Formal verification of embedded distributed systems in a prototyping approach

F. Kordon, I. Mounier, E. Paviot-Adet
LIP6-SRC, Université P.&M. Curie
4 place Jussieu, 75252 Paris Cedex 05, France
Fabrice.Kordon@lip6.fr
Isabelle.Mounier@lip6.fr
Emmanuel.Paviot-Adet@lip6.fr

&

D. Regep
PhD. student CS TELECOM
28, rue de la Redoute, BP 74
92263 Fontenay-aux-Roses Cedex, France
Dan.Regep@lip6.fr

Abstract: *This paper presents an evolutionary prototyping methodology dedicated to the design, verification and implementation of embedded systems. This methodology relies on **LfP**: a formalism combining UML-like structuring capabilities and a precise semantic suitable for both code generation and formal verification based on colored Petri nets. We apply this methodology on a small example and show how it enables system designers to detect non-trivial problems on the system.*

Keywords: *Prototyping, Formal verification, Petri Nets, UML.*

1. INTRODUCTION

Design and implementation of industrial systems is getting more and more complex **[9]**. This is a problem for embedded distributed systems for which a high quality is required. Several problems can be identified:

- Standard notation, such as UML **[15]** can be considered as an important contribution to describe a solution. However, it is more suitable at an early stage of application design and implementation. Thus, UML specifications are difficult to check due to their semi-formal semantics (dynamic aspects are not formally defined). A typical illustration is the interaction between components of a system. This information is dispatched into several diagrams: interaction, sequence and state.
- Once the specification of the system is completed, implementation needs to be done. Then, developers may interpret these specifications and we can get a program that is not exactly the image of the corresponding specification.
- Tests of the system are usually performed on the program. Then, when debugging the program, the initial specification tends to disappear: modifications on the program are not reported to the specification. Security of such modifications usually decreases with time over the maintenance of the system.

Evolutionary prototyping **[13]** is a good solution to these problems since it enhances the definition of a model serving as a basis for both the description of the system and automatic code generation. By reducing the production cost of an executable program, it promotes the model to be the center of the development process. Then, a system is elaborated by successive refinements of the following operations:

- design/refinement of the model,
- evaluation of the model,
- code generation,
- evaluation of the prototype,

In evolutionary prototyping, a strong correspondence between the model, programs and documentation may be maintained. However, evaluation of the system still relies on «traditional» testing techniques based on large benchmarks.

This paper presents an evolutionary prototyping technique. Our methodology relies on **L*f*P** (Language for Prototyping), a formalism dedicated to the description of embedded distributed systems **[18]**. **L*f*P** combines high level modeling facilities such as the ones of UML and a precise semantics suitable for both code generation and system verification by means of formal methods (instead of benchmarks-based testing).

Section 2. presents our prototyping methodology. Then, **L*f*P** is described in Section 3. Section 4. details an example of system specification using **L*f*P** and states some properties to be checked on this system. Finally, Section 5. shows how a formal specification can be generated from the **L*f*P** model and used to detect non-trivial errors.

2. METHODOLOGY

Our methodology is a model-based development in the sense of **[17]**: the model describes the system and serves as a basis for validation (in our case, formal verification) and code generation. Our methodological approach aims to implement evolutionary prototyping capabilities based on:

- *An integrative design approach*. **L*f*P** acts as a glue prototyping language **[2]** between state of the art specification formalisms (e.g. UML for system modeling, ODP as a distributed component framework **[8]**, Petri nets for formal verification).
- *An aspect oriented design framework*. **L*f*P** is based on a multi views approach to system prototyping **[7]**. Views are dedicated to a given prototyping aspect: software architecture, system implementation and formal property description.
- *A formalized development approach* to system behavior modeling and verification **[19]**. **L*f*P** relies on well formed Petri nets semantics **[3]** for formal verification.
- *A hierarchical, structured and modular approach* to system modeling **[4]**. **L*f*P** uses a component based approach allowing hierarchical specification and behavior refinement.

The main objective of **L*f*P** is to formalize relations between system modelling, formal verification and code generation of embedded distributed systems. Thus, we provide:

- transparent formal verification to enable its use in an industrial context without requiring specific training and skills **[12]**,
- strong correspondences between the detailed description of a system, its proofs and its implementation. In

other words: «what you check and what you implement is what you describe».

Figure 1 presents our methodology. Its input is an UML specification of a system. UML is not suitable for direct verification as noticed in the vUML project **[10]**. This is also true for distributed code generation from UML as mentioned in **[14]**. So, extensions to UML have to be considered. **L*f*P** has been elaborated for this purpose. **L*f*P** can be considered as an additional diagram that groups and enrich information provided in other diagrams. This unambiguous information is usefull to automate both code generation and formal verification.

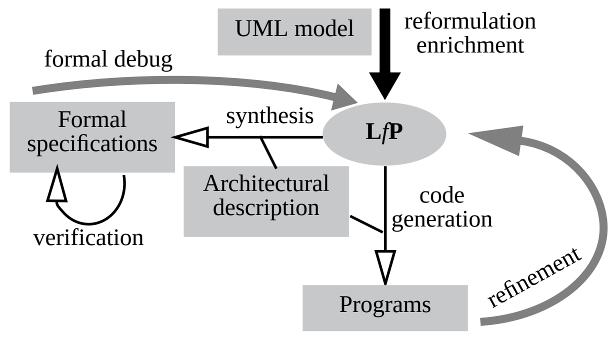


Figure 1 : *Our evolutionary prototyping methodology.*

The **L*f*P** model can be partially generated from UML standard diagrams. However, it contains enriched information compared to other UML diagrams: centralized description of components behavior by means of finite state machines (FSM), identification of properties to be verified and implementation directives. All this information is located in **L*f*P** and can be checked without being concerned with coherence problems between several diagrams.

Once the **L*f*P** model is produced, Petri nets synthesis can be performed. In Figure 1, «synthesis» corresponds to a set of transformations from **L*f*P** to Petri nets. Each one is dedicated to the verification of a given property according to a given strategy. This reduces the complexity of the proof: non relevant information can be discarded and thus, generated Petri nets are optimized.

Once all properties stated in the **L*f*P** model are verified (which may require some modification and several refinements on the diagram itself), code generation produces pieces of programs to be compiled and deployed in the target execution environment.

As shown in Figure 1, the main interest of evolutionary prototyping is to enhance the role of a model which enables: 1) several refinement of the system since production of the corresponding executable version is performed at low cost, 2) formal verification manageable by engineers since most of the process is hidden and performed automatically, 3) use of the **L*f*P** model, even during the maintenance phase.

To operate our methodology, we use a set of languages dedicated to each prototyping phase:

- UML for system specification and modelling,
- **L*f*P** diagram to centralize informations and to enable code generation as well as formal verification,
- Petri nets to apply formal verification procedures,
- Programming languages to implement the system.

3. THE L*f*P FORMALISM

This section summarizes the main features of **L*f*P**. Detailed information and rationale can be found in **[18]**.

