Generation of distributed programs in their target execution environment

Frédéric Gilliers, Jean-Pierre Velu
frederic.gilliers@sagem.com
jean-pierre.velu@sagem.com
SAGEM SA
Etablissement d'Eraggny, Avenue du Gros Chêne
95610 Eragny
B.P. 51 - 95612 Cergy Pontoise Cedex, France

Fabrice Kordon
Fabrice.Kordon@lip6.fr
Laboratoire d'Informatique de Paris 6/SRC
Université Pierre & Marie Curie
4, place Jussieu
F-75252 Paris CEDEX 05, France

***Abstract*— This paper presents how we use L*f*P, a formal-based, Object Oriented notation dedicated to the development of distributed application.**

The language comes with a development methodology which emphasizes the separation between the control aspect of the application, and the computational aspect. Specifications written in L*f*P focus on the control part of the application which is known to be a difficult issue of distributed applications. The corresponding code is then automatically generated to implement the behavior in the target execution environment.

This paper briefly presents the L*f*P language, how we handle connection between computational and control aspects. We then describes a prototype implementation of the code generator and the associated runtime.

Note: the work presented in this paper is being performed withing the MORSE project. MORSE is a French government founded research project (RNTL) with industrial partners (Sagem, Aonix) and academic partners (LIP6 - Univ. P. & M. Curie, LaBRI - Univ. Bordeaux I).

I. Introduction

The rapid advance of distributed technology has lead to systems stretching limits in terms of complexity and manageability [1]. This problem is crucial for reliable distributed systems which are required to have a deterministic behavior. A way to consider this development problem is to use "prototyping techniques".

Prototyping is defined by IEEE as *"A type of development in which emphasis is placed on developing executables early in the development process to permit early feedback and analysis in support of the development process"* [2]. However, this definition was variously interpreted and several types of prototyping are considered for various purpose [3].

For some kinds of systems, prototyping can be usefully considered as a development approach strongly supported by program generation techniques. This approach distinguishes two strong components [4]:

- a model on which any type of validation or verification techniques may be applied,
- the programs that implement this model ; in a prototyping based design, programs are generated from the model.

Such an approach becomes widely accepted under various names. As an example, MDA [5] (Model Driven Development) may be considered as a sort of prototyping approach. However, generated programs must interact with their execution environment: operating system or middleware as well as libraries providing routines for various purpose (i.e. device drivers).

A distributed application is made of two orthogonal aspects: the control aspect, and the computational aspect. The control aspect manages the global state of the application whereas the computational aspect covers the domain specific computational components.

Prototyping is of particular interest for distributed applications for the following reasons:

- They are very difficult to develop since they are very undeterministic ; thus, some apparently minor choices may have dramatic influences on the system behavior.
- The control aspects (the difficult part to build) do strongly interact with both the execution environment and computational aspects, these interactions have to be carefully studied.

We consider that distributed applications group together a control part managing the application (interaction protocol between the application's components, initialization, termination, etc..) and external components that contain routines to be appropriately invoked when the control part evaluates they have to be processed. Since these two aspects of a distributed system cannot be easily captured in one single semantics, it is necessary to separate them. Thus, the model specifies all the control aspects of the system and references external components to be appropriately inserted in the control code by the program generation tool.

This paper presents how we aim to provide such flexibility using **L*f*P** [6]: a modeling language dedicated to the prototyping of distributed systems. We present how we model the separation between the control code and the execution environment (the "external routines"). We also illustrate our technique using an experimentation to generate Java-RMI [7] code from a **L*f*P** specification.

Section II presents the methodology developed around **L*f*P** and the structure of the resulting applications. Section III describes the language itself using an example: a simple load manager for a group of server. This example is followed through the whole paper and lets us introduce the code generation techniques required to automatically generate the control part of the distributed application. Section IV presents the techniques we developed to implement the methodology associated to **L*f*P** focusing on three aspects: deployment of the generated application, the runtime required for code generation, and the generated code structure.

