Correctness properties in a control-parallel extension of Fortran

Gilbert Caplain

septembre 1994

N° 94-29

Correctness properties in a control-parallel extension of Fortran[1]

Gilbert Caplain

CERMICS, Ecole Nationale des Ponts et Chaussées
F–93167 Noisy-le-Grand Cedex

Abstract

We study a correctness property of programs in a subset of Fortran X3H5, a control-parallel extension of Fortran. This property is a semantic equivalence between a parallel program and its sequential version. We present an extended version of the correctness property outlined in the CERMICS report 93-18 [2].

Propriétés de validité dans une extension à parallélisme de contrôle de Fortran

Résumé

Nous étudions une propriété de validité dans un sous-ensemble de Fortran X3H5, une extension à parallélisme de contrôle de Fortran. Cette propriété est une équivalence sémantique entre un programme parallèle et sa version séquentielle. Nous présentons une version étendue de la propriété exposée dans le rapport de recherche CERMICS n^o 93-18 [2].

[1]This work was supported by the French Department of Research and Industry, Contract n^o 90 S 0961, and developed within a project undertaken together with René Lalement and Thierry Salset (same address). I am grateful to both of them for helpful comments.

Contents

1 Introduction

In [2], we studied conditions for correctness of parallel programs written in some parallel extension of Fortran. The parallel language we studied is a subset of Fortran-X3H5 [3]. We provided correctness proofs under some restrictions, and showed how to apply these properties to effectively check the correctness of parallel programs in a whole range of cases.

The aim of this paper is to extend these correctness results, and to make them more precise. The extension we consider mainly consists in allowing *dynamic variable reference* and *dynamic loop bound evaluation*, i.e. variables (other than loop indices and program parameters – see below) in subscript lists of arrays and in loop bounds.

We begin with an outline of the language subset we study. Then, we set up the correctness problem : a parallel program and its sequential version, that we define, must have the same observable behavior *(semantic equivalence)*. Afterwards, we describe the *precedence* and *dependence* relations, emphasizing the differences brought by the "dynamic" extensions considered here.

Then, after deriving a few preliminary results, we prove our main theorem of semantic equivalence : assuming the preservation of dependences (along the well known "dependence implies precedence" pattern) *as defined on the sequential version*, and a few other assumptions dealing with the sequential version too, we derive the semantic equivalence property for any possible run of the parallel program being considered.

Finally, we give some hints towards possible extensions of these results.

Although we consider a parallel extension of Fortran, these correctness results could straightforwardly be extended to other parallel languages, however restricted along the lines we mention.

2 The language studied

2.1 A subset of ANSI X3H5

The language we study is a subset of Fortran–77 extended by a subset of the X3H5 parallel constructs. The extensions we include are the combined parallel and worksharing constructs PARALLEL DO and PARALLEL SECTIONS, and the EVENT explicit synchronization (POST/WAIT pairs) ; inside a worksharing construct, the NEW statement may be used. We do not consider the more general PARALLEL regions, nor the mutual exclusion mechanisms (structured CRITICAL SECTION and unstructured LOCK synchronization), and the SCOMMON statement. On the sequential side, we have structured IFs and DO loops, assignment, scalar and array references ; we exclude GOTO, procedure CALL, DATA, EQUIVALENCE and SAVE statements.

A detailed specification of an abstract syntax for this subset has been given in [2].

We allow for *parameters* under the form of scalars (or possibly arrays) that are not written. Thus, in our framework, a *program* in fact represents a "class of programs", differing from one another by the values of parameters.

A **program instance** is obtained from a program by assigning constant values to the parameters.

Let us remind that loop indices cannot be written either.

In what follows, parameters and loop indices will not be termed as "variables". A *variable* is a *memory location* other than a location assigned to a loop index or a parameter. A *(variable) reference* is a syntactic element pointing to a variable. For instance, in the statement :

```
A = B(I)
```

where B is not a vector of parameters, "A" and "B(I)" are variable references ; if I is a loop index and this statement happens to be executed for $I = 3$, then, in this statement execution, the reference "B(I)" points to the variable $B(3)$.

In [2], some staticity conditions were assumed : subscript expressions in array references did not involve variables, but only parameters and indices of enclosing loops (*staticity of variable references*) ; loop bounds did not involve variables either (*loop staticity*). Here, our subset language will be extended : we will consider programs in which variables are allowed in loop bounds as well as in subscript expressions, however with a restriction : we retain the static reference assumption for event array references involved in WAIT statements.

As regards the loop bounds, and consistently with the Fortran usage, we assume that these loops are evaluated at the loop entry, and not reevaluated at every iteration.

Loops are *normalized*, i.e. their increment is set to 1.

Our language subset allows for *explicit private* variables (statement NEW). In [2], we described in detail how these variables should be dealt with, especially as far as dependences are concerned. In what follows, we will not specifically consider these variables ; the extension of our results in case NEW variables are involved, would be straightforward.

Synchronizations

We consider a mechanism for explicit synchronization : the use of *events*. An event has one of two defined values : *cleared* and *posted*. Undefined when created, an event gets a value by one of the statements POST and CLEAR. The WAIT statement, when reached, tests the value of its event, continues if this value is *posted*, and repeats this step at a later time otherwise. In our language subset, only POST and CLEAR statements modify the value of events.

In this paper, we will not consider ORDINAL synchronizations, but our results can be rather straightforwardly extended so as to contain them. Let us just mention that, in such an extension, an ordinal POST virtually contains a WAIT, and would therefore be concerned by the restrictions assumed here for the WAITs.

In the PARALLEL SECTIONS construct, our language subset provides for the possibility to specify, in a SECTION header, that this section shall wait for one or several (lexically previous) specified sections to complete execution (this is the *section wait* feature). In this paper, and *contrarily to the point of view adopted in [2]*, it appeared convenient to treat these section waits as ordinary waits : more precisely, *in the developments presented here, every section wait clause will be treated as though it were replaced by* POST/WAIT *pairs* so as to preserve the semantic properties we are studying. It is straightforward to prove that this is possible ; moreover, in these POST/WAIT pairs introduced thus, the event references are static.

Notion of statement instance

For the sake of convenience, in what follows, the *statements* we will consider will be only *simple* statements, not *structured* ones. Correlatively, we will consider as *statements* not only executable statements in the usual sense, but also such features as : heads and ends of loops and parallel constructs, IFs and ENDIFs.

Considering loops leads us to define a notion of **statement instance**. To every statement in the program, we will associate a set of statement instances, every instance corresponding to a possible execution of the statement, in such a way that two conditions are met : the set of statement instances associated to every statement is defined statically ; a statement instance is executed at most once in a given run (obviously, *whether it is executed or not* is not defined statically).

To every statement in the program, will be associated a (possibly empty) **index vector**, every component of which takes its values in the set of rational integers. A statement instance will then be obtained by assigning an integer value to every component of the index vector. The index vector is recursively defined as follows. Let a be a statement :

- If a is not contained in a loop, its index vector is empty : then, a generates one statement instance.
- If a is contained in a loop, let c be the header of the innermost loop containing a. Let $\boldsymbol{i}$ denote the index vector of c. The index vector of a is then obtained as the concatenation of $\boldsymbol{i}$ and a component j, denoted $\boldsymbol{i}::j$

Thus, through the latter rule, every (executed or not) instance $c(\boldsymbol{i})$ generates an infinite (on both sides) sequence of instances $a(\boldsymbol{i}::j)$

Through this formalism, a statement contained in a loop generates a countable infinity of statement instances but, in any given run, only a finite number of them will come to be executed.

2.2 Execution model

Let us outline the execution model of our language subset, as regards the process control. (As regards variables, some features of the execution model will be outlined in Section 6, when we introduce the notion of time step.)

- The program execution begins, from the program start, with an initial process.
- A process runs until one of these circumstances occurs :
  - it reaches the end of the program (*normal termination – this may occur only to the initial process*) ;
  - it encounters a WAIT ;
  - it encounters a parallel construct ;
  - it encounters the end of a parallel construct.
- When a process encounters a WAIT, it evaluates the event this WAIT statement instance involves :
  - if the event is *posted*, the process continues ;
  - if it is not, the process repeats this step at a later time.
- When a process encounters a parallel construct, it becomes the *base process* for this construct. This parallel construct specifies a number of *units of work* : each iteration of a PARALLEL DO and each section of a PARALLEL SECTIONS is a unit of work. A *team* of processes is created.

Every unit of work is then assigned to some process in this team, in some order. Thus, from this point on, every process will have one or several units of work in charge. (Since nested parallelism is allowed, this definition of units of work and process teams operates recursively : a unit of work may give place to subunits, a process team member may become itself a base process, and so on.)

- When a process has completed the execution of a unit of work, the execution passes to the next unit of work this process has in charge, if any (we will say that the next unit of work is *loaded*) ; if this process has completed the execution of all the units of work it had in charge, it waits for the other processes in the team to complete their work.
- If and when all processes in the team have completed their work, that means that all the units of work in the parallel construct have been executed. Then, the team is dissolved and its base process continues execution. (Only then, we will say that the END PARALLEL DO or END PARALLEL SECTIONS is *executed*.)

A *waiting statement* is a statement on which the control may come to a wait, till some condition is met. In our language subset, the waiting statements are the WAIT, the END PARALLEL DO and END PARALLEL SECTIONS, the WAIT clause of a section.

A unit of work which is waiting for an available process will be said to be *pending*. More precisely, as regards statement instances, the first instance in such a unit of work will be said to be *pending* at that time (we will see later why only the *first*)[1].

It is straightforward to prove that our language does not allow infinite loops (remember the absence of GOTOs and the fact that loop bounds are evaluated once at the loop entry). All programs come to an end, either a normal termination, or a deadlock.

