A Model Based Development Approach for Distributed Embedded Systems

Frédéric Gilliers[2,⋆], Fabrice Kordon[1], and Dan Regep[1]

[1] Laboratoire d'Informatique de Paris 6, 4 place Jussieu,
F-75252 Paris Cedex 05, France
{Fabrice.Kordon,Dan.Regep}@lip6.fr
[2] Sagem SA,
21 Avenue du Gros Chêne, 95610 Eragny,
BP51, 95612 Cergy Pontoise cedex, France
Frederic.Gilliers@lip6.fr

Abstract. Design of reliable distributed systems is stretching limits in terms of complexity since existing development techniques are usually not fully accurate for this type of applications. The main problem is the gap between the various notations used during the development process. Even if UML is a significant step forward, it is not fully suitable for model based development of distributed systems.
We present a model based development approach based on **L*f*P** (Language for Prototyping) applied to distributed systems. It emphasizes the use of a model serving as a basis for automatic code generation; strong connections with formal verification techniques enforce correcteness of the system. The paper focuses on the description of code generation techniques.

1 Introduction

The fast evolution of distributed technology has led to systems stretching that stretch the limits of complexity and manageability [12]. A major problem when distributed systems have to be certified resides in both the design and coding phases: collected requirements may be incomplete, inconsistent or misunderstood, and the numerous interpretations of a large specification often leads to unexpected implementation and additional debugging costs.

The problem comes from the gap between the various notations used in the software life cycle (natural languages, specification languages, programming languages). A first solution is to use a methodology that provides a coherent set of notations to solve this problem. The UML standard [18] represents a significant advance to system specification, however:

- UML semantics is not sufficiently formally defined to enable formal verification unless strong restrictions and hypotheses on its use are introduced (like in [2, 5]);
- Good code generators available for information systems are lacking for distributed applications since UML is not fully adapted to capture all aspects of distributed architectures [15];

⋆ This work is done within an industrial grant provided by SAGEM S.A.

M. Wirsing et al. (Eds.): RISSEF 2002, LNCS 2941, pp. 137–151, 2004.

- UML relies on object orientation, that can be easealy implemented using middleware such as CORBA [17] or Ada-DSA [7] that implement distributed objects. However, the use of other classes of middleware (such as MPI [14]) requires the implementation of adaptation components to handle object oriented mechanisms. Similarily, implementation of an UML specification according to profiles for embedded systems such as ravenscar [4] is delicate since dynamic mechanisms are forbidden in such systems;
- The behavioral semantics of UML will remain informally defined for several years since the 2.0 initial submission claims it essentially formalize static/structural aspects [19]. It introduces OCL to define constraints precisely but only a very limited number of pages are dedicated to the description of unambiguous behavior of a system.

Therefore, for distributed systems, UML is mostly valuable in the early stages of the software life cycle. When a preliminary object-oriented solution is sketch, there is a need for another type of description closer to implementation (e.g. that does not necessarily rely on object oriented middleware). This new description should enable both formal verification (a well accepted approach to ensure high quality in distributed systems) and automatic program generation (to ensure coherence between specification and program).

This paper presents our proposal for a model based development approach. It emphasises the use of a model that formally defines behavioral aspects of the system. This model is used to automatically generate the control code of a distributed application. It is also suitable for formal verification.

Our methodology is presented in section 2. It relies on a notation briefly described in section 3. We then focus on how programs can be derived from these specifications in section 4.

2 Model Based Development

Model-based development [22] focuses on the use of a model that serves as a basis for two main objectives: formal verification and automatic program generation. We favor this approach and consider that it corresponds to an evolutionary prototyping approach [11]. Our proposal relies on **L*f*P**, a high-level modeling language that unambigously describes the behavior of a distributed systems using Behavioral diagrams (automata with structuring facilities). Behavioral diagrams express contracts to be obeyed by components of a distributed system. For example, a class C may state that method M_1 has to be executed prior to the execution of mehod M_2. It may also state that method M_2 cannot be executed twice. **L*f*P** also provides facilities to insert assertions like invariants of temporal logic formulas into the model. This information is suitable for verification purposes and can be used by code generators to optimize programs.

