# Verifying Linear Time Temporal Logic Properties of Concurrent Ada Programs with Quasar

Evangelista S.
evangeli@cnam.fr

Kaiser C.
kaiser@cnam.fr

Pradat-Peyre J.F.
peyre@cnam.fr

Rousseau P.
rousseau@cnam.fr

CEDRIC - CNAM Paris
292, rue St Martin, 75003 Paris

## ABSTRACT

In this paper we present an original and useful way for specifying and verifying temporal properties of concurrent programs with our tool named QUASAR. QUASAR is based on ASIS and uses formal methods (model checking). Properties that can be checked are either general, like deadlock or fairness, or more context specific, referring to tasks states or to value of variables; properties are then expressed in temporal logic. In order to simplify the expression of these properties, we define some templates that can be instantiated with specific items of the programs. We demonstrate the usefulness of these templates by verifying subtle variations of the Peterson algorithm. Thus, although QUASAR uses up-to-date formal methods it remains accessible to a large class of practitioners.

## Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking, Formal methods*

## General Terms

Verification, Reliability

## Keywords

Software verification, Concurrency, Temporal logic, Petri nets

## 1. INTRODUCTION

In many applications, concurrency provides programmer with the ability to design and to organize its applications in an elegant and efficient way. Indeed, the possibility to define concurrent activities that can collaborate according to predefined synchronization mechanisms gives a lot of flexibility for mimicking the structure of the application domain that contains natural parallelism and cooperation. The application is often more simple, it gains in scalability, it is less error-prone and it is easier to prove its correctness than without the use of concurrency.

However, while providing many advantages, concurrency introduces specific difficulties due to the non determinism and to the many possible interactions between activities. One can quote the deadlock or the fairness problem.

Classical test methods are not sufficient to detect this kind of problems, in particular because of the difficulty to reproduce an error when it has occurred. So specific technologies have to be employed to enforce confidence in concurrent software.

We have developed a tool, named QUASAR [3], that addresses this problem. It is based on formal methods (model checking) but it remains accessible to a large class of practitioners because the formal part of its work is automated: the concurrent program (up to now a Ada program) is automatically translated into a formal model (a colored Petri net) and is analyzed through this formal model. Report of the analysis is made by giving references to the original program. Note that from a theoretical point of view, checking program properties is undecidable. So we do not always obtain a response to a query. But in most practical cases, it works fine.

If general properties, like deadlock or fairness, can be universally and simply defined, specific properties must be expressed with formal specification languages like temporal logics [19]. These formalisms are theoretically well suited for expressing this kind of properties but are often very difficult to use in concrete cases. Indeed, expressing temporal logic properties needs a particular cast of mind and a lot of experience in this theoretical domain.

In order to simplify the expression of these properties we propose in QUASAR a specific interface. This interface focuses on the most useful kind of properties and proposes four distinct templates that can be easily instantiated by designing parts of the code (by naming tasks, variables or entries or by selecting graphically part of the code). We present here these templates and we show on an example how they can be efficiently used to track down errors in concurrent programs.

The paper is organized as follow; we first present the use of ASIS in Quasar. Then we recall briefly the LTL definition and we present our property templates. We show through an example their usefulness and we conclude after reporting related works.

## 2. ASIS - GETTING INFORMATION FROM SOURCE CODE

When analyzing a program, Quasar proceeds in four steps :

- first, it slices the program by suppressing all the non relevant parts with respect to the studied property;
- second, it translates it into a formal model (a colored Petri net);
- third, structural or model checking techniques are applied on the formal model in order to determine if the checked property is verified or not;
- at last, it makes a report referring to the original program and the checked property.

Three of these actions (slicing, translation and report) manipulate directly the source code and it may be a very difficult task, in particular for high level language such as Ada.

Fortunately, the use of the ASIS library [12] (Ada Semantic Interface Specification) gives us an elegant and efficient way to perform these tasks using ASIS iterators and ASIS queries.

### 2.1 ASIS general use

ASIS is an Interface which provides means to navigate through the syntax tree of any Ada program. Each node of this tree is an *ASIS Element* defined by its *Kind* (declaration, statement, expression, ...).

The kinds of an element are defined hierarchically : for example, the kind of the element : `type M5 is mod 5` is at the same time a declaration, an ordinary type declaration and a modular type declaration. This hierarchy enables different levels of analysis. For example, one can choose to apply a treatment on all type declarations or only on modular type declarations.