A deadlock necessarily involves a WAIT statement the event of which persistently remains un*posted*. In the (usual) case when this WAIT is located within a parallel construct, a deadlock situation may be described as follows :

- one or several WAIT statement instances are reached but not executed ; so, the corresponding units of work remain uncompleted ;
- as a consequence, the execution of the parallel construct cannot be completed ;
- as a possible consequence too, some parallel units of work do not begin execution because they are assigned to a process after a deadlocked unit of work, though they would be "executable in principle". The first statement instance in each of these units of work is thus "persistently pending".
- in case of nested parallel constructs, a deadlock in an inner construct brings similar deadlock features in an outer construct.

A parallel construct may have the ORDERED condition. The meaning of this condition deals with the way units of work are assigned to the team of processes created when the execution arrives

[1]In case such an instance is a WAIT, we will say that it is *pending* as long as it is not loaded ; it gets *reached* as soon as it is loaded, and *waiting* if its event is not *posted* yet.

to the parallel construct. If this construct is ORDERED, this assignment is done sequentially : in the index ordering for a PARALLEL DO, in the section ordering for a PARALLEL SECTIONS. Otherwise, the assignment may be done in any order.

As a consequence, in case a parallel construct is ORDERED, the order in which units of work are assigned to some process is necessarily compatible with the index or section ordering. In other words, no unit of work is assigned to some process after another unit of work whenever the former unit ranks before the latter in the sequential order. As a matter of fact, this property of the ORDERED condition will be sufficient for the proofs we will give, though the X3H5 proposal for the ORDERED condition is more exacting.

2.3 Serial semantics. The notion of semantic equivalence

By definition, the *sequential version* of a parallel program is the result of the transformation of PARALLEL DO into DO, the deletion of PARALLEL SECTIONS, SECTION and END PARALLEL SECTIONS statements, the substitution of new names for names specified in NEW statements within their scope (this last point has been expanded in [2]), and the *disabling* of POST, WAIT and CLEAR statements : by "disabling", we mean that, in the sequential version defined here, they will function like ordinary CONTINUE statements, but we retain the possibility, in the following developments, to keep track of them, allowing ourselves to consider what occurs to event references as though they were indeed addressed in a sequential run.

In the framework we are considering here, the intended observable behavior of the parallel program is that of its sequential version, whose semantics is (admittedly) well-defined. The existence of a serial semantics is a rather peculiar requirement which does not necessarily hold in any study dealing with parallel programs. For instance, the full X3H5 extensions allow to write a parallel program to search one possible solution to a constraint problem by proceeding in several directions at once, and stopping as soon as one solution is found. Such a program gives non-deterministic outputs and is not required to find the same solution as its sequential version (of some kind which remains to be defined for the full language).

Thus, in our framework, a correct parallel program can be seen as some kind of parallelization of a sequential program, and not as a genuine parallel program. The improvement sought through the parallelization, in this context, lies only in the ability to run the program faster, by allowing several statements to be executed simultaneously, on several available processors.

Our aim is to prove the correctness (or lack thereof) of a parallel program, in that sense.

We would like to show that all variables coming to be computed must, in both versions, undergo the same computations and, therefore, display the same values.

It is easy to prove (by a recurrence along the run of the sequential version) that this **semantic equivalence** requirement we are considering can be expressed as follows :

- any statement instance executed in any parallel run is also executed in the sequential version, and conversely ;
- any variable reference used by that statement instance as input points to the same variable, and that variable has been computed by the same other statement instance, in any parallel run as in the sequential version.

To be more accurate, these properties imply the semantic equivalence under the extra condition that, in the sequential version, any variable used as input in a statement instance has been computed previously. (in other words, the sequential program does not use the random values the memory locations may contain when allocated to the program at the beginning.) In our framework, this *determinacy* condition will be considered as a prerequisite for correctness of the sequential version, assumed in our results.

Checking the semantic equivalence, especially as regards variable computations, boils down to checking that *race conditions* are avoided. Whenever two statement instances involve the same variable location, at least one of them modifying (i.e. writing) it, we will say that they are in a *dependence* relation. Then, we will have to check that the program control structure and the synchronizations preserve the order in which these statement instances will be executed – along the well known *"dependence implies precedence"* pattern [1].

3 Execution predicate

As long as we have not proved that there is no memory conflict between parallel processes, it is not possible to assume that an expression may be evaluated to a well-defined value. When the program is shown to be correct, the value of an expression in a statement instance is the value computed by the sequential version.

For every statement a, indexed by the (possibly empty) index vector $\boldsymbol{i}$, we would like to use a predicate $\mathrm{Exe}(a(\boldsymbol{i}))$ meaning that the statement instance $a(\boldsymbol{i})$ is executed. In the case of a waiting statement (e.g. a WAIT), "being executed" will mean "being passed". An important fact must be pointed out: $\mathrm{Exe}(a(\boldsymbol{i}))$ is not "well-defined" in the sense that it is not defined through a formula involving only $\boldsymbol{i}$ and the program parameters. This predicate depends on the specific run of the program, and it should be used only when it is sufficient to know that it is "run-time defined", more accurately run-time valued.

In order to avoid this problem, we will be interested in the condition $\mathrm{Exe}^{\mathsf{S}}$ for a statement to be executed in the *sequential* version. It is well-defined and its expression is rather straightforward for our language subset : it involves the boolean expressions of the embedding IF statements and the bounds of the embedding loops. Since arbitrary variables may occur in these IF and loop statements, this condition $\mathrm{Exe}^{\mathsf{S}}$ depends on the whole data environment.

For a statement a and an index vector $\boldsymbol{i}$ such that the instance $a(\boldsymbol{i})$ is executed (in the sequential version), we may consider the environment $curr\_env@a(\boldsymbol{i})$ in which the execution of $a(\boldsymbol{i})$ takes place. So, for any expression *exp* which happens to be evaluated through the execution of statement a, its value $[\![exp]\!]@a(\boldsymbol{i})$ is defined as $[\![exp]\!](curr\_env@a(\boldsymbol{i}))$.

Let us define $\mathrm{Exe}^{\mathsf{S}}(a(\boldsymbol{i}))$ when a is a statement of the parallel program, and $\boldsymbol{i}$ an index vector for a. Several cases have to be considered, depending on the nesting of a in a loop or in the body of a IF statement. In case a is nested, we consider the innermost loop or IF in the range of which a is nested.

- a is not contained in a loop nor an IF: then, $\mathrm{Exe}^{\mathsf{S}}(a) = true$.
- The innermost nesting of a is in an IF statement c, of boolean expression c.BEXP. Let $\boldsymbol{i}$ denote the index vector of c and a :
  - if a is in the THEN branch, $\mathrm{Exe}^{\mathsf{S}}(a(\boldsymbol{i})) = \mathrm{Exe}^{\mathsf{S}}(c(\boldsymbol{i})) \wedge [\![c.\mathrm{BEXP}]\!]@c(\boldsymbol{i})$

  – if a is in the ELSE branch, $\mathrm{Exe}^{\mathsf{S}}(a(\boldsymbol{i})) = \mathrm{Exe}^{\mathsf{S}}(c(\boldsymbol{i})) \wedge \neg([\![c.\mathrm{BEXP}]\!]@c(\boldsymbol{i}))$

- The innermost nesting of a is in a loop c of lower and upper bound expressions $c.$LB and $c.$UB respectively. Let $\boldsymbol{i}$ denote the index vector of c and $\boldsymbol{i}::j$ denote the index vector of a. We then have: $\mathrm{Exe}^{\mathsf{S}}(a(\boldsymbol{i}::j)) = \mathrm{Exe}^{\mathsf{S}}(c(\boldsymbol{i})) \wedge ([\![c.\mathrm{LB}]\!]@c(\boldsymbol{i}) \leq j \leq [\![c.\mathrm{UB}]\!]@c(\boldsymbol{i}))$

4 Precedences

The sequencing of parallel programs is specified by the sequential subset of the language and by the synchronization primitives. It is a partial order, as opposed to the complete precedence order we are used to in sequential programs. It will be expressed through a "precedence" predicate, and its definition is split into a *sequential precedence* and a *synchronization precedence.*

We wish to define a predicate $\mathrm{Pre}(a(\boldsymbol{i}), b(\boldsymbol{j}))$, expressing that: "If $a(\boldsymbol{i})$ and $b(\boldsymbol{j})$ are both executed, then the overall parallel program structure (sequential control and synchronization) implies that $a(\boldsymbol{i})$ is executed before $b(\boldsymbol{j})$".

It may be of interest to notice that *there may be several non equivalent predicates correctly expressing a precedence relation.* This comes from the following fact : through a predicate $\mathrm{Pre}(a, b)$, we wish to express that *"in case $a(\boldsymbol{i})$ and $b(\boldsymbol{j})$ are executed, the former is executed before the latter ; but we are not interested in what is expressed if one of these instances is not executed."*

Considering two statement instances α and β, for any predicate P correctly expressing that α precedes β in case both are executed, any other predicate Q such that :

$$\mathrm{Exe}(\alpha) \wedge \mathrm{Exe}(\beta) \wedge P \Rightarrow Q \Rightarrow (\mathrm{Exe}(\alpha) \wedge \mathrm{Exe}(\beta) \wedge P) \vee (\neg\,\mathrm{Exe}(\alpha) \vee \neg\,\mathrm{Exe}(\beta))$$

also correctly expresses this. Then, among these possible precedence predicates, it may be suitable to choose one instead of another. In what follows, we will be able to express control precedences through predicates not depending on the specific run of the program.

This same multiplicity of correct predicates will also hold for dependences ; furthermore, it will be straightforward to check that the "dependence preservation" property we will consider is (fortunately...) invariant by any change of correct predicates.

4.1 Control precedence

The control precedence predicate, hereafter denoted Pre^0, expresses precedence relations as they result from the control structure of the program, without considering the effect of synchronizations. As mentioned above, we will give expressions of Pre^0 which do not depend on the specific run considered – in fact, expressions independent of any variables.

Calculating precedences on index vectors

In order to express Pre^0, we have to express the precedence orders between index vectors in loops, denoted $\prec$. We remember that loops are normalized.