L*f*P is a graphical Architecture Description Language with coordination facilities. It is dedicated to the rapid prototyping of embedded concurrent systems. **L*f*P** enhances an existing UML model with information enabling automatic code generation of concurrent programs and formal verification.

To do so, **L*f*P** uses three orthogonal views adapted to some specification aspects:

- the *functional view* (implemented as a diagram),
- the *implementation view* (annotations on the diagram),
- the *property view* (annotations on the diagram).

The functional view describes the system behavior in terms of execution workflow of connected components and the coordination between component instances. It also describes the system software architecture.

The implementation view describes the system implementation constraints (target executive, used programming language, communication infrastructure) and the deployment topology.

The property view specifies properties to be verified by the system (similar to the B proof-assertions **[1]**). Such properties are stated by means of invariants (for example, to check mutual exclusion), temporal logic formulas (for example, to check availability or fairness of a service) or other statements that can be converted to a given formal method. This view can be exploited to perform computer-assisted formal verification but also introduces relevant information for code generation (i.e. rutime checks).

3.1. The L*f*P Structure

The **L*f*P** functional diagram contains:

- a declarative part defining management information (e.g. model name, author, version number, comments and the associated UML model if any) and formal declarations: types or constants. Elementary types are: integer range, ordered enumerations or the opaque type. The opaque type denotes variables which only support the affectation operation and, thus, cannot influence the execution workflow.
- a list of entities: classes and media.

A **L*f*P** class corresponds to a complete UML instanciable class. Thus, abstract or virtual UML classes have no correspondence in **L*f*P**.

A media is used to connect classes. It specifies both interaction contract and communication semantics. It corresponds to an UML association, aggregation or composition.

Table 1 presents the graphical representation for classes and media.

L*f*P Class	**L*f*P Media**
Class name	Media name

Table 1: *Graphical representation of classes and media*

3.2. L*f*P entities

As mentioned in Section 3.1., the **L*f*P** functional diagram contains classes and media. Their description strongly relies on **L*f*P**-FSM (Finite State Machine) supporting various elements of a class or media specification. Thus, we present the **L*f*P**-FSM structure prior to classes and media.

3.2.1. L*f*P-FSM

L*f*P-FSM uses a notation similar to the one of Petri nets and provides some modelling facilities. They are used in various parts of a specification; the main difference consists in the signification of transition labels. **L*f*P**-FSM contains:

- a declarative part specifying a list of variables representing the execution context.
- the FSM itself. It specifies the execution workflow of a class, a class role, a method or a media. A **L*f*P**-FSM contains basic elements (or nodes) «wired» together using connection links.

Variables of a **L*f*P**-FSM context are either local to a class or media instance or shared between all of them. Variables are typed (according to a visible defined type) and may hold a default value. As mentioned in Section 3.1., opaque variables only support the affectation operation.

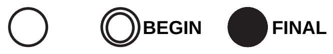
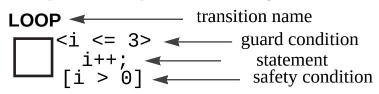
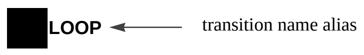
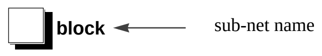
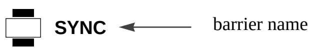
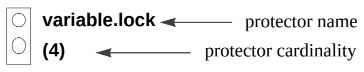
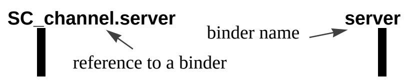
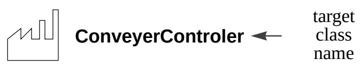
Symbol name	Icon
State	
Transition	 LOOP transition name <i <= 3> guard condition i++; statement [i > 0] safety condition
S-Transition	
H-Transition	
Barrier	
Protector	 variable.lock protector name (4) protector cardinality
Binder	 SC_channel.server binder name server reference to a binder
Constructor	

*Table 2: Graphical representation of **LfP**-FSM basic elements*

Table 2 summarizes nodes to be found in an **L*f*P**-FSM:

- *States* are execution steps. Two special states are distinguished: BEGIN and FINAL corresponding to the initial and final execution states. **L*f*P**-FSM has only one initial state.
- *Transitions* express potentially guarded actions. Guard conditions specify activation rules to be satisfied when firing a transition. The transition name may reference class role name (Section 3.2.2.), or a class method name. A statement is executed after the firing, it modifies state variables of the **L*f*P**-FSM visible at this level. These have an atomic execution semantics. Safety conditions express invariants useful for formal verification, debugging and testing
  Transitions may be linked to sequential code written using any programming language, to be inserted in the distributed application at code generation time. This code may change opaque variables values only and thus, cannot change the execution workflow.
- *Shadow Transitions* (S-Transitions) are graphical aliases to existing transitions proposed to simplify **L*f*P**-FSM.
- *Hierarchical Transitions* (H-Transitions) abstract sub-**L*f*P**-FSM to increase readability. Sub-**L*f*P**-FSM have one initial state and one terminal state. These are bound to the H-Transition input state and output state.
- *Barriers* are special shared transitions corresponding to a synchronization point between all concurrent instances of a **L*f*P**-FSM.
- *Protectors* are shared locks (multi-level semaphores or groups of semaphores) used to provide restricted access to a shared resource. They are used to define critical sections between concurrent instances of a **L*f*P**-FSM. A protector can be standalone or associated to one variable or group of variables. The protector cardinality specifies how many concurrent **L*f*P**-FSM instances may simultaneously get into the critical section.
- *Binders* are access points to media. **L*f*P**-FSM communicate through binders by means of messages. A message consists of three fields: 1) a message name known by the media, 2) message discriminants that can be modified by the media, 3) message arguments that must be opaque for the media. Binders are declared in media and referenced in classes.
- *Constructors* are used to create new class instances. An initialization context has to be specified for created instances.

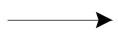
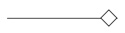
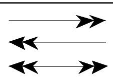
Arc	Protector link	Media link
		

*Table 3: Graphical representation of **LfP**-FSM connectors*

Table 3 presents connectors to be used in a **L*f*P**-FSM:

- *Arcs* are used to link a State to a Transition, S-Transitions, H-Transition or Barrier and vice versa. An arc express the execution sequence. **L*f*P**-FSM are sequential finite state machines. Thus, the number of input and output arcs of a Transition (S-Transition, H-Transition, or Barrier) is of exactly one.
- *Protector Links* connect Protectors to Transitions or S-Transitions and vice versa to define critical sections.
- *Media Links* are used to connect Binders or Constructors with Transitions or S-Transitions. Media Links specify the connection direction (in, out or inout). A Media Link specifies the binding contract between local context variables and messages (discriminant, name and arguments).

3.2.2. L*f*P Classes

A **L*f*P** Class corresponds to an UML implementation class and expresses some functional aspects of a system. It consists of:

- a declarative part specifying: 1) the class identifier, 2) for each binder, potential messages and their parameters (some of these messages correspond to public methods), 3) a list of private methods and their parameters, 4) definition of sequential code to be linked to transitions.
- a list of FSM defining : 1) the execution contract (main FSM), 2) class roles (optional), 3) methods.