ASIS provides two powerful mechanisms for navigating through the syntax tree. The first one is an iterator which allows to traverse the syntax tree by using depth-first method. During the traversal of each element, it permits to apply a pre and a post procedure. The pre procedure is applied when reaching the node, the post procedure is applied when the sub-tree of the node has been traversed. Figure 1 shows an example of syntax tree traversal and pre and post procedure executions order. The number near each arrow is the call order number, for instance, the arrow 1 correspond to the first call to a pre or a post procedure. As it is a black arrow, it is a pre procedure, as it points to the complete "while loop", it is the pre procedure performed before the node `while ... end loop`. Arrow 21 is the post procedure called at exit of the assignment node `I := I + 1;`.

The second method consists in using a set of queries. These queries give information about a node of the syntax tree (i.e. an ASIS element). But more than simple queries, these functions give the possibility to navigate through the syntax tree by the semantic correspondence between elements without using an iterator. For instance, in Figure 2, we see how to decompose the assignment `I := Inc(I);`. First, we have an ASIS query to get the declaration of the identifier (`Corresponding_Name_Declaration`), then another one to get the declaration of the called function (`Corresponding_Called_Function`), and if we search information about the code of this function, we can easily get its body with a third ASIS query (`Corresponding_Body`).

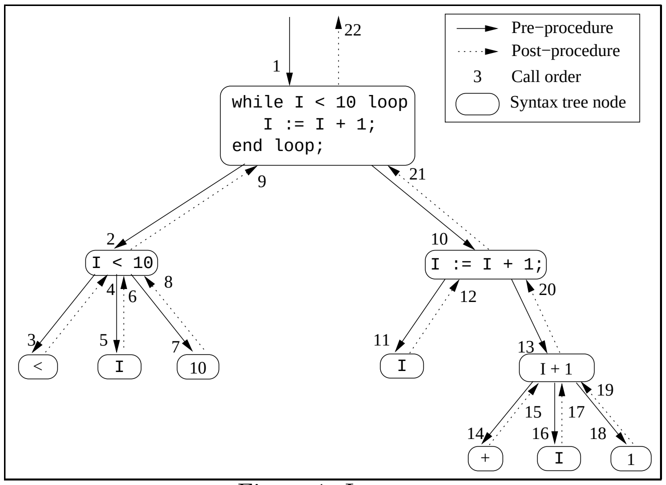
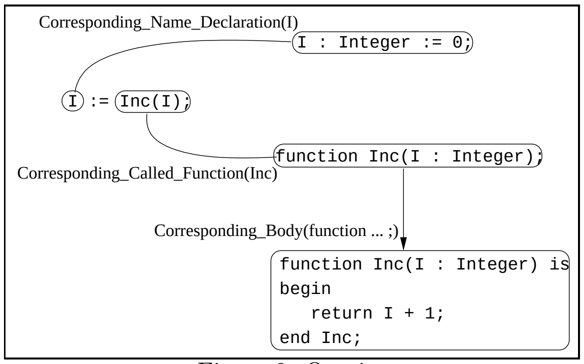


Figure 1: Iterator


Corresponding_Name_Declaration(I)
I : Integer := 0;
I := Inc(I);
function Inc(I : Integer);
Corresponding_Called_Function(Inc)
Corresponding_Body(function ... ;)
function Inc(I : Integer) is
begin
return I + 1;
end Inc;
Figure 2: Queries

With these tools, ASIS allows to get information on a source code in a much more simple and efficient way than with classical methods (grammar definition, syntax tree generation and parsing, ...).

### 2.2 Quasar specific use

The first two steps of the analysis requires information from the source code. According to the property to be checked, the slicing step decides which parts of the source code have to be kept. In the second step, each collected element requests some particular information to be translated into a colored Petri net. For instance, in order to translate a procedure declaration we need its name, its parameters and their types, and, the Petri net patterns.

ASIS allows us to collect all these information, but as a general library, it is not specific enough for our use. For instance, we have to collect all declaration by kind (tasks, variables, ...). In an other hand, using ASIS implies some complex sequences of queries which are reused several times. Some functions have to be specialized to simplify the code translation process.

All these requirements are met by writing a specific library based on ASIS iterator and ASIS queries. We use the iterators to collect lists of declarations (ordinary types declaration list, task declaration list, protected object declaration list, variable declaration list, ...). Queries are used and specialized to be more specific to our needs. For example, when a "for loop" statement is analyzed we need to know the bounds of the loop control parameter. Therefore, we made a simple function which uses the appropriate sequence of queries according to the way the range has been written (`for I in 1..4 loop / for I in T'Range loop / for I in ...`).

