A Hierarchical Approach to the Formal Verification of Distributed Systems

Mohamed Larbi, Rebaiaia* and Jihad Mohamad, Jaam**
*Computer Sciences Department,University of Batna, Algeria
Email : Rebaiaia@arabia.com
**Computer Sciences Department,Qatar University, Qatar
Email: jaam@qu.edu.qa

Abstract

With the increasing emergence of hardware/ software distributed systems, designers need in some ways efficient methods and tools to improve the safeness of such systems and to reduce drastically their design engineering time. In general, some traditional methods as simulation and testing are used to detect errors and bugs, but they are inadequate to certify the correctness of complex systems. To accelerate the design and to avoid distributed systems malfunctions, new mathematical-based techniques has been used to improve the deficiency of the old methods. During the last two decades, Model checking and theorems proving have been the major research verification fields intensively used to check the design correctness of some big projects (NASA lance rockets, Pentium chips, ect.) In fact, despite the generalization of formal methods, some problems remain in suspense in producing a coherent specification fully integrated semantics and avoiding the size complexity of designs.
In this paper, we present a hierarchical approach and an open environment to describe and to verify distributed and concurrent systems. The system currently uses UML notation and provides rewriting logic, model checking, theorem proving, and simulation techniques.

1. Introduction

Formal specification and verification are now widely accepted by the software engineering community and industrial practitioners. The correct functioning of distributed systems has been an issue of vital importance in the system development process. Nowadays, several formal verification techniques and tools are available. They are divided into two major fields, a proof-assistant program as ACL2 [8], which needs the interaction with specialist users by means of lemmas to verify theorems, and fully automatic programs called model-checking based essentially on temporal logics as for example CTL and LTL (Linear Temporal Logic) [5]. Following the development of model-checking programs a large number of tools called Model-checkers have been developed for different applications in different areas. For instance, SMV [10] for hardware systems, STEP for reactive systems, SPIN [20], for communication protocols and UPPAAL [16] for timed systems. Despite their well-founded mathematical provability, these tools lack in two major problems: a theoretical part, which is the state explosion that grows exponentially with the number of states and transitions and an informal part used to design the

different components of the systems, their relations and their objectives using a known formalism as UML (Unified Modeling Language) specification [4-17]. The result of the execution (i.e., verification) is the response of the system by YES, which mean that the model is correct and by No, saying that a part of the model could be erroneous and finally gives a counter example (i.e., the behavior execution trace). Several other approaches try to use process algebra logic like CCS [13] and LOTOS [3], embedded respectively in some tools like the Concurrency Workbench [18] and CADP [19]. The problem with such kind of formalisms remains the heaviness to describe multifunctional systems.

These considerations motivated the development of VALID-2, a specification, verification and simulation environment, which is the result of many discussions with Professor A. Attoui at LLP-CESALP Laboratory of Haute-Savoie University (France). The first version of a similar environment has been developed at the LAMOS (University of Clermont-Ferrand, France) and a naïve form called VALID is obtained. It is available on-line at the following web address www.univ-savoie.fr and published in [1]. VALID, as it was designed, is just a simple environment used for the formal specification of reactive systems.

The environment VALID-2 written in Java (JDK2), is completely different from VALID environment developed in C++ of Borland (a naïve C++ compiler). In fact, VALID-2 is based mainly on Statecharts to describe formal objects and their relationships, and on OCL (Object Constraint Language) to trace up all the functional rules of the systems [17]. VALID uses a simple graphical form to describe the system, its objects and the communication links. The dynamic behaviors are processed using rewriting rules. The termination of the generated rewriting system is checked using the tool ORME, a primitive proof-checker [9]. VALID-2, uses a specific programming language called SPECIF, which is a subset form of Maude language which is a well-known efficient executable system developed at Stanford University [6]. Such incorporation is due to the well-known rewriting logic and its reflective computation [12]. VALID did not embed neither a verification model-checker nor a simulation mechanism. Both the model-checker and the simulation program are available in VALID-2, in addition to two translators: one from UML description to Java code for the simulation, and the second for the specification language SPECIF. VALID-2 embeds also other functionalities, which cannot be fully presented in this article.

Briefly VALID-2 is based on four modules: *(1)* a graphical description like-UML module, using a visual interface, *(2)* a prototyping and code-generation module which translates the UML graphical description as input giving formal modules using a specification language based on rewriting logic semantic and a native reflexive Maude code, and a computable Java-code. *(3)* An animation and simulation module based on the Simjava package [11] and *(4)* a verification of specifications module operators translated as rewrite rules extended with some new operators. The system integrates modularity and abstraction and follows the main principles of an Object-Oriented approach inspired by Maude system and Java programming language.

This paper is organized as follows: Section 2, gives an overview of the system components, which are the syntax and the semantic of the SPECIF language, UML description as a specification language, the model-checker called VERIF. Section 3, presents the simulation module, while Section 4, conclude the paper.

