Verification in Concurrent Programming with Petri nets Structural Techniques

Kamel Barkaoui and Jean-François Pradat-Peyre
Conservatoire National des Arts et Métiers
Laboratoire CEDRIC
barkaoui@cnam.fr, peyre@cnam.fr

Abstract

This paper deals with verification of flow control in concurrent programs. We use Ada language model as reference. After translation of Ada programs into Petri nets (named Ada nets for Ada programs), we show how one can fully exploit the relationship between the behavior of the concurrent program and the structure of the corresponding Petri net. Using the siphon structure, we precise some structural conditions for behavioral properties such as deadlock-freeness and liveness that correct concurrent programs must satisfy. These conditions can be proved or disproved using efficient algorithms. We provide also a formal justification of guidelines (such as client/server paradigm) that programmers observe traditionally in order to built correct concurrent programs. Several examples are presented to show the effectiveness of using structure theory of Petri nets for static analysis of concurrent programs.

1. Introduction

1.1. Motivations

Because individual modules are small enough to be understood and are more likely to be correct than larger ones, the software construction for complex computer systems must be done in a modular fashion. However, more and more complex computer systems are built such that embedded computer systems where concurrent programming is required, and where faults in their design or construction lead to unacceptable consequences. So, these big and critical systems must be correct but not only "probably" correct. In fact, for such applications , "probably" almost means "probably not". For example, suppose that in a 1000-modules system , we have 0.999 confidence in each module (or task), then the probability of the whole system being error-free will be only $(0.999)^{1000} = 0.37$.

Even, if one can prove, with formal methods, that specifications correspond to user needs, and that the code is correct with respect to these specifications, the correctness of concurrent programs is ensured only if safety and liveness properties (in the sense of Owicki and Lamport [17]) are satisfied. Due to the non-deterministic behavior of concurrent programs, the proof of these properties is quite complex.

An important example of safety property is freedom from deadlock. Deadlock is a state in which all processes (tasks) are blocked waiting for something that will never happen. Detection techniques of deadlocks in concurrent programs generally split into two classes : static analysis techniques which do not require actual execution of the source program, and dynamic analysis techniques which requires the introduction of a specific monitoring task into the program and for which results are dependent on the scheduler characteristics.

In this paper, we provide a contribution for verification of concurrent programs [9]. For sake of simplicity, we focus here on concurrent Ada programs (called in the following Ada tasking programs) involving tasks and rendez-vous

concepts. However, the techniques proposed are based on structure theory of Petri nets, and can be applied to any synchronization and communication pattern [13, 2].

This paper is organized as follows : in section 2, we recall some basic definitions and results of the structure theory of Petri nets being used in this study. We give also a succinct overview of translation of Ada tasking programs into Ada nets and their main properties. In section 3 we give a necessary structural-liveness condition and a sufficient deadlock freeness condition of an Ada net based on the notion of controlled siphon. In section 4 , we present subclasses of Ada programs for which necessary or sufficient condition deadlock freeness (and liveness) conditions can be stated and efficiently proved. We justify hence formally some practices used by programmers to built deadlock-free concurrent programs. Finally in section 5, we conclude and discuss worthwhile future work.

1.2. On flow control analysis of concurrent programs

A program is correct when it fulfills certain properties [18]. A property of a concurrent program is some assertion that is true of every legal execution of that program. In general the properties are very dependent on the scheduling policy used. Static analysis of a program consists in checking properties without making assumptions on the support system on which the program is executed.

There are two classes of property that a correct concurrent program must satisfy : safety and liveness [17, 8].

1. safety properties : assert that nothing ”bad” will ever happen during an execution (the program will never enter a ”bad” state).
2. liveness properties : assert that something ”good” will eventually happen during the execution.

There are two important concurrent program properties : the deadlock freeness and the livelock freeness. A deadlock is a state in which processes are blocked waiting for something that will never happen. The livelock is a condition in which processes are executing instructions uselessly, in the sense that they will never make constructive progress.

If deadlock can be viewed as a violation of safety and liveness, livelock however is not usually considered a violation of safety.