II. **L*f*P**, PROTOTYPING AND EXECUTION ENVIRONMENT

A. Application development methodology

L*f*P is a language developed to support our prototyping methodology for the developpement of distributed systems. It focuses on the control aspect since this aspect is related to most of the specific issues that may be encountered when developping a distributed application. The components required to implement the computational aspect are not modeled in **L*f*P** and are therefore called external components.

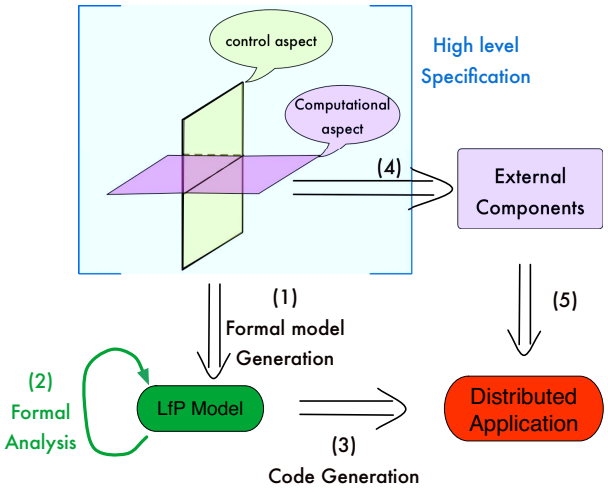

Fig. 1. **L*f*P** methodology for distributed applications developpement

Our methodology is presented on figure 1. It starts from a high level specification, for example written in UML. This specification should outline the distinction between the control aspect of the application and the external components that handle the computational aspect.

External components define a set of interfaces used by the programmer to link the computational aspect of the application to its control part where required. External data (required by external components to perform their task) are handled by the control part of the application using a mechanism very similar to private types defined in the Ada language [8].

Requirements of the application concerning the control aspect should be translated into assertions or properties. Assertions such as “a variable never has a given value” should be written in OCL, whereas properties which involve sequences of actions such as “if a service S is invoked, then it will always provide an answer” are written in temporal logic.

The control aspect of the application is then translated to a **L*f*P** specification (step 1 on figure 1). Given that (1) the control aspect of the application is now unambiguously defined and that (2) the external data are not modified outside of the external components, it is possible to apply formal techniques to the resulting **L*f*P** model (step 2 of figure 1). It is therefore possible to check that this specification meets all the application’s requirements expressed in the model.

Once the model is verified, the source code of the control aspect is generated (step 3 of figure 1). This code is linked with the external components using external calls defined in the **L*f*P** specification. Development of external components (steps 4 and 5 of figure 1) is out of the scope of our methodology and of this paper. They may be implemented using various techniques or come from legacy code, as long as they respect their specifications.

This paper focuses on the automatic code generation, formal verification is presented in [9], [10]. Let us now present a more detailed structure of the generated applications. In the following sections, we call a **L*f*P** component a control component automatically produced from a **L*f*P** model.

B. General structure of a *LfP* application

The development of distributed applications using **L*f*P** highly relies on the separation of the control aspect from the computational aspect. This leads to the application structure of figure 2. This figure outlines the communication scheme of an applications designed using our methodology. **L*f*P** components (control aspect) handle the interaction mechanisms between external components (computational aspect), and manipulates them by means of function calls.

In order to comply with this communication scheme, an external component must not :

- modify the current state of a **L*f*P** component;
- call a method of a **L*f*P** component;
- directly communicate with an other external component.

Breaking one of these rules may alter the control components behavior, therefore invalidating formal verifications performed on the model.

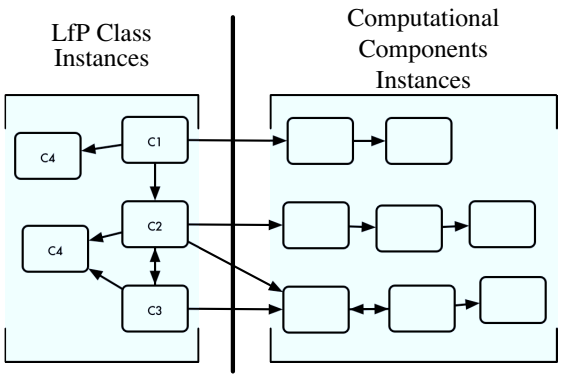

Fig. 2. Structure of an application generated from **L*f*P**

III. Presentation of **L*f*P** through an example

In order to model the control aspects of distributed applications, **L*f*P** provides a "protocol oriented" structure.

