Correctness properties in a shared-memory parallel language

Gilbert Caplain*

Abstract

We study a property of *correctness* of programs written in a shared-memory parallel language. This property is a semantic equivalence between the parallel program and its *sequential version*, that we define. The language we consider is a subset of a standard parallel language. Within this subset, this correctness property follows from the preservation of data dependences by the control flow and the synchronizations. Our result makes use of the semantics of the sequential version only. Hence, through our result, checking the correctness of some *parallel* program boils down to verifying properties of some *sequential* program.

1 Introduction

In the field of parallel programming, an important trend has been to provide program designers with automatic parallelizers which transform a sequential program source into a parallel program source in such a way that some correctness property remains true along the way. However, another possible approach consists in considering "directly" a parallel program source and wondering whether it meets some correctness specification. The latter approach may reveal interesting, for example, for a programmer who would design a program "directly" in a parallel form, in order to get a better performance and/or a good understanding of the parallelization obtained.

In view of this approach, it may be useful to provide a tool to statically verify parallel programs. In this paper, we present results which could be applied in the design of such a verification tool.

The programming model we address in this paper is the *shared-memory asynchronous model (MIMD-SM)*, as opposed to the *distributed-memory* one *(MIMD-DM)*. However, some recent developments allow to apply the shared-memory programming model on distributed message-passing architectures [15], through functionalities which allow to compile a shared-memory *program* towards a network of interconnected workstations (i.e. a distributed-memory *device*), hence an access to extended resources, without the extra programming burden classically associated to distributed-memory programming. Moreover, there are a few hints of an increasing interest in the shared-memory model these last years: let us mention the OpenMP standard [22], and some recent research works [27]. Such developments might contribute to popularize the shared-memory programming model which we consider here.

We study a property of *semantic correctness* of programs written in a shared-memory parallel language. Various semantic correctness properties have been considered in the literature.

*CERMICS, Ecole Nationale des Ponts et Chaussées, 6 & 8 avenue Blaise Pascal, Cité Descartes, Champs-sur-Marne, F–77455 Marne-La-Vallée Cedex2, France. **Email:** *caplain@cermics.enpc.fr* . This work was developed within a project undertaken together with René Lalement and Thierry Salset (same address). I am grateful to both of them for helpful comments.

Sequential consistency of a multiprocess device is the requirement that the result of any execution of a parallel program should be the same as if all operations executed by the several processes had been executed sequentially in a certain strict (unspecified) order compatible with the execution order of operations of every process [18]. **Linearizability** is a more stringent requirement, derived from sequential consistency by adding the constraint that the "equivalent" sequential execution of every operation lie within a specified time interval [14]. **Serializability** is a correctness condition roughly similar to sequential consistency, adapted to the study of *database systems* [24] liable to be accessed and modified by several *users* in parallel.

Sequential consistency and linearizability can be viewed as required properties of a memory management system in a shared-memory parallel machine. One may consider testing of these properties through an experimental study of some run of a parallel program. The complexity of such a test has been thoroughly studied in [12]: the general problem is NP-complete. An analogous complexity result has been obtained for the general problem of serializability [24].

The **sequential correctness** requirement that we will consider is different from the properties we have just alluded to. In those properties, it was required that any parallel run of some program be similar (in its observable effects) to *some, unspecified,* sequential run of this program. The property we will consider is more stringent, in the following sense: in the framework we are considering, a parallel program is viewed as a parallelization of some *given* sequential program – basically, this parallelization will consist in parallelizing loops and introducing event synchronizations –, and we require that the results of any run of the parallel program be identical to those of *the sequential program being considered*, and not merely to the results of "some possible sequential run". The property we are considering is a *semantic equivalence* between the parallel program and its *sequential version.* The improvement sought through the parallelization, in this context, lies only in the ability to run the program faster, by allowing several parts of it to be executed simultaneously, on several available processors. Considering this sequential correctness property seems relevant especially in many *scientific computing* applications.

The main purpose of this paper is to present and prove a theorem which states sufficient conditions for this semantic equivalence property. This result applies to a fairly general shared-memory parallel language. Although the corresponding general correctness problem is undecidable, there are prospects that our theorem may be applied through tools to verify a wide range of parallel programs. A preliminary application, dealing with a subset of our language, is developed in [7, 26].

Here we are interested in *static*, i.e. compile-time, correctness checking. Indeed, since there is an inherent indeterminism in the behavior of a parallel program, we could not rely on an observation of *one* run of this parallel program: we seek a proof that *any* possible run of this program will deliver correct results, in our sense. Besides, this static character will allow us to consider *parametered* programs, i.e. classes of programs differing from one another by the values of parameters.

For the sake of brevity, some extensions of our theorem (e.g. introducing *critical sections*) are not developed here; they are introduced in [4, 5].

Some of the basic concepts we use have been introduced in [3]. In the development of shared-memory parallel programs, the most difficult challenge is to avoid *data races*, a circumstance which corresponds to *data dependences.* A data dependence links a pair of accesses to the same variable (memory location) when at least one of these accesses is a write. In order to ensure that the parallelized program displays the same results as its sequential counterpart, we must check that

every data dependence is *preserved*, i.e. that the two corresponding accesses operate in the suitable order (this is the *dependence implies precedence* requirement).

This *dependence preservation* paradigm is fairly well-known, to the point that some of its intricacies may not be apparent at first sight. What does "dependence implies precedence" precisely mean? In first approximation, it should be interpreted as "dependence (defined in the sequential version) implies precedence (ensured during a run of the parallel version)". But, what if some statements executed in the sequential version are not executed in some parallel run? or if their execution in this parallel run does not involve the same variables as in the sequential run? We will not limit ourselves to *static control* programs, a limitation adopted very often in the literature (see e.g. [11]): in the language we will consider, loop bounds and subscript expressions in arrays may contain variables, which may influence the condition for a statement to execute and the designation of the inputs and outputs of that statement. As long as we have not proved the semantic equivalence we are interested in, therefore, such intricacies imply that a "dependence implies precedence" requirement *has no well-defined meaning* at this point.

As a way to deal with these intricacies, in our search for a static correctness criterion, the semantic equivalence theorem we present here states sufficient conditions *which refer to the semantics of the sequential version only,* i.e. involve predicates that are defined on the sequential version.

This reference to the sequential version has an interesting consequence regarding the possibility to use *dataflow analysis* in the process of applying our theorem. Dataflow analysis (see e.g. [20, 30, 21, 1, 11, 8, 9]) is inherently adapted to the study of data flows in a sequential program; it cannot be straightforwardly transposed within a parallel context, especially due to the crucial use of the strict time ordering in sequential execution, which is not preserved in a parallel run (let us however mention that some recent works, e.g. [10], develop some kinds of dataflow analyses for concurrent programs). However, due to our theorem, checking the semantic equivalence of a parallel program boils down to checking some properties of the *sequential version* of this program, i.e. *verifying properties of some sequential program*, hence the possibilities to use the resources of dataflow analysis when applying our theorem.

In Section 2, we describe the language we study – a standard imperative language with a few parallel features – and its execution model; then, we set up the correctness problem: a parallel program and its sequential version must have the same observable behavior *(semantic equivalence)*. In Section 3, we study data dependence and precedence relations. In Section 4, we derive some preliminary results.

In Section 5, we derive our theorem of semantic equivalence: assuming the preservation of dependences by the precedence relations *as defined under the semantics of 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.

Section 6 illustrates an application of our theorem on a little example, together with highlighting an interesting *incremental* property of a check-and-repair procedure making use of our theorem.

2 The language studied

2.1 Outline

The language we study is a parallel extension of a standard language[1].

On the sequential side, we have assignment, variables of integer, real, boolean types; variable arrays of these types (the subscript expressions in array references have type *integer*); usual arithmetic and logical operations. It would be straightforward to extend our results so as to include more complex, structured variable types, and pointers as well (as in C++, for instance), but applying our results may reveal more intricate then, especially when considering pointers. (The general concern here, is that the input/output references in a statement should be rather "clear-cut".)

We include the following structured statements:

- Static loops, denoted:

  ```
  DO  <index>=<lower_bound>,<upper_bound>  <statement_list>  ENDDO
  ```

  The bounds are evaluated at entry, and not reevaluated at every iteration. (It is in this sense that the loop is said to be *static*.) The index cannot be written within the loop. For the sake of convenience, loops are *normalized*, i.e. their increment is set to 1.

- Conditionals, denoted:

  ```
  IF  <test>  THEN  <statement_list>  ELSE  <statement_list>  ENDIF
  ```

- Dynamic loops, denoted:

  ```
  WHILE  <test>  <statement_list>  ENDWHILE
  ```

These structured statements may be nested.

We do not include GOTOs. It is a well-known result that any sequential algorithm can be implemented without using GOTOs; therefore, ruling them out is not a restriction here.

Our language contains subroutine and function calls, however with three important restrictions: all such calls should terminate; outputs should be functions only of inputs *(determinacy)*; input/output exchanges should occur only at the call (for inputs) and at the return (for outputs) – in other words, the call must be comparable to a simple statement execution as regards value exchanges (More on this later).

We introduce the following parallel features:

- Parallel static loops, denoted PDO, specify that iterations in the loop may execute in parallel.

  ```
  PDO  <index>=<lower_bound>,<upper_bound>  <statement_list>  ENDPDO
  ```

[1] Our previous work ([6, 7]) was developed in the framework of the Fortran X3H5 proposal [31, 23], but the result presented here is valid in a more general framework.