Let $\boldsymbol{i}$ be the loop index vector of a statement ; let k be the innermost index in $\boldsymbol{i}$; let $\boldsymbol{j}$ be the (possibly empty) "remaining" index vector, such that $\boldsymbol{i}$ is the concatenation of $\boldsymbol{j}$ with k ; we denote $\boldsymbol{i} = \boldsymbol{j}::k$.

- If k indexes a DO loop :

  $(\boldsymbol{i}_1 \prec \boldsymbol{i}_2) = (\boldsymbol{j}_1 \prec \boldsymbol{j}_2) \vee ((\boldsymbol{j}_1 = \boldsymbol{j}_2) \wedge (k_1 < k_2))$

- If k indexes a PARALLEL DO loop :

  $(\boldsymbol{i}_1 \prec \boldsymbol{i}_2) = (\boldsymbol{j}_1 \prec \boldsymbol{j}_2)$

- *(Starting the recurrence :)* If $\boldsymbol{j}$ is empty – let us denote $[\,]$ the empty index vector –, then we set :

  $([\,]_1 \prec [\,]_2) = \mathit{false}$; $([\,]_1 = [\,]_2) = \mathit{true}$

Now, we can express Pre^0. Let a and b be two statements such that a comes before b in the text of the program. We will give expressions of $\mathrm{Pre}^0(a,b)$ in the different cases. In what follows, we do not need to single out the special case when a and b are in two alternative branches of an IF since, due to the above remark (on multiplicity of correct predicates), the part of $\mathrm{Pre}^0(a,b)$ corresponding to mutually exclusive instances of a and b will be superfluous.

The case when a and b are not in the same PARALLEL SECTIONS

This includes the case when there are no PARALLEL SECTIONS *in the program.*

- a and b are not in the same loop. Then :

  $\mathrm{Pre}^0(a,b) = \mathit{true}$; $\mathrm{Pre}^0(b,a) = \mathit{false}$

- a and b are in the same loop : a indexed by $\boldsymbol{i}::\boldsymbol{j}$; b indexed by $\boldsymbol{i}::\boldsymbol{k}$; $\boldsymbol{j}$ and $\boldsymbol{k}$ disjoint (i.e. indexing distinct loops).

  Then :

  $\mathrm{Pre}^0(a,b) = (\boldsymbol{i}_a \prec \boldsymbol{i}_b) \vee (\boldsymbol{i}_a = \boldsymbol{i}_b)$

  $\mathrm{Pre}^0(b,a) = (\boldsymbol{i}_b \prec \boldsymbol{i}_a)$

The case when a and b are in the same PARALLEL SECTIONS

Whenever several nested PARALLEL SECTIONS contain both a and b, we consider the innermost of them.

Let c be the HEAD of this PARALLEL SECTIONS, and $\boldsymbol{i}$ the index of c.

- a and b are in distinct SECTIONs of this PARALLEL SECTIONS :

  $\mathrm{Pre}^0(a,b) = (\boldsymbol{i}_a \prec \boldsymbol{i}_b)$

  $\mathrm{Pre}^0(b,a) = (\boldsymbol{i}_b \prec \boldsymbol{i}_a)$

- a and b are in the same SECTION :

  a indexed by $\boldsymbol{i}::\boldsymbol{j}::\boldsymbol{k}$; b indexed by $\boldsymbol{i}::\boldsymbol{j}::\boldsymbol{l}$; $\boldsymbol{k}$ and $\boldsymbol{l}$ disjoint.

  Then :

  $\mathrm{Pre}^0(a,b) = (\boldsymbol{i}_a::\boldsymbol{j}_a \prec \boldsymbol{i}_b::\boldsymbol{j}_b) \vee (\boldsymbol{i}_a::\boldsymbol{j}_a = \boldsymbol{i}_b::\boldsymbol{j}_b)$

  $\mathrm{Pre}^0(b,a) = (\boldsymbol{i}_b::\boldsymbol{j}_b \prec \boldsymbol{i}_a::\boldsymbol{j}_a)$

Moreover : expression of $\mathbf{Pre}^0(a, a)$

a indexed by $\boldsymbol{i}$.

$\text{Pre}^0(a_{(1)}, a_{(2)}) = (\boldsymbol{i}_{(1)} \prec \boldsymbol{i}_{(2)})$

4.2 Precedence formula

To obtain the overall precedence relation, we have to combine the control precedence Pre^0 and the synchronization precedence relations realized through POST/WAIT pairs. These synchronization relations will be considered in a moment ; they will be denoted Sync^{S}. Since Sync^{S} will be defined referring to the sequential version, this overall precedence relation we will consider, denoted Pre^{S}, will represent *the precedence relations which would stand in the parallel program, assuming that all variables involved in the definition of these relations get their "sequential" values.* As a consequence, in all further derivations where the use of Pre would be expected, we will have to check the validity of using Pre^{S} instead of Pre.

As another consequence of this definition of Pre^{S}, the precedence relation expressed by Pre^{S} obviously holds in the sequential version. In other words, for any statement instances α and β executed in the sequential version, $\text{Pre}^{\mathsf{S}}(\alpha, \beta)$ implies that α is executed before β in the sequential version.

In order to obtain Pre^{S}, we have to compose Pre^0 with Sync^{S}. This composition is not exactly a transitive closure, as might be expected ; $\text{Pre}^{\mathsf{S}}(a(\boldsymbol{i}), b(\boldsymbol{j}))$ and $\text{Pre}^{\mathsf{S}}(b(\boldsymbol{j}), c(\boldsymbol{k}))$ do not imply $\text{Pre}^{\mathsf{S}}(a(\boldsymbol{i}), c(\boldsymbol{k}))$. Let us consider for instance :

```
      parallel sections (ordered)
      section
p:     post(E)
      section
       if (B)
       then
w:      wait(E)
       else
        A=1
       endif
a:      A=2
      end parallel sections
```

We see that p precedes w, and w precedes a, but p does not precede a, because the ELSE branch may be taken, and without waiting for E, a is then executed concurrently with p. We can only state: if w is executed, then p precedes a.

So, instead of transitivity, we have "transitivity modulo Exe^{S}" :

$$\text{Pre}^{\mathsf{S}}(a(\boldsymbol{i}), b(\boldsymbol{j})) \wedge \text{Exe}^{\mathsf{S}}(b(\boldsymbol{j})) \wedge \text{Pre}^{\mathsf{S}}(b(\boldsymbol{j}), c(\boldsymbol{k})) \Rightarrow \text{Pre}^{\mathsf{S}}(a(\boldsymbol{i}), c(\boldsymbol{k}))$$

The relation Pre^{S} will therefore be obtained, through this transitive closure modulo Exe^{S} along paths, and by disjunction between alternate paths, in a "conjunction in series, disjunction in parallel" manner, from Pre^0 and Sync^{S}. The transitive closure within Pre^0 is taken care of by the previously given expressions of Pre^0. Therefore, the precedence paths to consider in order to obtain $\mathrm{Pre}^{\mathsf{S}}$ alternate Pre^0 and $\mathrm{Sync}^{\mathsf{S}}$ links, in the following way :

$$\alpha \to \pi_1 \rightsquigarrow \omega_1 \to \pi_2 \rightsquigarrow \omega_2 \to \ldots \to \pi_n \rightsquigarrow \omega_n \to \beta,$$

where $\to$ denotes the Pre^0 relation, π_i denotes a POST, ω_i denotes a WAIT, and $\rightsquigarrow$ denotes the synchronization relation $\mathrm{Sync}^{\mathsf{S}}$. (In the special case when α is a POST and/or β is a WAIT, we must also consider paths described by the above formula where "$\alpha \to \pi_1$" is replaced by "$\alpha = \pi_1$" and/or "$\omega_n \to \beta$" is replaced by "$\omega_n = \beta$", respectively.)

The corresponding computation of $\mathrm{Pre}^{\mathsf{S}}$ will be realized through relations such as:

$$\begin{aligned}&\mathrm{Pre}^0(\alpha,\pi_1) \wedge \mathrm{Exe}^{\mathsf{S}}(\pi_1) \wedge \mathrm{Sync}^{\mathsf{S}}(\pi_1,\omega_1) \wedge \mathrm{Exe}^{\mathsf{S}}(\omega_1) \wedge \mathrm{Pre}^0(\omega_1,\pi_2) \wedge \\ &\qquad \ldots \wedge \mathrm{Sync}^{\mathsf{S}}(\pi_n,\omega_n) \wedge \mathrm{Exe}^{\mathsf{S}}(\omega_n) \wedge \mathrm{Pre}^0(\omega_n,\beta) \quad \Rightarrow \quad \mathrm{Pre}^{\mathsf{S}}(\alpha,\beta)\end{aligned}$$

4.3 Synchronization precedence

The elementary synchronization precedence relation between a POST and a corresponding WAIT is much less straightforward to consider than the above Pre^0.

The first difficulty stems from the fact that we allow dynamic variable reference in POST and CLEAR statements. If we wished to consider synchronization relations in a parallel run of the program, we would meet the same problems of "undefinedness" we have met above for the execution predicate Exe.

Therefore, as mentioned above, we will consider the synchronization precedence relation $\mathrm{Sync}^{\mathsf{S}}$, corresponding to the synchronizations which occur provided that the variables involved in the event references get their "sequential" values.

The second difficulty is of a different nature. Through a predicate $\mathrm{Sync}^{\mathsf{S}}(\pi,\omega)$, between a POST instance and a corresponding WAIT instance, we wish, here again, to express that *"if π and ω are both executed, then necessarily π is executed before ω."*. This supposes that no other POST instance ρ is susceptible to trigger the execution of ω, by posting the same event. Indeed, in case several non mutually exclusive POST statement instances will seem able to trigger the execution of one WAIT statement instance, no precedence relation will be guaranteed between any one of these POSTs and this WAIT, and the case will be intractable within our "precedence" framework. (Notice that, conversely, one POST may very well post to several WAITs : this brings no problem in our framework.)