We aim to formalize relations between system modeling, formal verification and code generation of distributed systems in order to provide:

- transparent formal verification to enable its use in an industrial context without requiring both a heavy training and some specific skills, as outlined in [13],

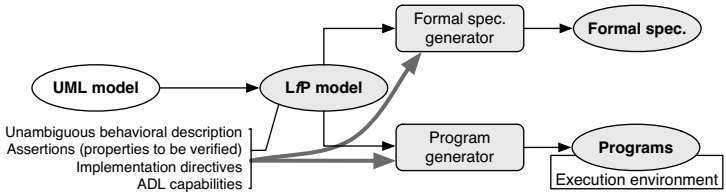

Fig. 1. Model based development using evolutionary prototyping.

– strong correspondence between the detailed description of a system, its proofs and its implementation. In other words: "*what you describe is what you check and implement*".

As shown in Fig. 1, we aim to plug our model based development approach into a "classical" requirement/analysis phase that produces an UML model. First, a reformulation of this model into **L*f*P** must be considered. Most of the work should be done by a tool (producing behavioral state machines from collaboration diagrams cannot be completely automated [9]) but designers have to add information such as the behavioral contracts and assertion to be verified (e.g. "this server has to provide an answer") that cannot be automatically deduced from UML diagrams. When program generation is used, designers also add informations used by code generators (e.g. "components affected to this host are coded in Java"). This additional information is sometimes located in UML tagged values supported by some CASE tools. However, such information is potentially non standard.

From this central description two types of operations can be performed:

– Formal specification generators produce formal specification to be verified: the transformation is optimized according to the property to be verified (i.e. the appropriate couple <*formal method, used technique*> is selected).
– Program generators produce source files to be compiled and integrated in the target execution environment. These generators have to deal with code generation techniques (how to translate the **L*f*P** semantics into a given language) but also with configuration and deployment of the system on top of the target architecture.

Model-based development, similarly to model driven architecture [20], is a development strategy promoting the use of various techniques: modeling (as precise as possible), verification techniques (formal is safer but any other evaluation techniques can also be considered as a first step), and program generation.

In that context, a modeling language such as **L*f*P** serves as an interface to elaborate, by successive refinements, a very precise view on the system, taking advantage of:

– information provided by formal verification (e.g. are all assertions verified?),
– information provided by the execution of previous prototypes (e.g. are performance goals met?).

This paper focuses on program generation techniques. More information concerning formal verification from **L*f*P** can be found in [10].

3 The L*f*P Formalism

This section summarizes the main features of **L*f*P**. It is an Architecture Description Language with coordination facilities that focus on distributed systems. In order to enhance UML models with information that enables automatic code generation of distributed programs as well as formal verification, we define three orthogonal views:

- The *functional view* describes the system software architecture, and links classes (that are execution units) to media (that are communication mechanisms). Both classes and media are described in terms of execution workflow in order to precisely establish behavioral contracts to be analysed and programmed.
- The *implementation view* describes the system implementation constraints (target executive, programming language, communication infrastructure) and the deployment topology.
- The *property view* specifies properties to be verified by the system (analogous to the notion of proof obligation in B [1]). Such properties are stated by means of invariants (for example, confirmation of a mutual exclusion), temporal logic formulas (for example, to the availability or fairness of a service) or other assertions that can be converted into a given formal method. This view can be exploited to perform computer-assisted formal verification. Moreover, it introduces relevant information for code generation (e.g. runtime checks).

The Gas Station Example. To illustrate **L*f*P** features, let us present a model first introduced in [6] and then widely used to demonstrate various verification techniques, as in [3, 24].

It models the simplified behavior of a self-service gas station (see class diagram in Fig. 2). When entering the station, a client prepays the operator, and receives a *ticket* bearing his *id*. The client then proceeds to the pump, inserts his *ticket*, pumps up to *sum* gas, and finishes his pumping operation. He then returns to the operator to get his change and receipt before leaving the station. Information that concern clients being processed is centralized in infosystem. The operator registers the client when it prepays, and unregisters him when he returns. The pump accesses the infosystem when a client activates the pump, and updates the credit when the client puts back the nozzle.

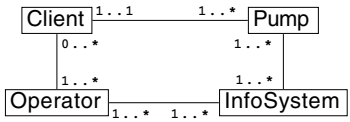

Fig. 2. Class diagram of the gas station.

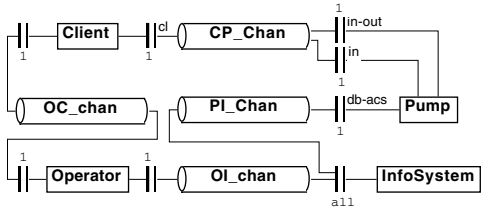

Fig. 3. L*f*P functional diagram of the gas station.