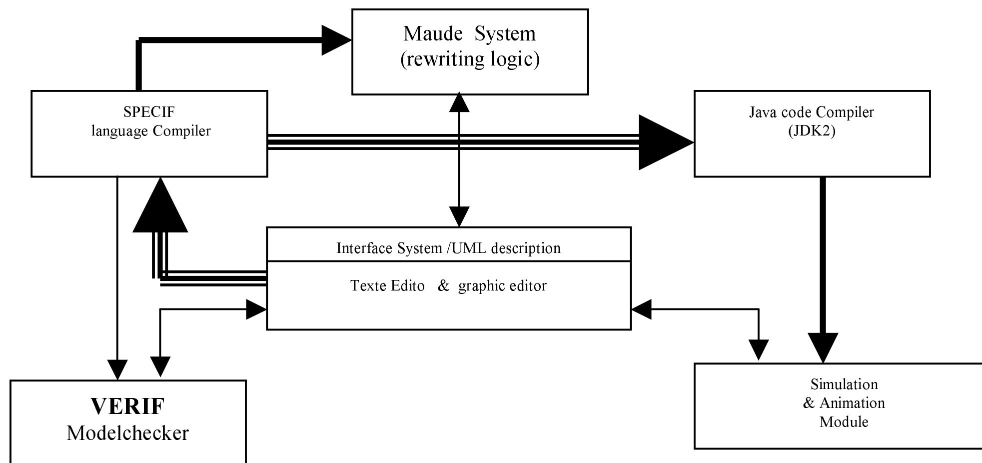

Figure 1 : The overall VALID-2 architecture

2. Overview of VALID-2 Environment

As it was already mentioned, the architecture of VALID-2 as depicted in figure 1, is centered on some independent programs and systems, which interact together and monitored by a supervisor. In this section we detail some of them, especially the SPECIF system, the UML (the description is introduced through an example), the model-checker VERIF and the simulation and animation program.

2.1. Syntax and Semantic of the SPECIF Language

In contrast to LOTOS and ESTEREL [2], the syntax of the SPECIF language is closely similar to the Maude language and it is based on Rewriting logic as developed by Meseguer et al. and well introduced in [12]. The most important particularity is its capacity to use a single theory for the formal specification of data structure and the dynamic behaviour of systems. The paradigm of theories in this case is built as a generalisation of the object-oriented and functional programming and retains the basic concept of the formal module. A formal module is allowed to describe a module as in Obj3 [12]. To be more accurate about Maude system the reader is invited to consult the site www.csl.sri.com.

Our approach is based on a simple theory representing an object as a Formal Module. In the SPECIF language, a system configuration is defined through a number of objects and messages and represents the formal description of a concurrent system. Each object (formal module) possesses intrinsic properties. Messages are generated by external stimuli or internal interruption. The same message can be sent to different objects. In another words SPECIF language takes also some advantages from the language developed by Attoui et al. in [1]. It presents the following characteristics to describe the syntactic form:

2.1.1. The alphabet of the SPECIF language

In algebraic specifications, as in most declarative languages, the notion of *Type* is used to distinguish different kinds of values, such as integers, strings, Boolean values, and so on. The declaration of names identifiers and symbols is performed using, "*Type* object, Attribute, Attributes, Msg, Configuration, Value, ObjectId, ClassId and AtributId". For example we declare names of type integer, Boolean and list of natural as follows :
types Nat Boolean List .

2.1.2. Hierarchical and structural relation between the syntactic entities (the objects)

In order-sorted algebraic specification [12], a Type T may have subtype. Intuitively, a subtype declaration is performed as :

$subType\ \ t' < t$

which means that the *subtype* t' is included in the (super) type t. That is, each element of t' is also an element of t. In SPECIF language the following declarations are accepted.

subType ObjectId, ClassId, AttributId < Value .
subType Attribut < Attributs .
subType Object, Msg < Configuration .

2.1.3. How to built operation of a formal module

We use function symbols (or operator symbols) to define the values of each type, and to define the functions on the domains. In the SPECIF language, a declaration of a function symbol has the following form :

op < _: _ > : ObjectId Value → Object .
op _ ; _ : AttributId Value → Attribut .
op _, _ : Attributs Attribut → Attributs [assoc comm id = Null Prior = Value] .
op _ _ : Configuration Configuration → Configuration [assoc comm id = Null] .

Where ObjectId, AttributId, Attributs, Attribut and Configuration are called the arity and the names at the right of each operation is called the co-arity. The symbol operators are the function symbols (e g. ":", ";" ect .). In the SPECIF language as in Maude it is possible to declare that an operation is commutative (comm), associative (assoc), the identity (id) and priority toward a set of operators.

We note that the arity of the symbol function may be empty, in such case the function is called a constant. For example :
Type Boolean .
op true : → Boolean . --- the symbols true and false are considered as a logical constant of type boolean ;
op false : → Boolean .

We note, that in formal methods all the symbol declarations (types and operations) are not pre-declared as in the usual programming languages (e g. C, C++, Java …). The users have to declare all the operations and type names he may need.

2.1.4. Building the rewriting rules in the SPECIF language

Rules in the SPECIF language are similar of those of the Maude language enriched by the adjunction of the Timing characteristic as an optional requirement. A concurrent rewriting rule is presented to the system as follows :

$M_1, \ldots , M_n$<O_1:C_1| $listeAt_1$> ... <O_m :C_m| $listeAt_n$> =>
<O_{i1}:C_{i1}| $listeAtn_1$> ... <O_{ik}:C_{ik}| $listeAtn_{ik}$>
<Q_1:D_1| $liste1At_1$> ... <Q_p : D_p| $liste1At_p$> $M'_1, \ldots , M'_q$ [Timing].