Conditions for the occurrence of livelock remain difficult to characterize while conditions for the occurrence of deadlock are well identified. However the strategies to cope with deadlock need techniques difficult to manage.

There are three strategies for coping with deadlock in concurrent programming. The first one, very difficult to apply, consists in introducing components into the program that can detect the occurrence of deadlock and when it has, carry out some recovery actions. The second one, called deadlock avoidance, consists in anticipating impending deadlock and to take avoiding action in advance. These two approaches need the introduction of external components and doesn’t ensure that system remains always in safe states. The third one, called deadlock prevention, consists in building a system such that the occurrence of deadlock is logically impossible i.e. in which one of the Coffman conditions [10] is denied. Generally, designers try to remove the circular ”wait condition” by ordering the resources requests. This policy leads generally to poor performance measures in terms of concurrency (throughput, resources utilization ratios, ...) and imposes restrictive programming rules difficult to respect.

An efficient deadlock prevention method consists in translating a concurrent program into a Petri net model [15]. This net is called Ada net in the case of multi-tasking Ada program. In such a translation, only Ada statements which can alter the control flow (such as If, Loop, and select statements) or constitute a Rendez vous (entry calls and accepts) are taken into account in the Petri net translation. More details on the translation of Ada programs into Ada nets can be founded in [15] or in [2, 13] for the modeling of the protected object in Ada95. For efficiency reasons, the reachability analysis of Ada net must be avoided since it is equivalent to Taylor’s algorithm that shows in [21] that the generation of the complete concurrency

history for a program has an exponential time complexity (in terms of the number of tasks). Recent works attempt to exploit reductions [22] and decomposition techniques [16] developed in Petri nets theory to reduce the cost of such analysis [20]. In [15] a static analysis method is developed combining Petri nets transition invariants with reachability graph generation.

The approach developed in this paper is purely based on structure theory concepts of Petri nets. Structure theory consists to relate the behavior of the Petri net to its structural objects, such like siphons, traps or invariants.

2. Basic Petri nets notions

2.1. Definitions

We suppose that the reader is familiar with basic Petri nets notions and its graphical representation.

In the following, we denote $\langle P, T, W^+, W^- \rangle$ a Petri net where P is the set of places, T is the set of transitions and W^- (resp. W^+) is the the backward (resp. forward) incidence application from $P \times T$ to $\mathbb{N}$.

Given a marking M and a transition t, one denotes by $M[t > M'$ the fact that t is fireable at M and accesses M'. The set of reachable markings of a Petri net from a marking M is denoted by $A(R, M)$.

One notes ${}^\bullet p = \{t \in T | W^+(p,t) > 0\}$ and $p^\bullet = \{t \in T | W^-(p,t) > 0\}$.

Definition 2.1 *Let (R, M_0) be a marked Petri net. (R, M_0) is said to be :*

- **live** *iff:* $\forall M \in A(R, M_0), \forall t \in T, \exists s \in T^*$ *such that :* $M[s > M'[t > M''$. *This property means that whatever the evolution of the system is, each action (transition) remains executable (fireable).*
- **weakly-live** *(or deadlock free) iff :* $\forall M \in A(R, M_0), \exists t \in T, M' \in \mathbb{N}^P$ *such that* $M[t > M'$. *This property means that the system never reaches a deadlock (not deadlockable) but does not ensure that the net is live.*

Important remark : Because we focus only on the control flow and the communication patterns of concurrent programs, valuations of the net model (Ada nets for Ada programs) belong to $\{0, 1\}$. **So, now on, we denote by Petri net, places/transitions nets for which valuations belong to** $\{0, 1\}$.

We define now two classes of Petri nets for which there exists a necessary and sufficient condition for liveness.