- Parallel sections, denoted:

```
PSECTIONS    SECTION  <statement_list>
             SECTION  <statement_list>
             ........
ENDPSECTIONS
```

  specify that several sections of code may execute in parallel.

- Explicit synchronizations: we consider *event variable synchronization* through POST/WAIT pairs.

```
POST ( <event_reference> )
WAIT ( <event_reference> )
CLEAR ( <event_reference> )
```

  These three statements are the only ones accessing the event variables. A POST (resp. a CLEAR) sets value *posted* (resp. value *cleared*) to the event variable it refers to. The event variables are initialized at *cleared*. A WAIT reads the event it refers to: if this event is *cleared*, the WAIT *waits* and tries again later; if this event is *posted*, the WAIT continues (only then, we will say that the WAIT *executes*); and the execution flow comes to the next statement. Thus, a WAIT is led to *wait for* the event to be *posted* by the execution of some POST statement, as was intended.

  Further explanations, on the behavior of synchronizations, will be provided later (§2.2), and an example will be described in §2.3 (figure 3).

We allow for **parameters** under the form of "variables" that get a value "once and for all" when the program starts, and are not written afterwards. Thus, in our framework, a *program* in fact represents a "class of programs", differing from one another by the values of parameters. (For instance, in many applications, dimensions of matrices will be such parameters. In a different way, considering programs designed to run several times on different *data*, these data will be parameters in our sense.)

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

In what follows, parameters and DO or PDO loop indices within their loop, 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 assignment:

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

where B is not an array 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 $B(3)$.

Variables will be allowed in DO and PDO loop bounds, test expressions in IFs and (obviously) in WHILEs, and subscript expressions in arrays *(dynamic variable reference)*, including event arrays.

The dynamic variable reference feature 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. 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.)[2]

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, unless otherwise stated; correlatively, we will consider as *statements* not only executable statements in the usual sense, but also such features as: heads and ends of DOs, IFs, WHILEs and parallel constructs; and the test expressions in WHILEs.

Considering DO, PDO and WHILE loops leads us to define a notion of **statement instance**. Classically (see e.g. [32]), since a statement within a loop may execute several times, each of these executions is termed as a statement instance. Thus, in a loop iterating 10 times, each statement generates 10 instances. This usual point of view brings a difficulty in our framework: since our language allows for variables in static loop bounds, and also for dynamic loops, the set of instances generated by one statements will generally not be known statically. Thus, we are led to introduce a different definition of a 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 DO, a PDO nor a WHILE, its index vector is empty: then, a generates one statement instance.

  Otherwise, let us consider the innermost loop containing a. Let c be the header of this innermost loop, and $\mathbf{i}$ be the index vector of c.

- If c is a DO or PDO header, the index vector of a is obtained as the concatenation of $\mathbf{i}$ and a component j, denoted $\mathbf{i}::j$. j corresponds to the iteration index of the loop.

- If c is a WHILE header, the index vector of a is obtained as the concatenation of $\mathbf{i}$ and a component j. This time, j will take positive integer values, numbering the successive WHILE iterations.

[2] In case our language would be extended to include pointers, this notion of indirection order would apply to pointer references as well.

Thus, through the two latter rules, every (executed or not) instance $c(\mathbf{i})$ generates an infinite (on both sides for DOs and PDOs, on one side for WHILEs) sequence of instances $a(\mathbf{i} :: j)$.

Through this formalism, a statement contained in a loop generates a countable infinity of statement instances but, in any given run not leading to an infinite loop, only a finite number of them will come to be executed.

2.2 Execution model

In order to obtain a good generality in our results, we must not specify the execution model of our language entirely; we will only specify a few properties supposed to hold in what follows. These specifications are inspired by the X3H5 proposal [31].

Two important notions will be introduced: the notions of *process* and *unit of work*.

- 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 parallel construct;
  - it encounters the end of a parallel construct;
  - it encounters a WAIT;
  - it encounters an execution fault.
- 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 PDO and each section of a PSECTIONS 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[3]. (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.)

  As regards variables, when the base process creates the team of processes, replicates of variables are made for every process in the team; computations are then performed locally in every process in the team.
- 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 processes in the team communicate the values of the updated variables to the base process; afterwards, the team is

[3] As an alternative, we could consider the possibility that this assignment of units to processes be *dynamic*: every unit of work still to be executed would be "waiting somewhere" till a process gets ready to run it. Such a variant could increase efficiency, but would not bring any essential change in the results to follow.

dissolved and its base process continues execution. (Only then, we will say that the ENDPDO or ENDPSECTIONS is *executed*.)[4]

- Such a variable updating also occurs when a process executes a POST or WAIT instance. We have outlined (§2.1) how a WAIT instance "waits for" a POST instance to have "posted" the matching event, and executes only then. To be more specific, when a process encounters a WAIT, it evaluates the event this WAIT statement instance involves. If this event is not *posted*, the process reiterates this step, till the event being considered gets *posted*, if at all[5]. When this condition is filled, the process realizes the variable updating and continues (only then, we will say that the WAIT is *executed*). This specification is consistent with the fact that a WAIT instance is aimed at *waiting for* the execution of some POST instance, presumably in order to ensure that – for instance – some value computed before the POST in its process is indeed available just after the WAIT in its process.

- Moreover, regarding the variable updatings we have mentioned, there may be memory conflicts, hence an inherent indeterminism. The aim of our study will be to detect whether such conflicts may occur (a circumstance which is *unwanted* in our framework) or whether we will be able to derive, from a static study of the program source, that such conflicts cannot happen (a circumstance which is desirable in our framework). In order our results to be as comprehensive as possible, we must not hypothesize exactly what happens, under our execution model, whenever such conflicts arise. This is why our execution model must not be specified entirely: hence our results will be valid for several nonequivalent execution models.

  Moreover, it is not assumed whether variable updatings occur in other cases than those mentioned above: termination of a parallel construct and event synchronization[6]. *However*, as regards the execution of a single statement instance within a process, we make the assumption that this instance gets its inputs if any, *then* performs its computations if any, *then* produces its outputs if any, without interference of variable updatings *during* these computations.

- As regards execution faults, they will be examined below.

It is important to point out the difference to be made between these two notions of *process* and *unit of work*. During the running of the program, the process generation is highly dependent on the

[4] In our language, we could introduce the notion of *private variable* (mentioned in the X3H5 proposal [31]), a temporary variable designed to be used locally in each process of the team, without communication among processes, and therefore, not involved in the variable updatings. Such variables would not be involved in the dependence preservations we will consider. Introducing such variables would be straightforward, as soon as "private" references and "shared" ones (our "variables") would be easily distinguishable. We have not done so, for the sake of brevity. *However*, the DO and PDO loop indices are supposed to hold this "private" status, in relation to parallel constructs the loop is nested in, if any.

[5] In the execution model of a WAIT instance involving a dynamic reference, a point is not specified here, regarding the variables involved in the event reference: are they reevaluated at every attempt, or "once and for all"? (in the latter case, the WAIT instance keeps on waiting for the same event.) We do not need to decide between these two possibilities: our results will be valid in both hypotheses.

[6] However, we have to notice that an efficient execution model will keep such variable updatings to a minimum, due to the cost of data transfers *(von Neumann bottleneck – see e.g. [32])*. Moreover, it would be possible to introduce *synchronizations with guards clause*, specifying the references to the variables to be updated, and thereby limiting these updates. We have not done so here, for the sake of brevity; such an extension, which was mentioned in [4], would be rather straightforward within our framework.

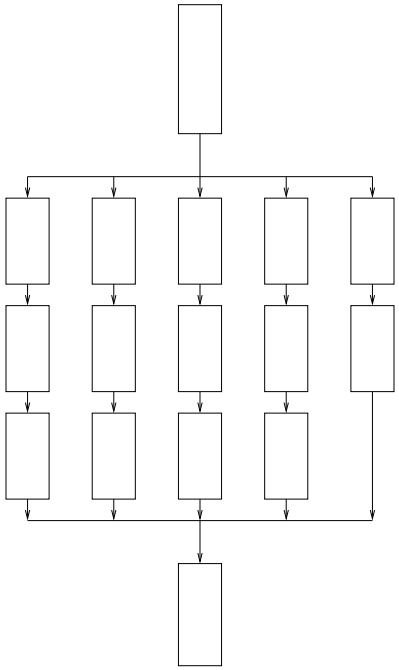

Figure 1: Loading units of work on processes

resources available at that moment: it is a *machine-dependent* phenomenon. In the contrary, the characterization of the units of work depends solely on the semantics of the program being run, as that semantics develops while the program is running; in case the program is *correct in our sense*, this characterization will be *machine independent*.

This is why we could not realistically hypothesize that one process per unit of work is created: in the contrary, we take care of the possibility that several units of work are assigned to one process, which will run them sequentially. Moreover, within our framework, we will require that the correctness of our programs be ensured independently of the number of processes which will turn out to be available for some run. This correctness will have to be guaranteed even in the extreme case when there is only one process available, or in the more common case when *some* parallel construct within the program will find only one process available for its execution, during a "multiprocess" run of the whole program.

A unit of work which is waiting for a process to run it 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*)[7].

As an example, Figure 1 depicts a possible case where a parallel loop creates 14 units of work (corresponding to 14 iterations) which are assigned to 5 processes (available at that point). The arrows indicate the order of execution, before the loop (upper part), within the loop (medium part: the 5 processes execute in parallel), and after the loop ends (lower part).