To the extent that one and only one POST statement instance should be able to trigger a WAIT statement instance, it is suitable to require that a CLEAR statement instance, dealing with the same event, not be in a race condition with this POST, nor with this WAIT.

We will express these restrictions through two assumptions, which are not independent, dealing with the use of synchronizations. These assumptions are rather general ; they will be sufficient for the theorem we will prove later. *However*, they presuppose that the synchronization relations $\mathrm{Sync}^{\mathsf{S}}$ are given, which is a weakness for two reasons : we do not provide a general expression of these relations $\mathrm{Sync}^{\mathsf{S}}$; moreover, these two assumptions contribute to the very existence of these synchronization relations, which brings a circularity.

Afterwards, however, we will give expressions of $\mathrm{Sync}^{\mathsf{S}}$ and of these assumptions in a restricted (but still rather general) case.

Assumption S1 (No race condition between POST/WAIT and CLEAR) Consider a CLEAR statement instance θ, and a POST or WAIT statement instance ω, both executed, and involving the same event variable ε, in the sequential version of a program instance. θ and ω must be related in a precedence relation Pre^{S}. Furthermore, in case ω is a WAIT instance and this precedence Pre^{S} goes from θ to ω, $\text{Pre}^{\mathsf{S}}(\theta, \omega)$ must be obtained without using a synchronization in which ω participates, i.e. through precedence paths ending with a Pre^0 link, not with a synchronization link involving ω.

Comment This assumption seems obviously reasonable. Strangely, it does not seem to be a requirement of the X3H5 proposal. Basically, it boils down to a "no race condition" requirement involving event variables, if we consider that WAIT statements read, and POST and CLEAR statements write such a variable.

Assumption S2 (Ensured precedence from POST to WAIT) Supposing Assumption S1, consider a WAIT statement instance γ executed in the sequential version of a program instance. There is at most one POST statement instance π, executed in this sequential version, meeting the following conditions :

- π and γ involve the same event variable ε in this sequential version.
- there is no CLEAR instance θ, executed and involving ε in this sequential version, such that the precedences Pre^{S} implied by Assumption S1 are from π to θ and from θ to γ.
- we have not $\text{Pre}^{\mathsf{S}}(\gamma, \pi)$.

Comment This expresses that at most one POST statement instance is *susceptible to trigger* the execution of the WAIT instance.

We will now give expressions of these assumptions under a restriction : we will consider the case when the precedence relations referred to in Assumption S1 are control precedences Pre^0, and not merely generalized precedences Pre^{S}. First, let us introduce a few notations. For any POST, WAIT or CLEAR statement instance γ executed in the sequential version, we denote $[\varepsilon_\gamma]^S$ the event variable reference γ involves in the sequential version. The relation $\equiv$ between event variable references means that they refer to the same event variable.

Under the mentioned restriction, Assumption S1 can be expressed thus :
For any CLEAR instance θ and any POST or WAIT instance ω :

$$\text{Exe}^{\mathsf{S}}(\theta) \wedge \text{Exe}^{\mathsf{S}}(\omega) \wedge ([\varepsilon_\theta]^S \equiv [\varepsilon_\omega]^S) \Rightarrow \text{Pre}^0(\theta, \omega) \vee \text{Pre}^0(\omega, \theta)$$

Under Assumption S1 thus restricted, Assumption S2 can be expressed as follows. For a POST instance π and a WAIT instance γ, let us define a predicate $\text{Sync}^\star$ as follows :

$$\begin{aligned}\text{Sync}^\star(\pi, \gamma) \quad = \quad & \text{Exe}^{\mathsf{S}}(\pi) \wedge \text{Exe}^{\mathsf{S}}(\gamma) \wedge ([\varepsilon_\pi]^S \equiv [\varepsilon_\gamma]^S) \wedge \neg\, \text{Pre}^{\mathsf{S}}(\gamma, \pi) \wedge \\ & (\forall \text{clear instance } \theta, \text{Exe}^{\mathsf{S}}(\theta) \wedge ([\varepsilon_\theta]^S \equiv [\varepsilon_\gamma]^S) \Rightarrow \text{Pre}^0(\theta, \pi) \vee \text{Pre}^0(\gamma, \theta)\ )\end{aligned}$$

$\text{Sync}^\star(\pi, \gamma)$ expresses that π is susceptible to trigger γ, in our sense.

With this notation, and still under restricted Assumption S1, Assumption S2 can be expressed as follows : for any POST instances π_i and any WAIT instance γ :

$$\text{Sync}^\star(\pi_1,\gamma) \wedge \text{Sync}^\star(\pi_2,\gamma) \Rightarrow \pi_1 = \pi_2$$

in which case $\text{Sync}^\star$ indeed expresses the synchronization relation Sync^{S} we were looking for.

The restriction we have just suggested – the required *control* precedence between event clearing and event use – seems to match a "sound" programming style, in which synchronization pairs are required to be rather clear-cut. On the other hand, it somewhat restricts the interest of allowing dynamic variable reference in POST and CLEAR statements.

It is important to keep in mind that these assumptions, under both the "general" and the "restricted" forms, refer to the semantics of the sequential version and do not depend on some specific parallel run.

5 Dependences

The reader may refer to [2] for the formal calculation of dependences. Let us just make a few reminders and further remarks.

Considering two statements a and b, indexed by $\boldsymbol{i}$ and $\boldsymbol{j}$ respectively, a predicate $\text{Dep}(a(\boldsymbol{i}), b(\boldsymbol{j}))$ will express that : "in case $a(\boldsymbol{i})$ and $b(\boldsymbol{j})$ are both executed in the sequential version, in this order, then they both access one same memory location, at least one of them writing it".

The reference to the sequential version is crucial here, because we will always be interested in the preservation, in the parallel version, of dependences *as they appear in the sequential version.* In other words, the *"dependence implies precedence"* condition must be interpreted as : *"dependence (as appears in the sequential version) implies precedence (ensured in the parallel version)"* This feature is quite classical, and we already considered it in [2].

However, extending the definition of dependences to fit our dynamic variable reference assumption, i.e. the possibility for subscripts to contain variables, brings an extra point which must be emphasized : inputs in an assignment are not only variables involved in the right-hand side, but also variables involved in the left-hand side subscript, if any. (This also holds for event arrays).

Mainly due to this dynamic reference feature we are introducing, it will often be impossible to specify exact dependence relations statically. Then, we will have to seek a conservative approximation of these dependences, i.e. an approximation from above, as we will see below.

6 Some preliminary results

The *dependence preservation* requirement (introduced in [1]) can be expressed as follows:

> If two statement instances $a(\boldsymbol{i})$ and $b(\boldsymbol{j})$ are in a dependence relation $\text{Dep}(a(\boldsymbol{i}), b(\boldsymbol{j}))$ – which implies that $a(\boldsymbol{i})$ comes before $b(\boldsymbol{j})$ in the sequential version –, and if both instances are executed, then the parallel program structure must ensure that $a(\boldsymbol{i})$ is executed before $b(\boldsymbol{j})$.

Using the predicates we have introduced: for all statements a and b and for all parallel executions,

$$\text{Exe}^{\mathsf{S}}(a) \wedge \text{Exe}^{\mathsf{S}}(b) \wedge \text{Dep}(a,b) \Rightarrow \text{Pre}^{\mathsf{S}}(a,b)$$

The aim of this requirement is to avoid *race conditions* : situations when the value received by a variable depends on which of two concurrent statement instances happens to be executed first.

Before proving our theorem in the next section, we will derive some preliminary results.

6.1 Conservative approximations of predicates

In many cases, it will be impossible (even in principle, sometimes) to statically produce exact expressions of the predicates involved here. This is mainly due to the dynamic variable reference and loop bound specifications we are introducing here. Then, we will have to seek *conservative approximations* of these predicates, i.e. approximations such that the use of them, instead of the unknown exact predicates, will never lead us to give a positive conclusion when the preservation property is not met – but may lead to a "don't know" answer in some cases when the property is indeed true.

The direction of the above implication makes clear the kinds of approximations which will be conservative : these will be approximations from above for $\mathrm{Exe}^{\mathbf{S}}$ and Dep, from below for $\mathrm{Pre}^{\mathbf{S}}$: we will then consider predicates $\mathrm{Exe}^{\mathbf{S}\star}$, $\mathrm{Dep}^{\star}$ and $\mathrm{Pre}^{\mathbf{S}}_{\star}$ such that $\mathrm{Exe}^{\mathbf{S}} \Rightarrow \mathrm{Exe}^{\mathbf{S}\star}$, $\mathrm{Dep} \Rightarrow \mathrm{Dep}^{\star}$ and $\mathrm{Pre}^{\mathbf{S}}_{\star} \Rightarrow \mathrm{Pre}^{\mathbf{S}}$, respectively meaning that "a statement instance may be executed", "a dependence may exist" and "a precedence must exist".

The computation of a $\mathrm{Pre}^{\mathbf{S}}$ relation involves predicates Pre^{0}, $\mathrm{Sync}^{\mathbf{S}}$ and $\mathrm{Exe}^{\mathbf{S}}$, through "transitive closure modulo $\mathrm{Exe}^{\mathbf{S}}$ along paths", as we have seen before. Pre^{0} will be rather easily computable ; $\mathrm{Sync}^{\mathbf{S}}$ is very vulnerable to any approximation. So, approximating $\mathrm{Pre}^{\mathbf{S}}$ from below may involve approximating $\mathrm{Exe}^{\mathbf{S}}$ from below, by a predicate $\mathrm{Exe}^{\mathbf{S}}_{\star}$ such that $\mathrm{Exe}^{\mathbf{S}}_{\star} \Rightarrow \mathrm{Exe}^{\mathbf{S}}$; and considering only some of the precedence paths.