Definition 2.2 *A Petri net is called*

- *a* **free choice** *net if and only if:* $\forall (p,q) \in P \times P, p^\bullet \cap q^\bullet \neq \emptyset \Longrightarrow p^\bullet = q^\bullet$
- *an* **asymmetric choice** *net if and only if:* $\forall (p,q) \in P \times P, p^\bullet \cap q^\bullet \neq \emptyset \Longrightarrow p^\bullet \subseteq q^\bullet$ *or* $q^\bullet \subseteq p^\bullet$

2.2. The controlled siphon property

Definition 2.3 *Let R be a Petri net.*

- *A subset of place D of P is a* **siphon** *if and only if,* ${}^\bullet D \subseteq D^\bullet$. [1]
- *A siphon D is said to be* **token-free** *for a marking M if* $\forall p \in D, M(p) = 0$.
- *A siphon D is said to be* **minimal** *if it contains no other siphon.*
- *A siphon D is said to be* **maximal** *if there is no siphon containing it.*
- *A siphon D is said to be* **controlled** *for M_0 an initial marking of R if for each reachable marking M of $A(R, M_0)$, D is not token-free for M.*

Definition 2.4 *A marked Petri net (R, M_0) fulfills the* **controlled siphon property (cs-property)** *if each siphon D of R is controlled for M_0.*

With this definition sufficient conditions for deadlock freeness and a necessary condition for liveness can be established.

[1] we denote by extension, $D^\bullet = \cup_{p \in D} \{p^\bullet\}$ the output transitions of D and ${}^\bullet D = \cup_{p \in D} \{{}^\bullet p\}$ the input transitions of D

Proposition 2.1 *The cs-property is a sufficient condition for deadlock freeness [5]; i.e. a marked Petri net that fulfills the cs-property is weakly-live.*

Proposition 2.2 *The cs-property is a necessary liveness condition [5]; i.e a live marked Petri net fulfills the cs-property.*

These two conditions can be unified in a necessary and sufficient condition for liveness in the case of asymmetric choice nets.

Proposition 2.3 *An asymmetric net (or a free choice net) is live if and only if it satisfies the cs-property [5].*

2.3. How cs-property can be checked

Two structural methods can be used to check the satisfiability of cs-property. The first one consists in checking that each minimal siphon contains a trap (a subset of places S with $S^\bullet \subset {}^\bullet S$) that is non token-free for the initial marking. This is the deadlock-trap property [19]. The second one consists in checking that for each siphon of the net, there exists a particular invariant place f that "covers" it and ensures, with respect of initial marking, that the siphon cannot become token-free [5].

Proposition 2.4 *Let (R, M_0) be a marked Petri and D a siphon of R. If there exists a subset of places S with $S^\bullet \subseteq {}^\bullet S$, $S \subseteq D$ and $M_0(S) > 0$ then D is controlled for M_0.*

In this proposition the subset S is a trap and this proposition is known as the Commoner's property.

Proposition 2.5 *Let (R, M_0) be a marked Petri and D a siphon of R. If there exists a vector of places $f = \sum_{p \in P} f(p).p$ such that*

1. $\forall t \in T, \sum_{p \in P} f(p).W^-(p,t) = \sum_{p \in P} f(p).W^+(p,t)$,
2. $\{p \in P / f(p) > 0\} \subseteq D$ and
3. $\sum_{p \in P} f(p).M_0(p) > 0$

then D is controlled for M_0 [5].

Remark : point 1 ensures that f is a place invariant of the net, point 2 that the positive support of f is included in the siphon D and point 3 that there exists a place p of D such that $f(p) > 0$ and $M_0(p) > 0$.

Checking the cs-property by the first method is polynomial for free choice nets [4, 11, 1], but it is theoretically exponential in general case because of the complexity of minimal siphon calculus.

However, experience shows that, in most cases, these verification methods work as they were polynomial [12], [8]. Furthermore, it is clear that for complex models it is often easier to check the cs-property than to build the reachability graph in order to ensure the deadlock freeness [19].

In section 3, we show how the cs-property can be used to characterize Ada-tasking programs.

3. On the use of the cs-property in concurrent programming

3.1. Modeling concurrent programs with Petri nets

We focus on Ada's language model [14] but others languages employing message passing, remote invocation, semaphore or monitors (like Occam, Pascal FC, Java, ...) can also be take into account with the proposed approach due to modeling power of Petri nets.