Such description shows a set of n concurrent objects declared in parallel (for example the first object O_1 belonging to the class C_1 with its list of attributes $listeAtn_i$ is declared as <O_1:C_1| $listeAt_1$>). The $M_1, \ldots , M_n$ are n messages which could be used to pass information to the objects and to guide the interaction between the objects. The arrow (=>) in the description, is the link between the left side and the right side of the rule, which means that the left side is transformed to the right side. After the interaction between objects (communication by message passing), a new message is created ($M'_1, \ldots , M'_q$) and perhaps new objects are also created. The symbol [Timing] is optional and it depends on the timing operators used in such case. We note that we have developed some symbolic timed operators as the “Until”, “Next”, “Before”, ”At”, which have been well-defined in [15].

2.2 Modeling Concurrent Systems

The description and the modeling steps are similar to those developed in UML. The method uses two different specification stages which are the following :

2.2.1. Architecture and Interface

In general, embedded systems are modelled in term of objects. An object is a representation of an entity in the real world; it contains information and offers services. A system is therefore composed of interacting objects. As objects belonging to a class of objects and evolve in an environment, they need some particularities to react with each others. In such case, the objects are defined formally as an information specification as developed in [7] and composed of the following elements:

1. A configuration of information objects.
2. The behaviour of those objects.
3. Environment contracts for the objects in the systems.

In such case, information objects are modelled directly by SPECIF objects. The roles that objects play in the information specification are modelled by SPECIF classes, whose members are the

objects exhibiting behaviour compatible with the ones identified by each role. The communities are compositions of information objects, and therefore are modelled by SPECIF configurations, which are multisets of objects and messages.

The general philosophy to represent the information of the whole system (environment) is as follow: The first stage consists of the representation of the different information objects of the system by processing through abstraction levels. It is imperative to respect the natural object hierarchy. At a given level only information objects and messages belonging to a referred level are excited and then highlighted. A complex system is then decomposed into subsystems using a composition relation. This stage is performed through a graphical editor and respects the hierarchical descendant description of the system. At a given abstraction level, the user defines the attributes of the corresponding information object (for example, attribute *state* of the objects Sender and Receiver of a communication protocol specification). The interface of each visible object is also described (its input and output messages and its attributes). At each level, each object is considered as a system itself and therefore it can be decomposed in the same manner as the system father with its referred attributes and messages. Objects without messages and attributes receive the Null value.

2.2.2. The System Behavior

The dynamics of the systems also called behavior, is performed by assigning to each object a set of rules which describe how the different events associated with the messages modify the state of each object. For example, assume that two objects (a sender and a receiver) communicate by sending information (messages). In the following, we demonstrate that SPECIF language authorizes the construction of synchronous and asynchronous communications.

The following example, represents the Bank operations in term of balancing from credit to debit and vice-versa of two accounts Custom1 (C1) and Custom2 (C2). To be more accurate, the information contained in such example could be manipulated using two kinds of rules: synchronous and asynchronous where (transfer Mont From C1 to C2), credit(C1, Mont) and debit(C2, Mont) are the messages. The following cases describe the synchronous and asynchronous rules:

Case1: *Synchronous rule*

```
crl  [rule1]    (transfer Mont From C1 to C2) < C1: Account | solde: S1> < C2: Account | solde: S2>
                    =>
               < C1: Account | solde: S1-Mont> < C2: Account | solde: S2 + Mont>   if   ( S1 – Mont  >= 0 )
```

Case2: *Asynchronous rules*

```
crl   [rule2] credit(C1, Mont) < C1: Account | solde: S1 >  => < C1: Account | solde: S1-Mont> if  S1– Mont >= 0
rl    [rule3] debit(C2, Mont)    < C2: Account | solde: S2>  =>   <C2: Account | solde: S2 + Mont>.
```

In the synchronous rule (case 1), the object C1 (< C1: Account | solde: S1 >) and the object C2 (< C2: Account | solde: S2>) interact concurrently. Influenced by the message (transfer Mont from C1 to C2), they exchange the value Mont and subtract it from solde: S1 and add it to solde: S2. The result of such concurrent rule (< C1: Account | solde: S1-Mont> < C2: Account | solde: S2 + Mont>) is a new configuration where the (transfer Mont From C1 to C2) have been deleted.

We can imagine from the structure of such operations, that for the case of asynchronous rules, there is no interaction between objects, they proceed sequentially. The first object will lose the quantity Mont deduced from solde: S1, and added to solde: S2 of the second object.

2.2.3. Relation between UML/OCL description and the SPECIF language

UML is the standard description language for object-oriented technologies (OOT). In that way, the proposed unification concerns only the notation and not the unification process. The system specification in UML is represented by diagrams (Class, UseCase, Collaboration, Sequence, State, Statechart and Activity diagrams) used to describe Statecharts and automata-graphs. To specify constraints, UML offers the Object Constraint Language (OCL). OCL is well-suited to specify invariants on classes and types. UML has been extended to UML-RT, to enable specification of real-time systems. Let's begin with a simple example as proposed in [7], the use of the UML class diagrams for the description of the structure of the communities. Such example is based on a library at a University. As in [7], we develop just a part of the specification description: all academic staff, postgraduate and undergraduate students. There are prescribed periods of loan and limits on the number of items allowed on loan to a borrower at any given time.
In such example, we can identify three special kinds of borrowers (academic staff, undergrads and postgrads), and two kinds of items (books and periodicals). Such example is represented by the UML class diagram as depicted in the following figure (figure 2). The correspondence between UML model classes and the SPECIF language as shown in [7], using Maude language is straightforward. In this paper, we demonstrate the translation between just some parts of the example as described in figure 2.