- `classes` manage interface with the external components, and handle the data required to manipulate these components;
- `media` provide a convenient way to relate **L*f*P** classes together using arbitrary complex protocols.
- an `architecture diagram` describes the application architecture .

A. The load manager system

This section introduces our language through a simple protocol example: a simple load_management system for a group of servers.

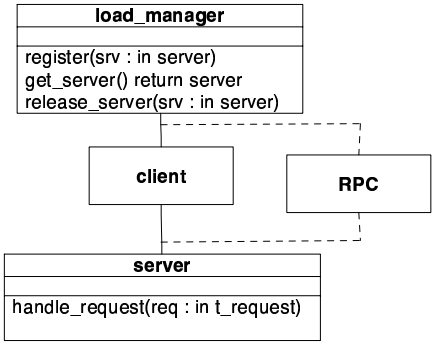

Fig. 3. Class diagram of the load manager system

The UML class diagram of this system is provided on figure 3. It displays four classes : `client` models a simple client that sends a batch of requests to the server; `server` receives the requests and handles them; `load_manager` provides the reference of the less loaded server to a client; class `RPC` is an interaction class that handles communications between the application's components. This model behavior can be succintly described as follows:

- clients ask for a server identifier with method `get_server` of class `load_manager`,
- then the client sends its requests to the server using method `handle_request` of class `server`,
- when the client does not need the server anymore, it releases it by calling method `release_server` of class `load_manager`.

B. A short presentation of the corresponding ***L*f*P*** model

Figure 4 shows the **L*f*P** architecture diagram of the "load_manager" system. The three main UML classes of the system are translated into **L*f*P** classes. The interaction class `RPC` is translated into a **L*f*P** media. From now we use the word "*component*" when no distinction needs to be done between a class or a media.

```
const nbr_of_server : integer := 3 ;
srv1, srv2, srv3 : server with() ;
client1, client2,client3, client4, client5 : client with() ;
load_mngr : load_manager with ();


-- port type for componant's interface
type simple_port is port ;
type client_port is port (simple_port);

-- Requests handled by the server
type t_request is opaque ;

-- return value of a request
type t_req_val is opaque

-- request manager (external componant)
type req_handler is opaque
    function handle_request(req : in t_request) ;
end ;
```

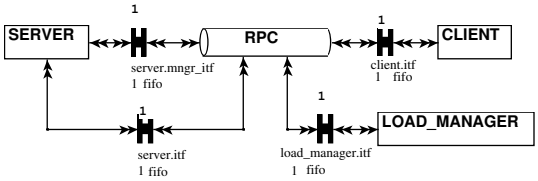

Fig. 4. Architecture diagram of the load_manager system

Some elements that cannot be modeled in UML also appear on this diagram: the `binders` model interaction points between the classes and the media. They describe characteristics of the buffer required for communications between components. Binders are handled inside the components through `ports`. Components ports are references to all the binders that the component may use to communicate.

At the binder level, every interaction between components is a message. A **L*f*P** message contains two sections: the discriminant contains the information needed by the media to handle the message and send it to its destination, the data section contains the data sent to the destination component. The structure of the discriminant for a given port is given by its type. Two types of ports are declared in figure 4: `simple_port` and `client_port`. When a class declares a `simple_port` port, it means that the class must send all its messages through this port without a discriminant; whereas

when a class sends a message to a `client_port` port, it must provide a discriminant that contains a reference to a `simple_port` instance. On the media side, the port type is only required when reading information from a port (to get the discriminant structure).

This diagram also shows the declaration of two external types: type `t_request` contains a request that the client sends to the server, and type `t_result` contains the results that the server sends back to the client. Since `t_result` and `t_request` do not declare any method for external calls, it is possible to use variable of these types as parameters for **L*f*P** methods.