As we said earlier, we only translate the relevant part of the program with respect to the studied property (deadlock or user defined property). To do this, we have to be able to decide which parts of the program have to be kept. For instance, in the Peterson example (given is section 3.3), we first look for concurrent statements and possibly blocking actions (for example, infinite loops or statements defined by the user). In the tasks (`T_One` and `T_Two`) and in the main program, we go through non protected calls to determine if they use statements related to concurrency. In `Peterson.Enter`, we can't determine if the "while loop" will finish or not: if the loop condition remains true, tasks will never go out the loop statement. So we have to keep this loop. Keeping this statement implies to keep variables used in the loop condition: `Candidate`, `Priority` and the parameter `X`. As we keep local variables we have to keep theirs declaration and as we keep a parameter, we have to keep all variables that are used in a call to `Peterson.Enter`. This leads to keep local variable `My_Id` of tasks `T_One` and `T_Two` (and then to keep the declarations of these local variables). We pursue this process in `Peterson.Quit` to determine all the elements that we have to keep.

The user can also decide to keep some specific declarations or statements (when it considers that they are involved in possible blocking situation). Also, we must keep all parts of the code that are related to these declarations or statements. This code is retrieved using ASIS queries (`Asis.Expressions.References`).

## 2.3 Translation step

High-level Petri nets with inhibitor arcs [13, 6] have the same expressiveness as the Ada language. So, the translation of an Ada program to a high-level Petri net doesn't raise any theoretical problem. In fact, translating an Ada program to a high-level Petri net is similar to a compilation process. In both cases, the aim is to map a program semantic from a particular formalism to another one. The real difficulty is to produce Petri nets which can be easily analyzed. In particular, we have to tackle the combinatory explosion which is an inherent problem to concurrent programs analysis. Therefore, in some very particular cases, we choose to replace the exact Ada behavior by an abstract one (we perform a "weak simulation" of the program). For instance, the elaboration part of a sub-program or of a task is considered to be atomic.

The translation mechanism proceeds in two steps:

1. Each Ada construction (statement, expression, declaration) of the program is translated into a predefined generic pattern. Thus the translation of the whole Ada program produces a set of small Petri nets components.

2. The produced components are elaborated with two simple operations: the merging operation, which combines two Petri nets by merging places having the same name, and the substitution operation, which replaces an abstract transition by a given sub net. By this way, we obtain a standard colored Petri net that can be directly analyzed with usual structural and model checking techniques.

This modular approach has numerous advantages. In particular, it allows us (1) to easily prove the correctness of the translated Petri net, and (2) to have a relatively language independent translation process : as many high-level languages have the same basic features (if-then-else statements, loop statements, variable assignment, ...), a great part of the work is generic. It is thus re-usable for doing the same work for another concurrent programming language such as Java.

As an example, let us examine how QUASAR proceeds when constructing the whole Petri net given figures 3 and 4 that corresponds to the following simple concurrent program:

```
task body Client is              task body Server is
begin                            begin
   loop                             loop
      Server.Service;                  accept Service;
   end loop;                        end loop;
end Customer;                    end Server;
```

When translating a loop statement, the enclosing statement is ignored. It is considered as an abstract transition. As it is translated the abstract transition of the `loop ... end loop;` net is replaced by the translated one. This is a substitution operation.

By applying successively two substitutions ($\prec$ operator) we obtain two independent Petri nets that we merge ($\infty$ operator) around the `E.RETURN` and `E.CALL` places to obtain the Petri net corresponding to the two loop statements.

This example illustrates some features of the translation process :

- To each task of the program is attributed an identifier which is used for instance during an entry call. For performance issues, theses identifiers are given statically (as initial tokens in places).

- Naming places allows to merge dependant parts of the program (e.g. an entry call and an accept statement on the same entry).

- Compound elements of programs (e.g. task bodies or loop statements) are composed of abstract transitions which are replaced with the substitution operation by the correct sub net.

# 3. LTL PROPERTIES IN QUASAR

Linear time Temporal Logic [19] (LTL for short) is recognized as being a powerful tool for expressing properties of concurrent systems. Its minimal syntax combined with its great expressiveness allows designers to define and to prove a large set of temporal related properties.