[7] 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.

Execution faults

An *execution fault* occurs whenever the execution of a statement instance creates an operation which is *forbidden* by the language and/or by the execution environment, so that the execution stops at that point *(fault deadlock)*.

In our execution model, we must address the possibility of execution faults. Indeed, whereas we will consider that the *sequential version* of our program will not produce execution faults, such an assumption cannot be made as regards our parallel program, because we aim at deriving semantic equivalence properties, referring only to the sequential semantics.

We will make the following assumption regarding execution faults: *considering a parallel program, and a statement instance coming to be executed in some run of this program, the occurrence of an execution fault at the execution of this instance depends only on the values of the inputs of this instance and the operations it attempts to perform using these inputs; it does not otherwise depend on the specific run considered.* As a consequence, whenever there are processes running in parallel to the one on which the fault occurs, *we hypothesize* that they are not stopped thereby – as long as they do not produce an execution fault themselves. This is a simplifying assumption which is not restrictive in our framework: indeed, we will derive conditions under which *there is no execution fault* meeting this assumption (*primary* execution fault); therefore, under the same conditions, and supposing that we do not make this simplifying assumption, there will not be *induced-in-parallel* faults either.

Deadlocks and infinite loops

In our language, there are three ways in which a program may not stop normally: it may come to a *fault deadlock*, or to a *waiting deadlock*, or it may enter an *infinite loop*.

A waiting 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 waiting deadlock condition 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 a similar deadlock situation in an outer construct.

A fault deadlock will induce effects quite similar to those of a waiting deadlock.

An infinite loop can occur only due to a WHILE construct (remember the absence of GOTOs and the fact that DO and PDO loop bounds are evaluated once at the loop entry). The corresponding ENDWHILE statement instance never gets executed: we will say that it is *persistently pending*. (Thus,

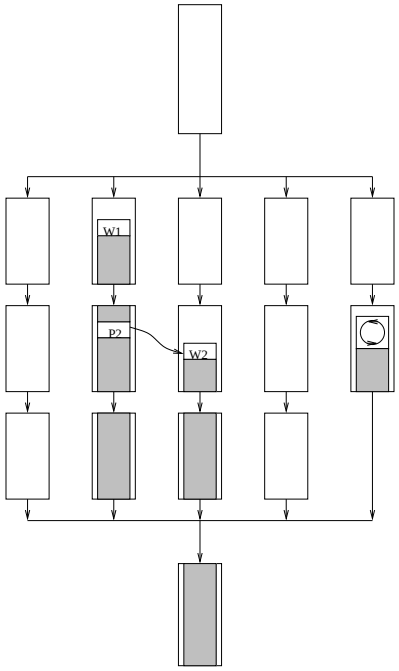

Figure 2: Parallel execution: a few possible pathologies

there are two different ways in which some statement instance may become *persistently pending*; in spite of the difference in nature, we will notice some similarity between the two, hence the same name for these two features.)

Whenever the WHILE construct is nested in a parallel construct, an infinite loop in this WHILE brings a situation similar to the one brought by a deadlock, described previously.

Figure 2 depicts a situation which is similar to the one represented previously (Figure 1) as regards units of work and processes, but exhibits various pathologies leading the parallel loop to deadlock. The shaded parts represent the instances unexecuted because of the deadlocks. Here, we get a deadlocking WAIT instance `W1` the event of which never got *posted*. As a consequence, the units of work loaded next to it on the same process are prevented from executing: therefore, the POST instance `P2` does not "post" towards the corresponding WAIT instance `W2`, which therefore deadlocks too *(induced waiting deadlock)*. Independently of these waiting deadlocks, an infinite WHILE loop is depicted on the right of the figure: it too prevents the normal termination of the parallel loop.

The sequential assignment

About the assignment of units of work to processes, we will make an assumption as regards our language:

Assumption SA (Sequential assignment) *In all parallel constructs containing synchronization statements (i.e.* POST*,* WAIT*,* CLEAR *statements), the assignment of units of work to available*

```
a0:   A(0)=...                          A(0)=...
p0:   post(E(0))                        continue
      pdo I=1,N                         do I=1,N
        ...                               ...
w:      wait(E(I-1))                      continue
b:      ...=A(I-1)                        ...=A(I-1)
a:      A(I)=...                          A(I)=...
p:      post(E(I))                        continue
        ...                               ...
      endpdo                            enddo
```

Figure 3: An example of a parallel loop with synchronizations (left) and its sequential version (right)

processes is made in such a way that the units of work assigned to every process are assigned sequentially: in the index ordering for a PDO*, in the section ordering for a* PSECTIONS*.*

Comment: Let us consider a parallel construct, and a POST/WAIT pair linking two units of work in this construct. If there were no assignment requirement whatever, it could randomly occur that the unit of work containing the WAIT be assigned to the *same* process as the unit of work containing the corresponding POST, and *before* it, hence a waiting deadlock situation. One possible way to avoid such "spurious deadlocks" is to state Assumption SA as a requirement in our execution model: indeed, this requirement is consistent with the following fact: in case the parallelization is correct in our sense, in a POST/WAIT pair, the POST instance precedes the WAIT instance in the sequential order, because such a synchronization pair is designed to *keep the sequential execution order* between statement instances which would otherwise be allowed to execute in parallel.

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 PDO into DO, the deletion of PSECTIONS, SECTION and ENDPSECTIONS statements, and the *disabling* of POST, WAIT and CLEAR statements: by "disabling", we mean that, in the sequential version defined here, they are converted into statements which do nothing (here denoted CONTINUE), but we retain the possibility, in the following developments, to keep track of event references, allowing ourselves to consider what occurs to these references as though they were indeed addressed in a sequential run.

Figure 3 shows a typical example of a parallel loop with event synchronizations, together with the sequential version of this loop, which sets the intended semantics: the output of statement instance `a(I-1)` is used as input in statement instance `b(I)`, as specified by the sequential order of execution of the DO loop. The synchronization statements **w** and **p** have been introduced in the parallel version, in order to preserve this execution order once the DO is parallelized into a PDO: the WAIT statement **w** at iteration `I` waits for the event `E(I-1)` to have received the value *posted* through the POST statement **p** at iteration `I-1`, or the statement **p0** (for `I=1`). Of course, the events `E(I)` must not have been *posted* before this loop without having been *cleared* meanwhile, nor have been *posted* elsewhere in parallel, in case this loop is nested in a larger parallel construct.

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 *(semantic equivalence)*.

In our language, we assume a **determinacy** condition, as a prerequisite for correctness of our program: *in the sequential version, any variable used as input in an executed statement instance has been initialized previously; similarly, any* DO *or* PDO *loop index used as input variable after the loop, has been initialized after the loop and before this use.*

Under this determinacy condition, we express the semantic equivalence requirement we are considering 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.* (As a consequence, that variable will indeed get the same value in both runs.)

Checking the semantic equivalence requires checking that the parallel program will not produce a waiting deadlock, whatever the number of available processes; and also that *data races* will be avoided. Whenever two statement instances involve the same variable location in the sequential version, at least one of them modifying (i.e. writing) it, we will say that they are in a *dependence* relation (denoted Dep). Then, we will have to check that these statement instances are in a *precedence* relation (denoted Pre), i.e. that the program control structure and the synchronizations preserve the order in which these statement instances will be executed together with ensuring the updating of variables meanwhile – along the well-known *"dependence implies precedence"* pattern [3] mentioned above.

For instance, under our execution model outlined above, a POST/WAIT synchronization is a precedence in that sense (remember the "variable updating" specification in our execution model, together with the succession in time).

3 Dependences and precedences

The theorem we present here refers to the semantics of the sequential version only. The gist of this result is to show that checking "dependence implies precedence" *under the sequential semantics* ("sequential" values of variables, etc.) indeed ensures the semantic equivalence between the parallel program and its sequential version. Under this theorem, we will be led to consider the condition for a statement instance to execute *in the sequential version* (predicate $\mathrm{Exe}^{\mathsf{s}}$), which is well-defined in our language; the dependence predicate Dep which deals with the sequential version, by definition; and the precedence predicate $\mathrm{Pre}^{\mathsf{s}}$ expressing *the precedence relations which would stand in the parallel program, as a consequence of the execution model, assuming that all variables involved in the definition of these relations get their "sequential" values.*[8]

In the derivation of this theorem (Sections 4 and 5), we will be led to consider the parallel counterparts of $\mathrm{Exe}^{\mathsf{s}}$ and $\mathrm{Pre}^{\mathsf{s}}$ – Exe and Pre respectively – which, as long as we have not proved the semantic equivalence, are defined only in reference to some specific run of the parallel program (they may differ for different runs of *the same* parallel program). Considering such a parallel run,

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

the precedence predicate Pre associated to this run expresses *the precedence relations which stand in this run, as a consequence of the execution model, considering the values* in this run *of the variables involved in the definition of these relations.* We will develop this point later (§3.4).

3.1 Sequential execution predicate

Let $\mathrm{Exe}^{\mathsf{s}}$ be the condition for a statement instance to be executed in the sequential version, *under the extra condition that the sequential program terminates*, i.e. does not enter an infinite WHILE loop. Then, $\mathrm{Exe}^{\mathsf{s}}$ is well-defined (although it is generally not computable!) and its expression is rather straightforward for our language.