L*f*P classes are "active classes" which means that they define an execution unit of the application (very similar to a thread). When an instance of a class is created, it executes its automata as long as it finds executable transitions. When the automata comes into a state only followed by one or several methods, it holds its execution until one of the pending method is called from another class.

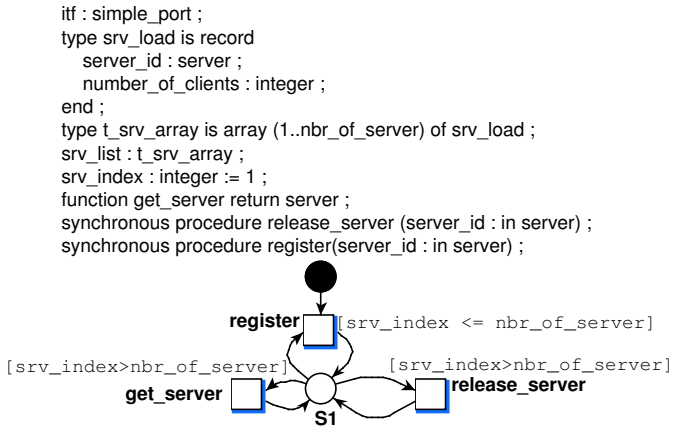

Fig. 5. behavioral diagram for the load manager class

A class behavior is modeled using a class hierarchical automata, such as the one of figure 5. Attributes and methods declarations appear on the main behavioral diagram (for this class, the one presented in figure 5). Sub-diagrams provide a hierarchical mean to simplify the description of the automata. When a sub-diagram is named after a method, it represents the execution flow of the corresponding method. The behavior of class `load_manager` can be described as follows (see figure 5):

1) when activated the class initializes its local variables (`srv_list` and `srv_index`) with default values specified in the declaration part,
2) then the class waits for a call to `register` whose behavior is presented on figure 6 and which registers a server on the load manager and increments `srv_index` by one,
3) once `register` has been executed the class reaches state `S1` which may be described as follows:
   - if `srv_index` is less or equal to `nbr_of_servers`, the only activatable method is register,
   - if `srv_index` is greater than `nbr_of_servers`, then `get_server` or `release_server` may be executed.
4) once the selected method is executed, the component jumps back to state `S1`.

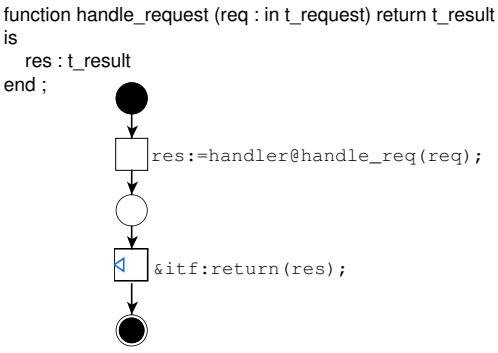

Fig. 6. behavioral diagram for method register

Figure 7 displays the behavioral diagram of class server. The declaration part of this class shows the declaration of an external component type (`request_handler`) which provides method `handle_request` which executes a request and returns its result. Just after this type declaration, an instance of the external component is declared and initialized. On the contrary to types `t_request` and `t_result`, `request_handler` declares an interface, therefore a variable of this type cannot be a parameter of a **L*f*P** method.

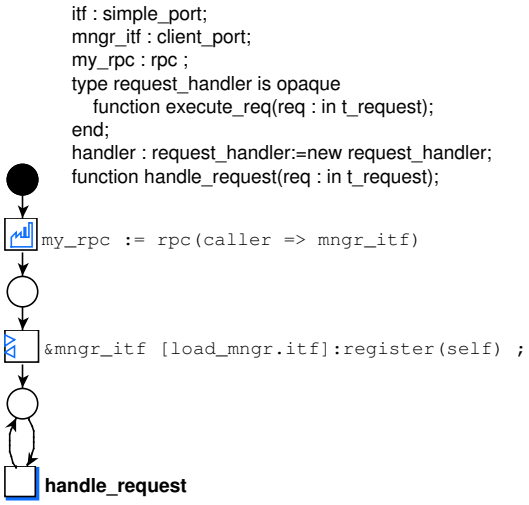

Fig. 7. behavioral diagram for class `server`

The behavioral diagram of `server` performs the following actions: (1) create an instance of media `RPC` to handle communications with other **L*f*P** components, then (2) wait for method `handle_request`.