Parallel systems programming can be expressed in Ada through the concept of task. An Ada task is an Ada program unit which is defined by a specification and a body. A task specification defines the name of the task and names of its entries. Each entry defines a synchronization or communication point reachable by others tasks. The synchronization mechanism is based on the one-to-one Rendez-vous mechanism. This mechanism assumes that both called and calling tasks are ready to achieve the Rendez-vous. In order to provide code reutilization, a Rendez-vous is asymmetric : the called task is not made known to the calling task. An entry without parameter means that this entry is only used to synchronize tasks (no communication). Furthermore, tasks can make selective calls or selective accepts by the use of the constructor "select".

Petri nets are well-suited to describe communication patterns and control flow of an Ada tasking program. Basic works on this approach are described in [15] or in [14]. The idea is to associate to each task a Petri net state-machine and to represent communications and synchronization by shared places. The Ada program is then modeled by a set of communicating state-machines.

In term of communication patterns, we have to distinguish several cases : the called task may do a simple or a selective accept, there may be one or many tasks calling a particular entry and a calling task may do a simple or a selective call. To each case corresponds a particular modeling in term of Petri nets.

Translation of Ada-tasking programs into a Petri net model called Ada-net is done according to the rules presented in [15]. This Ada-net is bounded (conservative) by construction. Its initial marking represents the state in which the program modeled by the net is ready to start execution. In order to ensure strong connectivity, and then to use the results of structure theory of Petri nets, we consider that each task is encapsulated in an external loop. This transformation does not alter the behavior of the original programs [15].

We define now the relations linking the behavior of an Ada tasking program and the properties of the corresponding Ada net.

Definition 3.1 *An Ada tasking program is said to be* **statically executable** *if and only if its corresponding Ada net is weakly-live.*

Such a program has always something to do i.e. there is no reachable state in which tasks can be indefinitely blocked waiting for something that will never happen. However, it does not ensure that all instructions can be executed : a part of the program may be blocked. We are then faced to the difficult problem of livelock : "condition in which tasks are executing instructions uselessly, in the sense that they will never make constructive progress" [7].

The following definition characterizes Ada program in which a livelock condition can never appear.

Definition 3.2 *An Ada tasking program is said to be* **concurrently correct** *if and only if its corresponding Ada net is live.*

Correct programs fall within the class of concurrently correct programs and then, several researchers have given a characterization of this class with Petri nets theory.

The first method to verify that an Ada tasking program is concurrently correct is to construct the reachability graph of the Ada net [20] : i.e. the graph which describes the complete enumeration of the state space. Since an Ada net is by construction bounded (each marking place is bounded by an integer), it is possible to generate this graph. However, this way is not efficient since it is equivalent to trace the program's control flow by generating all possible reachable states (Taylor's algorithm [21]). The complexity is exponential : $O(n^T)$ where T is the number of tasks and n the number of statements related to concurrency.

So, alternative methods have been proposed. The first method was to use the theory of invariants [15] to obtain a necessary condition on liveness. The second one was to try to reduce the Ada net state space by applying reduction theory of Petri nets [22] or more recently by using a decomposition method [16]. Unfortunately, all these methods remain exponential in time and space.

We show in the next section that controlled siphon property is a necessary and/or sufficient liveness condition.

3.2. Deadlock detection

3.2.1 A necessary liveness condition

Proposition 3.1 *If the Ada net of an Ada program contains a siphon token-free for the initial marking then the Ada program is not concurrently correct. Moreover, this condition can be checked in linear time (as a function of the number of arcs) by the following algorithm [3].*

Algorithme 1 : maximal siphon

1. $S = \{p \in P | M(p) > 0\} \cup \{p \in P | \exists t \in {}^\bullet p \text{ with } {}^\bullet t = \emptyset\}$
   – S is the set of places that cannot belong to any siphon
2. While $S \neq \emptyset$ do

(a) take p in S; $S = S \setminus \{p\}$

(b) forall t in $p^\bullet$ do
if ${}^\bullet t = \{p\}$ then $S = S \cup \{t^\bullet\}$ fi ; – outputs of t cannot belong to a deadlock