For example the class Library is written in SPECIF as follows:

class Library | Address : String, Institution : String , LoanPeriods : Date, MaxLoans : Nat , SuspendedUsers : Boolean .

In our example, the Borrow is modeled by the following class:

Type BorrowerStatus .
ops suspended released : → BorrowerStatus .
class Borrower | address : String, borrowedItems : Nat, fines : Money, status : BorrowerStatus, name : string .

To define the subclass related with the class Borrower, one can declare what follows :

subclasses Academic Undergrat Postgrat < Borrower .

Such three subclasses are automatically defined by inheritance. Before that, we have to declare such subclasses as classes in the following way :

class Academic | dept : String, faculty : String .
class Postgrat | dept : String, tutor : String .
class Undergrat | tutor : String, faculty : String .

The other classes and their subclasses could be defined in a similar way.
The description of the system's behavior could be interpreted by the following dynamic rule. Such operation concerns for example the borrower borrowing of a book; the book, a librarian, the library, and the Calendar. To borrow a book, one needs that the book not be on the loan, the borrower not to be suspended and the number of borrowed books be smaller than the allowed number. Such possible procedure is performed using the following rule as in [7] :

crl [borrow-books] : borrow(O, I, B) < B : Borrower | borrowedItems : N > < I : Book | status : free > < L : Library | mawLoans : ML, loanPeriod : LP, suspendedUsers : False> <O: Librarian | >< C : Calendar | date : Today > => < B : Borrower | borrowedItems : N+ 1 > <I : Book | status : onloan > <L : Library | > <O: Librarian | >< C : Calendar | > <A : Loan | dueDate : Today + LP(class(<B : Borrower | >), Book)], initialDate: Today, borrower : B, item : I> **If** suspendedUsers = False and N < ML[class(<B : Borrower | >)).

We stop now our explanation concerning the transformation of UML diagrams into SPECIF language. This could be engaged us to develop the entire system.

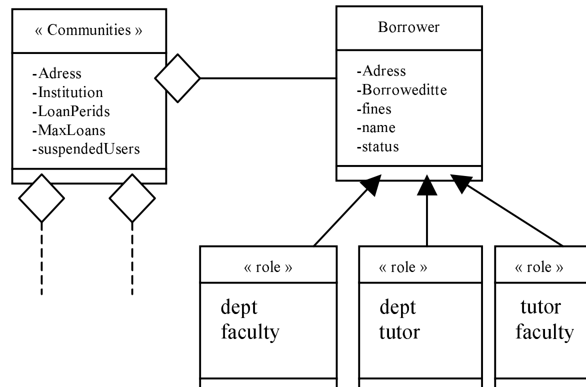

Figure 2 : Structure of a side of the library community.

2.3 Model Checking embedded inside VERIF

2.3.1 Model Checking

Model checking (see figure 3) is a formal analysis technique used to automatically validate behavioural specifications of hardware and software systems. Such specifications are described using a common language called temporal logic or some words generated using Buchi automata [21-24]. There are two mainly approaches to model checking. First, symbolic model checking [22-23], uses binary decision diagrams and their extension to represent specification and implementation as a whole formula in the form of a fixpoint computation [5]. Second, Explicit

state model checking, a technique based on the representation of the problem using Kripke Structure—a state transition graph. The properties are described as temporal logic formulae. The solution is performed using particular algorithms and it depend on the choice of the temporal logic. For example in case of Computational Tree Logic (CTL), Clarke & Emerson technique is a well suited solution for the branching non deterministic problems [5]. The Linear Temporal Logic (LTL) could be applied to find states or traces, which satisfy a requirement specification. There are two types of algorithms to verify LTL formulae, a Buchi automata-based due to Vardi et al. [24]. The other one is a tableau-based also called axiomatic method. Both CTL and LTL formulae properties could be verified using fixpoint computations as in mu-calculus logic.
The operation in critical systems imposes constraints on the system (software and hardware). These constraints gather all the problems involved in the time and the reliability. The problem of the time constraints consists in being able to sufficiently and quickly process so that the resulting action makes sense, while remaining within the limits of operation of the system

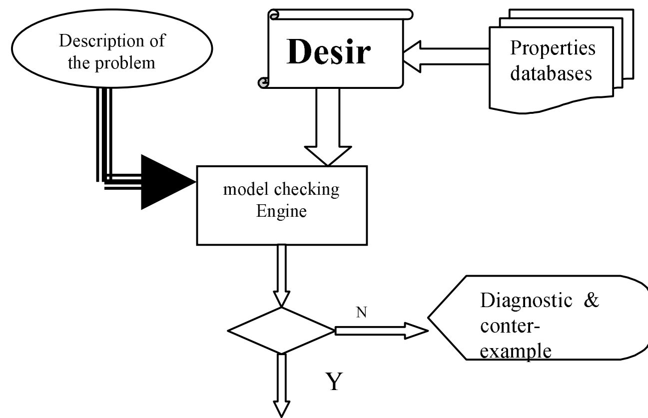

Figure 3 : Model Checker Architecture

The first part of the verification module (VERIF) is built using extended rewriting rules to handle timing requirement. The complete operational semantic rules is presented in [15].
The second part implemented in VERIF module is the axiomatization of the Propositional Temporal Logic, a well-known temporal logic introduced by Pnueli in [14], and well adapted for the verification of reactive systems. Thus, using such logic, one can more easily specify how we want the system to behave without thinking about a particular behaviour. The following code is just a segment of the PTL model-checker:

2.3.2 Syntax of the Propositional Temporal Logic

Linear time prepositional temporal logic (PTL) has been developed by Manna & Pnueli in [2] and discussed in many works [3]. Formulae of PTL are built from a set *AP* of atomic propositions and are closed under the application of Boolean connectives, the unary X (next, written O or @), the binary temporal connective U (until) and two auxiliary operators : the unary operator [] (Always) and another operator <> (eventually). PTL is interpreted over computations, which are executions (Path's). The interpretation function assigns truth values to the elements of

AP at each time instant (natural number). Such computations are viewed as infinites words over the alphabets 2^{AP}. The formal meaning of temporal logic formulae is defined using the notion of Kripke structure.