For a statement a and an index vector $\mathbf{i}$ such that the instance $a(\mathbf{i})$ is executed in the sequential version, we may consider the environment in which the execution of $a(\mathbf{i})$ takes place. For any expression *exp* which happens to be evaluated through the execution of $a(\mathbf{i})$, its value is defined in this environment: it will be denoted $[\![exp]\!]@a(\mathbf{i})$. (Due to our assumption that subroutine calls terminate, the evaluation of an expression always terminates, in our language.) We must emphasize that $[\![exp]\!]@a(\mathbf{i})$ is undefined whenever $a(\mathbf{i})$ is *not* executed. This leads us, in the expressions to follow, to make use of the *sequential conjunction*, denoted &, which differs from the *logical conjunction*, denoted $\wedge$, as follows: if A and B are boolean expressions (taking values *true*, *false* or *undefined*), $A \,\&\, B$ is false whenever A is false, even if B is undefined, whereas $A \wedge B$ is undefined in this case[9].

Let us give the expression of $\mathrm{Exe}^{\mathsf{s}}(a(\mathbf{i}))$ when a is a statement of the parallel program, indexed by $\mathbf{i}$. Several cases have to be considered, depending on the nesting of a in a loop or IF structured statement. In case a is nested, we consider the innermost DO, PDO, IF or WHILE a is nested in.

- a is not contained in a DO, a PDO, a IF nor a WHILE construct: then, $\mathrm{Exe}^{\mathsf{s}}(a) = \mathit{true}$.
- The innermost nesting of a is in a IF construct of header c, with boolean expression $c.\mathrm{BEXP}$. Let $\mathbf{i}$ denote the index vector of c and a:
  - if a is in the THEN branch, $\mathrm{Exe}^{\mathsf{s}}(a(\mathbf{i})) = \mathrm{Exe}^{\mathsf{s}}(c(\mathbf{i})) \,\&\, [\![c.\mathrm{BEXP}]\!]@c(\mathbf{i})$
  - if a is in the ELSE branch, $\mathrm{Exe}^{\mathsf{s}}(a(\mathbf{i})) = \mathrm{Exe}^{\mathsf{s}}(c(\mathbf{i})) \,\&\, \neg([\![c.\mathrm{BEXP}]\!]@c(\mathbf{i}))$
- The innermost nesting of a is in a DO or PDO loop of header c, with lower and upper bound expressions $c.\mathrm{LB}$ and $c.\mathrm{UB}$ respectively. Let $\mathbf{i}$ denote the index vector of c and $\mathbf{i} :: j$ denote the index vector of a. We then have: $\mathrm{Exe}^{\mathsf{s}}(a(\mathbf{i} :: j)) = \mathrm{Exe}^{\mathsf{s}}(c(\mathbf{i})) \,\&\, ([\![c.\mathrm{LB}]\!]@c(\mathbf{i}) \leq j \leq [\![c.\mathrm{UB}]\!]@c(\mathbf{i}))$
- The innermost nesting of a is in a WHILE loop of header c. Let $\mathbf{i}$ denote the index vector of c and $\mathbf{i} :: j$ denote the index vector of a. Let b be the first statement in the WHILE body, i.e. the test statement, of boolean expression $b.\mathrm{BEXP}$.
  - If a is not b: $\mathrm{Exe}^{\mathsf{s}}(a(\mathbf{i} :: j)) = \mathrm{Exe}^{\mathsf{s}}(b(\mathbf{i} :: j)) \,\&\, ([\![b.\mathrm{BEXP}]\!]@b(\mathbf{i} :: j))$
  - If a is b: $\mathrm{Exe}^{\mathsf{s}}(b(\mathbf{i} :: 1)) = \mathrm{Exe}^{\mathsf{s}}(c(\mathbf{i}))$ and, for the other iterations: $\mathrm{Exe}^{\mathsf{s}}(b(\mathbf{i} :: j)) = \mathrm{Exe}^{\mathsf{s}}(b(\mathbf{i} :: (j-1))) \,\&\, ([\![b.\mathrm{BEXP}]\!]@b(\mathbf{i} :: (j-1)))$

[9] Consistently with this sequential aspect, in what follows, conjunctions are *left-associative*: e.g. $A \wedge B \,\&\, C \wedge D$ is interpreted as: $((A \wedge B) \,\&\, C) \wedge D$. Moreover, in all cases when some expression C may be *undefined* if some other expression A is *false*, we require that C appear only in expressions $(\ldots \wedge A \wedge \ldots) \,\&\, C \ldots$, so that these resulting expressions are always *defined*.

It is possible to extend the expression of $\mathrm{Exe}^{\mathsf{s}}$ so as to include cases when the sequential program infinitely loops, however in function of the explicit data of the looping WHILE. Let $c(\mathbf{i})$ be the corresponding WHILE header, and $e(\mathbf{i})$ the matching ENDWHILE. For $e(\mathbf{i})$ and all instances $a(\mathbf{j})$ standing after it in the sequential order, the above expression of $\mathrm{Exe}^{\mathsf{s}}(a(\mathbf{j}))$, which besides is not necessarily well-defined then, should be replaced by: $\mathrm{Exe}^{\mathsf{s}}(a(\mathbf{j})) = \mathit{false}$.

The condition that the sequential program terminates, which is of course undecidable in general, can then be formally expressed here. Using the notations we have just introduced regarding WHILE loops, we get:

Sequential termination condition:
For all WHILE *loops, of header* c *and test* b,

$$\mathrm{Exe}^{\mathsf{s}}(c(\mathbf{i})) \Rightarrow \exists\, j\ \ \mathrm{Exe}^{\mathsf{s}}(b(\mathbf{i}::j))\ \&\ \neg([\![b.\mathrm{BEXP}]\!]@b(\mathbf{i}::j))$$

In case of infinite loop, the above implication is trespassed only for the WHILE instance on which the infinite loop occurs.

3.2 Dependences

Considering two statements a and b, indexed by $\mathbf{i}$ and $\mathbf{j}$ respectively, a predicate $\mathrm{Dep}(a(\mathbf{i}), b(\mathbf{j}))$ will express that: "in case $a(\mathbf{i})$ and $b(\mathbf{j})$ are both executed in the sequential version, in this order, then they both access one same memory location (not corresponding to an event variable), at least one of them writing it"[10].

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)"*

Let $a(\mathbf{i})$ and $b(\mathbf{j})$ be two statement instances respectively involving references *expa* and *expb*, referring to variables (other than event variables), these two references *not both being input references.* In case $a(\mathbf{i})$ executes in the sequential version, $[\mathit{expa}@a(\mathbf{i})]^S$ denotes the variable *expa* refers to during that execution. The relation $\equiv$ between variables to which two distinct references are made, means that they are the same variable. $\ll$ denotes the sequential order. We can give an expression of Dep:

$$\mathrm{Dep}(a(\mathbf{i}), b(\mathbf{j})) = (a(\mathbf{i}) \ll b(\mathbf{j})) \wedge ([\mathit{expa}@a(\mathbf{i})]^S \equiv [\mathit{expb}@b(\mathbf{j})]^S)$$

This expression of Dep is not necessarily defined when $a(\mathbf{i})$ and/or $b(\mathbf{j})$ does not execute in the sequential version. Therefore, in what follows, we will be led to use Dep in expressions such as:

$$\mathrm{Exe}^{\mathsf{s}}(a(\mathbf{i})) \wedge \mathrm{Exe}^{\mathsf{s}}(b(\mathbf{j}))\ \&\ \mathrm{Dep}(a(\mathbf{i}), b(\mathbf{j}))$$

making use of the sequential conjunction & introduced previously (§3.1).

[10] In the special case when $a(\mathbf{i})$ and $b(\mathbf{j})$ are the same instance accessing a variable, first as input, second as output – e.g an instance of a statement such as: "`x=x+1`" – such a "reflexive" dependence will necessarily be preserved, in our sense, through the precedence "input precedes output" guaranteed during the execution of one instance (§2.2). Therefore, we will not have to take care of such reflexive dependences $\mathrm{Dep}(a(\mathbf{i}), a(\mathbf{i}))$.

```
        ...                              ...
ps:     psections                        continue
          section                          continue
            ...                              ...
a:          A(N)=...                         A(N)=...
          section                          continue
            ...                              ...
b:          ...=A(N)+...                     ...=A(N)+...
c:          ...=A(N+1)*...                   ...=A(N+1)*...
          section                          continue
            ...                              ...
d:          N=...                            N=...
            ...                              ...
        endpsections                     continue
        ...                              ...
```

Figure 4: Studying data dependences: an example

Let us study an example of a portion of program supposed to be executed (Figure 4). The sequential version is shown on the right. We examine the dependences dealing with variable array $A()$. Let us first consider the case when we can statically check that the integer variable N is not written between `ps` and `c` on the sequential version. (Such a check may be quite easy in a Fortran-like language without procedure calls, but more intricate, or even intractable, whenever e.g. pointers or procedure calls are used.) Then, statements `a` and `b` refer to the same memory location in the sequential version (because $[\![N]\!]@a = [\![N]\!]@b$), and `a` writes it. Hence we have a dependence: $\mathrm{Dep}(a, b) = \mathit{true}$. On the other hand, we have no dependence (at least as regards references visible here) between `a` and `c`.

Now, considering the more difficult case when we cannot statically know what may occur to N between `ps` and `c` in the sequential version, we will have to assume (as a conservative approximation) that there is a dependence from `a` to `b` *and* from `a` to `c` (unless otherwise proved, we may very well have $[\![N]\!]@a = [\![N]\!]@c + 1$!).