(c) $P = P \setminus \{p\}$ (and remove all arcs linked to p)

done

3. The necessary condition is fulfilled if and only if $P = \emptyset$

Let A be an Ada net and let D be the maximal siphon computed by the previous algorithm. Each task that has non alternative statement and having a transition included in $D^\bullet$ is deadlocked; i.e. the program reaches surely a state from which any instruction of this task can not be more executed. Moreover, it is possible to anticipate about dynamic program behavior. As soon as an alternative of a task including a dead transition ($t \in D^\bullet$) is chosen, then this task will be also deadlocked. If the program contains n tasks, as soon as $n - 1$ tasks are proved deadlocked, then the program is not static executable (its Ada net is deadlocked).

Let us demonstrate the power of this simple necessary condition on significant examples. For the two models of producer-consumer given in [15] (Ada-tasking program 1), we can compute in both case a token-free siphon. The Ada program and the corresponding Ada net of the first example are given below (figure1). One can verify that dashed places form a siphon token-free for the initial marking. This program is not concurrently correct. Task Producer is deadlocked (there is no alternative statement). Since task Buffer accepts entry "produce", this task becomes also deadlocked. So, we can conclude that, since task Buffer accepts entry "produce", the program is deadlocked.

Consider now the following Ada program (Ada-tasking program 2) from [22] modeling the activities in a gas station when a customer comes to fill up with petrol. The corresponding Ada net is represented figure2.

In the corresponding Petri net one can compute in linear time a siphon that is token-free for the initial marking (blacked places on figure 2). So, without any more consideration, we can conclude that the Ada program is not concurrently correct. Since tasks Customer and Pump have no select statement and got state places in this siphon, we can conclude that necessarily these tasks will be

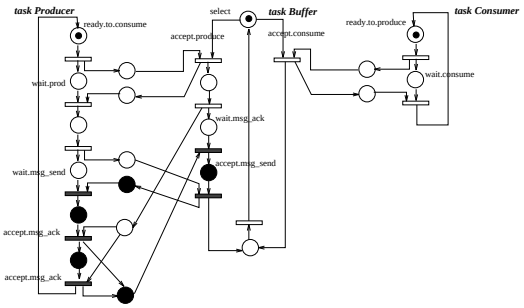

Figure 1. The Ada net of the program 1 (the "bad producer-consumer")

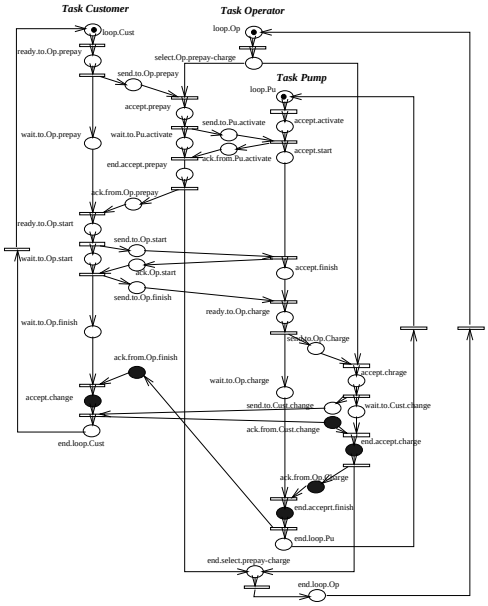

Figure 2. Ada net of Pgm 2 (the gas station program)

deadlocked. As the Ada program contains only three tasks, the program is deadlockable.

3.2.2 A sufficient deadlock freeness condition

We can claim through these significant examples that most ”bad” Ada programs can easily be detected by our previous algorithm. However, a program that pass this filter is not necessary ”good”. So, in order to characterize ”not bad” program, we give now a sufficient condition on weakly-liveness.

Proposition 3.2 *If the Ada net of an Ada program fulfills the cs-property then the Ada program is* **statically executable**.

Proof :

As a direct consequence of proposition 2.1.