Definition of the main FSM is mandatory. It represents the execution workflow of a class instance. Transitions in the main FSM may reference class roles (if any) or class methods.

Class roles correspond to alternative class behaviors; their definition is optional. Each role is described using a **L*f*P**-FSM. Transitions in a role may reference methods.

Class methods are also described by means of a **L*f*P**-FSM defining the execution workflow (i.e. the method execution contract).

Table 2 summarizes the graphical representation of the class main **L*f*P**-FSM, a class role or of a class method.

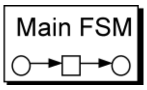
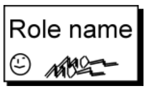
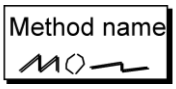
the main L*f*P-FSM	**a class role**	**a class method**
		

Figure 2 : *Graphical representation of class components.*

3.2.3. L*f*P Media

Media connect instances of **L*f*P** Classes. It is possible to use them as basic components or to assemble them into more sophisticated communication patterns. Media connecting two or more **L*f*P** classes correspond to an UML association, aggregation or composition. Media can also be used to implement shared resources (list, FIFO, stack, etc.).

A media specifies binding constraints and communication protocol semantics. We base our approach on the ODP contract definition **[8]**. A media consists of:

- a declarative part defining : 1) new types declaration 2) media variables which are similar to class variables, 3) the interaction contract consisting of several binding constraints.
- the main FSM representing the communication contract (communication protocol semantics).

The binding constraints specify: 1) a reference to the connected binding point, 2) the communication mode (`synchronous` or `asynchronous`), 3) the accepted messages and their arguments, 4) the binding multiplicity (one, `all` or any): one means that the binder is connected to only one class or media instance; `all` specifies that the binder is shared by all the connected classes instances; any leaves this unspecified.

A media cannot play various roles, has no methods nor associated constructors. Media carry on information on classes request.

4. AN EXAMPLE

Let us consider a set of conveyers circulating on a path divided in N segments as shown in Figure 3. A segment may contain only one conveyer. Conveyers may cross between segments where a crossing zone is defined (noted Z_i in the Figure). When two conveyers cross, the first one gets into a special path in the crossing zone and lets the other one get out before entering in the segment.

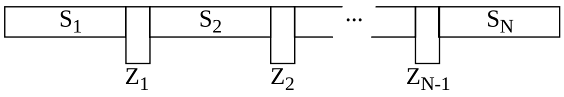


Figure 3 : *The conveyer system.*

4.1. Conveyer Behavior in the System

Conveyers, segments and crossing zones are locally driven by an embedded Control application. Command centers drive conveyers movements (using the `MOVE` message). However, these are not part of the system but correspond to an «external» component (e.g. a piece of code that already exists and has to be linked to the generated programs). Behavior of this component (instead of its implementation) is also represented to enable formal verification.

Therefore, the system contains three classes: SegmentControl (noted `SC`), ConveyerControl (noted `CC`) and CrossingZoneControl (noted `CZC`). Interactions between classes are defined using the following rules:

1) Upon receiving a `MOVE` command, a `CC` has to require (using `DEM` message) an authorization provided by the `SC` of the segment it wants to get in (if different from the current one). When it gets a positive answer (`AUT` message), it may come in.
2) This authorization may be refused (`REF` message), then, the conveyer must get into the crossing zone.
3) A conveyer stopped in a crossing zone is not considered to be in any segment.
4) The `SC` replies `AUT` when it is empty.
5) The `SC` replies `REF` when it contains a conveyer. It then stores the query in a local FIFO to reactivate the demanding conveyer when it is empty.
6) The `CC` sends `DEM` when it wants to leave a segment, just before entering in a new one.
7) The `CC` leaves the crossing zone when it gets a `GO` message from the `SC`. This message is sent when the conveyer occupying the segment leaves it.
8) When a `CC` leaves a segment (to get into another one or to get into a crossing zone), it notifies the corresponding `SC` by means of an `OUT` message. This message is also sent when a `CC` leaves a `CZC`.
9) When entrance in a segment is refused, `CC` checks (message `EMPTY`) if the `CZC` it has to get in is empty.
10) The `CZC` answers to `EMPTY` using `OK` (it is empty) or `PB` (it already contains a conveyer).
11) When `PB` is sent by a `CZC`, the `CC` sends `ALARM` to other conveyers and the entire system stops in an error state.

Figure 4 presents the static structure of the system as an UML class diagram.

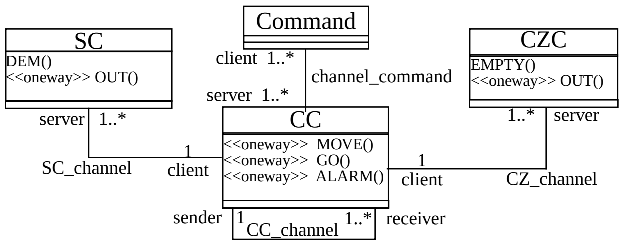


Figure 4 : *The UML class diagram of the conveyer system.*

Let us illustrate the rules exposed in Section 4.1. with UML sequence diagrams. The one of Figure 5 corresponds to a first scenario. Conveyer «c» located in segment «1» wants to get into segment «2». It sends `DEM` and gets `AUT`, enters in segment «2» and sends `OUT` to segment «1».

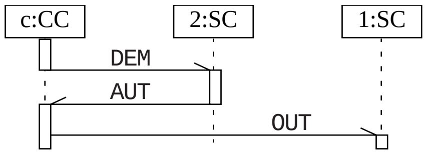


Figure 5 : *Sequence diagram of scenario 1.*

The UML sequence diagram of Figure 6 corresponds to a second scenario. A conveyer «c1», located in segment «1», wants to get into segment «2» where another conveyer «c2» is located. When «2» refuses entrance, «c1» gets into the crossing zone «z» after having checked if it is empty. When «c2» leaves «2», «c1» is waken up by «2» and leaves «z».

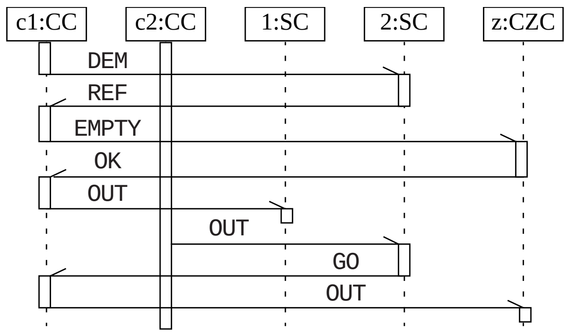


***Figure 6 :** Sequence diagram of scenario 2.*

4.2. The L*f*P Specification

In order to build the **L*f*P** diagrams we reuse information found in the UML models.