Definition 1 (Kripke structure) A Kripke structure is a tuple $M = (S, \rho, s^0, \pi)$ where

- S is a countable set of states,
- $\rho \subseteq S \times S$ is a transition relation satisfying $\forall s \in S : \exists s' \in S : (s, s') \in \rho$,
- $\pi : S \rightarrow 2^{AP}$ is an interpretation function on S,
- $s_0 \in S$ is the set of initial states.

Definition 2 (Path). A path in Kripke structure is an infinite sequence of states $(s_0, s_1, s_2, \ldots) \in S^{\omega}$ such that $(s_i, s_{i+1}) \in \rho$ for all $i \geq 0$.

A path is thus an infinite sequence of states such that between successive states transitions do exist. For path $\sigma = s_0 s_1 s_2 \ldots$ and integer $i \geq 0$, $\sigma[i]$ denotes the (i+1)-th state of σ. The set of paths that start in state s is denoted Paths(s). As each state in a Kripke structure is required to have at least one successor, it follows $Paths(s) = \emptyset$ if $s \notin S_0$ for any state s.

The following example gives an overview of the principles elements of the above definitions.

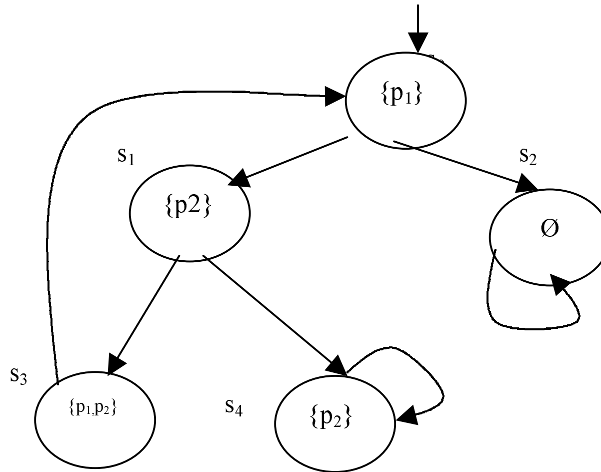

Figure 4 : A simple Kripke structure

In such structure we define the following components :

$AP = \{p_1, p_2\}$; $M = (S, \rho, s_0, \pi)$ such that :
$S = \{s_0, s_1, s_2, s_3, s_4\}$; $\rho = \{(s_0,s_1), (s_0,s_2), (s_1,s_2), (s_1,s_3), (s_1,s_4), (s_2,s_2), (s_3,s_0), (s_4, s_4)\}$
$s^0 = s_0$; $\pi(s_0) = \{p_1\}$; $\pi(s_1) = \{p_1\}$; $\pi(s_2) = \emptyset$; $\pi(s_3) = \{p_1, p_2\}$, $\pi(s_4) = \{p_2\}$.

The two computations in *M* : $(s_0,s_1,s_3,s_0,s_1,s_3, \ldots)$ and $(s_0,s_2,s_2,s_2, \ldots)$ correspond to the infinite sequences :

$(\pi(s_0, \pi(s_1), \pi(s_3), \pi(s_0), \pi(s_1), \pi(s_3), \ldots) = (\{p_1\}, \{p_1\}, \{p_1, p_2\}, \{p_1\}, \{p_1\}, \{p_1, p_2\}, \ldots)$
and
$(\pi(s_0), \pi(s_2), \pi(s_2), \pi(s_2), \ldots) = (\{p_1\}, \emptyset, \emptyset, \emptyset, \ldots.)$

The path (s_0, s_1, s_2) is not an execution because it is finite, and the path $(s_1, s_3, s_0, s_1, s_4, s_4, s_4, \ldots)$ either is not an execution because it is not prefixed by the initial state s_0.
The infinite sequence $(\{p_2\}, \{p_1\}, \{p_2\}, \{p_1\}, \{p_2\}, \{p_1\}, \ldots)$, is not a sequence defined on the language L_M, because such type of execution is not contained in M.

Definition 3. Let *AP* be a set of atomic propositions. The propositional temporal logic over *AP*, denoted PTL(*AP*), is defined as the least set closed under the following rules:

1. $2^{AP} \subseteq$ PTL(*AP*)
2. If $f, g \in$ PTL(*AP*), then $f \vee g \in$ PTL(*AP*).
3. If $f \in$ PTL(*AP*), then $\neg f \in$ PTL(*AP*).
4. If $f \in$ PTL(*AP*), then $[]f \in$ PTL(*AP*).
5. If $f \in$ PTL(*AP*), then $0f \in$ PTL(*AP*).
6. If $f, g \in$ PTL(*AP*), then f U $g \in$ PTL(*AP*).
7. If $f \in$ PTL(*AP*), then $<>f \in$ PTL(*AP*).