System Architecture. Fig. 3 presents the architectural diagram of the station example. It contains additional information that may be not present in the UML diagram, in order to specify communication patterns between classes. Typically, relations between UML classes (sometimes represented using associations) lead to the creation of media (here *OC_chan*, *CP_chan*, *OI_chan* and *PI_chan*) to describe communication semantics (behavior of communication elements).

Media and classes are connected by means of binders. This notion is inspired from the notion of binding points in RM-ODP [8]. Binders define interaction points between a class instance and a media instance. They correspond to interface buffers associated with characteristics like maximum size, management strategy (FIFO, etc.) and overflow strategy (message loss, client blocked, etc.). Deployment of binders is defined by means of the cardinality (`1` or `all`) specifying if they are shared or not. To avoid overloading Fig. 3, we only list the binders related to *CP_chan* and pump (cl, in-out and db-acs); they are referenced later in the paper.

Let us illustrate how the cardinalities of the gas station model should be interpreted. Fig. 4 shows a class architecture and Fig. 5 corresponds to the corresponding object architecture with two instances of A and C and three of B and D. Each instance of class A (as well as those of class B) has its own buffer connected to the communication system while each instance of class C has its own buffer. Instances of D share a single buffer.

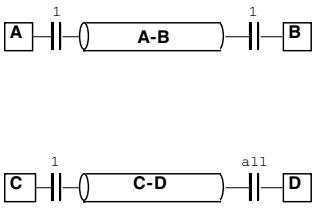

Fig. 4. Class connections: examples.

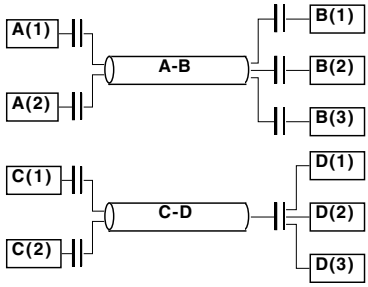

Fig. 5. Class connections: instanciation.

There may be several interaction points (and thus binders) between a class and a media when different characteristics are required. This is the case between *CP_chan* and pump, in Fig. 3: in-out is a two way binder (to support remote method invocation) and sb-acs is a one way binder (used to propagate events).

The behavior of classes and media introduced in the functional diagram is formally described using *behavioral diagrams* (**L*f*P**-BD). They are hierarchical state machines defining what action must be executed based on the internal state of a class instance.

The architecture diagram also declares the number of classes and media instances to be elaborated when the system starts (this is not represented in Fig. 3).

Behavior of a Class. The behavioral contract of a class deals with methods and triggers (an activation condition + code to be executed when the condition is satisfied).

Methods are invoked by other components and triggers are activated by internal conditions. Methods and triggers cannot be executed in parallel. The behavior of a class is expressed using **L*f*P**-BDs, a notation to express state machines. The main level defines the activation conditions of methods and triggers; each transition of the automaton correspond to a method or a trigger. Then, each method and trigger behavior is described using a **L*f*P**-BD, where transitions represent atomic actions to be performed.

Fig. 6, defines the relationship between pump's methods: when `start` is operated, `pump_gas` can be executed until `finish` is called. Asynchronous methods can be compared to message passing (like the *asynchronous* pragma in Ada-DSA [7]). This diagram also declares variables known to the class. In pump, there are only local variables (i.e. each instance of pump has its own copy) but class variables can also be declared (i.e. one copy for all instances of a given class).

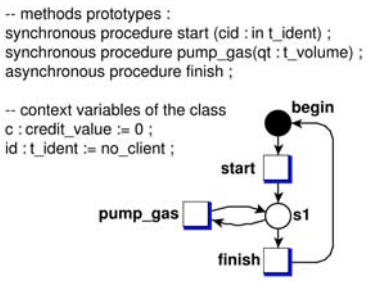

Fig. 6. Behavioral diagram of class pump.

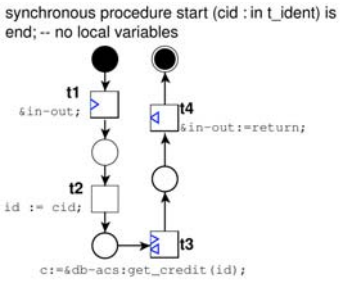

Fig. 7. L*f*P-BD diagram of method `start`.