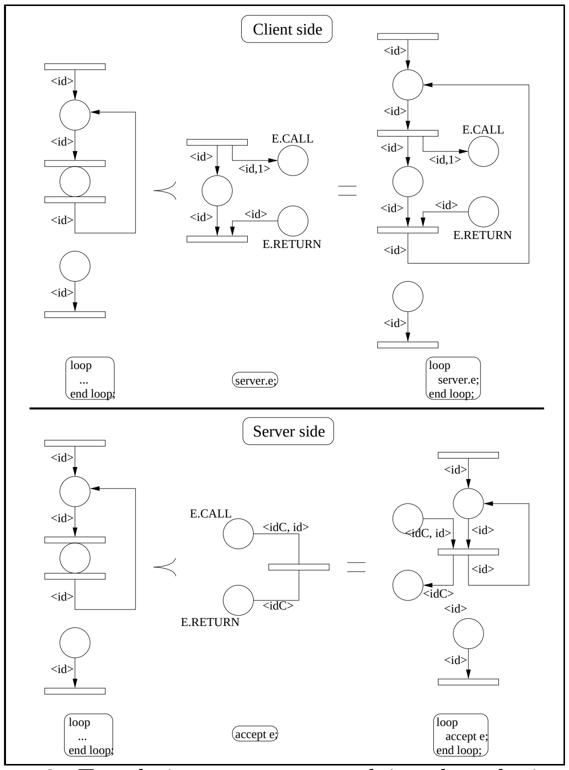


Figure 3: Translation patterns : applying the substitution operator

## 3.1 Some useful formal definitions

If $f$ is a LTL formula then $\neg f$ (not $f$) is also a formula. If $f_1$ and $f_2$ are two formulae, then $f_1 \bigwedge f_2$ ($f_1$ AND $f_2$), $f_1 \bigvee f_2$ ($f_1$ OR $f_2$), and $f_1 \implies f_2$ ($f_1$ implies $f_2$) are also formulae. If we define $AP$ as a set of *atomic propositions* then any $p \in AP$ is a LTL formula. Furthermore, a LTL formula can be defined with temporal operators using the following syntax:

$$\begin{array}{lll} f & ::= f_1 U f_2 & (\textbf{Until operator}) \\ & ::= G f_1 & (\textbf{Always operator}) \\ & ::= F f_1 & (\textbf{Finally operator}) \\ & ::= X f_1 & (\textbf{Next operator}) \end{array}$$

Let us explain the semantic of these four operators [1]. Let $\sigma$ be an infinite execution of the system. We note $\sigma(i)$ the execution beginning at the ith state of $\sigma$ (thus $\sigma = \sigma(0)$) and $\sigma_i$ the ith state of $\sigma$ (thus $\sigma = \sigma_0.\sigma_1 \ldots \sigma_i \ldots$). The infinite sequence $\sigma$ satisfies :

- *an atomic formula* $p$ iff $p$ is satisfied at $\sigma_0$.

  We consider here that an atomic proposition is expressed as a condition on markings of Petri nets (atomic state properties) [2].

- *the formula* $f_1 U f_2$ iff there exists an index $i$ such that $f_2$ holds for $\sigma(i)$ and $f_1$ holds at $\sigma(j)$ for each $j$ in $[0, i-1]$. Intuitively, "$f_1$ is true until $f_2$ becomes true" means that $f_2$ must necessarily be satisfied in the future and that $f_1$ holds as long as $f_2$ does not.

- *the formula* $G f_1$ iff for each $i$, $\sigma(i)$ satisfies $f_1$. Intuitively, "always $f_1$" means that the property $f_1$ remains always true during the execution.

- *the formula* $F f_1$ iff there exists $i$ such that $\sigma(i)$ satisfies $f_1$. Intuitively, "eventually $f_1$" means that a state $\sigma_i$ verifying $f_1$ will be necessarily reached in the future.

- *the formula* $X f_1$ iff $\sigma(1)$ satisfies $f_1$. Intuitively, "next $f_1$" means that $f_1$ will be verified in the next state of $\sigma$.

[1]One can notice that the operators $F$ and $G$ are redundant with the operator $U$ but they are often given in order to simplify property expression.

[2]It is also possible to define an action based logic by expressing atomic propositions as conditions on transitions occurrences.