Intuitively the following temporal formula states that :

- $[]\, f$, is the proposition that from every time instant, now and in the future the formula f will be true.
- $< >\, f$, is the proposition that the formula f will ever perhaps now but definitely sometime in the future, become true.
- f U g, is the proposition that the formula f will be true at least for all time instant from the present until (but not including) the time when g becomes true, and the latter must eventually happen.
- $@\, f$, is the proposition that f is true in the next time instant. The one immediately following the present.

For a set *AP*, each formula $f \in$ PTL(*AP*) denotes a set $\|f\|$ of runs over *AP*, defined by the following rules,

1. $\|S_0\|$ is the runs $s_0\, s_1\, s_2 \ldots \in AP^{\omega}$ such that $s_0 \in S_0$, for all $S_0 \subseteq AP^{\omega}$.
2. $\|f \vee g\| = \|f\| \cup \|g\|$ ($\cup$ the union of two sets).
3. $\|\neg f\| = AP^{\omega} \setminus \|f\|$ (all the runs except those of $\|f\|$) .
4. $\|[]f\|$ is the set of runs $s_0\, s_1\, s_2 \ldots \in AP^{\omega}$ such that the run $s_i\, s_{i+1} \ldots$ is in $\|f\|$ for all i.
5. $\|\mathrm{O} f\|$ is the set of runs $s_0\, s_1\, s_2 \ldots \in AP^{\omega}$ such that the run $s_1\, s_2 \ldots$ is in $\|f\|$ for all i.
6. $\|f \mathrm{U}\, g\|$ is the set of runs $s_0\, s_1\, s_2 \ldots \in AP^{\omega}$ such that there is an $i \geq \mathrm{O}$ such that $s_i\, s_{i+1} \ldots$ is in $\|g\|$ and $s_j\, s_{j+1} \ldots$ is in $\|f\|$ for all i for all $j < i$.

Other Boolean connective could be defined as follows :

- False = $\neg$True
- f $\wedge$ g = $\neg((\neg f) \vee (\neg g))$ (conjunction)
- f $\rightarrow$ g = $(\neg f) \vee$ g (implication)

and other temporal operators :

- ◊ f = True U g (Eventually)
- □ f = ¬◊¬f (Henceforth)

and others like the release operator (R) and the week until operator (W).

The following module gives the approximation of the PTL syntax written in Maude logic.

```
Formal_Module PROP-TEMPORAL-LOGIC is

    type Formula .
    ops true false : → Formula .
    op _*_ : Formula Formula → Formula  [assoc comm prec 15] .
    op _+_ : Formula Formula → Formula  [assoc comm prec 19] .
    op _++_ : Formula Formula → Formula  [assoc comm prec 24] .
    op !_ : Formula -> Formula [prec 10] .
    op _->_ : Formula Formula → Formula  [prec 21] .
    op _<->_ : Formula Formula → Formula [prec 23] .
    op []_ : Formula → Formula [prec 13] .
    op @_ : Formula → Formula [prec 11] .
    op <>_ : Formula → Formula [prec 12] .
    op _Uw_ : Formula Formula → Formula [prec 15] .
    op _U_ : Formula Formula → Formula [prec 14] .
    vars X Y Z V : Formula .
    eq !(true) = false .
    eq !(false) = true .
    eq <> false = false .
    eq <> true = true .
    eq <> <> X = <> X .
    eq <> [](<> X) = [](<> X) .
    eq [] @ X = @([] X) .
    eq [] (X Uw Y) = [] (X + Y) .
    eq @(X * Y) = @ X * @ Y .
    eq @(X + Y) = @ X + @ Y .
    eq X U (Y + Z) = (X U Y) + (X U Z) .
    eq false U X = X .
    eq X U false = false .
    eq true U X = <> X .
          |
          |
  endformalModule
```

As it was introduced in the last presentation, the semantics of the PTL logic could be defined using a satisfaction relation noted M, s |= *f*, where M is a Kripke structure, s an initial state of M and *f* a formula of PTL(*AP*). The meaning of such relation is to give a satisfaction function

between the states of a path (execution) and the formula *f*. In other words it is to say that the formula *f* holds for each state of a path with s its initial state.
Such satisfaction relation is declared in Maude using the following module :

```
Formal_Module SATISFACTION is
Protecting PROP-TEMPORAL-LOGIC .
Type State < States .
Op _|=_ : State Formula → State .
endformalModule
```

Using such code program (Model-checker), we have verified some well-known benchmarks as depicted in the following table.

Table 1. Experimental Results of the PTL model-checker

Model name	Time PTL-tool	rewrites Number	Time (ms)
Muller_C	0.1 s	70	40
Lars1	3.0 s	124	0
adder	0.1 s	195	0
Mul4	158.7s	216	10
Dff1	0.0	53	20
Josko3	105.6 s	21	10
Josko4	18.0 s	20	10
Mutex1	1.3 s	91	30
Mutex2	19.1 s	93	20
Peterson	-	103	160
Scc-event	-	23	10
Josko5	-	20	10
Muller-C2	-	20	10
Mutex	-	65	10
CallHear	-	34	10
Hailpern	-	70	20

3. Simulation and Animation Module

Simulation technique is nothing more than the imitation of reality. It proceeds by building a discrete model of reality. A *random generator* is used to take samples from, for example, order arrivals, processing times, batch sizes, etc., using a user-defined distribution. Well known *distributions* are: negative exponential, normal, Erlang and uniform laws.
A *package* is usually designed and a concept such as delay is translated into processing. Packages can make an *animation* of the simulated process and sees if the model is running the way it is supposed to, and reporting execution results. The correctness of the model and the correctness of the results are checked using some statistical tests (e g. χ^2 test).
When simulating a distributed system, visualization of the system behavior seems necessary and become more critical for larger specifications. The system of a specification is given using the notion of traces or runs which can be taken as the composition of concurrent actions labeling transition system and taking into account the states and the transitions of a run. The behavior of a system is the set of all the runs also said paths. For example figure 4 represents the well-known Alternating Bit Protocol (ABP), the animation protocol of the simulation and the generated traces.