Methods have to be connected to binders through which they get parameters, send results or invoke services provided by other classes. Triggers may also be connected to binders if they send/receive information from other classes. These connections are defined when describing the execution flow of a method (or trigger). Fig. 7 shows the **L*f*P**-BD that specifies the execution flow of `start`:

- at **t1** and **t2**, the parameter `cid` of the method is extracted from the `in-out` connection point when the query is issued and then copied into variable `id`,
- at **t3**, the credit value for the client is requested through an invocation of method `get_credit`; the query message is issued and the pump instance waits until the value is available to assign to variable `c`,
- at **t3**, an empty message is sent back to the client that issued the query to signal the execution end (`start` is declared as a synchronous method in Fig. 6).

In order to get a complete view of a class behavior, the sub-diagrams that describe individual methods are inserted in the behavioral contract. For instance, the begin state of the `start` automata is merged with begin in the main diagram and the end state of the `start` automata is merged with s1.

Media Behavior. Media have no methods. The associated **L*f*P**-BD describes the communication semantics to be supported.

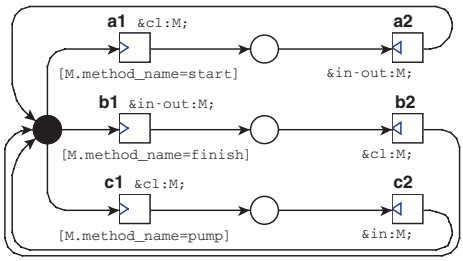

Fig. 8. Partial behavioral description of the *CP_chan* media.

Let us consider the specific protocol associated with media *CP_chan*, that connects a client to a pump (Fig. 8). The **L*f*P**-BD states that:

- only one message is handled at a time by a given instance of media **CP_chan**,
- variable `M` stores a **L*f*P** messsage,
- messages coming from the binder cl are routed according to the method parameter they transport; this is stated by means of the guards associated to transitions **a1** and **c1** (reference to the predefined operator `method_name`). Method `start` goes to binder in-out (an answer is expected, it will be sent via transition **b1**) and `pump` goes to binder in (asynchronous method, no answer expected),
- messages coming from binder in-out are all routed to cl,
- no message originates at binder in.

Other capabilities of **L*f*P** that are not listed here (but presented in [10, 23]) are:

- definition of constructors to dynamically create new instances of a given class,
- use of enhanced data structures such as arrays, records and bags,
- definition of critical sections to protect shared variables,
- use of predefined instructions to label transitions (basically, loops and tests),
- assertions that are "proof obligations" for verification, and may lead to the generation of runtime checks.

4 Code Generation from L*f*P

Code generation translates structures and operators into calls to the primitives of a runtime that provides procedures and services required by the **L*f*P** semantics. Generated programs handle distributed control of the application and thus are executed over several hosts. We first study the general architecture of generated applications and then focus on the translation procedure itself.

4.1 Architecture of the Generated Code

Each host that supports execution runs a partition of the system (we call it a *node*) according to the architecture presented in Fig. 9. A partition consists of classes, media and/or binders instances.

The generated application is built on top of a system-dependent layer, the *runtime*, which provides a set of standard subroutines required to support the **L*f*P** semantics. The interface provided by the runtime to the generated programs remains the same, whatever the target architecture. Therefore, for a given language, the generated code may be deployed on various operating systems for which a runtime is available. To ease the porting of the runtime, it is split in high level services containing non-platform specific services (garbage collection if any, buffer management, etc.), and low-level services containing platform specific services (such as threading mechanisms, memory allocation, etc.). Low-level services rely on the execution environment (operating system and/or communications libraries and/or middleware).

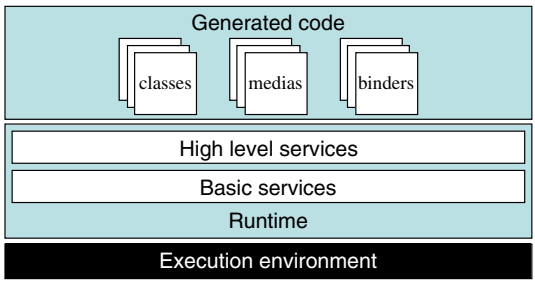

Fig. 9. Architecture of the generated Code and its environment.

To ensure that generated code has a minimal footprint, high-level and low-level services are divided into components corresponding to operations in **L*f*P**. The component corresponding to a given operation is embedded in the runtime if it is activated in the corresponding node.

When a class or media is tagged *external* (e.g. it is legacy software), only interfaces are generated. These interfaces should be enriched if necessary to fit the implementation of the **L*f*P** interactions strategy. For example, if one wants to use sockets, a media provides an abstraction of socket mechanisms that is used, first for modeling and verification, and subsequently to generate an empty interface that can be invoked by the generated code. Mapping of the generated interface to sockets primitives has to be done manually. External components are also a way to insert hand written code into an application when the generated programs do not respect performances requirements.