4.2.1. Description of the System

Figure 7 presents the main **L*f*P** diagram. The static structure of this is derived from the UML class diagram (Figure 4). It declares three classes (corresponding to the UML ones) and four media. The first three media correspond to the three UML associations and specify the communication protocol between classes as well as their interaction contract with the media. The last media is a local FIFO used by SC instances to store unsatisfied queries.

model_name := 'conveyeurs' ;
author := 'Dan Regep';
version := 0.0.1;
comments := '';
UML_model := 'conveyers.mdl';

const NBC := 2;
const NBS := 4;
const NBZ := 3;
type station **is integer range** 1..NBS;
type start_bound **is integer** (1,2,3);
type end_bound **is integer** (1,2,3);

static instances:
CC **with** HID **in range** 1.. NBC;
SC **with** HID **in range** 1..NBS ;
CZC **with** HID **in range** 1..NBZ;
properties:

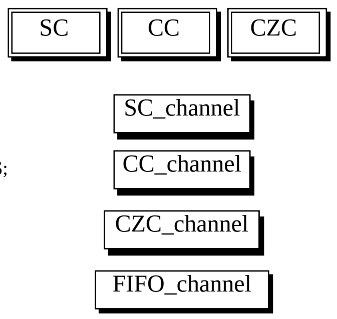


Figure 7 :** The* ***LfP *main diagram of the conveyer example.*

The declarative part of the **L*f*P** main diagram consists of:

- *general model information*. Model name, author, version, associated UML model.
- *declaration of constants and new data types*. `NBC`, `NBS` and `NBZ` constants respectively define the number of conveyers, segments and crossing zones. A new type (`station`) defining valid station identifiers. Two enumerated types (`start_bound` and `end_bound`) representing valid bounds of a segment in terms of stations (`stations` are numbered). We assume here that there is only one station per segment. This may be changed without modifying the structure of the **L*f*P** model.
- *declaration of static class instances with their initial context*. We find two conveyer instances, four segment instances and three crossing zone instances.
- *a list of properties to be verified for the system*. These declarations belong to the property view of the system. We provide some interesting properties in Section 4.3.

4.2.2. Description of the SC Class

Figure 8 presents the `SC` class. The declarative part specifies for each connected media binder, the list of accepted messages (marked as `in`) and possible outgoing messages (marked with `out`). According to the sequence diagrams of the two scenarios (Figure 5 and 6), a `SC` class instance may receive an entrance demand (`DEM`) or a notification message (`OUT`). A segment controller may reply to the demanding conveyer controller using two alternative messages: entrance authorization (`AUT`) or entrance reject (`REF`). It also sends `GO` to let a waiting conveyer come in from a crossing zone.

messages:
from SC_channel.server
in DEM();
in OUT();
out AUT();
out REF()
out GO();
from FIFO_channel.in
out WRITE (**in opaque** data);
from FIFO_channel.out
in READ (**out opaque** data);
internal_methods:
procedures:

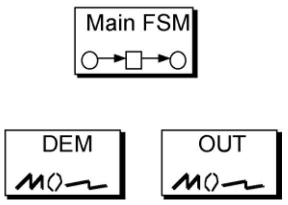


***Figure 8 :** The SC (Segment Control) class.*

The main **L*f*P**-FSM (Figure 9) specifies the execution contract of the `SC` class. Its purpose is to merge together the two alternative behaviors corresponding to the execution scenarios from Figure 5 and Figure 6.

The main **L*f*P**-FSM states that `DEM` and `OUT` methods should be mutually exclusive. Moreover, `OUT` can be executed only if the segment contains a conveyer (segment state is `full`).

The main **L*f*P**-FSM declares three local variables (duplicated in any class instance): `status` represents the execution state of a class instance (`empty` or `full`); `index` stores the number of pending demands; `HID` represents the class instance identifier. `HID` corresponds to a unique instance identifier.

When constructing new class instances, all context variables have to be initialized.

context :
local status **is** ('empty', 'full');
local index **:=** 0 **is integer range** 0..NBC;
local HID **is integer range** 1..NBS;

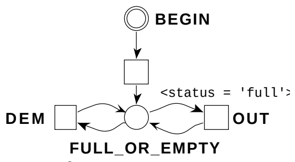


***Figure 9 :** The main LfP-FSM of the SC class.*

Figure 10 presents the `DEM` method **L*f*P**-FSM. It defines `conveyer_ID`, a local variable used to store the identifier of a demanding conveyer.

The `DEM` execution contract has two branches:

- When the segment is empty, access is granted and the segment state changes to `full`.
- When the segment is full, the query is stored in a `FIFO` media for further process and the index of pending demands is incremented. Then, a negative response (`REF`) is sent back to the conveyer.

Communication with the `FIFO` has oneway asynchronous message passing semantics.

Figure 11 presents the **L*f*P**-FSM of the `OUT` method. As for `DEM`, it also contains a local variable to store conveyer identities. Behavior of the `OUT` method consist of two alternative branches:

- If there is no pending request (`index` = `0`) then `status` of the segment is changed to `empty`.
- If there is at least one pending requests, the oldest demand is retrieved from the `FIFO` and a `GO` message is forwarded to the corresponding conveyer.

context : **local** conveyer_ID **:=** 0 **is integer range** 0..NBC;

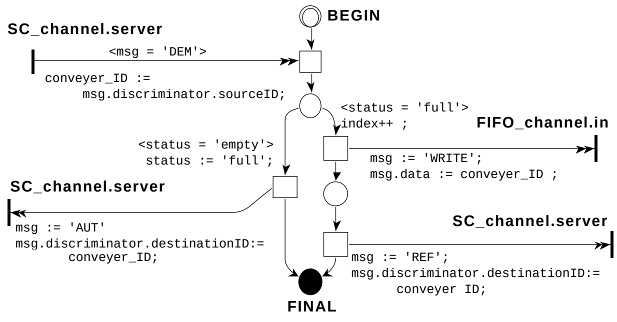


Figure 10 : *LfP-FSM of the DEM method.*

Due to space reasons the **L*f*P** representations of `CC` and `CZC` classes are not presented in this paper.

context : **local** conveyer_ID **:=**0 **is integer range** 0..NBC;

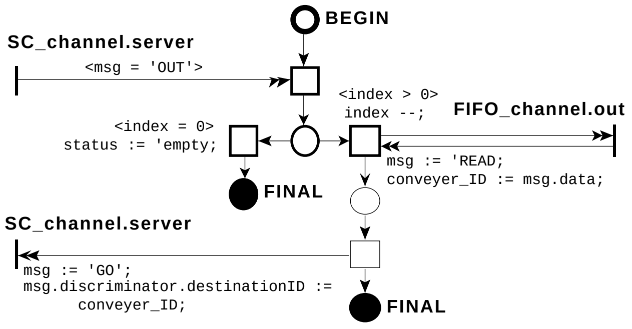


Figure 11 : *LfP-FSM of the OUT method.*

4.2.3. Description of the SC_channel media

The structure of the `SC_channel` media is presented below in Figure 12. Its context consists of two local variables: `client_ID` represents the identifier of the connected `CC` class and `message` is used to encapsulate the contents of an incoming message.