On this example, let us now consider the dependence relations regarding variable N. This variable location is an output of `d` and an input of `a`, `b` and `c`, hence a dependence relation from `a` to `d`, from `b` to `d` and from `c` to `d`. There is no mutual dependence among `a`, `b` and `c`, regarding the variable N, since these statements refer to N as input.

As this example shows, mainly due to the *dynamic variable reference* feature, 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" (More on this later: §4.1).

3.3 Precedences

The precedence predicate $\mathrm{Pre}^{\mathsf{s}}$ expresses the precedence relations which would apply in the parallel program, as a consequence of the execution model, *assuming* that the variables involved in the definition of these relations have their "sequential" values. We will characterize $\mathrm{Pre}^{\mathsf{s}}$ from a *control precedence* Pre^{0} and a *synchronization precedence* $\mathrm{Sync}^{\mathsf{s}}$ (corresponding to the POST/WAIT pairs). 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 $\mathrm{Pre}(\alpha, \beta)$ involving two statement instances α and β, we wish to express that *"in case α and β are executed, the former is executed before the latter (in such a way that variable updatings happen meanwhile); but we are not interested in what is expressed if these instances are not both executed."* 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. This same multiplicity of correct predicates also holds for dependences; furthermore, it is straightforward to check that the "dependence preservation" property we will consider is (fortunately...) invariant by any change of correct predicates.

Let us now express *control* precedences through a predicate Pre^0 independent of the specific run of the program – in fact, a predicate independent of any variables and even parameters. Afterwards, we will be interested in the synchronization precedences and the way they combine with the control precedences.

Expression of control precedences

Calculating precedences on index vectors

In order to express Pre^0, we have to express the precedence order between index vectors, denoted $\prec$.

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

- If k indexes a DO or WHILE loop:

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

- If k indexes a PDO loop:

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

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

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

This directly leads to the expression of $\mathrm{Pre}^0(a, a)$ for a statement a. Let $\mathbf{i}$ be the index vector of a; let $a(\mathbf{i}_1)$ and $a(\mathbf{i}_2)$ be two instances of a; we set:

$\mathrm{Pre}^0(a(\mathbf{i}_1), a(\mathbf{i}_2)) = (\mathbf{i}_1 \prec \mathbf{i}_2)$

Expression of Pre^0 *between different statements*

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 a 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.

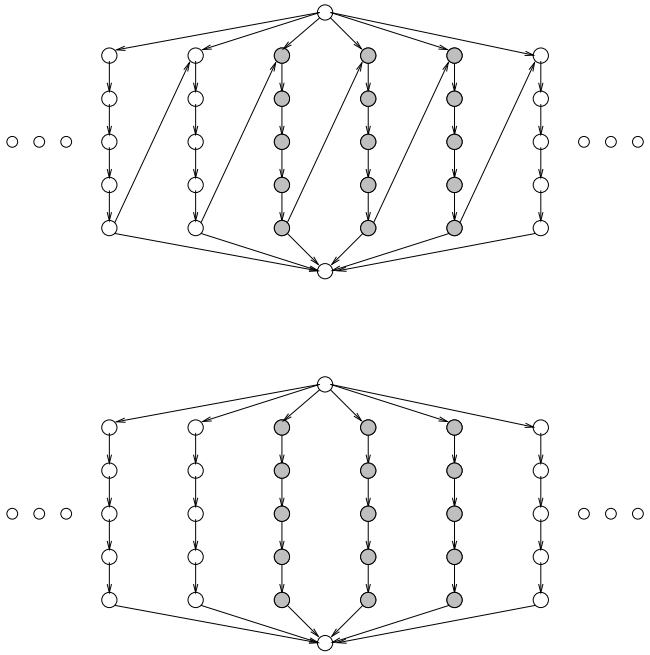

Figure 5: Control precedences in a DO (top) and a PDO (bottom). Only six iterations are shown here, but loop instances in fact extend infinitely on both sides. The control precedence links Pre^0 (arrows) are independent of the specific run and the specific iterations coming to execute in this run (three iterations shaded in the example).

In case a and b are not in the same loop or PSECTIONS, we get:
$\mathrm{Pre}^0(a,b) = \mathit{true}$; $\mathrm{Pre}^0(b,a) = \mathit{false}$

Otherwise, we consider the innermost loop or PSECTIONS containing both a and b. Let c be the header of this structured statement, and $\mathbf{i}$ be the index vector of c.

- If c is a loop header, let $\mathbf{j} = \mathbf{i} :: h$ be the index vector common to a and b. (h denotes the loop index; the index vectors of a and b are concatenations of $\mathbf{j}$ with possibly empty disjoint index vectors $\mathbf{k}$ and $\mathbf{l}$ respectively.) We get:

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

  $\mathrm{Pre}^0(b(\mathbf{j}_b :: \mathbf{l}_b), a(\mathbf{j}_a :: \mathbf{k}_a)) = (\mathbf{j}_b \prec \mathbf{j}_a)$

In the two next cases, when c is a PSECTIONS header, the index vector common to a and b is $\mathbf{i}$. Again, the index vectors of a and b are concatenations of $\mathbf{i}$ with possibly empty disjoint index vectors $\mathbf{k}$ and $\mathbf{l}$ respectively.

- If c is a PSECTIONS header and a and b are in the same SECTION of this PSECTIONS:

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

  $\mathrm{Pre}^0(b(\mathbf{i}_b :: \mathbf{l}_b), a(\mathbf{i}_a :: \mathbf{k}_a)) = (\mathbf{i}_b \prec \mathbf{i}_a)$

- If c is a PSECTIONS header and a and b are in distinct SECTIONs of this PSECTIONS:
  $\mathrm{Pre}^0(a(\mathbf{i}_a :: \mathbf{k}_a), b(\mathbf{i}_b :: \mathbf{l}_b)) = (\mathbf{i}_a \prec \mathbf{i}_b)$
  $\mathrm{Pre}^0(b(\mathbf{i}_b :: \mathbf{l}_b), a(\mathbf{i}_a :: \mathbf{k}_a)) = (\mathbf{i}_b \prec \mathbf{i}_a)$

As an example, Figure 5 shows control precedences in a DO and a PDO loops.

Combining control and synchronization precedences

To obtain the overall precedence relation Pre^s, we have to combine the control precedence Pre^0 and the synchronization precedence relations Sync^s realized through POST/WAIT pairs (we will consider the latter in a moment; meanwhile, we consider them as given). This composition of Pre^0 with Sync^s is not exactly a transitive closure, as might be expected; $\mathrm{Pre}^s(a(\mathbf{i}), b(\mathbf{j})) \wedge \mathrm{Pre}^s(b(\mathbf{j}), c(\mathbf{k}))$ does not necessarily imply $\mathrm{Pre}^s(a(\mathbf{i}), c(\mathbf{k}))$ (consider the case when $b(\mathbf{j})$ is not executed...). Instead, we have "transitivity modulo Exe^s":

$$\mathrm{Pre}^s(a(\mathbf{i}), b(\mathbf{j})) \wedge \mathrm{Exe}^s(b(\mathbf{j})) \wedge \mathrm{Pre}^s(b(\mathbf{j}), c(\mathbf{k})) \Rightarrow \mathrm{Pre}^s(a(\mathbf{i}), c(\mathbf{k}))$$

Considering the directed *precedence graph*, whose vertices are the statement instances and edges are the precedence links Pre^0 and Sync^s, the relation Pre^s will 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 Pre^s alternate Pre^0 and Sync^s links, in the following way:

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

where $\rightarrow$ denotes the Pre^0 relation, π_i denotes a POST, ω_i denotes a WAIT, and $\rightsquigarrow$ denotes the synchronization relation Sync^s.

The corresponding computation of Pre^s will be realized through relations such as:

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

In fact, in what follows, we will be led to include the conditions of execution of π and ω within the $\mathrm{Sync}^s(\pi, \omega)$ predicate. Hence, the above expression becomes:

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

Let us consider again the precedence paths shown above, alternating the $\rightarrow$ and $\rightsquigarrow$ links. In case α is a POST, we must also include precedence paths where "$\alpha \rightarrow \pi_1$" is replaced by "$\alpha = \pi_1$". In other words, a precedence path expressed through Pre^s may begin with a synchronization. *In the contrary*, in case β is a WAIT, *we do not allow ourselves* to consider paths in which "$\omega_n \rightarrow \beta$"

```
n1:   N=1
      psections
        section
          ...
n2:       N=2
p:        post E(N)
          ...
        section
          ...
w:        wait E(N)
          ...
      endpsections
```

Figure 6: A case of dynamic event reference in a WAIT

is replaced by "$\omega_n = \beta$". In other words, a precedence path expressed through Pre^{s} may *begin*, but not *end*, with a synchronization link. Why do we introduce this difference? We have been led to do so in consideration of the specific behavior of the WAIT statements.

The problem of the dynamic references in the WAIT*s*

In our language, we allow dynamic references in WAITs. This ability triggers specific problems, illustrated through the example of Figure 6.