4.2 Generating Code for Classes

Classes are the smallest unit of concurrency. Therefore the hierarchical automaton of a class, defined by means of **L*f*P**-BD, is translated into a sequential program. It appears useful to maintain the hierarchical structure for two reasons: readability/traceability, and optimization of the code. The code generator must consider the following elements:

1. *data types* used by the class,
2. *local variables* that are the non static attributes of the class,

3. ***LfP-BD methods*** that describe the execution flow of a method,
4. *shared variables* that are the static attributes of the class (i.e. the variable is shared among all the instances),
5. *critical sections* that specify synchronizations in the **L*f*P** model,
6. *evaluation of transitions guards* that select the next transition to execute,
7. *evaluation of assertions* to perform runtime checks on the model,
8. *connections to binders* that correspond to synchronization points with a media,
9. the ***LfP-BD class behavioral contract*** that implements the **L*f*P**-BD diagram of the class.

The architecture of a class is presented in Fig. 10. The generated code for the **L*f*P** class is made of several interacting software units. For example, the code embedded in `PumpPackage` contains of the following elements: `PumpTask` which implements the class's contract, `PumpMonitor` to synchronize the access to binders, `PumpImplementation` that embeds the class's methods and local attributes, `PumpPredicates` which contains the assertions and the guards associated to the class, `PumpSharedVariables` which contains shared variables, and `PumpTypes` which implements local data types.

The class implementation is also related to global units: `GlobalTypes` defining model level types, and some `Binder` elements implementing the connected binders. For the pump class of the example, they are: *in*, *in-out* and *db-acs* (see Fig. 3).

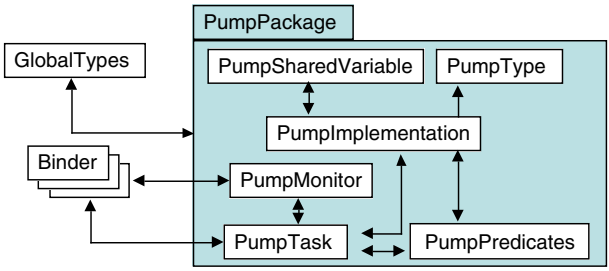

Fig. 10. Structure of the code generated for the pump class.

Global and specific types are embedded into separate packages that easily translated into instructions since construction rules in **L*f*P** types are very similar to those proposed in conventional programming languages. Types and variables visibility is preserved to avoid complex renaming and ensure readability. Global types definitions are produced in a `GlobalType` component that is imported by the `PumpPackage`.

Shared variables of an **L*f*P** Class are handled by a specific package (`PumpSharedVariables`). Instances of a class may be created on several nodes of the application and variables remain global whatever the distribution is. Therefore, the generated code handles consistency of these variables and provides synchronization as well as `set` and `get` primitives.

Transitions guards enable or disable transitions. If a guard only refers to local variables, it is implemented as a boolean condition. If it refers to the body of an incoming message, it is implemented via a two phase transaction handled by the related binders. There are three primitives: `Get` retrieves the message from the buffer, `commit` completes the current transaction when the guard is satisfied, and `rollback` aborts it. When a state is followed by alternative transitions, a select-like structure is generated, as shown for `s1` in the behavioral contract of pump (see Fig. 6).

Assertions are also implemented as boolean expressions that may raise an exception at runtime. Assertions and guards are embedded into the `PumpPredicates` program unit.

Methods and local variables are both implemented in `PumpImplementation`. Code generation reproduces the automata hierarchical structure: code for transitions performs operations, states are associated with labels. For a given `State` the execution sequence is: *1)* evaluation of guards, *2)* execution of the transition body, *3)* verification of assertions if any, *4)* jump to next state.

Relation to binders are implemented using a monitor (`PumpMonitor`). There is a monitor for each class instance. Its role is to synchronize execution of the class with incoming messages. When a message is required to fire a transition, the program registers with the binder and waits until the message arrives. If a message is already in the queue, the invocation of the registration primitive is non-blocking. When a message is read, the `PumpTask` unit (that handles the execution contract) may evaluate a guard to decide if the message has to be consumed or not.

The behavioral contract is implemented as a separate task for each instance and located in `PumpTask`. It only accepts messages corresponding to methods that are enabled in its current state. The current state is encoded as a local variable that selects an case alternative.