Figure 8 presents method `handle_request` that contains a call to the external method `execute_request`. The result

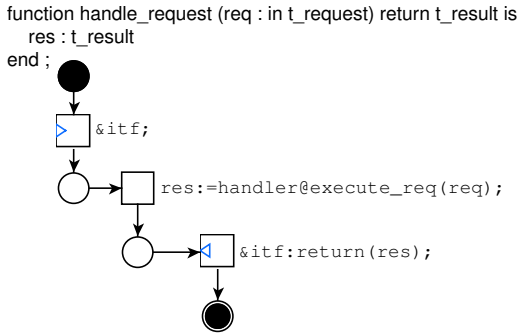

Fig. 8. behavioral diagram of method `handle_request`

returned by this call is then returned to the component that invoked the method.

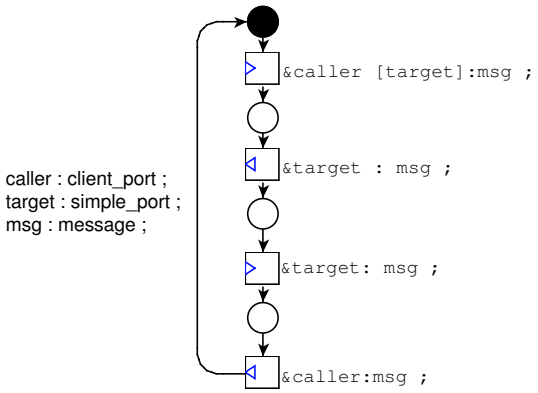

Fig. 9. behavioral diagram for class `RPC`

Media are active components dedicated to the modeling of communication protocols between classes. The only interaction scheme known at the media level is message passing. Figure 9 presents the behavioral diagram of media `RPC` in our example. This media waits for a message coming from an input port (connected to the corresponding binder instance) and copy it into a local variable: `msg`. Then it forwards this message to an output port (connected to another binder), waits for the return message and then forward back this return message to the sender. Let us note that the output port (`target`) is a routing parameter provided by the caller. This is the way to model point-to-point communication (the output port is computed using a reference to a binder instance).

IV. Implementation of a **L*f*P** model

This section presents the translation of **L*f*P** models to programs to be deployed in the final execution environment. This is a two phases process: the user first specifies the deployment of its model, then the code generator uses both the model and the corresponding deployment data to produce the code corresponding to the control part of the application. The generated code relies on a set of low-level functions that creates a common execution environment.

A. Deployment of a **L*f*P** specification

Deployment is a very important aspect of a distributed application. Within our methodology, deployment maps the application behavior defined in the **L*f*P** model on the physical architecture. A **L*f*P** model is typically deployed over several hosts linked by a network; every host that belongs to the application is called a node. Deployment is expressed using an external configuration file interpreted by the code generator and allocating **L*f*P** components and instances on the target execution architecture.

The **L*f*P** approach provides a unified interaction scheme between components via a limited set of operators (remote procedure call or message passing), regardless of the deployment criteria. This means that the way a model is deployed does not influence its behavior. The code generator and the runtime handle a deployment scheme provided by the user and generate the appropriate network interactions. Therefore, the user only has to specify the nodes where static instances of the models are instantiated.

B. The **L*f*P** runtime

Let us now present the **L*f*P** runtime. We first present its requirements, and then comment a prototype implementation in java.

*1) The **L*f*P** runtime requirements:* We define the **L*f*P** runtime as the set of low-level functions required to provide the execution environment needed by the generated code. The runtime is a set of functions and components that handle thread management, binder implementation, naming services and memory management.