context :
local client_ID **is integer range** 1..NBC; **local** message **is opaque;**

binders:

client: **asynchronous** ;	**server: asynchronous;**
type msg.discriminator is	**type msg.discriminator is**
sourceID : integer range 1..NBC;	sourceID : integer range 1..NBS;
destinationID: integer range 1..NBS;	destinationID: integer range 1..NBC;
multiplicity := 1 ;	**multiplicity** := all ;
messages: in DEM ();	**messages**: **out** DEM ();
in OUT ();	**out** OUT ();
out AUT ();	**in** AUT ();
out REF ();	**in** REF ();

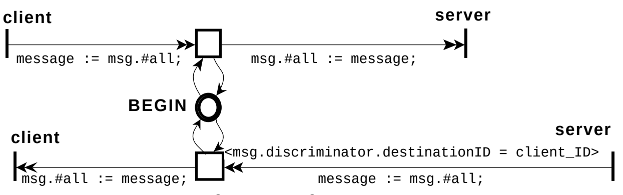


Figure 12 : *LfP-FSM of the SC_channel media.*

`SC_channel` has two binders (`client` and `server`) through which it is connected to a client (a `CC` class instance) and to all server instances together (all `CS` class instances). The connected conveyer is the client (`client` multiplicity is `one`) and all connected segments are servers (multiplicity of the `server` binder is marked as `all`).

The media may transport several messages. Possible messages and their parameters are enumerated for each binder. The communication is asynchronous through both binders.

When receiving a message through the client binder, the media dispatches it to the concerned server. This is achieved using a simple copy of the message contents from the incoming binder to the output one.

In order to match an incoming message from a server, the message destination should be identical to `client_ID`.

4.3. Properties to be Verified

Two kinds of properties may be considered with regards to modular specification:

- properties local to a module,
- properties global to the complete specification.

Local properties concern the internal behavior of a component independently from its environment. If we consider the `SegmentControl` class, local properties express the link between the demand of a conveyer and the answer of the segment, such as:

i. if SegmentControl gets request to enter an empty segment, it answers «`OK`» to the conveyer,

ii. if SegmentControl gets request to enter a full segment, it answers «`REF`» to the conveyer.

Global properties concern the behavior of the complete system; verification thus requires the specification of the entire system. An example of such a property is :

iii. the system is deadlock free.

5. FORMAL VERIFICATION

L*f*P specifications cannot be used «as is» to perform formal verification. Thus, a translation into a verification language is necessary. The generated formal specification is not as easy to read as the one in **L*f*P**, but handles automated formal verification.

It is usually impossible to perform formal verification without abstraction and reduction of the system at the formal level. However, as most abstractions and reductions rely on the semantics of the property to be verified, we produce one formal specification per property to verify. Thus, the obtained formal specification is equivalent to the **L*f*P** one regarding the considered property.

We choose well formed colored Petri net **[3]** because, in addition of excellent capabilities for the description of concurrent systems, they support both structural and behavioral verification methods.

Let us use the conveyers example to illustrate validation of the behavioral properties stated in Section 4.3. To validate this system, we use CPN-AMI, a Petri net based CASE environment **[11]**.

5.1. Colored Petri Nets

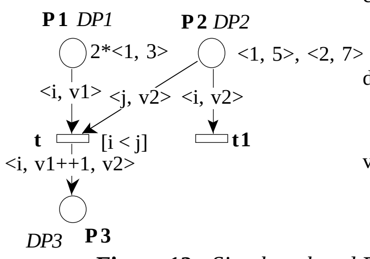
This section informally presents colored Petri nets.

A colored Petri net is a 5-uple <P, T, Pre, Post, Types,

M_0> where:

- P is a set of places (depicted by circles).
- T is a set of transitions (depicted by rectangles).
- Pre[t] is the precondition function for transition t.
- Post[t] is the postcondition function for transition t.
- Types is the set of basic types. A basic type is a finite set.
- M_0 is the initial marking.

Figure 13 depicts a simple colored Petri net with 3 places (P1, P2 and P3) and 2 transitions (t and t1).



```
class
  Id is 1..3;
  Value1 is 1..10;
  Value2 is 4..15;
domain
  DP1 is <Id, Value1>;
  DP2 is <Id, Value2>;
  DP3 is <Id, Value1, Value2>;
var
  i, j in Id;
  v1 in Value1;
  v2 in Value2;
```

Figure 13 : *Simple colored Petri net example.*

To each place p, a domain Dom(p) is associated: Dom(p) is the cartesian product of some basic types. In Figure 13, basic classes are Id, Value1 and Value2. The domain of P1 is the cartesian product of Id and Value1, the one of P2 is the cartesian product of Id and Value2 and the one of P3 is the cartesian product of Id, Value1 and Value2. Dom(p) corresponds to the set of token color that place p can possibly contain.

A marking M(p) is associated to each place p: M(p) is a multi-set over Dom(p). Therefore, a marking M is the function that associates a marking to each place p of P. An element of a marking in a place is called a token. In Figure 13, the initial marking associates:

- two tokens with color <1, 3> to P1,
- tokens <1, 5> and <2, 7> to P2,
- the empty multi-set to P3.

Pre and Post functions describe how a marking is modified when an action is performed. Since actions are associated to transitions, instead of «an action is performed» we say: «a transition is fired».

To each transition, a set of variables Var(t) is associated. Each variable is defined over a basic type. In Figure 13, Var(t) = {i, j, v1, v2} and Var(t1) = {i, v2}. Variables i and j are defined over the basic class Id, variable v1 is defined over Value1 and variable v2 is defined over Value2

Let us call a binding of transition t the association of a value to each variable of Var(t). Let x a binding of t, Pre[t][p, x] returns a multi-set over Dom(p). A transition t can be fired for a marking M with a biding x iff:

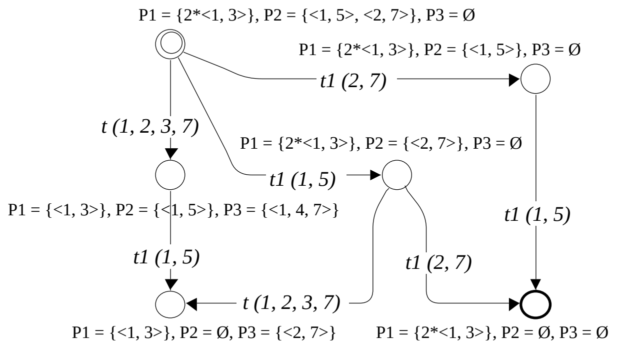
- constraints over the binding are satisfied (they are called guards), [i < j] is a guard associated to transition t in Figure 13.
- Pre[t][p, x] is included in M(p) for all p of P,

Post[t][p, x] also returns a multi-set over Dom(p). If t can be fired for binding x, then a new marking M' can be computed: M'(p) = M(p) - Pre[t][p, x] + Post[t][p, x]. In order to synchronize different values, successor (v++n) and predecessor (v--n) functions are defined (see postcondition of t in Figure 13).