To read messages from binders, the task first `reset`s the associated `monitor`, then `register`s itself with binders. When a message can be read (e.g. there is a message respecting the transition guard), the class consumes it, unregisters from the binders, executes the selected method, verifies the corresponding assertions and goes to the next state. When no message is available or conform to the guard, the task waits on the monitor (`sleep` entry of the monitor) until a message arrives to one of the connected binders.

Writing into a binder is simpler, a class instance `put`s the message into the binder.

Fig. 11 illustrates the structure of the code generated for state `s1` of class pump (see Fig. 6). In this state two methods are enabled; the one to be fired depends on the next binder that will receive a message. First the class resets its monitor and register itself with the two binders. Then it loops to check the content of both binders. If any valid message (e.g. satisfying the transition's guard) is found, the class executes corresponding code, unregisters itself from the binders and jump to the next state. If no valid message is found (there is no message or the first one is not valid) the class waits on the monitor. Invalid messages are left in the binder.

```
S1_State:
   -- reset the monitor
   Pump_Monitor.reset();
   -- register to connected binders
   in-out.register(this);
   in.register(this);
   loop
      -- non-blocking retrieval of an eventual message
      in-out.get(message);
      -- if a message exist in the in-out binder
      if message /= null then
         -- if it is a pump call and guard is true
         if Pump_Predicates.Pump_gas_pre(message) then
            -- consume the message
            in-out.commit();
            -- execute the pump_gas Method
            Pump_Implementation.Pump_gas(message.qt);
            -- check assertions
            Pump_Predicates.Pump_gas_assert();
            -- unregister from binders
            in.UnRegister(this);
            in-out.UnRegister(this)
            -- jump to next state (S1_State)
            goto(S1_State);
         else
            -- unexpected messages or unverified
            -- guard, rollback transaction and
            -- try another binder
            in-out.rollback();
         end if;
      end if;

      -- non-blocking retrieval of an eventual message
      in.get(message) ;
      -- if a message exists in the in binder
      if message /= null then
         -- if it is a finish call and guard is true
         if Pump_Predicates.Finish_pre(message) then
            -- consume the message
            in.commit();
            -- unregister from binders
            in.UnRegister(this);
            in-out.UnRegister(this);
            -- execute the finish method
            Pump_Implementation.Finish();
            -- check assertions
            Pump_Predicates.Finish_assert();
            -- jump to next state (Begin_State)
            goto(Begin_State);
         else
            in.rollback();
         end if;
      end if;
      -- Sleep on the monitor until next message on
      -- registered binders
      Pump_Monitor.Sleep(wait_delay);
   end loop
```

Fig. 11. Pseudo-code of pump related to state s1 and activation of methods `pump_gas` and `finish`.

4.3 Generating Code for Media

Media are communication mechanisms between classes. Two strategies are considered.

When it is tagged "*external*", the media corresponds to legacy software (a communication library). Only an interface defining the functions required to interact with associated binders has to be generated. The media definition is used first for modeling and verification purposes, then to generate an empty interface to be invoked by classes. For example if a designer wants to use sockets, the corresponding media provides appropriate interfaces and abstraction. Mapping to the socket library has to be done manually; the resulting component is reusable. Off-the-shelf media will be provided (such as sockets or RPC).

When it is tagged "*internal*", a media is translated into an automaton implementing the specified protocol.

Media also allow support the definition of implementation-independent higer level communication mechanisms. For example, a channel media may encapsulate various types of implementations: sockets, shared data segments, etc. Such abstractions are of interest to ensure portability over several target architectures (hardware + operating system).

4.4 Generating Code for Binders

Binders are connection objects between instances of Classes and Media. They are implemented as distinct code units. There are several implementation schema depending on their characteristics: multiplicity, blocking/non-blocking primitive access, etc. Implementation strongly relies on the **L*f*P** runtime that provides generic message passing and instantiation services.

```
generic class Blocking_fifo (Size, Message_type, Client_ref_type)
begin
     protected entry Put (Message : in  Message_type) when not transaction;
     protected entry Get (Message : out Message_type) when not transaction;
     protected entry Commit () when transaction;
     protected entry Rollback () when transaction;

     protected entry Register (Client : in Client_ref_type);
     protected entry UnRegister (Client : in Client_ref_type);

  private :
     Buffer is array (1..Size) of Message_type;
     First, Last := 1;
     No_items := 0;
     Client_list is list of Client_ref_type;

     Transaction is Boolean := FALSE;
end;
```

Fig. 12. Specification of a blocking FIFO.