`Thread management` mainly includes priority management, instantiation and termination of execution supports for components. Since **L*f*P** does not specify any specific priority management or specific operation on thread, most current thread libraries should work. The goal of this section of the runtime is rather to provide a unified access to thread management in order to avoid modification of the code generator for every thread management system.

`Binders` are the communication means for **L*f*P** components. Therefore, they also handle distribution and network interface. Basically, binders are buffers containing messages. They implement a producer / consumer model. The tricky thing is that the read / write operations are distributed: the message sender (resp. reader) may be instantiated on a distant node, therefore binder instances must be included in the naming service.

The `naming service` allows to reach a component (or one of its binders) from every part of the model. In the example of section III, class `client` gets a reference to a server, and extracts the corresponding binder instance. Figure 10

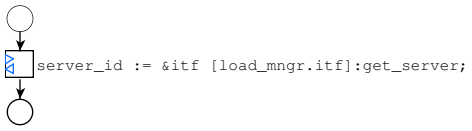

Fig. 10. Part of class `client` behavioral diagram

presents a part of class `client` which shows this sequence of instructions. The implementation of these operations rely on the naming service. The **L*f*P** semantic says that a variable which has the type of a component of the model or of a binder is a reference to the corresponding instance, regardless of distribution. The naming service must also be able to handle dynamic creation / termination of instances.

`Memory management` in the runtime is mainly focused on the instantiation and termination of **L*f*P** components. **L*f*P** allows dynamic instantiation of component, and this instantiation also instantiate the required elements for the component, that is: the thread or process that holds its execution, and its related binders. At termination, the **L*f*P** component itself, and all its resources must be destroyed, including the corresponding entries in the naming service.

2) A prototype implementation using java: We have implemented a prototype of the **L*f*P** runtime. In order to reduce its size in terms of lines of code, we use java and RMI (Remote Method Invocation). This is currently only a proof of feasibility, and performance was not a concern for this implementation.

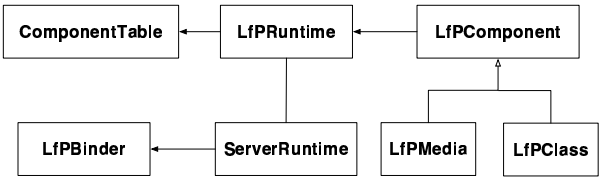

Fig. 11. Class diagram of the runtime

Figure 11 shows the runtime simplified class diagram. Every class (resp. media) of the model extends `LfPClass` (resp. `LfPMedia`); these class both extend `LfPComponent` which is linked to the runtime itself. This is mainly a way for the component to access to runtime primitives. This class also provides the required interface for naming management.

Class `LfPRuntime` provides the interface required by components. This includes message management, naming services, and instantiation service. This class also directly handles the hosts component table (class `ComponentTable`) which provides a way to retrieve a local component from its **L*f*P** reference. This class also forwards all the requests to the appropriate `ServerRuntime` instance. If the target node is the local host, then the call is a direct method invocation, otherwise the runtime uses the RMI protocol.

Class `ServerRuntime` executes the forwarded requests. It is the only declared RMI class, since its methods may be remotely invoked by the instances of `LfPRuntime` that forward component's requests to distant hosts. This class handles the node's components via the local instance of `LfPRuntime`.

Class `LfPBinder` implements the **L*f*P** binders services (message queues). Its local instances are stored in the node's local instance of `ComponentTable` to be retrieved by the runtime when required for message operations.

C. Generated code structure

Let us now present the general structure of the generated code. The runtime provides the set of primitives required to implement each instruction of the model. Therefore, code generation must implement the automaton of every component of the model, translating **L*f*P** instructions in the target programming language, or inserting runtime calls where apropriate. Let us present an outlook of the structural aspect of the generated code:

- **L*f*P** components (classes and media) are implemented as java classes,
- **L*f*P** types are implemented as java classes, each sort of type (enumerated type, record type etc…) has a "pattern" to build the corresponding java classes,
- **L*f*P** methods are mapped to methods of the component java class,
- attributes of the **L*f*P** class are mapped as attributes of the java class,
- local variable of the **L*f*P** specification are declared in the instruction block that implements the construction that declares them.