Using this property it is possible to verify that programs which seems to be concurrently correct (the last necessary condition is satisfied) are at least statically executable. Theoretical complexity of this verification is not polynomial in the general case due to the theoretical complexity of minimal siphon computation. However, experiences reveal [12] that in most cases, siphon can be computed in polynomial time and so, checking cs-property can be done efficiently.

On the following example (Ada-tasking program 3 and fig.3) taken from [15], the cs-property is satisfied (all every minimal siphon contains a marked trap), then we can ensure that the corresponding Ada-tasking program is at least weakly-live. Indeed, we will show in the next section, that this program is live because it belongs to a class of Petri nets (asymmetric choice nets) for which the cs-property is a sufficient liveness condition.

4. Sub classes of concurrent programs

Many books about concurrent programming give the following guideline : ”a task must be an active entity (client) or a passive one (server) but not both !”.

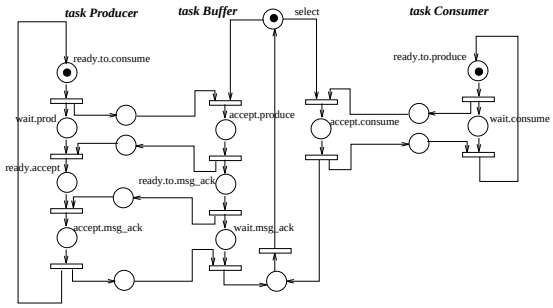

Figure 3. Ada net of program 3 (the "good" producer-consumer)

We must not give to this guideline a restrictive meaning. In our context, this sentence must be interpreted as : ”at each Rendez-vous occurrence, one and only one participating task may be on a selective statement”. At this point, only one task controls, in some way, the manner in which it is used. In this sense, this task acts as a server, the other as a client. It is possible to say that the client/server paradigm is satisfied locally.

One can note that this paradigm is thinner than the classical ”client/server paradigm” which imposes that a task must be devoted to be either a client or a server. In particular, a task may be a client at one point of its life and a server at an other point. Furthermore, a task may choose, using a select statement, to be either a server or a client. The only restriction is that two tasks on a Rendez-vous cannot be both on a select statement (even if one make only calls and the other only accepts). In this case, both may control the manner in which they are used and then, both act as a client.

We can now justify formally how this empirical guide help programmers to ensure the concurrency semantic correctness of their programs.

4.1. Asymmetric choice programs

Definition 4.1 *A tasking Ada program is an* **asymmetric choice Ada-tasking program** *if at each Rendez-vous occurrence at most one of both task is on a select statement.*

By construction, Ada nets of asymmetric choice

Ada-tasking program are asymmetric choice nets. It has been proved, in Petri nets theory, that such nets are live as soon as they are weakly-live [5, 6].

So, under this property, livelock condition cannot appear in asymmetric choice Ada-tasking programs. They are either totally blocked or concurrently correct. Since detection of livelock is much less straightforward, it becomes obvious that building asymmetric choice Ada-tasking programs is an efficient way to build correct concurrent programs.

Proposition 4.1 *A* **necessary and sufficient condition** *for an asymmetric choice Ada-tasking program to be* **concurrently correct** *(live) is that the corresponding Ada-net fulfills the cs-property.*

Proof :

Since Ada-nets obtained from asymmetric choice Ada-tasking program is asymmetric choice net, we have, prop. 2.3, that cs-property is a necessary and sufficient condition for the program to be concurrently correct.

We may observe that several significant examples belong to this class (all programs presented previously are in this class). For instance, the ”good” program of consumer-producer (Ada-tasking program 4) is an asymmetric choice program. As we show in the previous section, this program is statically executable (weakly-live), so, using the previous condition, it is possible to conclude that this program is concurrently correct (live).

If we consider the problem of the dining philosophers, and if we imposed that philosopher take forks in a particular order we have the following Ada-tasking program.

Using the algorithm given in [3], it is possible to check that this program is an asymmetric choice program and that the corresponding Ada net satisfies the cs-property.

So, this solution of the dining philosophers problem is concurrently correct.

4.2. Free choice programs

In Petri nets theory it is shown [4] that for some classes of bounded Petri nets(extended free choice nets and non self-controlling nets) liveness is decidable in polynomial time using cs-property.