6.2 A lemma about execution predicates

It will be useful to derive in which cases, and in what sense, the execution of some statement instance α in a parallel run, strictly depends on the execution of some statement instances β such that $\mathrm{Pre}^{0}(\beta, \alpha)$. This will be the object of the following lemma :

Lemma 1 *Considering a parallel program, for any run of this program, and for any statement instance α of any statement except the first one, the condition, denoted $\psi(\alpha)$, for α to be* reached or pending, *is fully determined by the execution of one or several statement instances β such that* $\mathrm{Pre}^{0}(\beta, \alpha)$. *All or some of these instances β are specified independently of the run considered ; the other ones, if any, are specified by the execution of the former.*

As for the execution of α, we are in one of the following cases :

- *α is an instance of the first statement in a* PARALLEL DO *or in a* SECTION *of a* PARALLEL SECTIONS *: then, $\psi(\alpha)$ expresses the condition for α to be executed or persistently pending ; the latter possibility may occur in case of a deadlock.*

- *α is an instance of a* WAIT *w : then, $\psi(\alpha)$ expresses the condition for α to be reached (or the condition for α to be reached or pending, in case w is both a* WAIT *and the first statement in a parallel construct body). Under this condition, however, α may be persistently waiting (or persistently pending), instead of finally executing, in a deadlock situation.*

- *In all other cases,* $\mathrm{Exe}(\alpha) = \psi(\alpha)$.

Proof : We will refer to the previously specified execution model. We will successively examine all possible cases in our language subset.

We consider some statement instance α, an instance of a statement a other than the first one.

- a is a WAIT : then, in all interesting cases, it is not true that the execution of α depends on instances preceding α Pre^0-wise. But the condition for α to be *reached* – not meaning that it is executed – will conform to everything we will derive now, as shown by fictively inserting a CONTINUE statement just before the WAIT statement, and considering which of the following cases this CONTINUE statement fits in.

- a is the first statement in a PARALLEL DO body : then, let c be the loop header ; let $\boldsymbol{j}$ be the (possibly empty) index vector of c and $\boldsymbol{j}::k$ be the index vector of a. For any instance $\alpha = a(\boldsymbol{j}::k)$ to be executed, it is necessary that the corresponding instance $c(\boldsymbol{j})$ be executed ; conversely, the execution of $c(\boldsymbol{j})$, through the evaluation of its loop bounds, fully characterizes which instances $a(\boldsymbol{j}::k)$ are *reached or pending*. Thus, $\psi(a(\boldsymbol{j}::k))$ is fully characterized by the execution of $c(\boldsymbol{j})$; and we have $\text{Pre}^0(c(\boldsymbol{j}), a(\boldsymbol{j}::k))$. *However*, an instance $a(\boldsymbol{j}::k)$ such that $\psi(a(\boldsymbol{j}::k))$ may be persistently pending instead of finally executing, in deadlock situations.

- a is the first statement in a SECTION body, within a PARALLEL SECTIONS construct : then, let c be the PARALLEL SECTIONS header ; let $\boldsymbol{j}$ be the (possibly empty) index vector of a and c. For any instance $\alpha = a(\boldsymbol{j})$ to be *reached or pending*, it is necessary and sufficient that $c(\boldsymbol{j})$ be executed. So, $\psi(\alpha)$ fully depends on other statement instances which precede it Pre^0-wise. However, there again, α may be persistently pending, in deadlock situations.

At this point, we have just examined the three cases when the execution of a statement instance is not fully determined by the execution of statement instances which precede it Pre^0-wise. However, such a full determination will stand for a WAIT instance to be *reached* – not meaning that it will be executed – ; for an initial instance in a unit of work to be *executed or pending* – not meaning that it will finally be executed – ; and for an initial instance in a unit of work which happens to be a WAIT, to be *pending or reached* – not meaning that it will finally get reached, or executed.

Now, let us examine the other cases.

- a is a END PARALLEL SECTIONS. Then, let c be the corresponding head of the parallel construct, and $\boldsymbol{j}$ the index vector common to c and a. The execution of $a(\boldsymbol{j})$ is fully determined by the execution of the statement instances which end all the units of work in the parallel section. All these instances precede $a(\boldsymbol{j})$ Pre^0-wise. They are specified independently of the run.

- a is a END PARALLEL DO. Then, let c be the corresponding loop head and $\boldsymbol{j}$ be the index vector common to c and a. Then, the condition for $a(\boldsymbol{j})$ to be executed is that $c(\boldsymbol{j})$ be executed and that, in case the index range is not empty (a circumstance determined by the execution of $c(\boldsymbol{j})$), the statement instances which end all the parallel units of work be executed. All these instances precede $a(\boldsymbol{j})$ Pre^0-wise, and are specified by the execution of $c(\boldsymbol{j})$.

- a is the first statement in a DO loop body. Let c be the loop head, $\boldsymbol{j}$ be the index vector of c and $\boldsymbol{j}::k$ be the index vector of a. Then, the execution of $c(\boldsymbol{j})$ fully determines the range of values of k which will be considered, and for any of these values k, the condition for $a(\boldsymbol{j}::k)$

to be executed is the execution of $c(\boldsymbol{j})$ and (for the iterations other than the first) of the last statement instance in the loop body corresponding to the previous iteration. Both statement instances precede $a(\boldsymbol{j}::k)$ Pre^0-wise, and the latter is specified by the execution of the former.

- a is a END DO. Then, let c be the corresponding loop head and $\boldsymbol{j}$ be the index vector common to c and a. Then, the condition for $a(\boldsymbol{j})$ to be executed is that $c(\boldsymbol{j})$ be executed and that, in case the index range is not empty (a circumstance determined by $c(\boldsymbol{j})$), the last loop body instance (specified by the execution of $c(\boldsymbol{j})$) be executed. Both statement instances precede $a(\boldsymbol{j})$ Pre^0-wise.

- a is the first statement in the THEN or ELSE part of a IF. Let c be this IF. The execution of an instance of a is fully determined by the execution of the corresponding instance of c.

- a is a END IF. Let c be the corresponding IF, and $\boldsymbol{j}$ be the index vector common to a and c. The condition for $a(\boldsymbol{j})$ to be executed is the execution of $c(\boldsymbol{j})$ and the execution of some instance – ending the THEN part or the ELSE part – specified by the execution of $c(\boldsymbol{j})$. These two instances precede $a(\boldsymbol{j})$ Pre^0-wise.

- The remaining case is the most straightforward : a has an immediate predecessor b, of same index vector $\boldsymbol{j}$, and the condition for $a(\boldsymbol{j})$ to be executed is exactly that $b(\boldsymbol{j})$ be executed.

◀

This lemma derives its main interest from the previously mentioned observation that Pre^0 is independent of the specific parallel run considered, contrarily to Sync. Its meaning can be summarized as follows : with the exception of the WAIT statements and some other ones susceptible to be pending, the fact that some statement instance is executed in some run of the program depends upon statement instances which *are bound to execute* before it (by the control structure of the program), and not just upon statement instances which merely *happen to execute* before it in some run being considered, as implied by plain causality.

6.3 A notion of time step

In order to derive our main result in the next section, we need to introduce a notion of *time step*. This will be the object of the following result :

Discretized time step lemma : *Considering a parallel program, for any run of this program, we assume the three following properties :*

i. For any statement instance α executed in this run, there exists a time lag $]t_\alpha, t'_\alpha[$ (of physical time), with $t'_\alpha > t_\alpha$, within which the execution of α takes place.

ii. For any statement instances α and β executed in this run, if $\mathrm{Pre}(\alpha, \beta)$, then $t_\beta \geq t'_\alpha$.

iii. Whenever the execution time lags of two statement instances α and β overlap and α outputs a variable x which is an input of β, the value computed by α is not *available as input for β.*

To every statement instance α executed in this run, may be associated a positive integer $\tau(\alpha)$, called the time step of execution of α, *with the following properties :*

1. $\tau(\alpha)$ *nondecreasingly depends on* t_α.

2. For any two statement instances α *and* β *executed, whenever* $\mathrm{Pre}(\alpha,\beta)$, *we have* $\tau(\alpha) < \tau(\beta)$.

3. τ *is* causally defined, *i.e.* $\tau(\alpha)$ *depends only on the execution time lags of* α *and the instances beginning execution before* α.

4. Computational causality : *for any statement instances* α *and* β, *a value output by* α *cannot be used as input by* β *unless* $\tau(\beta) > \tau(\alpha)$.

Comment : Before proving this lemma, let us mention that (iii) is implied by the execution model assumed in the X3H5 proposal [3]. This lemma, and the theorem to follow, would not hold within a different execution model in which a shared memory location could receive a value as soon as this value is computed. We must also mention that, consistently with our definition of the "execution" in case of a WAIT instance, the execution time lag of such an instance does not contain the waiting time ; it does not begin before the involved event has been detected to be *posted*.

Proof : Considering a run of the parallel program, let us rank the (finite) set of statement instances α executed in this run, in the increasing order of the initial times t_α. In case several initial times are equal, we rank the corresponding instances arbitrarily. The sequence of instances obtained thus will be denoted $\alpha_1, \alpha_2, \ldots \alpha_N$. For convenience, the execution time lag for α_i will be denoted $]t_i, t'_i[$.

The time step function τ will be defined by the following procedure :

1. Set $\tau(\alpha_1) = 1$ and $i = 1$.

2. For integers j following i, if any, such that α_j exists and $t_j < \min(t'_k \mid i \le k < j)$, set $\tau(\alpha_j) = \tau(\alpha_i)$

3. If the sequence of instances α is not exhausted yet, let j be the index of the first remaining α. Set $\tau(\alpha_j) = \tau(\alpha_i) + 1$; set $i = j$ and go to [2.].

It is straightforward to check that the function τ thus defined meets the required properties : we notice that instances associated to the same time step have execution time lags which mutually overlap : hence, (iii) implies the computational causality feature ; moreover, any two instances the execution time lags of which are disjoint – especially, any two instances which are in a precedence relation Pre – have different time steps. ◀

As far as program semantics will be concerned, statement instances associated to the same time step will be considered as though they executed "at the same time". Thus, for convenience, we will say that "some statement instance executes at some time step". The causality features (properties 3 and 4) are crucial here : they ensure that the time step at which a statement instance is said to execute does not depend on what may happen after this execution, and that the result of a computation "made at some time step" is not available before the next time step.