L*f*P classes extend the class `LfPClass`, media extend the class `LfPMedia`.

```
public class Load_Manager{
    public Simple_Port itf;
    private class Srv_Load {
        Server server_id;
        Integer number_of_clients;
    }
    private class T_Srv_Array {
        int first = 1;
        int last = nbr_of_server;
        Srv_Load value;
    }
    public T_Srv_Array srv_list;
    public Integer srv_index;
}
```

Fig. 12. Declaration of the attributes of class `Load_Manager`

Type declaration also depends on the **L*f*P** type's visibility: types declared in the architecture diagram such as `simple_port` on figure 4 will be generated as public classes in a separate java file. Types declared inside a **L*f*P** component such as `srv_load` will be declared as private classes inside the component's class. Figure 12 presents the declarations of class `Load_Manager`: The declarations of the attributes are

self-explanatory. Local types are mapped to private classes: `Srv_Load` implements a record type, every attribute is a field of the record. Class `T_Srv_Array` implements an array type, which requires three fields: `first` is the lowest valid value of the index, `last` is the highest valid value of the index, and at last value is the array itself.

```
while(true) {
    switch(next) {
        case initial:
            request.addMethod(register);
            msg = runtime.getMessage(request);
            if (msg.method.equals("register"){
                register((Server) msg.paramters[0]);
                runtime.setnMessage(msg, this);
            } break ;
            ...
    }
}
```

Fig. 13. Structure of the generated code

The dynamic aspect of the diagram is handled by creating code for the automaton states and transitions. The main diagram is generated in a method called `run` which is a default mean to implement the "main" function of a thread in java. Code generation for methods follows the same pattern, only the java method's name changes. Since the java language does not implement a "goto" statement, we "emulate" it with a "switch" structure inside a while loop: Every state is labeled with a number, and the `next` variable contains the label of the next state to execute. This solution remains efficient because the number of states per automaton is generally quite reduced since a dedicated loop is produced for every sub-diagram.

Figure 13 displays the structure of the code generated for the initial state of the main diagram of figure 5. Since this state is followed by a list (here reduced to one element) of method transitions, we apply the following pattern:

- for every outgoing transition whose guard is true, add the name of the method in the list of valid methods,
- send the request to the runtime,
- get the activation message and execute the corresponding method,
- send the return message with the return value.

The first two items allow to build and send a request that will ask for all activatable method(s). The runtime call returns a message that contains both the name of the activated method and the parameters. The content of the message is directly used as parameters for the method call, since java only uses references for object parameters, **L*f*P** `inout` and `out` parameters are updated in the message. If the method is a function, the message also stores the return value. At last, if the method is a synchronous procedure or returns a value, the message is sent back in the binder, with the method's updated parameters or return value.

The code generated for a state followed by a set of transitions is much simpler: evaluated the outgoing transition's guard, and "jump" to the corresponding label if the guard is true. This "jump" means to set the value of `next` to the transition's label.

If several transitions follow one state, **L*f*P** does not specify a default evaluation order for the guards. If required, it is possible to specify it on the model by giving priorities to every outgoing arc of a state.

```
public T_Result handle_request(T_Request req) {
    Request_Handler handler = new Request_Handler();
    T_Result res ;
    next=S1;
    loop:
    while (true) {
        switch (next) {
        case S1:
            next = T1;
            break;
        case S2:
             next = T2;
             break;
        case T1:
            res = handler.execute_request(req);
            next= T2;
            break;
        case T2:
            return (res);
        }
    }
}
```

Fig. 14. Structure of the code generated for `handle_request`

Transitions are relatively easy to translate since the instructions available in **L*f*P** often have a direct equivalent in every structured programing language, however some constructions specific to the **L*f*P** language are harder to implement. The code for the `handle_request` method of figure 8 is displayed on figure 14. This figure displays the general structure of the code generated for methods and the call to the external component (`handler.execute_request(req)`) which has been initialized on instantiation of class `server`. Communication instructions are harder to translate and require the runtime interface for message manipulation, the generated code is not presented in this paper.