In this example, we assume that N is not present elsewhere than indicated. The underlying intention is to ensure a synchronization from p to w, involving event $E(2)$. Indeed, *following the sequential semantics*, N has got value 2 at p and w. *However*, in a parallel execution, N may have value 1 when the control gets to w, which will then wait for the event $E(1)$ (whereas p will still post $E(2)$). Therefore, the dependence relation $\text{Dep}(n2, w)$ involving variable N is not preserved through the precedence path $n2 \rightarrow p \rightsquigarrow w$. This is the reason why we have defined Pre^{s} so as to exclude such precedence paths ending with a synchronization link. Considering the execution model (§2.2), this restriction we bring here allows us to ensure the following property for the precedence predicate Pre^{s}: *if α is an executed instance and w is a waiting* WAIT *instance,* $\text{Pre}^{\mathsf{s}}(\alpha, w)$ *implies that α executes before w begins waiting (and in such a way that variable updatings happen meanwhile).* Thus, in our example, if $\text{Pre}^{\mathsf{s}}(n2, w)$ comes to be ensured, through a precedence path not apparent here, then the output of N at $n2$ precedes (in the sense of our precedences) its input when w begins waiting: w will then wait for $E(2)$ (unless N is written meanwhile).

3.4 The synchronization precedence Sync^{s}

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

The first difficulty stems from the fact that we allow dynamic variable reference in POST, WAIT and CLEAR statements. This possibility, together with the fact that Exe is defined only in reference to a given run of the parallel program, implies that the POST/WAIT synchronizations which show up during some specific run of the program, and the precedence brought by them as a consequence of the execution model (§2.2 and 2.3), essentially depend upon the run being considered. Considering

such a run, let ω be an executed WAIT instance. Then, there is at least one POST instance π (and possibly several of them) which set value *posted* to the involved event, and thus *made possible* the execution of ω together with ensuring the corresponding precedence from π to ω. We denote $\mathrm{Sync}(\pi, \omega)$ the predicate, essentially dependent on the specific run of the program, expressing the synchronization precedence thus realized.

The precedence predicate Pre for a specific run of the parallel program, is then obtained from Pre^0 and Sync, exactly along the same lines as $\mathrm{Pre}^{\mathsf{s}}$ is obtained from Pre^0 and $\mathrm{Sync}^{\mathsf{s}}$ (§3.3).

Since Sync, and therefore Pre, thus essentially depend on some specific run of the parallel program, we are led to consider, instead of Sync, a predicate $\mathrm{Sync}^{\mathsf{s}}$ describing the synchronizations which arise under the assumption that the variables involved in these synchronizations (through the execution predicate and/or the dynamic event references) get their "sequential" values. Let us remember that, when we consider the sequential version of our program, the synchronizations are disabled, but we keep track of the event references they involve (§2.3). This will allow us to consider "the execution of synchronizations in the sequential version".

The second difficulty is of a different nature – and it will lead us to specify some properties we will prescribe regarding synchronizations. Through a predicate $\mathrm{Sync}^{\mathsf{s}}(\pi, \omega)$, between a POST instance and a corresponding WAIT instance, we wish to express that *"supposing the sequential semantics, if π and ω are both executed, then necessarily π is executed before ω."*. This supposes that no other POST instance 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 – in such a case, we will not have a $\mathrm{Sync}^{\mathsf{s}}$ relation from any of these POSTs to 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 two POST instances involving the same event not execute in parallel. Moreover, a CLEAR statement instance, dealing with the same event, should not be in a data race condition with this POST, nor with this WAIT.

We will express these restrictions through two assumptions dealing with the use of synchronizations. These assumptions will allow us to characterize the synchronization predicate $\mathrm{Sync}^{\mathsf{s}}$. Let us remind a few notations (§3.2). For any POST, WAIT or CLEAR statement instance γ executed in the sequential version, we denote $[\varepsilon_\gamma]^S$ the event variable reference that γ involves in the sequential version. The relation $\equiv$ between variable references means that they refer to the same variable.

Assumption S1 (No race condition involving synchronizations) *Let θ and ω be two instances of synchronizations* POST, WAIT *or* CLEAR. *Except in the case when one of these two instances is a* POST *and the other is a* WAIT, *and in the case when both are* WAITs, *we have:*

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

For a POST instance π and a WAIT instance γ, let us define a predicate $\mathrm{Sync}^\star$ as follows:

$$\mathrm{Sync}^\star(\pi, \gamma) \quad = \quad \mathrm{Exe}^{\mathsf{s}}(\pi) \wedge \mathrm{Exe}^{\mathsf{s}}(\gamma) \;\&\; ([\varepsilon_\pi]^S \equiv [\varepsilon_\gamma]^S) \wedge \neg \mathrm{Pre}^0(\gamma, \pi) \wedge$$

$$(\forall \text{CLEAR instance } \theta,$$
$$\text{Exe}^{\mathsf{s}}(\gamma) \wedge \text{Exe}^{\mathsf{s}}(\theta) \;\&\; ([\varepsilon_\theta]^S \equiv [\varepsilon_\gamma]^S) \Rightarrow \neg(\text{Pre}^0(\pi,\theta) \wedge \text{Pre}^0(\theta,\gamma)) \;)$$

$\text{Sync}^\star(\pi,\gamma)$ expresses that, under the sequential semantics, π is *susceptible to trigger* the execution of γ, in the sense that π and γ both execute, and involve the same event reference, in the sequential version; that γ does not precede π Pre^0-wise; and that no CLEAR instance involving the same event is bound to interfere between the two, Pre^0-wise.

Notice that, under Assumption S1, the term $\neg(\text{Pre}^0(\pi,\theta) \wedge \text{Pre}^0(\theta,\gamma))$ in the definition of $\text{Sync}^\star(\pi,\gamma)$, is equivalent to $(\text{Pre}^0(\theta,\pi) \vee \text{Pre}^0(\gamma,\theta))$.

Now we can express Assumption S2.

Assumption S2 (Ensured precedence from POST to WAIT) . *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.*

Comment This expresses that, under the sequential semantics, at most one POST statement instance is *susceptible to trigger* the execution of the WAIT instance, in our sense, in the given program instance. However, that POST instance may depend on the program instance considered, i.e. on values of parameters; for example, it will often occur that two POSTs posting the same event lie in two alternative branches of a IF: this is not contrary to our assumption because these two POSTs are mutually exclusive.

In the derivation of the theorem to follow, Assumptions S1 and S2 will be used through the following consequence, dealing with the case when a POST instance π and a WAIT instance γ do *not* form a synchronization pair:

$$\text{Exe}^{\mathsf{s}}(\pi) \wedge \text{Exe}^{\mathsf{s}}(\gamma) \;\&\; ([\varepsilon_\pi]^S \equiv [\varepsilon_\gamma]^S) \;\&\; \neg\, \text{Sync}^{\mathsf{s}}(\pi,\gamma) \;\Rightarrow\; (\text{Pre}^0(\gamma,\pi) \;\vee$$
$$(\exists \text{CLEAR instance } \theta, \text{Exe}^{\mathsf{s}}(\gamma) \wedge \text{Exe}^{\mathsf{s}}(\theta) \;\&\; ([\varepsilon_\theta]^S \equiv [\varepsilon_\gamma]^S) \wedge \text{Pre}^0(\pi,\theta) \wedge \text{Pre}^0(\theta,\gamma) \;) \;)$$

It would be possible to extend assumptions S1 and S2, by replacing the control precedences Pre^0 with generalized precedences Pre^{s}, *however* under the condition that the synchronization relations Sync^{s} involved in Pre^{s} are given *a priori*; such a generalization (which we mentioned in [4], and under which the theorem to follow still holds) does not allow to *derive* Sync^{s}. In other words, such an extension of S1 and S2 brings a circularity, in the sense that it presupposes that the synchronization relations Sync^{s} are given, whereas these two assumptions contribute to the very existence of these synchronization relations.

It is important to keep in mind that assumptions S1 and S2 refer to the semantics of the sequential version only, and do not depend on some specific parallel run.

Figure 7 shows an example of control and synchronization precedences in a parallel loop.

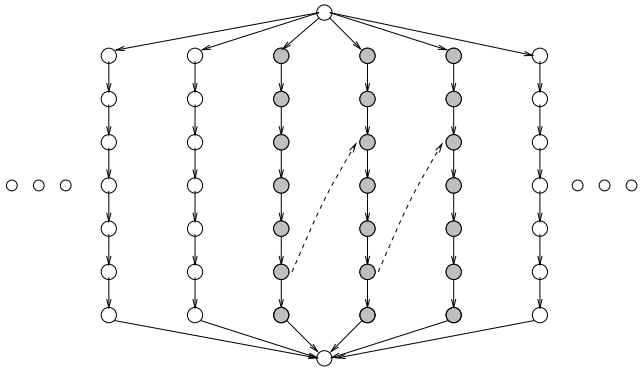

Figure 7: Control precedences in a PDO (straight arrows, again) and synchronization precedences $\mathrm{Sync}^{\mathsf{s}}$ (dashed arrows). Contrarily to the former, the latter precedences are defined referring to the sequential semantics. These synchronization precedences might not be ensured in some specific run of this parallel loop. In the example, the three iterations shaded are those which *should* be executed, according to the sequential semantics.

4 Some preliminary results

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

4.1 Conservative approximations of predicates

In light of the theorem we will prove in the next section, checking the semantic equivalence property will mainly require checking the following implication:

$$\mathrm{Exe}^{\mathsf{s}}(a(\mathbf{i})) \wedge \mathrm{Exe}^{\mathsf{s}}(b(\mathbf{j})) \;\&\; \mathrm{Dep}(a(\mathbf{i}), b(\mathbf{j})) \Rightarrow \mathrm{Pre}^{\mathsf{s}}(a(\mathbf{i}), b(\mathbf{j}))$$

for all statement instances $a(\mathbf{i})$ and $b(\mathbf{j})$ in the given program.