In this paper we define a first sub-class of asymmetric choice programs called free choice programs for which concurrently correctness can be checked in polynomial time.

Definition 4.2 *An asymmetric choice Ada-tasking program is a* **free choice Ada-tasking program** *if none of its tasks has a select statement and each of its entry is called once only.*

Proposition 4.2
A **necessary and sufficient condition** *for a free choice Ada-tasking program to be* **concurrently correct** *(live) is that the corresponding Ada-net fulfills the cs-property. Moreover, the cs-property can be checked in polynomial time.*

Proof :

Since free choice Ada-tasking programs are a particular case of asymmetric Ada-tasking programs the first part of the proof is clear. ◇◇◇

It is shown in [4] that the cs-property is equivalent to the deadlock-trap property. Furthermore deadlock-trap property can be checked in polynomial time for bounded (marking of each place is bounded) free choice nets by using algorithms proposed in [4] or in [11]. Since free choice Ada nets are bounded, the cs-property can be checked in polynomial time.

It is important to point out that, in spite of the limitations imposed by definition, the nested Rendez-vous is allowed in a free choice Ada-tasking program. For instance, let us consider the following program (Ada-tasking program 5) taken from [16] in which we have introduced a nested Rendez-vous. In [16], this example is used to illustrate the efficiency of the method proposed by the authors which is exponential in time.

Such a program is a free-choice Ada tasking program. So, we can prove in polynomial time, that it is concurrently correct using only the cs-property.

5. Conclusions

The crucial need for verification techniques which can be applied to concurrent software for complex systems has motivated our study. The main contribution of this work is to show that concepts and results of structure theory of Petri nets can lead to an efficient and automatic support for static analysis of concurrent programs. Moreover, we provide the formal justification of empirical guidelines that programmers observe frequently in order to built concurrent correct (safe and live) programs.

Some areas for future research are suggested by this work : first, we try to combine all structural analysis techniques (based on graph theory and or linear algebra) of Petri nets to highlight the livelock structural conditions and to improve tools and methodologies for concurrent software verification.

Secondly, we project to apply these techniques to Java programs in order to propose safe concurrent patterns, offering high level synchronization mechanisms for secure intranet applications.

References

[1] K. Barkaoui, J. Couvreur, and C. Duteilhet. On liveness in extended non self-controlling nets in application and theory of Petri nets. *LNCS*, 935, 1995.

[2] K. Barkaoui, C. Kaiser, and J. Pradat-Peyre. Petri nets based proofs of Ada95 solution for preference control.
In *Proc.of the Asia Pacifi c Software Engineering Conference (APSEC) and International Computer Science Conference (ICSC)*, Hong-Kong, 1997.

[3] K. Barkaoui and M. Minoux. Deadlocks and traps in Petri nets as horn satisfi ability solutions and some related polynomially solvable problems. *Discrete Applied Mathematics*, No. 29, 1990.

[4] K. Barkaoui and M. Minoux. A polynomial time graph algorithm to decide liveness of some basic classes of bounded Petri nets. *LNCS*, No. 616:62–75, 1992.

[5] K. Barkaoui and J. Pradat-Peyre. On liveness and controlled siphons in Petri nets. In Reisig, editor, *Petri Nets, Theory and Application*, number 1091 in LNCS. Springer-Verlag, 1996.

[6] E. Best. Structure theory of Petri nets : The free choice hiatus. In G. W.Brauer, W.Resig, editor, *LNCS*, volume No. 255. Springer-Verlag, 1986.

[7] A. Burns, M. Lister, and A. Wellings. A review of Ada tasking. In *LNCS*. Springer-Verlag, 1987.

[8] A. Burns and A. Wellings. *Concurrency in Ada*, chapter 6.11, pages 134–137. Cambridge University Press, 1995.

[9] J. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, Vol. 22(No. 3), 1996.

[10] C. E.G., E. M.J., and S. A. Systems deadlocks. In *Computing Survey, 3 (2)*, 1971.