3.1. The Simjava Packages

Java is an object-oriented programming language for writing portable applications which can be embedded in html. Java tool offers some advantages as for example, security, simplicity and reliability and the use of libraries called packages. The Java's most interesting feature is its support for concurrency through threads which are small parts of the java program that can be executed in parallel and uses timed parameter. Such features are very useful to develop simulation software. All such characteristics have been embedded in the core of Simjava which is a discrete event simulation package [11], written by Ross McNab and Fred Howell in Java Language and based on Hase++ library developed in C++. Conceptually Simjava is based on the three packages: eduni.SimJava, eduni.SimAnim and eduni.SimDiag.

3.2. Illustration by Example

In the following, we illustrate the graphical system using a rapid description of the ABP protocol (Alternating Bit Protocol) [10]. The ABP protocol is composed of a sender, a receiver, a data channel, a medium link between the sender and the receiver, and the acknowledge channel linking the receiver with the sender (receiver to the sender). A producer generating messages (data-frames), is simulated using a random numbers generator, and a consumer which is the last element of the system (figure 6).

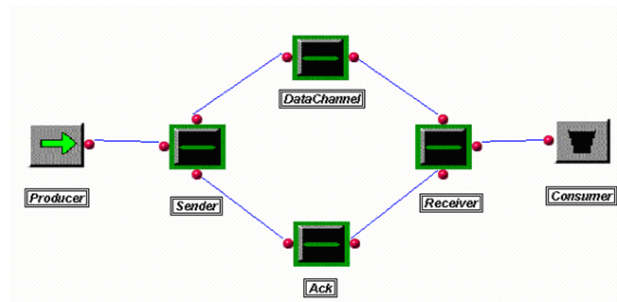

Figure 5 : Graphical description of the ABP protocol.

We will give more detail of the behavior and the functionality of the ABP protocol. As we can see in figure 5 graphic, the protocol is a distributed system composed of the following objects: the producer at the left side, the sender object, in the center side both the data channel (top) and the acknowledge channel (bottom) and finally, the receiver and the consumer at the right side.

In this work we will not explain how we have created such graphic, we give just a part of the generated Java-code, for example, we will detail the sender object. First of all we have to declare the following libraries of the Java programming language and some extension libraries of the SimJava.

```
import java.awt.*;
import java.awt.event.*;
import eduni.simjava.*;
import eduni.simanim.*;
import java.applet.*;
```

```
//****The Sender code is:
public class  Sender  extends Sim_entity  {
//*** The sender has three communication ports—1 input and 2 outputs
   private Sim_port  Port1;
   private Sim_port  Port3;
   private Sim_port  Port2;
   static int state=0;
public Sender ( String name,int x,int y )
{    super ( name,"Inter",x,y );
//*** Declaration  of the ports position in the sender box
Port1 =new Sim_port("Port1","portRed",Anim_port.LEFT,10);
                  |
                  |
                  |
 add_port(Port1);
Port3 =new Sim_port("Port3","portRed",Anim_port.BOTTOM,10);
 add_port(Port3);
  {   Protocol_ABP.tab[0]=0;
       Protocol_ABP.jobField.insertText("Envoie encore le "+Protocol_ABP.i +" message\n",1);
 sim_schedule(Port2, 0.0, 1,Protocol_ABP.tab);
     if(Protocol_ABP.show_msg) sim_trace(1, "S Port2 M"+Protocol_ABP.i );
}    else{   Protocol_ABP.jobField.insertText("Bien\n",1);
       state=0;
}}}}}
```

The other objects code are generated in a similar way, except for the producer object code which contains a random generation function that could be chosen by the user. For example in the code generated using the ABP protocol (figure 5), such declaration is in the following form :

```
public class  Producer  extends Sim_entity  {
private Sim_port  Port1;
private Sim_uniform_obj  delay;
private int max;
public Producer ( String name, int max, int x, int y )
{
   super ( name, "Emetteur", x, y );
   Port1 =new Sim_port("Port1","portRed",Anim_port.RIGHT,10);
   add_port(Port1);
   delay = new Sim_uniform_obj("",0.2,10,(int)System.currentTimeMillis() )};
```

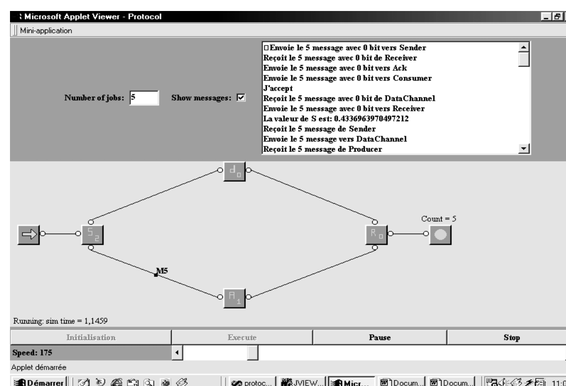

Figure 6 : Simulation and Animation of ABP protocol run.