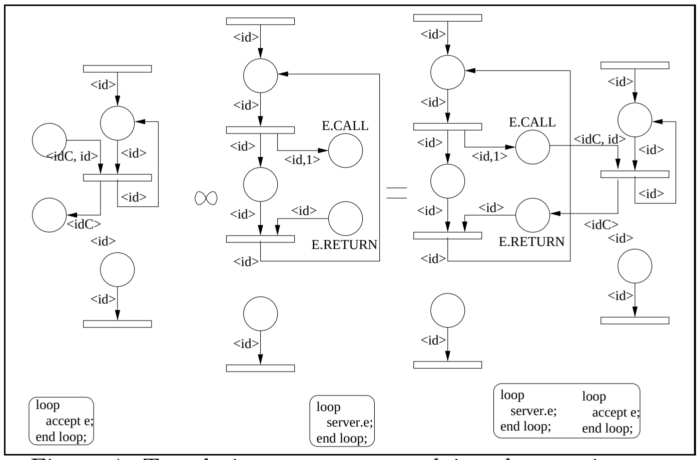


Figure 4: Translation patterns : applying the merging operator

Finally a LTL property is satisfied by the system if all infinite executions of the system that begin at the initial state satisfy the property (when a sequence is not infinite we consider that it loops on the last state of the sequence).

Checking a property $\phi$ can be done in three steps (remember that from a theoretical point of view the problem to test if a Petri net satisfies a state base LTL property is undecidable):

1. Construct a Büchi automaton corresponding to the negation of $\phi$[3].

2. Synchronize the automaton with the reachability graph (that defines all possible reachable states) of the Petri net model in order to obtain another Büchi automaton.

3. Check if there is an infinite execution accepted by the synchronized product (note that this check may not end due to the possible infinite size of the reachability graph).

If such an execution exists then it means that at least one execution verifies the negation of the property, and thus the property is not verified. If no execution is accepted then the property holds.

This process can also be done "on-the-fly" during the reachability graph generation. This has the advantage to

[3]A Büchi automaton is an automaton that recognizes infinite words. The rule for accepting a word is not "end in an accepting state" but "traverse infinitely often accepting states"

not generate the whole reachability graph if the property doesn't hold. More details on this process can be found in [21].

## 3.2 Specific properties definition

LTL is a very powerful formalism for defining properties on concurrency. However, it does not always provide the most intuitive manner to describe these properties.

We defined in Quasar a simple and graphical interface allowing the definition of the most usual properties. These properties refer to atomic propositions involving states of tasks, values of variables (when defined) and conditions on entry queues (number of waiting tasks or presence of a particular task in the queue). Once again, the link between the code and the formal model (here the LTL formulae) is made with the help of the ASIS library.

Tasks, variables and entries are defined by their name. States of tasks are defined by selecting lines in the code or using states of an automata that is automatically produced at the first stage of the translation process and that displays the state evolution of a task.

For defining temporal properties, we propose four templates that correspond to the most usual properties (an expert mode is also available in which any LTL property can be defined). These templates are instantiated with specific task state, variable or entry queue using the graphical interface. We will see in the next section an example of the use of these templates. These templates describe :

- **State accessibility**

  This template refers to the accessibility of a state $s_1$ as soon as a state $s_0$ is reached. In many cases, $s_0$ corresponds to a state in which initialization has been performed (or the beginning of a sequence that must lead to $s_1$). Both of states are described using task states, variables, and entry queues and the state $s_0$ may also be defined as the initial state. This template is decomposed into three different sub-templates (in these formulae we use $s_0$ and $s_1$ to denote atomic propositions defining corresponding states) :

  - **Inevitable state** : $s_1$ is inevitable from $s_0$ if it is necessarily reached in each execution as soon as $s_0$ has been reached. This property corresponds to a formula $\neg s_0 U(s_0 \Rightarrow F s_1)$.
  - **Inevitable state with condition** : in this case, we verify also that the state $s_1$ is necessarily reached from $s_0$ but we impose that from $s_0$ to $s_1$ an atomic proposition $Cond$ remains true (the proposition $Cond$ is also defined using task state, value of variable or length of entry queues). This templates corresponds to a formula $\neg s_0 U(s_0 \Rightarrow (Cond\, U\, F s_1))$
  - **Home state** : $s_1$ is a home state as soon as $s_0$ has been reached if it remains always reachable. This template corresponds to a formula $\neg s_0 U(s_0 \Rightarrow G(F s_1))$

- **Bounded wait**

  Each time a task is in a state $s_1$ (for instance waiting for some resources) then, necessarily, it will access in a finite future a state $s_2$ (for instance a state in which it gets its needed resources). It corresponds to a formula $G(s_1 \Rightarrow F s_2)$

- **Critical section**

  A non atomic sequence of statements is a critical section if it can't be executed by several tasks at once. It corresponds to a formula $G(\neg s)$ where $s$ denotes the subset of tasks states in which they perform a statement that is in the critical section.

- **Stable property**

  Once a property holds it holds forever; it corresponds to a formula $\neg f U G(f)$ where $f$ is the studied property.