[11] J. Esparza and M. Silva. A polynomial-time algorithm to decide liveness of bounded free-choice nets. *T.C.S*, N 102:185–205, 1992.

[12] J. Ezpeleta and J. Couvreur. A new technique for fi nding a generative family of siphons, traps and st-components. In *proc of the 12th International Conference on Application and Theory of Petri-Nets*, Aarhus, Denmark, June 1991.

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

[14] D. Mandrioli, R. Zicari, C. Ghezzi, and F. Tisato. Modeling the Ada task system by Petri nets. *Computer Languages*, Vol. 10(NO. 1):43–61, 1985.

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

[16] 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.

[17] S. Owicki and L. Lamport. Proving liveness property of concurrent programs. In *ACM Transaction on Programming Languages and Systems, Vol.4, N.3, pp. 455-495*, 1982.

[18] M. R. Communication and concurrency. In *Prentice Hall Ed.*, 1989.

[19] W. Reisig. *EATCS-An Introduction to Petri Nets*. Springer-Verlag, 1983.

[20] S. Shatz, K. Mai, D. Moorthi, and J. Woodward. A toolkit for automated support of Ada-tasking analysis. In *Proceedings of the 9th Int. Conf. on Distributed Computing Systems*, pages 595–602, June 1989.

[21] R. Taylor. A general purpose algorithm for analyzing concurrent programs. *Communication of ACM*, Vol. 26(No. 5):362–376, May 1983.

[22] S. Tu, S. 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.

Annexe

```
task body Producer is
begin
loop
    Buffer.produce;
    Buffer.msg_send;
    accept msg_ack;
    accept msg_ack;
end loop;
end Producer;
```

```
task body Buffer is
begin
loop
   select
        accept produce
        Producer.msg_ack;
        accept msg_send;
    or
        accept consume;
    end select;
end loop;
end Buffer;
```

```
task body Consumer is
begin
loop
   Buffer.consume;
end loop;
end Consumer;
```

Ada-tasking program 1: A "bad" producer-consumer

```
task body Customer is
begin
loop
    Operator.Prepay;
    Pump.Start;
    Pump.Finish;
    accept Change;
end loop;
end Customer;
```

```
task body Operator is
begin
loop
    select
        accept Prepay do
            Pump.Activate;
        end Prepay;
    or
        accept Charge do
            Customer.Change;
        end Charge;
    end select;
end loop;
end Operator;
```

```
task body Pump is
begin
loop
    accept Activate;
    accept Start;
    accept Finish do
        Operator.Charge;
    end Finish;
    end Pump;
```

Ada-tasking program 2: The gas-station program

```
task body Producer is
begin
loop
    Buffer.produce;
    accept msg_ack;
end loop;
end Producer;
```

```
task body Buffer is
begin
loop
   select
       accept produce
       Producer.msg_ack;
   or
       accept consume;
   end select;
end loop;
end Buffer;
```

```
task body Consumer is
begin
loop
   Buffer.consume;
 end loop;
 end Consumer;
```

Ada-tasking program 3: A "good" producer-consumer

```
task body Philo1 is
begin
loop
   Forks.f1;
   Forks.f2;
end loop;
end Philo1;
```

```
task body Philo2 is
begin
loop
   Forks.f3;
   Forks.f2;
end loop;
end Philo2;
```

```
task body Philo3 is
begin
loop
   Forks.f1;
   Forks.f3;
end loop;
end Philo3;
```

```
task body Forks is
begin
loop
   select
       accept f1;
   or  accept f2
   or  accept f3;
     end select;
  end loop;
  end Forks;
```

Ada-tasking program 4: A version of the dining philosophers

```
task body T1 is
begin
loop
   T2.E;
end loop;
end T1;
```

```
task body T2 is
begin
loop
   accept E do
      T3.F;
   end E;
end loop;
end T2;
```

```
task body T3 is
begin
loop
   accept G;
   accept F do
       accept H;
   end F;
end loop;
end T3;
```

```
task body T4 is
begin
loop
   T3.G;
   T3.H;
end loop;
end T4;
```

Ada-tasking program 5: An Ada free choice program