The reciprocal of the latter property is not true : having $\tau(\beta) > \tau(\alpha)$ does not imply that an output of α can be used as input by β (besides, the execution time lags of α and β may overlap). But we must point out that, under the execution model we are considering, this availability will stand if $\mathrm{Pre}(\alpha,\beta)$ (provided, of course, that the variable is not computed again meanwhile).

Let us emphasize that we will make use of the mere fact that a time step function *exists* ; we will not need to be able to *effectively compute* it.

6.4 Notion of indirection order

The extension to dynamic variable references we are considering here makes it useful to introduce the notion of *indirection order*. A variable reference will be said to be of indirection order 0 whenever it is a scalar or an array the subscript list of which involves only loop indices and program parameters. Thus, in [2], we restricted ourselves to considering references of indirection order 0.

A reference will be said to be of indirection order $n > 0$ whenever it is an array the subscript list of which involves variable references whose indirection orders are less than or equal to $n - 1$, with equality for at least one of them.

In everyday programs, the indirection order is seldom greater than 2.

7 The theorem of semantic equivalence

We will now derive our result of semantic equivalence. We will make use of Lemma 1 we have derived in the previous section, and Assumptions S1 and S2 dealing with synchronizations. We will also use the notion of time step we have just introduced – therefore, we will assume the hypotheses (regarding the physical time lags of execution) of the discretized time step lemma.

We will also assume another hypothesis, which may be termed as follows :

Assumption A1 All parallel constructs in which there are synchronization statements (i.e. POST, WAIT, CLEAR statements and/or section WAIT clauses) are ORDERED.

Comment This assumption, though not a requirement of the X3H5 proposal, is suggested in it, as a hint towards "good" programming. It can easily be checked at compile time.

A straightforward consequence of Assumption A1 deals with deadlocks. As we already know, a deadlock always involves a WAIT statement the event of which persistently remains un*posted*. When this WAIT occurs in a parallel construct (which is the usual deadlock case), it prevents the completion of this construct ; it may then happen that some pending statement instances, i.e. instances waiting for an available process, turn out to be *persistently pending*. Now, under Assumption A1, this parallel construct is necessarily ORDERED ; hence, *any persistently pending statement instance is ranked after some deadlocking* WAIT *statement instance, in the sequential order.*

Another simple consequence of Assumption A1 deals with section WAITs. The X3H5 proposal requires that a section waiting for another section be lexically ranked after the latter. Under this condition, which can easily be checked at compile time, Assumption A1 implies that a section WAIT cannot trigger a deadlock. More precisely, a section WAIT can indeed be involved in a deadlock situation, whenever some waited for section cannot complete, but such a situation can originate only in an "ordinary" WAIT deadlock.

A **single process** run is a run of the parallel program, obtained when there is only one process available. The X3H5 proposal requires that a compliant parallel program not deadlock whatever the number of available processes. Therefore, a single process run should not deadlock.

We will be interested in the "ordered single process run", defined as the single process run which treats all parallel constructs as if they were ORDERED. Under the assumption that this ordered single process run does not deadlock – an assumption in the theorem below – its behavior matches the one of the sequential version exactly, not only from the point of view of semantic equivalence, but also referring to the execution order of the statement instances. The only difference between the sequential version run and the ordered single process run will be that, in the latter, events are indeed dealt with ; but, due to the assumed absence of deadlock, this does not bring any significant behavioral change. Hence we will refer to the ordered single process run as "sequential run" below.

Theorem 1 *Under the following hypotheses:*

i. Assumption A1 ;

ii. Assumptions S1 and S2 ;

iii. No deadlock in the ordered single process run ;

iv. For all statements a and b, $\mathrm{Exe}^{\mathsf{S}}(a) \wedge \mathrm{Exe}^{\mathsf{S}}(b) \wedge \mathrm{Dep}(a,b) \Rightarrow \mathrm{Pre}^{\mathsf{S}}(a,b)$ *;*

the execution predicate Exe *is well-defined and equals* $\mathrm{Exe}^{\mathsf{S}}$*, and the parallel program is semantically equivalent to its sequential version. Especially, no parallel run can deadlock.*

Proof : We consider a program instance, by giving values to the parameters. Then, there is only one sequential run (in the sense defined above) of this program instance, whereas there are generally several possible runs of the parallel version. We consider one of them. We will first derive the semantic equivalence extended to all statement instances executed in this parallel run and all variables involved in them (points 1 to 4) ; finally, we will prove that, conversely, all instances executed in the sequential run are executed in this parallel run (point 5).

1) We will consider the *time step function* associated to the parallel run we are considering.
Let τ be a time step such that the following recurrence assumption holds:

> *Semantic equivalence up to time step τ:* for any statement instance α executed strictly before τ in this parallel run, α is also executed in the sequential run ; moreover, any variable reference involved in α as input (including the event references) points to the same variable in both runs, and that variable underwent the same computations, due to the same statement instances, in both runs up to the point reached, in the parallel run, just before time step τ.

We wish to prove that this semantic equivalence extends to time step τ. Considering that it obviously applies to the program start, that will ensure the semantic equivalence along all the parallel run.

We have to show that, for any statement instance γ which happens to be executed at time step τ, the semantic equivalence propagates to γ. First, we will prove that the semantic equivalence extends to the *inputs* of γ, in case γ is not a WAIT (point 2), and in the special case when it is (point 3). Then, we will easily show that the semantic equivalence extends to the *outputs* of γ (point 4).

2) In case a statement instance γ, coming to be executed at time step τ, is not a WAIT, according to Lemma 1, the fact that γ is *reached or pending* (not implying that it is executed) is fully determined by some statement instance(s) β such that $\mathrm{Pre}^0(\beta,\gamma)$, and which have all been executed (therefore, before τ). Due to the recurrence hypothesis of semantic equivalence up to time step τ, these same statement instances β execute in the sequential run, and identically determine that γ is reached or

pending in the sequential run. Therefore, γ is executed in the sequential run (due to hypothesis *(iii)*, no instance remains *pending* in the sequential run) : we have $\mathrm{Exe}^{\mathsf{S}}(\gamma)$.

Let us consider some variable reference ξ used by γ as input. In order to ensure the semantic equivalence for this input, since we assume the semantic equivalence up to time step τ, we just need to rule out two possibilities :

1. the possibility that the reference ξ in γ does not point to the same variable in the sequential run ; or, in case it does (let then x be the variable ξ points to in both runs),
2. the possibility that the value of x used by γ as input is not obtained by the same computations in both runs.

We will first show, by a recurrence on the indirection order of ξ, that ruling out Possibility 1 reduces to ruling out Possibility 2. Possibility 1 cannot arise if ξ is of indirection order 0, since such a reference statically points to the same variable (i.e. memory location) in any run. Now, if ξ is of indirection order $n > 0$, let us suppose that possibilities 1 and 2 have been ruled out for all inputs of γ of indirection order less than n. Then, the semantic equivalence extends to all such inputs, and especially to all variable references contained in the subscript list of ξ. Therefore, ξ points to the same variable x in both runs, and it is then sufficient to rule out Possibility 2 for this input x.

So, considering a variable reference ξ pointing to the same variable x in both runs, we have to rule out Possibility 2 by making sure that the value of x used by γ as input has been similarly computed in both runs.

Let β be the statement instance which computes the value of x used by γ as input *in the sequential run.* β exists, due to the determinacy condition. We will first show that $\mathrm{Exe}(\beta)$ and $\mathrm{Pre}(\beta, \gamma)$, which will imply that β is executed in this parallel run before τ. The recurrence hypothesis of semantic equivalence will then imply that x is similarly computed by β in both runs, and the precedence thus obtained will imply that this value of x is available as input for γ in the parallel run, *unless* some other computation of x interferes between β and γ, a circumstance that we will rule out afterwards.

We have $\mathrm{Exe}^{\mathsf{S}}(\beta)$ because it is β which computes x for γ in the sequential run; we have $\mathrm{Exe}^{\mathsf{S}}(\gamma)$ as we have seen ; we have $\mathrm{Dep}(\beta, \gamma)$ because β computes a variable used by γ *in the sequential run* on which Dep is defined. Therefore, according to *(iv)*, we get $\mathrm{Pre}^{\mathsf{S}}(\beta, \gamma)$. This is not sufficient for immediately implying that β be executed before time step τ, because we must remember that $\mathrm{Pre}^{\mathsf{S}}$ is defined referring to the sequential run. (We do not even know, as yet, whether β is executed at all in this parallel run...) However, a closer look will allow us such an implication, which will result from the following, more general, lemma.

Lemma 2 *We assume the hypothesis of semantic equivalence up to time step τ, and a statement instance γ executed at τ, as above. For any statement instance α such that $\mathrm{Exe}^{\mathsf{S}}(\alpha)$ and $\mathrm{Pre}^{0}(\alpha, \gamma)$, we have $\mathrm{Exe}(\alpha)$ and $\mathrm{Pre}(\alpha, \gamma)$. More generally, for any statement instance α such that $\mathrm{Exe}^{\mathsf{S}}(\alpha)$ and $\mathrm{Pre}^{\mathsf{S}}(\alpha, \gamma)$, we have $\mathrm{Exe}(\alpha)$ and $\mathrm{Pre}(\alpha, \gamma)$, with the following restriction : if γ is a* WAIT *instance, the precedence $\mathrm{Pre}^{\mathsf{S}}(\alpha, \gamma)$ must be obtained without using a synchronization $\mathrm{Sync}^{\mathsf{S}}$ in which γ participates.*

These results also hold if we replace γ here by any statement instance executed before τ in this run.

Proof (of the lemma). We will prove the result involving γ ; the latter extension will be straightforward. After giving a preliminary remark, we will prove the part involving Pre^0 ; then, we will derive the extension to $\text{Pre}^{\mathbf{S}}$.