Final states of components main diagram (not presented in the example of this article) mean the deallocation of the **L*f*P** component that reaches them. In java, this mainly means to remove all the class instances that implement this component from the naming service data structures (i.e.: set their references to "null"). This removes the only references to these class instances, thus allowing the garbage collector to perform the effective deallocation. This work is done by a runtime primitive called in the code generated for the diagram final state.

V. Conclusion and future work

The **L*f*P** language is dedicated to the prototyping of distributed applications. This language combine several concepts:

1) formal description techniques suitable for distributed systems,
2) a process oriented design, suitable for distributed systems,
3) an object oriented design, suitable for the analysis of a system (such as UML).

We presented in this paper how we intend to use our language to automatically generate distributed applications. The **L*f*P** model describes the control part of the system. We have shown how it can be connected to "external" components (that do not participate in the control but contain actions that have to be executed). This is a difficult part since **L*f*P** is based on a formal notation to enable formal verification.

Thus, we have elaborated a mechanism inspired from the notion of "private types" found in the Ada programming language. This allows the definition of variables that can only be transported from a **L*f*P** component to another one. Methods to be executed when the control part of the system decides can be associated to these types.

We presented how the control part and the external components can be merged into a coherent architecture implemented on top of a runtime providing basic services for the execution of a **L*f*P** model. In MDA terms, the **L*f*P** specification is then a PIM (Platform Intependant Model). Combined with deployment directives, it becomes a PSM (Platform Specific Model) suitable for automatic program generation.

L*f*P is a foundation of the MORSE[1] project that explores the design and implementation of highly reliable distributed systems. MORSE focuses on the design of the asynchronous part of such systems and aims to be used for critical applications and covers the development of the control part of the application from a high level specification written in UML to code generation. The **L*f*P** code generator is being written and the general prototyping approach will be validated on real size projects within the context of MORSE partners.

REFERENCES

[1] N. Leveson, "Software engineering: Stretching the limits of complexity," *Communications of the ACM*, vol. 40(2), pp. 129–131, 1997.

[2] C. Booth and G. Kurpis, "The new IEEE standard dictionary of electrical and electronics terms [including abstracts of all current IEEE standards], 5th edition," Institute of Electrical and Electronics Engineers, Tech. Rep., 1993.

[3] F. Kordon and J. Henkel, "An overview of Rapid System Prototyping today," *to appear in Design Automation for Embedded Systems*, vol. 8, no. 4, pp. 275–282, december 2003.

[4] F. Kordon and Luqi, "An Introduction to Rapid System Prototyping," *IEEE Trans. Softw. Eng.*, vol. 28, no. 9, pp. 817–821, 2002.

[5] OMG, "Model Driven Architecture (MDA), Document number ormsc/2001-07-01," OMG, Tech. Rep., 2001.

[6] D. Regep and F. Kordon, "**L*f*P**: A Specification Language for Rapid Prototyping of Concurrent Systems," in *Proceedings of the 12th International Workshop on Rapid System Prototyping*. IEEE Computer Society, 2001, pp. 90–97.

[7] "Java remote method invocation," 1997, http://splash.javasoft.com/pages/rmi.html.

[8] ISO, *Information Technology – Programming Languages – Ada*. ISO, Feb. 1995, ISO/IEC/ANSI 8652:1995.

[9] F. Kordon, I. Mounier, E. Paviot-Adet, and D. Regep, "Formal verification of embedded distributed systems in a prototyping approach," in *Monterey Workshop 2001: on Engineering Automation for Software Intensive System Integration*, June 2001.

[10] D. Regep, Y. Thierry-Mieg, and F. Kordon, "Modélisation et vérification de systèmes répartis: une approche intégrée avec LfP," in *Proceedings of AFADL'03*, January 2003.

[1] `http://morse.lip6.fr`