REMARK 1. *Note that LTL formulae describe the behavior of* **all** *possible executions. For expressing potential execution one must use an other temporal logic (for instance CTL).*

Once a property is specified with one of this template, it is automatically translated into the corresponding LTL formula and checked with a model-checker (for the moment we use Prod [22] or Maria [15]). By default no assumption is made on the scheduler. In particular, the scheduler is not necessarily supposed to be fair (it can decide to always choose the same task). However, it is possible to impose a verification under weak or strong fairness hypothesis. The weak fairness hypothesis will exclude all executions in which an action is continuously possible without being ever performed (for instance a task remains always in the "ready" state and is never chosen by the scheduler). The strong fairness hypothesis will exclude all executions in which an action is infinitely often possible and never performed. This case is more subtle, and corresponds, for instance, to a task waiting for a semaphore that is infinitely often taken and released although the waiting task will never get it. These fairness assumptions can be imposed to all tasks or to specific ones. It is important to note that these hypothesis make the verification of the property more difficult.

## 3.3 Example

In order to illustrate the use of these property patterns, let us consider the program given below. This program defines two tasks, named `T_One` and `T_Two`, that access a controller through a procedure named `Set_Controller_Instruction`. In order to avoid hardware malfunction, operations on the controller must be done in critical section (a sequence of actions on the controller that has been started by a task must be ended before a new sequence begins). We also suppose that for obscure reasons, mutual exclusion is not programmed with a protected object (nor with a server task) but with busy waiting loops similarly to variations of the Peterson algorithm for two tasks. This code corresponds to the package `Peterson`.

Three different versions of this algorithm are proposed by the development team. Only one is correct. These three versions have a common part presented here (they only differ in the code of the procedure `Enter`).

```
with Text_IO; use Text_IO;
with Peterson; use Peterson;

procedure Prog1 is

   procedure Set_Controller_Instruction (X : in Id) is
   begin
      -- a complicate code that reads and writes registers of a
      -- controller in order to give it some instructions; all
      -- these manipulations must be done in critical section;
      -- we do not detail here these statements
      Put_Line ("Actions_on_controller_for_" & Id'Image (X));
      delay (1.0);
   end Set_Controller_Instruction;

   task T_One;
   task T_Two;

   task body T_One is
      My_Id : Id := 1;
   begin
      loop
         Put_Line ("Before_actions,_task_" & Id'Image (My_Id));
         Peterson.Enter (My_Id);
          Set_Controller_Instruction (My_Id);
         Peterson.Quit (My_Id);
         Put_Line ("After_actions_section,_task_" &
                   Id'Image (My_Id));
      end loop;
   end T_One;

   task body T_Two is
      My_Id : Id := 2;
   begin
      loop
         Put_Line ("Before_actions,_task_" & Id'Image (My_Id));
         Peterson.Enter (My_Id);
          Set_Controller_Instruction (My_Id);
         Peterson.Quit (My_Id);
         Put_Line ("After_actions,_task_" & Id'Image (My_Id));
      end loop;
   end T_Two;

begin
   null;
end Prog1;
```

```
package Peterson is

   type Id is range 1 .. 2;

   procedure Enter (X : in Id);
   procedure Quit (X : in Id);

end Peterson;
```

```
package body Peterson is

   type Tab_Candidate is array (Id) of Boolean;

   Priority   : Id := 1;
   Candidate : Tab_Candidate := (others => False);

   procedure Enter (X : in Id) is
      -- depends on version
   end Enter;

   procedure Quit (X : in Id) is
   begin
      Candidate (X) := False;
   end Quit;

end Peterson;
```

The code of the procedure `Enter` is either one of these three versions :

### 3.3.1 *First version*

In the first version, a task `X` runs for election by updating the array `Candidate`; it gives the priority to the other task in case of conflict (assignment `Priority := Other`) and it waits as long as it is not true that it is candidate and it is its turn.

```
procedure Enter (X : in Id) is
   Other : Id := (X mod 2) + 1;
begin
   Candidate (X) := True;
    Priority := Other;
   while not ( (Candidate (X)) and (Priority = X) ) loop
      null;
   end loop;
end Enter;
```

### 3.3.2 *Second version*

The second version differs from the previous one by the waiting condition : in this version, a task waits as long as the other task is candidate and it's the turn of the other.

```
procedure Enter (X : in Id) is
   Other : Id := (X mod 2) + 1;
begin
   Candidate (X) := True;
    Priority := Other;
   while (Candidate (Other)) and (Priority = Other) loop
      null;
   end loop;
end Enter;
```

### 3.3.3 *Third version*

At last, in the third version, condition on barrier is modified in order to enforce protection of the critical section : a task can enter in the critical section only if the other is not candidate nor it is its turn.

```
procedure Enter (X : in Id) is
   Other : Id := (X mod 2) + 1;
begin
   Candidate (X) := True;
    Priority := Other;
   while (Candidate (Other)) or (Priority = Other) loop
      null;
   end loop;
end Enter;
```

### 3.3.4 *Properties expression*

Given the previous program, several properties can be checked with QUASAR. At first, one can check that there is no deadlock. For the three versions the response is `No deadlock found` that give some confidence on the code. It's not surprising since there is no blocking operations in the program.