Preliminary remark. In a deadlock situation, let α be a persistently waiting or pending statement instance. No instance β such that $\text{Pre}^0(\alpha, \beta)$ can be executed. This straightforwardly results from the execution model and the definition of Pre^0.

Considering Pre^0. We straightforwardly have $\text{Pre}(\alpha, \gamma)$, because the control precedence Pre^0 is common to all runs. Suppose that some instance α executed in the sequential run and such that $\text{Pre}^0(\alpha, \gamma)$, is not executed in this parallel run (then, α is clearly not the first statement in the program : Lemma 1 applies to α). We will derive that, in this case, some other instance α_1 such that $\text{Pre}^0(\alpha_1, \alpha)$ is also executed in the sequential run but not in this parallel run, which will then lead to a contradiction.

In case α is not a WAIT, according to Lemma 1, $\text{Exe}^{\mathbf{S}}(\alpha)$ depends on some instance(s) β_i executed in the sequential run, and such that $\text{Pre}^0(\beta_i, \alpha)$, hence $\text{Pre}^0(\beta_i, \gamma)$ (here, remember that, according to *(iii)*, there is no *persistently pending* statement instance in the sequential run). Therefore, if all these β_i were executed in this parallel run, they would be executed before time step τ, hence the semantic equivalence, which would imply that α be reached or pending. So, in the parallel run, the non execution of α would imply one of two things. Either all β_i are indeed executed in this parallel run but α is persistently pending, thus participating in a deadlock situation. This possibility is ruled out by the above preliminary remark : having $\text{Pre}^0(\alpha, \gamma)$ would prevent γ from being executed, as it is assumed to. There remains the possibility that at least one of these β_i is not executed in this run : let it be denoted α_1.

In case α is a WAIT, there are two possibilities. If α is not reached, we apply the above reasoning, making use of Lemma 1, and similarly find a "preceding" α_1. If α is reached, the fact that it is not executed means that there is a deadlock on α. This possibility is ruled out by the above preliminary remark : having $\text{Pre}^0(\alpha, \gamma)$ would prevent γ from being executed, as it is assumed to.

Thus, assuming that α is not executed in this parallel run implies that some other instance, α_1, preceding α Pre^0-wise, is also not executed in this run, though it is in the sequential run.

This argument may be repeated for α_1 : thus, we would find an infinite sequence ($\alpha_0 = \alpha$, α_1, α_2,...) such that every α_i would be executed in the sequential run and preceded (Pre^0-wise) by the next one in the sequence. This contradicts the simple fact that there are a finite number of time steps between the program start and any step it reaches, in any run[2].

Extending to $\text{Pre}^{\mathbf{S}}$*, with the mentioned restriction.* Suppose that some instance α such that $\text{Pre}^{\mathbf{S}}(\alpha, \gamma)$ and not $\text{Pre}^0(\alpha, \gamma)$ is executed in the sequential run. $\text{Pre}^{\mathbf{S}}(\alpha, \gamma)$ is realized through synchronizations, i.e., as previously explained, through one or several paths of the form:

$$\alpha \to \pi_1 \text{ or } \alpha = \pi_1$$

$$\pi_1 \rightsquigarrow \omega_1 \to \pi_2 \rightsquigarrow \omega_2 \to \cdots \to \pi_n \rightsquigarrow \omega_n$$

$$\omega_n \to \gamma$$

where, again, $\to$ denotes a Pre^0 relation, π_i denotes a POST, ω_i denotes a WAIT, and $\rightsquigarrow$ denotes a synchronization link $\text{Sync}^{\mathbf{S}}$; moreover, all the π_i and ω_i are executed in the sequential run

[2]In this reasoning, it is crucial to have $\text{Exe}^{\mathbf{S}}(\alpha_i)$, together with $\text{Pre}^0(\alpha_{i+1}, \alpha_i)$, to obtain the contradiction, since the ordering Pre^0 is not *well-founded* (because of the infinity of statement instances generated in loops).

(remember the "transitive closure modulo $\mathrm{Exe}^{\mathsf{S}}$ " involved in $\mathrm{Pre}^{\mathsf{S}}$). We have $\omega_n \to \gamma$ and not $\omega_n = \gamma$ because of the restriction we introduce : we exclude the case when γ is a WAIT instance and participates in a relation $\mathrm{Sync}^{\mathsf{S}}$ involved in $\mathrm{Pre}^{\mathsf{S}}$.

We have $\omega_n \to \gamma$, i.e. $\mathrm{Pre}^0(\omega_n, \gamma)$; therefore, according to the first part of this lemma, ω_n is executed before time step τ in this run. By upward recurrence, we will prove that all π_i and ω_i, and finally α, are executed before τ in this run. Let us assume that ω_i is executed before τ. Then, the recurrence hypothesis applies to ω_i and any variable involved in ω_i, i.e. in the event involved in ω_i, ε_{ω_i} : all the past computations of ε_{ω_i} at and before the execution of ω_i are identical in both runs. So, since $\mathrm{Sync}^{\mathsf{S}}(\pi_i, \omega_i)$, π_i was executed in this parallel run before ω_i and posted this same event for ω_i, and we have $\mathrm{Pre}(\pi_i, \omega_i)$.

Let us now consider the case $i > 1$ and derive the execution of ω_{i-1}. Since $\mathrm{Pre}^0(\omega_{i-1}, \pi_i)$, and $\mathrm{Exe}^{\mathsf{S}}(\omega_{i-1})$, according to the first part of the lemma, ω_{i-1} was executed before π_i, hence before time step τ, in this run. Thus, we conclude that ω_1 is executed before τ in this run. The above reasoning then ensures that π_1 too is executed before τ in this run. Now, we have either $\mathrm{Pre}^0(\alpha, \pi_1)$ (and $\mathrm{Exe}^{\mathsf{S}}(\alpha)$), or $\alpha = \pi_1$, which implies that α is indeed executed before τ in this run ; furthermore, we have $\mathrm{Pre}(\alpha, \gamma)$, by transitive closure modulo Exe.

◀

So, having $\mathrm{Exe}^{\mathsf{S}}(\beta)$ and $\mathrm{Pre}^{\mathsf{S}}(\beta, \gamma)$, together with the fact that γ is not a WAIT, implies that β is executed in this run, and that $\mathrm{Pre}(\beta, \gamma)$. This is what we had to prove. Now, in order to confirm that the semantic equivalence extends to the input x of γ, we need to prove that, in the parallel run, the statement instance which computes x for γ is indeed β, and not some other statement instance δ interfering between β and γ. δ would execute before τ, so that its output x would be available for γ ; so, due to the semantic equivalence hypothesis, δ would execute in the sequential run too, after β, and compute the same variable x. Since β indeed computes x for γ in the sequential run, this would imply that δ executes *after* γ in the sequential run. If this were the case, we would have $\mathrm{Dep}(\gamma, \delta)$, which, together with $\mathrm{Exe}^{\mathsf{S}}(\delta)$ and $\mathrm{Exe}^{\mathsf{S}}(\gamma)$, would imply $\mathrm{Pre}^{\mathsf{S}}(\gamma, \delta)$. δ executes in the parallel run before τ, so the latter extension of Lemma 2 applies to δ : noticing that δ is not a WAIT (because it has an output), $\mathrm{Exe}^{\mathsf{S}}(\gamma) \wedge \mathrm{Pre}^{\mathsf{S}}(\gamma, \delta)$ would imply $\mathrm{Pre}(\gamma, \delta)$, which would contradict the fact that γ executes no earlier than τ.

3) In case a statement instance γ, coming to be executed at time step τ, is a WAIT, according to Lemma 1, the fact that γ is *reached or pending* (not implying that it is executed) depends on instance(s) β such that $\mathrm{Pre}^0(\beta, \gamma)$, and which have been executed, therefore before τ. Due to the recurrence hypothesis, γ is also reached in the sequential run – and executed, because of hypothesis *(iii)*. So, we have $\mathrm{Exe}^{\mathsf{S}}(\gamma)$.

Let ε_γ be the event involved in γ in the parallel run. Due to the assumption of static reference in WAIT statements, this same event is also involved in the sequential run, and it is the only variable input to γ. Due to the semantic equivalence hypothesis, all computations of ε_γ before τ are identical in both runs. Let ρ be the statement instance which last wrote the variable ε_γ in this past common history. Since γ is executed at time step τ in the parallel run, ρ exists and is a POST (and not a CLEAR).

Let π be the statement instance which writes ε_γ for γ in the sequential run. Since γ is executed, π is a POST and (π, γ) is a synchronization pair : we have $\mathrm{Sync}^{\mathsf{S}}(\pi, \gamma)$.

We have to show that π is ρ. If this were not the case, Assumptions S1 and S2 would imply that, either $\mathrm{Pre}^{\mathsf{S}}(\gamma, \rho)$ (then, ρ would execute *after* γ in the sequential run), or there would be a

CLEAR instance θ involving event ε_γ in the sequential run, such that $\text{Pre}^{\mathsf{S}}(\rho,\theta) \wedge \text{Pre}^{\mathsf{S}}(\theta,\gamma)$. Let us successively rule out these two cases.

> *The case* $\text{Pre}^{\mathsf{S}}(\gamma,\rho)$. Since ρ executes before τ in the parallel run, Lemma 2 applies : noticing that ρ is not a WAIT, $\text{Exe}^{\mathsf{S}}(\gamma) \wedge \text{Pre}^{\mathsf{S}}(\gamma,\rho)$ would imply $\text{Pre}(\gamma,\rho)$, which would contradict the fact that ρ executes before γ in this parallel run.
>
> *The case* $\text{Pre}^{\mathsf{S}}(\rho,\theta) \wedge \text{Pre}^{\mathsf{S}}(\theta,\gamma)$. Assumption S1 would furthermore imply that $\text{Pre}^{\mathsf{S}}(\theta,\gamma)$ is obtained without using a synchronization involving γ. So, according to Lemma 2, having $\text{Exe}^{\mathsf{S}}(\theta)$ and $\text{Pre}^{\mathsf{S}}(\theta,\gamma)$, with this restriction, would imply that θ is executed before γ in the parallel run. Then, the fact that $\text{Pre}^{\mathsf{S}}(\rho,\theta)$ would imply a similar precedence in the parallel run (due to Lemma 2 applied to θ) : ε_γ would be cleared by θ between ρ and γ in this parallel run, which would contradict the fact that ρ is the last statement instance writing ε_γ before τ.