Binders are implemented as instances of generic templates to enable pattern reuse. Fig. 12 shows the pseudo-code of binders interface (here, a FIFO buffer). This template has three generic parameters: `size` (capacity of the buffer), `message_type` (type of transported data), `client_ref_type` (reference type to designate clients).

Private data implement a fixed size FIFO, a list of subscribers and a boolean variable to indicate if a `get` transaction is currently running.

Access to the template services are protected (i.e. both mutually and self exclusive). `Put`, `get`, `commit` and `rollback` are I/O primitives while `Register` and `Unregister` are dedicated to client management. `Get` starts a transaction. `Commit` or `rollback` entries end the transaction. Pending `Put` or `Get` are executed only when no transaction is running.

4.5 The L*f*P Runtime

The runtime provides a set of services to handle the **L*f*P** semantics using primitives of the target execution environment. The runtime provides the following services:

- *Task management service* deals with creation, synchronization and termination of the tasks that handle the execution contract of classes and media (e.g. `PumpTask` in Fig. 10). It is also a basis for implementing class monitors (e.g. `PumpMonitor` in Fig. 10).
- *Resource management service* handles creation, initialization and destruction of **L*f*P** code elements (classes, media and binders instances).
- *Registry service* (in the meaning of Java-RMI [16]) stores global references to generated code units and runtime units at execution time. It is required for distributed deployment when naming conventions have to be preserved over several adresses spaces. This is the case when a prototype is deployed on two middlewares (e.g CORBA and Ada/DSA).

Fig. 13 presents in the form of an UML class diagram the relationship between the **L*f*P** runtime and the application. This interaction model is inspired by RM-ODP [8]. The runtime contains three logical units dedicated to management:

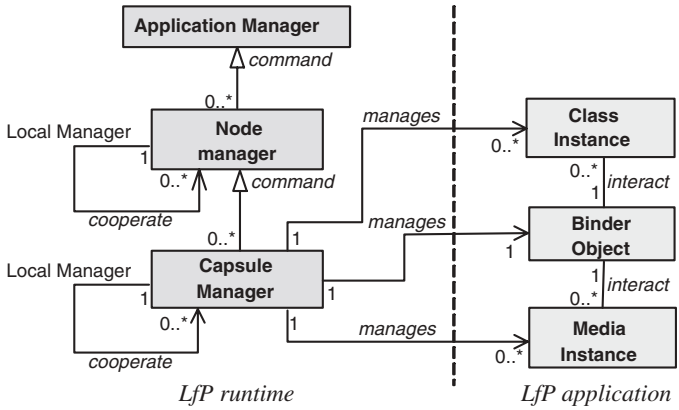

Fig. 13. Relationship between the **L*f*P** runtime and applications.

- The *application manager* handles initialization, termination and error management for the application. It relies on node managers to supervise hosts specific tasks. The Registry service is implemented in the application manager since it is used by all the components of the application.
- Each *node manager* handles a partition of the application on a given node. It implements process creation which is part of the runtime task management service.
- Each *capsule manager* handles instances of a given class or media within a given partition. It supports both task and resources management services.

The runtime implementation depends upon the selected execution environments. There are two types of execution conditions:

- Some applications focus on the use of thick execution environment such as CORBA, Ada-DSA or JAVA-RMI. These environments offer sophisticated services such as naming, dynamic remote creation of objects, etc.
- Other applications have critical time and/or memory constraints. They require thin execution environment such as QNX [21]. Code generation also requires specific strategies to minimize memory footprint or optimize execution time.

The runtime architecture presented in Fig. 13 is able to fit those two types of constraints with tolerable performances. The use of a thick execution environment is not a problem since they support most of the functions required in **L*f*P**. The runtime is then minimal but relies on more complex services. The use of thin execution environments requires a more complex runtime that relies on very simple but efficient services. However, it is possible to write most of the **L*f*P** capabilities with respect to the requirements of embedded systems. For instance a partition may include a static scheduler that handles instances of **L*f*P** classes as threads taken from a pool whose size is fixed at compilation time and yet be compatible with an interpretation of Fig. 13; then, many components are reduced to very limited code.

5 Conclusion

This paper presents a model based development approach for distributed applications. It relies on **L*f*P**, a notation to capture the behavioral semantics of such systems, and serves as a basis for both formal verification and automatic code generation.

