 'Policies Engine' and 'Control Plane' need to exchange.
❒ The different kinds of policies engines (reactive, deliberative ...)
❒ The basic skeleton of an operational or command procedures implementation, and so on ...

One of the advantages of the library is that it is independent from the final technology the user selects for the component. Component Compiler must generate code appropriate technology selected.

MOIRAE library presents different hierarchies for those elements who need a detailed level of particularization. It is the case of policy engines and communication mechanisms between agents. It does not matter the kind of policy engine, all of them work in the same way because they provide the same methods inherited from class at the top level.

5.2 Parser + IR

In this section we will explain the two auxiliar components used by Code Generator, MOIRAE Parser and IR structure.

MOIRAE Parser. MOIRAE Parser makes lexical, syntactic and semantic analysis of a MOIRAE component definition.

This component has two elements. The first one (called Scanner), performs lexical analysis of the MOIRAE file, this file has *.moi* extension. The second one, named Parser, contains the syntactic and semantic analyser.

In general, MOIRAE Parser has as an input file with *.moi* extension (the definition of the component to be implemented), and as an output, stores all useful data in the internal storage structure named IR.

IR (Internal Representation). The IR, is a intermediate storage structure, composed by lists, which stores all information from MOIRAE files. Each list is a set of instances of classes which representes concepts like services, commands, events or relationships of a MOIRAE component. IR is generated by MOIRAE Parser, and it is used as a bridge tool between Code Generator and Parser.

In addition, IR structure is requested by the rest of the MOIRAE tools. Once the MOIRE Parser has been executed the IR is taken as the validated representation of the component. An IR structure stores all information from the component definition files. This representation is not a file format rather than a memory structure to publish component description for other MOIRAEToolKit modules.

All the modules of the system are linked with the MOIRAE Parser code in order to call it to obtain an IR representation from a *.moi* file.

6 Example

This example shows how a component named `Account` is developed using the MOIRAEToolKit environment. `Account` component implements the typical functions that we can do with a bank account like deposit and withdraw an amount of money, and gets actual balance of the account.

The next code, represents a MOIRAE definition of the Account component, following MOIRAE syntaxis. As part of the design this component has a control event `redNumbers`, that the component will throw when the balance of account is negative.

```
// Set middleware CORBA_MICO
#option middleware = CORBA_MICO
component Account {
  // Put implementation class
  #option imple_modulo_op = "Implementation"
  events { // Events Declaration
     redNumbers() ; }
  services { // Services Declaration
     void deposit ( long amount );
     void withdraw( long amount ) throw (redNumbers);
     long balance (); }
}
```

The implementation class has the implementation of the three services (`deposit`, `withdraw` and `balance`). Once the *.moi* file is programmed, the code generator program is ran with the sentences shown in:

```
[jdoe@server] moirae Account.moi
[jdoe@server] make
[jdoe@server] Account myAccount accounts
```

When we typed the first sentence `moirae` program has generated the files shown in the next table.

Generated Files		
Files depending of CORBA middleware		
Account.idl	*Main.cpp*	*Account_impl.hpp/cpp*
Files inherited from Moirae Library		
Component_Account.hpp/cpp		*ControlPlane_Account.hpp/cpp*
OperationalPlane_Account.hpp/cpp		*ControlInterface_Account.hpp/cpp*
OperationalModule_Account.hpp/.cpp		*PoliciesEngine_Account.hpp/cpp*
OperacionalInterface_Account.hpp/cpp		
EventSensor_redNumbers_Account.hpp/cpp		
Files depending of program language, C++		
Makefile		*MakeVars*

The Makefile generated is used by the `make` tool. The second sentence shows how the `Account` executable is built. The third sentence is needed to run the component. The first parameter specifics the name of the instance, the second parameter specifics the context name where the instances are registered. Now, *myAccount* instance is running on the server side. On the client side, we run a C++ program which uses the Account component.

```
[jwhy@client] Client myAccount accounts
```

This program uses a reference of *myAccount* object. The output of the client program is the set of next sentences: `Deposit: 700 euros`

```
Balance: 700 euros
Withdraw: 600 euros
Balance: 100 euros
Withdraw: 150 euros[1]
Exception: MOIRAE[2]
```

The trace message printed by the server is: `Throw redNumbers event`[3]

[1] At this moment `myAccount` component throws `redNumbers` event, and its control plane captures the event and deals with the procedures described by the policy rules. One of the possible actions proformed by the control plane could be user notification using a remote exception, other possible actions are accept the negative balance or report to higher level component about possible penalties.

[2] Client captures MOIRAE exception and prints the message "Exception: MOIRAE".

[3] The object server named `myAccount` throws `redNumbers` event: Control plane summoned.

7 Conclusion and Future Lines

MOIRAE is a new generic architecture for distributed component environments. One of the key elements of this new architecture is the possibility to deal with distributed control policies in order to provide hierarchical management facilites. Using simple, but flexible, distributed control mechanisms, it is possible to change during runtime the behaviour and control procedures the system is ruled by.

MOIRAEToolKit is a development environment designed to create MOIRAE components. Using its tools a new component can be defined using a description syntax. Code Generator tool uses this component description to create the code wrapper for the services described by the component. Developers only need to implement the remote services provided by the component, all the requirements in terms of communication, component registration and advanced control features are generated by the tool.

Policy Verifier and Document Generator are both tools from the same suit that checks policy rules against the component definitions and generates documents with component description.

At the moment, the current version of the MOIRAEToolKit provides only very simple control plane implementations. This part is the main development line of this project. A second line is the extension of the validation mechanisms in order to validate complex situations like event domain edge conditions.

Other minor issues are to extend the number of options that users can specify for their components. It would be a good idea to provide as much as possible options, about providing different middlewares, policies engines, communication methods and languages. At the present moment only C++ and CORBA objects are supported. This suit can be downloaded from HPDA project web site[4].

References

1. Object Management Group. Ccm tutorial. Document: omg/00-06-01, June 2000.
2. José M. Peña. Distributed Control Architecture for Data MiningSystems. PhD thesis, DATSI, FI, Universidad Politécnica de Madrid, Spain, June 2001. Spanish title: Árquitectura Distribuida de Control para Sistemas con Capacidades de Data Mining'.
3. José M. Peña, F. Javier Crespo, Ernestina Menasalvas, and Victor Robles. Parallel data miningexep erimentation using flexible configurations. LNAI, 2475:441–448, 2002.
4. José M. Peña and Ernestina Menasalvas. Towards flexibility in a distributed data mining framework. In Proceedings of ACM-SIGMOD/PODS 2001, pages 58–61, 2001.
5. Dale Rogerson. Inside COM: Microsoft's Component Object Model. Microsoft Press, 1997.
6. Sun Microsystems. Enterprise JavaBeans 2.0 specification. Whitepaper, Sun Microsystems, 1999.

[4] HPDA Project: `http://nova.ls.fi.upm.es/hpda`