In Figure 13, many bindings can be found for transition t, like i = 3, j = 5, v1 = 7, v2 = 6. However, t cannot be fired for this binding since <3, 7> is not a token in P1 for the initial marking. The following binding allows t to be fired : i = 1, j = 2, v1 = 3, v2 = 7 (token <1, 3> belongs to P1 and token <2, 7> belongs to P2, there is no precondition for P3 and the guard is satisfied since i < j). When t has been fired a new marking M_1 is computed:

- P1 contains the token <1, 3>,
- P2 contains the token <1, 5>,
- P3 contains the token <1, 4, 7>.

From this new marking no binding can be found for t to be fired (the only possible binding would be i = 1, j = 1, v1 = 3, v2 = 5 and it does not satisfy the guard i < j). Figure 14 shows the reachability graph of the net figure Figure 13. The double circled state corresponds to the initial marking (M_0) of the net.



P1 = {2*<1, 3>}, P2 = {<1, 5>, <2, 7>}, P3 = Ø
P1 = {2*<1, 3>}, P2 = {<1, 5>}, P3 = Ø
t1 (2, 7)
t (1, 2, 3, 7)
P1 = {2*<1, 3>}, P2 = {<2, 7>}, P3 = Ø
t1 (1, 5)
P1 = {<1, 3>}, P2 = {<1, 5>}, P3 = {<1, 4, 7>}
t1 (1, 5)
t1 (1, 5)
t1 (2, 7)
t (1, 2, 3, 7)
P1 = {<1, 3>}, P2 = Ø, P3 = {<2, 7>}
P1 = {2*<1, 3>}, P2 = Ø, P3 = Ø

Figure 14 : *Reachability graph of the simple colored Petri net.*

5.2. From L*f*P to Colored Petri Nets

We have to ensure that results computed for the Petri net can be translated into **L*f*P** terms. Therefore, the translation process has to preserve the component structure of **L*f*P** models. This strategy also enables modular verification when it is possible (e.g. for local properties).

Therefore, we work at the module level (modules are deduced from **L*f*P** classes). We then compose them to produce a complete Petri net of the system.This procedure has two main steps:

- Generation of Petri net modules from **L*f*P**-FSMs.
- Composition of Petri net modules

To illustrate the translation procedure, we consider the specification of the SegmentControl.

5.2.1. Generation of Petri Net Modules

Structure of the Petri net. To obtain the structure of the Petri net, we consider **L*f*P**-FSM of the input model:

- In the Main-FSM, transitions corresponding to methods are replaced by the corresponding **L*f*P**-FSM.
- Places initiating methods are identified with the BEGIN place in the corresponding **L*f*P**-FSM
- Method output places are identified with FINAL places in the corresponding **L*f*P**-FSM.
- We preserve at the Petri net level, names of **L*f*P** places and transitions. Unnamed **L*f*P** places and transitions are given an arbitrary name in the Petri net. Naming is requested by some verification tools.

If we consider the SegmentControl class, we obtain the Petri net of Figure 15. Black places and transitions are those of the Main-FSM. Transitions `DEM`, `t1`, `t2`, `t3` and places `P1`, `P2` describe the method `DEM`. Transitions `OUT`, `t4`, `t5` and place P3 describe the behavior of method `OUT`. Transition `DEM` (respectively `OUT`) are shared by the Main-FSM and the method `DEM` (respectively `OUT`). `Channel_SC_server` and `FIFO_channel_out` correspond to media.

The verification we consider does not matter with the implementation of communication channels. We may thus abstract their specification with a single place. However it requires their implementation to respect the following property: *channels are deadlock and loss free*. This assertion has to be inserted as an implementation note used by code generators.

Color-domains, valuations and initial marking. Once the structure of the Petri net obtained, it is necessary to define variables management using color classes and domains, variables, valuations and guards. A color domain representing the information required to determine the state of the system is associated to places in the Petri net model.

We thus consider the variables identified in FSMs:

- Information depicted by variables declared in the main FSM is associated to place,
- Places derived from methods are enriched by local variables.
- Information in a channel contains three parts : the source, the destination and the value of the message.

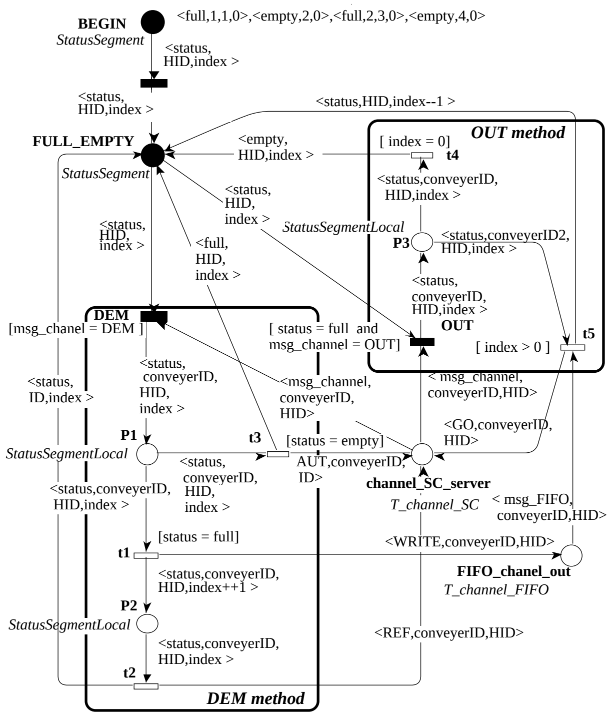


Figure 15 : *Petri net of the SC (SegmentControl) class.*

Let us illustrate this on the Petri net in Figure 15 and the corresponding declarative part in Figure 16. Places of the main FSM carry: status of a segment (empty or full), identity of the segment (integer between 1 and 4) and an index that indicates the number of conveyers waiting for segment release. This information is represented by the `StatusSegment` domain. Places derived from methods also contain the identity of the conveyer willing to enter the segment. Therefore the new domain, `StatusSegmentLocal`, is defined. Each of the two communication channels has its own domain (`T_channel_SC` and `T_channel_FIFO`).

The following declaration contains the classes, domains and variables that are necessary for the description of the SegmentControl class.

```
CLASS
  Tstatus is [empty,full];
  TCC is 1..2;
  TSC_CZC is 1..4;
  Tindex is 0..2;
  Tmsg_channel_SC is [OUT,DEM,AUT,REF,GO,VIDE,OK,PB];
  Tmsg_FIFO_channel is [ WRITE,READ ];
DOMAIN
  StatusSegment is <Tstatus,TSC_CZC,Tindex>;
  StatusSegmentLocal is <Tstatus,TCC,TSC_CZC,Tindex>;
  T_channel_SC is <Tmsg_channel_SC,TCC,TSC_CZC>;
  T_channel_FIFO is <Tmsg_FIFO_channel,TCC,TSC_CZC>;
VAR
 status in Tstatus;
 conveyerID2 in TCC;
 conveyerID in TCC;
 HID in TSC_CZC;
 index in Tindex;
 msg_channel in Tmsg_channel_SC;
 msg_FIFO in Tmsg_FIFO_channel;
```

Figure 16 : *declarative part of the SC (SegmentControl) class.*

Arcs valuation and transitions guards are also deduced from the **L*f*P** specification. Let us consider transition `DEM` in Figure 15.