4. Conclusion

We have presented a system based on a specification module called SPECIF, a simulation and a verification module called VERIF. The VERIF system uses a functional description of temporal rewriting logic and the PTL logic embedded in the system using an implementation based on the axiomatization of PTL. The three modules are monitored by an internal program to link the communication between them and to translate the graphical description into Java programs and into rewriting modules. The results of the rewriting logic module are executed using Maude system. As experimental results we have used VERIF module to verify some well-known benchmarks as presented in Table 1. We have also shown how the translator program could proceed to translate a graphic form as UML models to SPECIF form and to Java code used to supply the simulation program to get the dynamic behavior by animation. We think that, the idea of using formal methods to specify and to verify distributed software in an object programming framework seems easy and could improve the understanding of many problems due to the complexity in space and in time. We are now sure, from our experience in the development of VALID-2, that the system could take more advantages from its open front-end to incorporate existing specification and verification tools and new description as for example the notion of time in RT-UML and temporal constraints. All such characteristics are easily monitored by the rewriting logic of the Maude language. Until now, SimJava simulation packages seem well adapted for sequential deterministic processes and the notion of multithreading are well consumed, plus the timed primitives can solve many problems. Finally, we can say that VALID-2 has been lastly improved by incorporating a module ROBDD (reduced ordered binary decision diagram) used for the symbolic simulation of hardware systems.

5. References

[1] A. Attoui, M. Schneider, formal Approach to the Specification and the Behavior Validation of Real-Time Systems Based on Rewriting Logic, *Kluwer Academic Publischers*. Vol. 10, pp. 5-22, 1996.

[2] G. Berry, G. Gonthier,, The ESTEREL, Synchronous Programming Language: Design, Semantics, Implementation, *Science of Computer Programming,* vol. 19, no 2, pp. 87-152, 1992.

[3] T. Bolognesi et al, Introduction to the ISO Specification Language LOTOS, In P. H. J. Van Eijk, C. Vissers and M. Diaz. Editors, the Formal Description Technique Lotos, North-Holland, 1988.

[4] G. Booch, J. Rumbaugh ,and I. Jacobson, Unified Modelling Language, User Guide, *Addison-Wesley*,1999.

[5] E.M Clarke, E. A. Emerson, Design and Synthesize of Synchronisized Squeletons using Branching Time Temporal Logic, In Proc. *Workshop on Logic of Programs*, LNCS 131, Springer-Verlag, pp. 52-71, 1981.

[6] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and J. F. Quesada, A Maude Tutorial, presented at the European Joint Conference on Theory and Practice of Software, *ETAPS2000*, Berlin, Germany, March 25, 2000. http://maude.csl.sri.com/tutorial.

[7] F. Duran and A. Vallecillo, Writing ODB Information specification in Maude, Available at http://www.lcc.uma.es/ãv/Publicaciones/01/ITI-2001-10-pdf.

[8] M. Kaufmann, and J.S Moore, ACL2: an Industrial Strenght of Nqthm, In Proc. Conference of Computer Assurance (COMPASS-96), pp. 23-34, IEEE, computer Society Press, 1996.

[9] P. Lescanne, Elementary Interpretation in Proofs of Termination, Technical Report, Centre de Recherche en Informatique de Nancy (CNRS), INRIA-Lorraine, France, 1994.

[10] K.L. McMillan, Symbolic Model Checking: An Approach to the State Explosion Problem, Phd thesis, Carnegie-Mellon University, 1992.

[11] R, McNab, A guide to the Simjava Package, Department of Computer Science, University of Edinburgh, UK, 1996.

[12] J. Meseguer, A Rewriting Logic as a Unified Model of Concurrency, Proc. of the Concur90 Conference, Amsterdam, August 90, pp. 384-400, LNCS 458-1990.

[13] R., Milner, A Calculus of Communication Systems, *Springer* LNCS 92, 1980.

[14] A. Pnueli, The temporal logic of programs, In Proc. 18th Annual Symp. Foundations of Computer Science, pages 46–57, IEEE, 31 October–2 November 1977.

[15] M.L Rebaiaia, M. Benmohamed, J. M. Jaam, A. M. Hasnah, A Toolset for the Specification and Verification of Embedded Systems, *Proc. of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03*), Las Vegas, Nevada, USA, June pp. 23-26, 2003.

[16] The Toolkit Uppaal. BRICS, Aalborg University, DENMARK and Uppsula University, SWEDEN, 1995.

[17] J.B., Warmer, A.G., Kleppe, The Object Constraint Language, *Addison Wesley*,1999.

[18] http://www.dcs.ed.ac.uk/home/cws. 2003.

[19] http://www.inria.fr/vasy/tutorial/ 8 Nov 2003.
.

[20] G. J. Holzmann, The Model Checker Spin, *IEEE transaction on Software Engineering,* vol. 23(5), pp. 279-295, 1997.

[21] K. Etassami, G. J. Holzmann, Optimizing Buchi automata, LNCS,1877, pp. 153-167.

[22] R.E.Bryant, Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams, Carnegie Mellon, Technical Report CMU-CS-92-160, 1992.

[23] K. L. McMillan,. Symbolic Model Checking. Kluver Academic Press, 1993.

[24] P. Gerth, D. Peled, M.Y. Vardi, P. Wolper, Simple on the-fly-automatic Verification of Linear Temporal Logic . In Protocol Specification Testing and Verification, page 3-18, 1995.