We consider that such an approach is a valuable extention to UML based design methods. The combined approaches (UML for object-oriented design and **L*f*P** for a process-based implementation) offer a way to move from an object oriented design to a communicating processes oriented implementation (which is more natural for distributed systems) and provides independence from middleware. Our approach also enables the use of formal methods as described in [10].

We propose a mapping of **L*f*P** concepts to a generic architecture that can be implemented on top of various execution environments. This is a way to help engineers to design and implement complex systems without getting into the often complex and delicate task of using sophisticated middleware services.

Our generic architecture relies on a runtime that virtualizes the execution environment. This is of particular interest when the application executes on several hosts running different operating systems. Effort expended on the implementation of the runtime on a given target architecture can be reused for future applications.

Future work aims to provide a set of coherent tools based on **L*f*P**. This is the goal of a project founded by RNTL (Réseau National des Technologies Logicielles, a french label and founding provided by the government for cooperation between industry and universities), dedicated to embedded distributed systems, that started in July 2003.

References

1. J. Abrial. *The B-book*. Cambridge University Press, 1995.
2. P. Bose. Automated translation of UML models of architectures for verification and simulation using SPIN. In Robert J. Hall and Ernst Tyugu, editors, *14th IEEE International Conference on Automated Software Engineering, ASE'99*. IEEE, 1999.
3. J. Corbett. Evaluating deadlock detection methods for concurrent software. *Software Engineering*, 22(3):161–180, 1996.
4. B. Dobbing and A. Burns. The Ravenscar tasking profile for high integrity real-time programs. In *Proceedings of SigAda'98*, Washington, DC, USA, November 1998.
5. S. Gnesi, D. Latella, and M. Massink. Model checking uml statechart diagrams using jack. In *4th IEEE International Symposium on High-Assurance Systems Engineering*. IEEE, 1999.
6. D. Helmbold and D. Luckham. Debugging Ada tasking programs. *IEEE Software*, 2(2):47–57, March 1985.
7. ISO. *Information Technology – Programming Languages – Ada*. ISO, February 1995. ISO/IEC/ANSI 8652:1995.
8. ITU-T. Open Distributed Processing, X.901, X.902, X.903 and X.904 standard. Technical report, ITU-T, 1997.
9. I. Khriss, M Elkoutbi, and R. Keller. Automating the Synthesis of UML StateChart Diagrams from Multiple Collaboration Diagrams. In Jean Bézivin and Pierre-Alain Muller, editors, *First International Workshop on The Unified Modeling Language, UML'98: Beyond the Notation*, volume 1618 of *LNCS*, pages 132–147. Springer-Verlag, 1999.

10. F. Kordon, , I Mounier, E. Paviot-Adet, and D. Regep. Formal verification of embedded distributed systems in a prototyping approach. In *International Workshop on Engineering Automation for Software Intensive System Integration*, June 2001.
11. F. Kordon and Luqi. An introduction to rapid system prototyping. *IEEE Transaction on Software Engineering*, 28(9):817–821, September 2002.
12. N. Leveson. Software engineering: Stretching the limits of complexity. *Communications of the ACM*, 40(2):129–131, 1997.
13. Luqi and J. Goguen. Formal methods: Promises and problems. *IEEE Software*, 14(1):73–85, January / February 1997.
14. S. Marc, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, 1996.
15. N. Medvidovic and R. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering*, 26(1):70–93, 2000.
16. SUN Microsystems. Java Remote Method Invocation (RMI), version 1.3. Technical report, SUN, 2001.
17. OMG. The common object request broker: Architecture and specification, revision 2.2. Technical report, OMG, 1998.
18. OMG. Omg unified modeling language specification, version 1.3. Technical report, OMG, 1999.
19. OMG. Initial Submission to OMG RFP's: ad/00-09-01 (UML 2.0 Infrastructure) ad/00-09-03 (UML 2.0 OCL). Technical report, OMG, 2001.
20. OMG. Model Driven Architecture (MDA), Document number ormsc/2001-07-01. Technical report, OMG, 2001.
21. QNX. System Architecture Guide - QNX RTOS v6, 2002.
22. D. Quartel, M. van Sinderen, and L. Ferreira Pires. A model-based approach to service creation. In *7th IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, pages 102–110. IEEE Computer Society, 1999.
23. D. Regep and F. Kordon. **L*f*P**: a specification language for rapid prototyping of concurrent systems. In *12th IEEE International Workshop on Rapid System Prototyping*, June 2001.
24. J. Zhao. A slicing-based approach to extracting reusable software architectures. In *CSMR*, pages 215–223, 2000.