To be fired it requires one token from place `FULL_OR_EMPTY` (domain `StatusSegment`) and one token from `channel_SC_server` (domain `T_channel_SC`). The arc from place `FULL_OR_EMPTY` to `DEM` is valuated by the tuple <`status`,`HID`,`index`>; the one from `channel_SC_server` to `DEM` is valuated by the tuple <`msg_channel`, `conveyerID`,`HID`>.

So, to fire `DEM`, a message must be sent to a segment identified as `full` or `empty` (this segment is not responding to a request) and `HID` variables must be the same in both valuation. The token stored in the `DEM` output place is a combination of the inputs: <`status`,`conveyerID`,`HID`,`index`>.

Transition `t3` authorizes (`t2` refuses) the entrance in the segment; `t3` has the following guard `[status = empty]` (respectively `[status = full]` for `t2`). These guards correspond to the preconditions defined in the **L*f*P**-FSM (Figure 10).

This way, places domain, arcs valuation and transitions guard are computed from the **L*f*P** specification. Petri net of Figure 15 and declaration in Figure 16 represent the complete Petri net specification of `SegmentControl` Class.

The initial marking of the Petri net corresponds to the static instantiation of classes. For `SegmentControl`, we indicate which are the full segments; for `ConveyerControl` we indicate the segment identifier where each conveyer is and, we also indicate that all `CrossingZoneControl` instances are empty.

Petri net reduction. As the Petri net is automatically synthesized from the **L*f*P** specification, its structure may be not optimized. Some reductions are possible regarding the class of properties to verify. These reductions concern the Petri net structure. Therefore, combinatorial explosion of the corresponding state graph is reduced and verification becomes easier.

If we consider deadlock freeness, reduction techniques presented in **[6]** can be applied. If we consider verification of temporal properties, reduction techniques presented in **[16]** are necessary.

The first technique is compatible with deadlock freeness property (property iii.), the second one is compatible with Properties i. and ii. Some reductions, as the following one, belong to the two techniques. The rule we apply aims to identify two transitions (t_a and t_b) where the bindings of t_b depends only on the bindings of t_a. Such a reduction is possible between transitions DEM and (t1 and t3).

Figure 17 shows the reduced Petri net. Black transitions replace the three ones that have been reduced. The place between DEM, t1 and t3 has been suppressed.

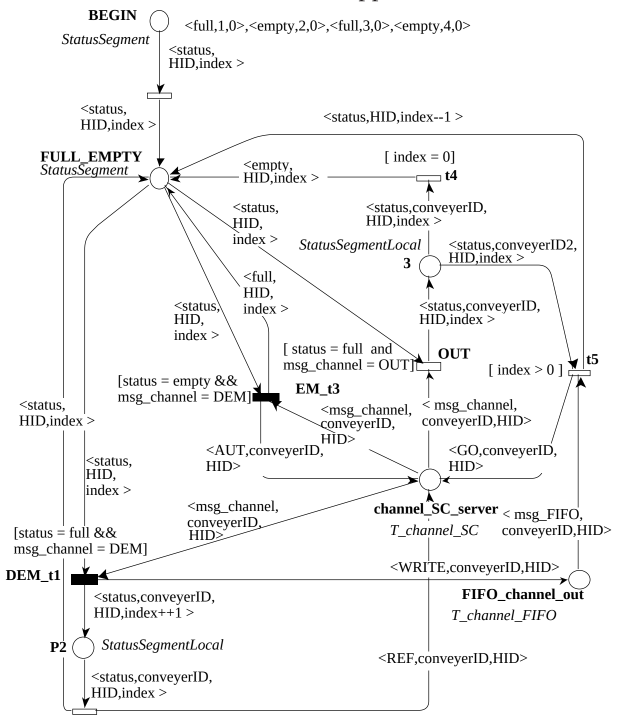


Figure 17 : *Reduced Petri net of the SC (SegmentControl) class.*

5.2.2. Composition of Petri Net Modules

Composition. The composition of modular Petri nets is obtained by identification of channel binding points. In our case, as each channel is represented by a single place, we perform the fusion of all the places representing the same channel.

Abstraction of system environment.To verify our system, we need a representation of its environment. At the verification level, this environment consists of the Command class (Figure 4) and channel_command.

Due to possible communications between Command and CC classes, this environment can be represented by means of a message generator coming from the communication channel. Command sends all possible messages since we have no constraints on it. Therefore, even if we consider a particular initial configuration, the evolution of the system leads to all possible configurations.

The complete model.The assembled model, obtained by composition of reduced modules, contains 20 places, 28 transitions and 92 arcs.

5.3. Verification of properties

We consider two types of properties: local and global.

5.3.1. Local properties

Let us consider SegmentControl with properties i. and ii. defined in Section 4.3. The formal language we use to express such properties is a temporal logic **[5]**, properties i. and ii. are interpreted as :

- ***i.:*** if transition DEM is fired with the binding (status = empty, conveyerID, HID, index, msg_channel = DEM) then place channel_SC_server will eventually contain the token <AUT,conveyerID,HID>,
- ***ii.:*** if transition DEM is fired with the binding (status = full, conveyerID, HID, index, msg_channel = DEM) then place channel_SC_server will eventually contain the token <REF,conveyerID,HID>

To verify such a property, it is not necessary to consider the entire system specification. The Petri net of ControlSegment class associated to an abstraction of its environment with an adapted initial marking is sufficient.

We use PROD **[20]**, a model checker dedicated to colored Petri nets and integrated as a component in CPN-AMI **[11]** to verify the temporal logic specification of these porperties. These properties are verified. Once all classes individually verified, we can consider global properties.

5.3.2. Global Properties

We now want to verify property iii. Theorem provers are able to check such a property without considering a specific instantiation of the system (e.g. a given number of segments and conveyers). However, such proofs cannot be automated.

To enable automated proofs based on the reachability graph, we have to instantiate the system. Such instantiations can be deduced from the expected size of the system. A well accepted strategy is to start with a small number of resources and components to check if the property is correct. Then we use realistic dimensions of the system for a safer verification.

So, let us start with two conveyers and four segments. We first compute the reachability graph of the complete Petri net and look for terminal nodes (i.e. specification deaddocks). We also used PROD to evaluate this property.

The computed reachability graph holds 3072 nodes, 6209 arrows and 33 terminal nodes. Thus, our specification is not correct. PROD helped us to extract a path between the initial state and one of the terminal nodes. This path builds a scenario explaining why our specification is not deadlock free.The scenario is the following :

- Initially conveyer 1 is in segment 1 and conveyer 2 is in segment 3,
- Conveyer 2 asks for segment 2 and enters in it.
- Conveyer 1 then asks for segment 2 and is not allowed to enter it.
- Conveyer 1 therefore asks to enter in crossing zone 1.
- It is allowed to enter this crossing zone which is immediately set to occupied even if conveyer 1 has not yet left segment 1.
- Conveyer 2 then asks for segment 1, it is not allowed to enter since conveyer 1 has not yet left the place.
- Therefore it asks to enter crossing zone 1. This raises a problem and the system stops.