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. 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 $\mathrm{Exe}^{\mathsf{s}}$ and Dep, from below for $\mathrm{Pre}^{\mathsf{s}}$: we will then consider predicates $\mathrm{Exe}^{\mathsf{s}\star}$, $\mathrm{Dep}^{\star}$ and $\mathrm{Pre}^{\mathsf{s}}_{\star}$ such that $\mathrm{Exe}^{\mathsf{s}} \Rightarrow \mathrm{Exe}^{\mathsf{s}\star}$, $\mathrm{Dep} \Rightarrow \mathrm{Dep}^{\star}$ and $\mathrm{Pre}^{\mathsf{s}}_{\star} \Rightarrow \mathrm{Pre}^{\mathsf{s}}$, respectively meaning that "a statement instance may be executed", "a dependence may exist" and "a precedence must exist".

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

4.2 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.

Considering a parallel run, let $\psi(\alpha)$ denote the condition for α to be executed *or* persistently waiting *or* persistently pending (let us remind that α is said to be *reached* in the two first cases). If α is the (only) instance of the first statement in the program, $\psi(\alpha) = \mathit{true}$. For all the other instances, we have 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, $\psi(\alpha)$ 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. If at least one of these β produces an execution fault, this implies $\psi(\alpha) = \mathit{false}$.*

We have $\mathrm{Exe}(\alpha) = \psi(\alpha)$ except in the three following cases:

- *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 persistently 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 (or in an infinite loop situation in the latter case).*

- *α is an instance of the first statement in a* PDO *body or in a* SECTION *of a* PSECTIONS*, without being an instance of a* WAIT*: then, $\psi(\alpha)$ expresses the condition for α to be executed or persistently pending; the latter possibility occurs in case of a deadlock or infinite loop.*

- *α is an instance of a* ENDWHILE*: then, $\psi(\alpha)$ expresses the condition for α to be executed or persistently pending; the latter possibility occurs in case the* WHILE *infinitely loops, or in case of a deadlock or infinite loop within an iteration.*

The proof of this lemma is tedious but not difficult. It is provided in Appendix A.

This lemma derives its main interest from the previously mentioned fact that Pre^0 is independent of the specific parallel run considered. 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.

4.3 A notion of execution date

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

Discretized time lemma: *For any run of a parallel program, we consider the precedences* Pre *and* Sync *associated to this run (defined in Section 3). We assume the four following properties:*

i. To any statement instance α executed in this run, is associated a time lag $[t_\alpha, t'_\alpha]$ (of physical time), with $t'_\alpha \geq t_\alpha$, called the execution time lag *of α.*

ii. 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 written by α is not *available as input for β.*

iii. As a consequence of (ii), for any statement instances α and β executed in this run, if $\mathrm{Pre}(\alpha, \beta)$ *or* $\mathrm{Sync}(\alpha, \beta)$*, then $t_\beta > t'_\alpha$.*

iv. For any time t, only a finite number of instances α have begun execution before t.

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

1. *$\tau(\alpha)$ nondecreasingly depends on t_α.*
2. Computational causality: *for any statement instances α and β, a value output by α cannot be used as input by β unless $\tau(\beta) > \tau(\alpha)$.*
3. *As a consequence, for any two statement instances α and β executed, whenever* $\mathrm{Pre}(\alpha, \beta)$ *or* $\mathrm{Sync}(\alpha, \beta)$*, we have $\tau(\beta) > \tau(\alpha)$.*
4. *τ is* causally defined*, i.e. $\tau(\alpha)$ depends only on the execution time lags of α and the instances beginning execution before α.*

Comment: Hypotheses *(i)* and *(ii)* can be interpreted thus: for any instance α coming to be executed in some run, α gets its input (if any) at or shortly before some time t_α, then executes without any input/output exchange, till some time t'_α when – or shortly after which – α provides its output (if any). (We allow ourselves to set $t_\alpha = t'_\alpha$ when α performs no computation.) Hypothesis *(iii)* is a consequence of the variable updating feature embedded in the relations Pre and Sync. Hypothesis *(iv)* is, obviously, meaningful only when the run is endless, a case we have to consider too; this hypothesis is then justified by the finiteness of available resources (especially the finite number of processes).

Let us 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 (countable) 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. Even in case the run is endless, this ordering of statement instances is *well-founded*, due to *(iv)*: these instances are then ordered into a sequence.

The (finite or infinite) sequence of instances obtained thus will be denoted $\alpha_1, \alpha_2, \ldots \alpha_N, \ldots$. For convenience, the execution time lag for α_i will be denoted $[t_i, t'_i]$.

The date 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 $t_j \leq \min(t'_k \mid i \leq k < j)$, 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 date have execution time lags which mutually overlap: hence, *(ii)* 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 or Sync – have different dates. ◀

Let us emphasize that we will make use of the mere fact that a date function *exists*; we will not need to be able to *effectively compute* it. Besides, the date function depends on the specific run being considered, even if the parallel program is semantically equivalent to its sequential version.

As far as program semantics will be concerned, statement instances associated to the same date will be considered as though they executed "at the same time". Thus, for convenience, we will say that "some statement instance executes at some date". The computational causality feature (property 2) is crucial here: it ensures that the result of a computation "made at some date" is not available before the next date. (The causal definition feature (property 4) will not be used here: it is a byproduct.) So to speak, what we are considering here is a *causality-preserving time discretization*.

The reciprocal of property 2 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 this availability will stand if $\mathrm{Pre}(\alpha, \beta)$ (provided, of course, that the variable is not written again meanwhile); or $\mathrm{Sync}(\alpha, \beta)$ (in which case, besides, α writes an event variable read by β).

The need to introduce a notion of execution date is the main reason why we consider *simple* statements rather than *structured* ones: very often, a structured statement instance extends on several dates, in our sense. This is not due to the time length of execution of this structured statement, but rather to the existence of input/output exchanges *within* the execution time lag. Thus, a subroutine or function call is attributed a single date in our framework, however long its execution time may be, provided that it follows the specification we mentioned above: that inputs/outputs occur only at the beginning and at the end of the subroutine or function[11].

Other notions of date

The notion of execution date we introduce here displays some similarities, and some differences, with the well-known notion of *linear time* proposed by Lamport [17, 25]. Lamport considers sequential *processes*, communicating with one another by sending and receiving *messages*. Events occurring on every process are assigned dates by a local clock; every message includes the date when it is sent (for the local time of the emitting process), and triggers an update of the local time of the receiving

[11] In the same flavor, in some languages – such as *C* – where several operations may be condensed into one seemingly "simple" statement, such a statement may have to be considered as structured in our sense, and will then extend on several dates.

process, if necessary, so that, so to speak, the message is received "after" it is sent. This mechanism permits to effectively express causality through a *global clock*, realized only from local clocks – without a need for a common time device with which all processes would have to communicate.

This causality feature of linear time is obviously what makes it somewhat similar to "our" execution date. An important difference between our scheme and linear time (or more complex time schemes outlined in [25]) is that the latter aim at *effectively computing* dates allowing to order events, whereas, as we mentioned before, it is sufficient for us to know the *existence* of a date function $\tau()$ endowed with some interesting properties; we will have no need whatever to *effectively* compute it.

4.4 The ordered single process run

A **single process** run is a run of the parallel program, obtained when there is only one process available. As mentioned before, it will be required that a parallel program not produce a waiting deadlock whatever the number of available processes. Therefore, a single process run should not produce a waiting deadlock.

We will be interested in the **ordered single process run**, defined as the single process run in which Assumption SA (sequential assignment) above (§2.2) extends to all parallel constructs, and not only to those containing synchronizations. Under the assumption that this ordered single process run does not produce a waiting 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. (As a consequence of the semantic equivalence, the ordered single process run produces no execution fault.)

On the other hand, if the ordered single process run deadlocks, its behavior matches the one of the sequential version exactly, in the same sense, *up to* the WAIT statement instance on which the deadlock occurs.

It may be of interest to express the condition that the ordered single process run does not deadlock. Due to the above remark, we have to express that *any* WAIT *instance executed in the sequential version is not a deadlocking* WAIT*, assuming that the ordered single process run proceeded up to that point, with semantic equivalence to the sequential run up to that point.* Considering this semantic equivalence, we can express the no-deadlock condition ($\ll$ expresses the sequential execution order):

No-deadlock condition for an ordered single process run:[12]

$$\begin{aligned}\forall \text{ WAIT instance } \gamma,\ \mathrm{Exe}^{\mathsf{s}}(\gamma) \quad \Rightarrow \quad & \exists \text{ POST instance } \pi, \mathrm{Exe}^{\mathsf{s}}(\gamma) \wedge \mathrm{Exe}^{\mathsf{s}}(\pi) \\ & \wedge(\pi \ll \gamma)\ \&\ ([\varepsilon_\pi]^S \equiv [\varepsilon_\gamma]^S) \\ & \wedge(\ \forall \text{ CLEAR instance } \theta, \\ & \mathrm{Exe}^{\mathsf{s}}(\gamma) \wedge \mathrm{Exe}^{\mathsf{s}}(\theta)\ \&\ ([\varepsilon_\theta]^S \equiv [\varepsilon_\gamma]^S) \\ & \Rightarrow (\theta \ll \pi \vee \gamma \ll \theta\ )\ )\end{aligned}$$

[12] According to what we have mentioned above (§3.1) about the expression of $\mathrm{Exe}^{\mathsf{s}}$, this condition is formally more satisfactory under the hypothesis that the ordered single process run terminates (an assumption in our theorem), since otherwise, the expression of $\mathrm{Exe}^{\mathsf{s}}$ explicitly depends on the location of the infinite loop.