However, no deadlock does not mean no problem. Indeed, if we ask to QUASAR "*Is the body of procedure* `Set_Controller_Instruction` *a critical region*" the response is `Yes` for the second and the third version of `Enter` and `No` for the first one (and the tool provides in this case a sequence that shows the property violation). So, the version1 of procedure `Enter` is incorrect with respect to the protection of the critical section.

At this point, version2 and version3 seem to be correct : no deadlock and the respect of the critical section is guaranteed. Nevertheless, it is not the case. Indeed, if we define a state $s_1$ as "*one task (T_One or T_Two) is in critical region*" and if we ask to QUASAR "*is $s_1$ a home state from the initial* *state*" the response is `Yes` for the second version and `No` for the third one. Indeed, in this last version, the counter example shows that the two task can both enter the `while` loop of procedure `Enter` and never go out the loop (both are and remain candidate). This is not a deadlock (both tasks remain active and use CPU time but none can progress) : it is a livelock.

Some other properties may be checked. For instance, one can verify that with the Peterson algorithm (the correct version) a task that does not want to enter the critical section does not block a task that wants to enter it. This property corresponds to the *Inevitable state under condition* template in which the condition is "*task T_Two does not access the critical section*" and the state to reach is "*task T_One access the critical section*" (from the initial state).

This little (but sometimes subtle) example shows that verifying concurrent properties is easier with a tool like QUASAR. Indeed, in this example, simulation does not efficiently reveal the problem while manual code analysis is a tedious task that is often error-prone. Using QUASAR, not only can one efficiently detect possible problem but one can have a sequence leading to the error state (that gives the opportunity to correct the problem). Furthermore, the templates proposed in QUASAR simplify greatly the expression of the properties that has to be checked.

## 4. RELATED WORKS

Many works concern the analysis of concurrent programs with formal methods. We can cite :

- Ada83 code analysis with Petri net techniques and particular with structural techniques like invariants or net reductions has been done by Murata and its team [17], [20], [18]. The use of ordinary Petri nets seems to us a major drawback of these works since it's very difficult with this formalism to express complex programming patterns.
- Symbolic data flow analysis framework for detecting deadlocks in Ada programs with tasks [1].
- Verification of distributed applications described in Promela or in C with the tool FeaVer or with the Spin model-checker at Bell Laboratories [11], [9], [7], or with the tool VeriSoft [5], [4].
- Multi-threaded Java source code verification with model checking techniques, using an adaptation of the Spin tool in the Bandera project [2] at Kansas State University.

¿From our experience, all these tools suffer from a intermediate language (Promela or internal description language) less mature and studied than Colored Petri nets. We claim that using colored Petri nets allows us to model complex program constructions [14] while limiting the combinatory explosion by combining easily and efficiently structural techniques (that work directly on the model) and optimized techniques based on the underlying state graph exploration. Furthermore, many efficient tools can be used to verify LTL property of a colored Petri.

At last, at our best knowledge, only few tools aim to simplify the use of formal temporal logic (except the Time-Line Editor [16] or the graphical version of the Spin model-checker [8]).

Our approach tries to combine the exactness of the LTL formalism with an intuitive interface and pre-defined templates that correspond to the most useful concurrent properties in the context of program verification.

## 5. CONCLUSION

We have presented in this paper an original and useful way for specifying and analyzing temporal properties of concurrent programs. This approach is based on the use of pre-defined property templates that can be instantiated using a graphical interface. All these new features are available in the QUASAR tool.

Current works include :

- the experimentation of QUASAR with other real life Ada programs (QUASAR is a *free* software and we encourage anyone to download it from `quasar.cnam.fr` and to use it for validating his concurrent programs);
- the definition of new property patterns using CTL (or equivalent logic) and referring to *potential reachability* (not for all execution but for at least one execution). Using CTL will require to adapt some techniques for reducing the cost of the verification process [10];
- extend the scope of analyzable Ada constructions, such as tagged types or dynamic task creation;
- extending the analysis capacity using stochastic or temporal Petri nets;
- developing specific verification techniques that take advantage of the way the formal models are produced (they correspond to program patterns and have thus particular behavior).