This deadlock is due to asynchronous communication between classes : conveyer 1 has not yet considered the authorization from the crossing zone since the crossing zone considers it is already in. Then, our specification does not ensure that the number of occupied segments and crossing zones remain equal to the number of conveyers. The verification process shows an implicit property that should be explicitly expressed in the **L*f*P** verification view.

To solve this problem, a communication protocol between classes has to ensure atomicity of the following actions:

- a crossing zone (or a segment) accepts a conveyer's demand,
- this conveyer gets into the crossing zone (or segment),
- the leaved crossing zone (or segment) moves to the empty state.

A transactional system should ensure such a property. Thus, the systems designer has to update the **L*f*P** specification according to this observation. Such an operation corresponds to what we called «formal debug» in Figure 1.

6. CONCLUSION

In this paper, we have presented an evolutionary prototyping methodology that promotes formal verification and debugging of a specification as well as code generation of distributed programs.

This methodology relies on **L*f*P**: a formalism offering structuration capabilities and having a precise semantics suitable for the description of interaction between components of an embedded distributed system. The strong semantical definition of **L*f*P** aims to eliminate problems observed on a standard notation such as UML when it comes to code generation and formal verification. However, **L*f*P** remains connected to UML since we consider it as an additional diagram. Some parts of this diagram may be deduced from classical UML diagrams but system designers have to provide new information regarding cooperation between classes in the system.

We illustrated our methodology on a small example: a conveyer system. This example showed that non-trivial errors can be detected on a system that appears to be correctly described. The detected problem deals with sophisticated behavioral aspects of the system which are due to some unspecified aspects on the system (here, some communication issues were not properly stated).

Based on the study presented in this paper, it appears that our methodology has some «nice» capabilities when designing a system:

- It is connected to a standard UML-based approach. In our methodology, UML design fits the early conception.
- **L*f*P** is used as a basis for detailed description of the system and a basis for code generation and formal verification.
- Our transformation techniques preserve a strong correspondence between the model level (**L*f*P**), the formal level and the program level.
- The use of formal methods to check properties may be hidden to the end user. Then, it can be used by engineers having a low knowledge on formal methods.
- Formal verification techniques enable formal debug at the model level. Then, if we assume that no bug is introduced by code generators, system implementation should behave according to the verified properties.

Our methodology is partially implemented in CPN-AMI, a Petri net based CASE environment.

7. REFERENCES

[1] J.R. Abrial, "The B-book", Cambridge University Press, 1995

[2] M. Björkander, "Graphical Programming Using UML and SDL", in Computing Practice, Vol. 33, No. 12, December 2000, <http://www.computer.org/computer/co2000/rztoc.htm#rz030>

[3] G. Chiola, C. Dutheillet, G. Franceschini & S. Haddad, "On Well-Formed Coloured Nets and their Symbolic Reachability Graph", High Level Petri Nets. Theory and Application. Edited by K. Jensen G.Rozenberg, Springer Verlag 1991

[4] M Elkoutbi, R. Keller :"Modeling Interactive Systems with Hierarchical colored Petri nets", In Proceedings of the 1998 Advanced Simulation Technologies Conference, pages 432-437, Boston, MA, April 1998. The Society for Computer Simulation International. HPC98 Special session on Petri-Nets.

[5] E. Emerson, "Temporal and Modal Logic" Handbook of Theoretical Computer Science, Chapitre 16, pages 995-1072 , Elsiever Science, 1990

[6] S. Haddad, "A Reduction Theory for Coloured Nets", LNCS : High Level Petri Nets. Theory and Application. Edited by K. Jensen , G. Rozenberg, Springer Verlag 1991

[7] V. Issarny, T. Saridakis and A. Zarra : "Multi-view Description of Software Architectures", in Proceedings of the 3rd ACM SIGSOFT International Software Architecture Workshop, pages 8184, November 1998, http: //www.irisa.fr/solidor/doc/../doc/ps98/isaw98.ps.gz

[8] ITU-T : "Open Distributed Processing", X.901, X.902, X.903 and X.904 standards, <http://www.itu.int/itudoc/itu-t/rec/x/x500up>

[9] N. Leveson, "Software Engineering: Stretching the Limits of Complexity", Communications of the ACM, Vol 40(2), pp 129-131, February 1997.

[10] J. Lilius, I. Porres Paltor :" vUML: a Tool for Verifying UML Models", in Proceedings of the 14th IEEE International Conference on Automated Software Engineering, October, 1999. <http://dlib.computer.org/conferen/ase/0415/pdf/04150255.pdf>

[11] the LIP6-SRC team, the Mars project homepage, <http://www-src.lip6.fr/logiciels/mars/>

[12] Luqi & J. Goguen : "Formal Methods: Promises and Problems", IEEE Software, Vol 14, N°1, pp 75-85, January 1997.

[13] Luqi, V. Berzins, M. Shing, R. Riehle and J. Nogueira : "Evolutionary Computer Aided Prototyping System (CAPS)", in Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34), August 2000, Santa Barbara, California

[14] N. Medvidovic, Richard N. Taylor : "A Classification and Comparison Framework for Software Architecture Description Languages", IEEE Transactions on Software Engineering, vol. 26, no. 1, january 2000. <http://dlib.computer.org/ts/books/ts2000/pdf/e0070.pdf>

[15] OMG, "OMG Unified Modeling Language Specifiacation", version 1.3, June 1999, <http://www.omg.org/cgi-bin/doc?ad/99-06-09.zip>

[16] D. Poitrenaud and J.-F. Pradat-Peyre, "Pre and Post-agglomerations for LTL Model Checking", proceedins of 21st International Conference on Application and Theory of Petri Nets, Aarhus, Denmark, June 2000, pages 387-408, Springer-Verlag

[17] D. Quartel, M. van Sinderen, L. Ferreira Pires : "A model-based approach to service creation", in Proceedings of the Seventh IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems, pages 102-110. IEEE Computer Society, 1999. <http://wwwhome.cs.utwente.nl/~quartel/publications/Ftdcs99.pdf>

[18] D. Regep, F. Kordon, "**L*f*P**: a specification language for Rapid prototyping of Concurrent Systems", to appear in proceedings of the 12th IEEE International Workshop on Rapid System Prototyping, Monterey, June 2001

[19] J. Saldhana and S. M. Shatz, "UML Diagrams to Object Petri Net Models: An Approach for Modeling and Analysis," Proceedings of the Int. Conference on Software Engineering and Knowledge Engineering (SEKE), Chicago, July 2000, pp. 103-110.

[20] K. Varpaaniemi, J. Halme, K.Hiekkanen & T.Pyssysalo, "PROD reference manual", Technical Report B13, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland, August 1995