4) At this point, we have proved that, given the recurrence hypothesis of semantic equivalence up to time step τ and a statement instance γ executed at τ in the parallel run being considered, γ is also executed in the sequential run and the semantic equivalence extends to all the input references of γ : any such reference ξ points to the same variable (denoted x) in both runs, and x contains the same value, similarly computed, at the execution of γ, in both runs.

This input equivalence implies that any output reference η of γ points to the same variable (denoted y) in both runs. To make sure that the semantic equivalence extends to the output y just after time step τ, it is sufficient to check that there is no conflict, i.e. no dependence relation, among the statement instances γ_i coming to be executed at time step τ in the parallel run. Due to the semantic equivalence of all input references, input variables and output references of these γ_i, such a dependence would also stand in the sequential run, as a Dep relation, which, according to *(iv)*, would imply a precedence Pre^{S} between two instances γ_i, say $\text{Pre}^{\mathsf{S}}(\gamma_1,\gamma_2)$. If γ_2 is not a WAIT, Lemma 2 applies : having $\text{Exe}^{\mathsf{S}}(\gamma_1)$ and $\text{Pre}^{\mathsf{S}}(\gamma_1,\gamma_2)$ would imply $\text{Pre}(\gamma_1,\gamma_2)$, which contradicts the fact that γ_1 and γ_2 execute at time step τ. If γ_2 is a WAIT involving an event reference ε, $\text{Dep}(\gamma_1,\gamma_2)$ means that γ_1 is a POST or CLEAR writing ε (in both runs, due to the semantic equivalence of references). Then, $\text{Pre}^{\mathsf{S}}(\gamma_1,\gamma_2)$ implies that γ_1 executes in the sequential run before γ_2. However, we have just seen (point 3) that some POST π, executing before τ in this run (hence distinct from γ_1), is the last statement instance writing ε before γ_2, in both runs. So, π executes after γ_1 in the sequential run. This contradicts the recurrence hypothesis, that ε undergoes the same computations, due to the same instances, in both runs up to the execution of π.

5) We have thus proved that any statement instance executed in some parallel run is also executed in the sequential run, and that any variable involved in this statement instance undergoes the same computations (and therefore receives the same values) in both runs up to the last point reached in this parallel run.

There remains to prove that, conversely, any statement instance executed in the sequential run is also executed in any parallel run. Let us suppose by contradiction that there are statement instances which are executed in the sequential run and not in some parallel run we are considering, and let α be the earliest one, in the sequential order.

In case α is not a WAIT, according to Lemma 1, the execution of α in the sequential run is dependent on statement instances β which precede α Pre^0-wise and are all executed in the sequential run, before α. By definition of α, these β are executed in this parallel run, with semantic equivalence, as shown previously. Therefore, α is executed in this parallel run, *unless* it is persistently pending. The latter possibility is ruled out by Assumption A1 : we have previously mentioned that a statement instance may be persistently pending only if some WAIT instance ω, ranking before it in the sequential order, is reached and deadlocks in the parallel run. Being reached, ω is reached too, and executes, in the sequential run. Its deadlock in the parallel run, and its preceding α in the sequential run, would contradict the definition of α.

The case when α is a persistently pending or unreached WAIT is ruled out by the same argument.

There remains the case when α is a reached and deadlocking WAIT. Let ε_α be the event involved in α (this event is the same in both runs, due to the assumption of static reference in WAIT statements). All instances previous to α in the sequential run are executed – with semantic equivalence – in this parallel run. This is the case, therefore, for the POST instance π which sets ε_α for α in the sequential run. So, the deadlock on α would imply that, in the parallel run, some CLEAR statement instance θ executes and clears ε_α after π, and before α used it. θ also executes in the sequential run and clears the same event (as shown above, about the semantic equivalence extended to all instances executed in the parallel run).

Assumption S1 implies that there is a relation $\mathrm{Pre}^{\mathsf{S}}$ between θ and π, and between θ and α. We cannot have $\mathrm{Pre}^{\mathsf{S}}(\theta,\pi)$ because this would imply (Lemma 2 applied to π) that θ executes before π in the parallel run. So, we have $\mathrm{Pre}^{\mathsf{S}}(\pi,\theta)$. We cannot have $\mathrm{Pre}^{\mathsf{S}}(\theta,\alpha)$ because, together with $\mathrm{Pre}^{\mathsf{S}}(\pi,\theta)$, this would imply an execution order in the sequential run : π before θ before α, and π would not post for α, as it is supposed to. So, we have $\mathrm{Pre}^{\mathsf{S}}(\alpha,\theta)$.

θ executes in the parallel run, so Lemma 2 applies to θ : α executes in the sequential run and we have $\mathrm{Pre}^{\mathsf{S}}(\alpha,\theta)$; furthermore θ is not a WAIT ; so, α executes in the parallel run, before θ, which contradicts the deadlock on α.

This ends the derivation of our theorem.

◀

8 Possible extensions

Extending dynamic reference to the WAITs ?

When considering synchronization precedences and the predicate $\mathrm{Sync}^{\mathsf{S}}$, we noticed that the assumption of dynamic variable reference in POST and CLEAR statements turns out not to be so interesting as one could believe, for reasons linked with the necessity to check that Assumptions S1 and S2 are met. For this reason, it is not necessarily interesting to extend this dynamic variable reference to WAIT instances. Such an extension would besides bring problems illustrated by the following example :

```
n1:   N=1
      parallel sections (ordered)
      section
       ...
n2:    N=2
p:     post(E(N))
       ...
      section
       ...
w:     wait(E(N))
       ...
      end parallel sections
```

The underlying intention is to have a synchronization from p to w. Indeed, referring to our definitions, we have $\mathrm{Pre}^{\mathsf{S}}(n2, w)$, i.e. a precedence between $n2$ and w *assuming that variables get their "sequential" values* ; however, the dependence $\mathrm{Dep}(n2, w)$ is not preserved, because N may have value 1 when w is reached. Extending dynamic variable reference to WAIT statements would require an extra condition : *whenever a dependence involves a subscript variable reference in a* WAIT*, it must be preserved through a precedence which does not involve a synchronization in which this* WAIT *participates.* Such an extension would also require that another point be specified in the semantics of WAIT statements : when a WAIT statement instance is reached, are subscript variables (if any) evaluated only once, or reevaluated each time the instance is tried again ? In the latter case, under the extra condition that event $E(1)$ is not posted in the above example and N is not written otherwise than indicated, the semantic equivalence is preserved here. Dealing with such complexities does not seem interesting in our framework.

Introducing WHILE features

Though the WHILE construct does not exist in standard Fortran, it may be interesting to consider such a feature, because it exists in many other languages and it allows for the possibility of infinite loops. A Fortran equivalent of a while may be constructed as follows :

```
C  Fortran equivalent of a while
    1  if(B) then
       ...
       B=...
       ...
       goto 1
       endif
```

Extending our language subset so as to include this construct would boil down to allowing GOTOs in the special case exemplified here – adding the requirement that there would be no synchronization link between the WHILE construct and the rest of the program. In this extension, it would be rather straightforward to prove an extended theorem of semantic equivalence : provided that the sequential run terminates (i.e. no infinite loop in the WHILE constructs), so does any parallel run. This is done by applying the previous recurrence derivation, considering that every statement in the WHILE construct, including the IF, is instanciated in an infinite sequence of instances (numbered 1, 2, 3,...), only a finite number of them coming to be executed in a given run. The semantic equivalence propagates along the considered parallel run, thus ensuring that every WHILE construct terminates similarly in both the given parallel run and the sequential run.

Introducing subroutines and functions

Introducing subroutine and function calls would be rather straightforward, provided that we can specify exactly what are the inputs, and the outputs, of every call. Then, every call may be treated as a statement. This requires that all variables within the subroutine, except the specified inputs and outputs, are strictly local. Especially, there must be no synchronization link between the subroutine and the rest of the program. However, parallelization within the subroutine is then allowed.

9 Conclusion

In this report, we have presented a semantic analysis of a subset of Fortran X3H5. Especially, we have derived a theorem of semantic equivalence which extends, and makes more precise, a result outlined in a previous report [2]. This result states the semantic equivalence (i.e. similarity in observable behavior) between a parallel program and its sequential version, that we define, under a few assumptions (mainly preservation of dependences) referring to the semantic behavior of the sequential version (more accurately, of a one-process execution of the program).

Though presented in the specific framework of Fortran X3H5, this analysis could apply to other parallel shared-memory languages, however under such restrictions as those we have described. The main restriction is the existence of a "sequential version" whose well-defined semantics provides the reference for the observable behavior we wish the parallel program to display. Another important restriction deals with the explicit synchronizations we allow in our language subset, and especially with the requirement that some assumptions about them be checkable. In case the synchronizations are used in some program in an intricate way, this intricacy may make it impossible to check the required assumptions within our framework. However, we feel that introducing such complexities in synchronization handling does not match a "sound" programming style, and that future efforts at extending our results should not address synchronization handling in intricate cases, but rather handling other kinds of synchronization not addressed here, such as critical sections, locks, rendezvous synchronizations.

Future efforts should also aim at developing a formalism, for the expression of parallel execution models, within which such correctness results could be made more rigorous and more general.

References

[1] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of event synchronization in a parallel programming tool. In *2nd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 21–30, Seattle, march 1990. ACM Press.

[2] G. Caplain, R. Lalement, and T. Salset. Semantic analysis of a control-parallel extension of Fortran. Technical Report 93-18, CERMICS, 1993.

[3] X3H5. *FORTRAN 77 Binding of X3H5 Model for Parallel Programming Constructs.* ANSI, September 1992. (draft version).