All material used in this paper, as well as an alpha version of Quasar and related documentations, are available on `http://quasar.cnam.fr`.

## 6. REFERENCES

[1] J. Blieberger, B. Burgstaller, and B. Scholz. Symbolic Data Flow Analysis for Detecting Deadlocks in Ada Tasking Programs. In *Proc. of the Ada-Europe International Conference on Reliable Software Technologies*, Potsdam, Germany, 2000.

[2] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[3] S. Evangelista, C. Kaiser, J. F. Pradat-Peyre, and P. Rousseau. Quasar: a new tool for analysing concurrent programs. In *Ada-Europe 2003*, LNCS. Springer-Verlag, 2003.

[4] Patrice Godefroid, Robert S. Hanmer, and Lalita Jategaonkar Jagadeesan. Model checking without a model: An analysis of the heart-beat monitor of a telephone switch using verisoft. In *International Symposium on Software Testing and Analysis*, pages 124–133, 1998.

[5] Patrice Godefroid. Verisoft: A tool for the automatic analysis of concurrent reactive software. In *Computer Aided Verification*, pages 476–479, 1997.

[6] C. Girault and J.F. Pradat-Peyre. Les réseaux de Petri de haut-niveau. In M. Diaz, editor, *Les réseaux de Petri: Modèles fondamentaux*, number ISBN 2-7462-0250-6, chapter 7, pages 223–254. Hermès (French), 2001.

[7] G.J.Holzmann and Margaret H. Smith. An automated verification method for distributed systems software based on model extraction. *IEEE Trans. on Software Engineering*, 28(4):364–377, April 2002.

[8] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[9] G.J. Holzmann. Logic verification of ansi-c code with spin. pages 131–147. Springer Verlag / LNCS 1885, Sep. 2000.

[10] S. Haddad and J.F. Pradat-Peyre. New powerfull Petri nets reductions. Technical report, CEDRIC, CNAM, Paris, 2003.

[11] G.J. Holzmann and Margaret H. Smith. Software model checking - extracting verification models from source code. pages 481–497, Kluwer Academic Publ., Oct. 1999. also in: Software Testing, Verification and Reliability, Vol. 11, No. 2, June 2001, pp. 65-79.

[12] ISO/IEC 15291. *Ada Semantic Interface Specification (ASIS)*, 1999.

[13] K. Jensen. Coloured Petri nets : A high level language for system design and analysis. In Jensen and Rozenberg, editors, *High-level Petri Nets, Theory and Application*, pages 44–119. Springer-Verlag, 1991.

[14] C. Kaiser and J.F. Pradat-Peyre. Comparing the reliability provided by tasks or protected objects for implementing a resource allocation service : a case study. In *TriAda*, St Louis, Missouri, november 1997. ACM SIGAda.

[15] M. Makela. Maria user's guide. Technical report, Helsinki Univ. of Technology, Finland, 2002.

[16] G.J. Holzmann M.H. Smith and K. Etessami. Events and constraints, a graphical editor for capturing logic properties of programs. pages 14–22, Toronto, Canada, Aug. 2001.

[17] T. Murata, B. Shenker, and S.M. Shatz. Detection of Ada static deadlocks using Petri nets invariants. *IEEE Transactions on Software Engineering*, Vol. 15(No. 3):314–326, March 1989.

[18] M. Notomi and T. Murata. Hierarchical reachability graph of bounded Petri nets for concurrent-software analysis. *IEEE Transactions on Software Engineering*, Vol. 20(No. 5):325–336, May 1994.

[19] A. Pnueli. The temporal semantics of concurrent programs. In *Theoretical Computer Science*, number 13, pages 45–60, 1981.

[20] S. Tu, S.M. Shatz, and T. Murata. Applying Petri nets reduction to support Ada-tasking deadlock detection. In *Proceedings of the 10th IEEE Int. Conf. on Distributed Computing Systems*, pages 96–102, Paris, France, June 1990.

[21] M.Y. Vardi. An automata-theoretic approach to linear temporal logic. In F. Moller and G. Birtwistle, editors, *Logics for Concurrency: Structure versus Automata*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer-Verlag, Berlin, 1996.

[22] K. Varpaaniemi, Halme J., Hiekanen K., and Pyssisalo T. prod reference manual. Technical Report 13, Helsinki Univ. of Technology, Finland, 1995.