5 The theorem of semantic equivalence

We will now derive our theorem of semantic equivalence. We consider a parallel program written in the language we have previously defined (Section 2). We consider the sequential version of this program, which is supposed to conform to the rules of our language regarding sequential programs (among others, determinacy, and absence of execution faults). Considering a few properties of the ordered single process run (§4.4), we will derive the semantic equivalence between our parallel program and its sequential version.

Theorem 1 *Under the following hypotheses:*

i. Assumptions S1 and S2 (§3.4);

ii. No waiting deadlock in the ordered single process run;

iii. No infinite loop in the ordered single process run;

iv. For all statement instances $a(\mathbf{i})$ *and* $b(\mathbf{j})$,
$\mathrm{Exe}^{\mathsf{S}}(a(\mathbf{i})) \wedge \mathrm{Exe}^{\mathsf{S}}(b(\mathbf{j}))$ & $\mathrm{Dep}(a(\mathbf{i}), b(\mathbf{j})) \Rightarrow \mathrm{Pre}^{\mathsf{S}}(a(\mathbf{i}), b(\mathbf{j}))$;

the parallel program is semantically equivalent to its sequential version. Especially, no parallel run can deadlock (in waiting or in fault), nor infinitely loop.

Proof : We consider a program instance, obtained by giving values to the parameters. Then, there is only one ordered single process run of this program, which will be denoted **S**, whereas there are generally many possible runs of the parallel version. We consider one of them, which will be denoted **P**. In what follows, the predicates Exe and Pre, defined in Section 3, are the execution and precedence predicates associated to this parallel execution **P**.

We will first derive the semantic equivalence extended to all statement instances executed in this parallel run **P** and all variables involved in them **(points 1 to 4)**; finally, we will prove that, conversely, all instances executed in the ordered single process run **S** are executed in this parallel run **P (point 5)**.

Point 1 : We will consider the *execution date function* (§4.3) associated to the parallel run **P** we are considering.

Let τ be a date such that the following recurrence assumption holds:

> *Semantic equivalence up to date* $\tau - 1$: for any statement instance α executed strictly before τ in **P**, α is also executed in **S**; moreover, any variable reference involved in α as input (including the event references) points to the same variable in both runs, and that variable has been written by the same other instance in both runs. Correlatively, all variables written, in **P**, before date τ (including event variables), underwent the same computations, due to the same instances, in the same order, in **S**.

This hypothesis indeed expresses a semantic equivalence between **P** and **S** as regards all the inputs and outputs of all statement instances executing before τ in **P**. (As a consequence, especially, **P** does not produce any execution fault before τ.)

We wish to prove that this semantic equivalence extends to date τ. Considering that it obviously applies to the program start, that will ensure the semantic equivalence along all the parallel run $\mathbf{P}$.

First of all, we have to introduce a lemma.

Lemma 2 *We assume the hypothesis of semantic equivalence up to date $\tau - 1$, and a statement instance γ executed at τ in $\mathbf{P}$. 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 $\mathrm{Pre}(\alpha, \gamma)$.*

This result also holds if we replace γ here by any statement instance executed before τ in $\mathbf{P}$.

The proof of this lemma is provided in Appendix B.

We have to show that, for any statement instance γ which happens to be executed at date τ in $\mathbf{P}$, the semantic equivalence propagates to γ. First, we will prove that the semantic equivalence extends to the *inputs* of γ. We will be led to consider separately the case when such an input is not an event **(point 2)**, and the case when it is **(point 3)**. Then, we will notice that this execution of γ does not produce any execution fault; afterwards we will show that the semantic equivalence extends to the *outputs* of γ, by deriving that there is no race condition between instances executing at date τ **(point 4)**. Thus, the semantic equivalence up to date τ will be derived.

We consider a statement instance γ coming to be executed at date τ in $\mathbf{P}$. According to Lemma 1 (§4.2), the fact that γ is *reached or persistently 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, since Pre^0 is common to all runs). Due to the recurrence hypothesis of semantic equivalence up to date $\tau - 1$, these same statement instances β execute in $\mathbf{S}$, and identically determine that γ is reached or persistently pending in $\mathbf{S}$. Therefore, γ is executed in $\mathbf{S}$ (due to hypotheses *(ii)* and *(iii)*, no instance remains *waiting* nor *pending* in $\mathbf{S}$): 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 date $\tau - 1$, 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 $\mathbf{S}$; 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 (§2.1) 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 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.

Point 2 : Let us first consider the case when x is not an event variable.

Let β be the statement instance which computes the value of x used by γ as input in $\mathbf{S}$. β exists, due to the determinacy condition (§2.3). We will first show that $\mathrm{Exe}(\beta)$ and $\mathrm{Pre}(\beta, \gamma)$, which will imply that β is executed in $\mathbf{P}$ 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 $\mathbf{P}$, *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 $\mathbf{S}$; we have $\mathrm{Exe}^{\mathsf{s}}(\gamma)$ as we have seen; we have $\mathrm{Dep}(\beta, \gamma)$ because β computes a variable (which is not an event) used by γ *in* $\mathbf{S}$, the run in reference to which Dep is defined. Therefore, according to *(iv)*, we get $\mathrm{Pre}^{\mathsf{s}}(\beta, \gamma)$. According to Lemma 2, having $\mathrm{Exe}^{\mathsf{s}}(\beta)$ and $\mathrm{Pre}^{\mathsf{s}}(\beta, \gamma)$ implies $\mathrm{Exe}(\beta)$ and $\mathrm{Pre}(\beta, \gamma)$. Now, in order to confirm that the semantic equivalence extends to the input x of γ, we need to prove that, in $\mathbf{P}$, the statement instance which computes x for γ is indeed β, and not some other statement instance δ interfering between β and γ.

Such an instance δ would execute before τ, so that its output x could be available for γ; so, due to the recurrence hypothesis, δ would execute in $\mathbf{S}$ too, and compute the same variable x. Therefore, there would be a dependence Dep between β and δ, which, together with $\mathrm{Exe}^{\mathsf{s}}(\beta)$ and $\mathrm{Exe}^{\mathsf{s}}(\delta)$, would imply a precedence $\mathrm{Pre}^{\mathsf{s}}$ between β and δ, according to *(iv)*. In what direction would this precedence stand? If we had $\mathrm{Pre}^{\mathsf{s}}(\delta, \beta)$, Lemma 2 applied to β would then imply $\mathrm{Pre}(\delta, \beta)$, which, together with $\mathrm{Pre}(\beta, \gamma)$, would prevent the value of x computed by δ from being read by γ. Therefore, we would get $\mathrm{Pre}^{\mathsf{s}}(\beta, \delta)$: δ would indeed execute after β in $\mathbf{S}$. Since β indeed computes x for γ in $\mathbf{S}$, this would imply that δ executes *after* γ in $\mathbf{S}$. 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 would execute in $\mathbf{P}$ before τ, so Lemma 2 would apply to δ: $\mathrm{Exe}^{\mathsf{s}}(\gamma) \wedge \mathrm{Pre}^{\mathsf{s}}(\gamma, \delta)$ would imply $\mathrm{Pre}(\gamma, \delta)$, which would contradict the fact that δ executes before τ in $\mathbf{P}$. Therefore, there is no such instance δ.

Point 3 : Let us now consider the case when x is an event variable. This implies that γ is a WAIT.

According to the recurrence hypothesis, all computations of x before τ are identical in both runs. Let ρ be the statement instance which last modified the event x in this past common history. Since γ is executed at date τ in $\mathbf{P}$, ρ exists and is a POST (and not a CLEAR).

Let π be the statement instance which writes x for γ in $\mathbf{S}$. Since γ is executed in $\mathbf{S}$, 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. This will end our Point 3. If this were not the case, Assumptions S1 and S2 (§3.4) would imply that, either $\mathrm{Pre}^0(\gamma, \rho)$ (then, ρ would execute *after* γ in $\mathbf{S}$), or there would be a CLEAR instance θ involving event x in $\mathbf{S}$, such that $\mathrm{Pre}^0(\rho, \theta) \wedge \mathrm{Pre}^0(\theta, \gamma)$. Let us successively rule out these two cases.

> *The case* $\mathrm{Pre}^0(\gamma, \rho)$. Since ρ executes before τ in $\mathbf{P}$, Lemma 2 applies: $\mathrm{Exe}^{\mathsf{s}}(\gamma) \wedge \mathrm{Pre}^0(\gamma, \rho)$ would imply $\mathrm{Pre}(\gamma, \rho)$, which would contradict the fact that ρ executes before γ in $\mathbf{P}$.
>
> *The case* $\mathrm{Pre}^0(\rho, \theta) \wedge \mathrm{Pre}^0(\theta, \gamma)$. According to Lemma 2, having $\mathrm{Exe}^{\mathsf{s}}(\theta)$ and $\mathrm{Pre}^0(\theta, \gamma)$ would imply $\mathrm{Pre}(\theta, \gamma)$: θ would execute before τ in $\mathbf{P}$. Then, the fact that $\mathrm{Pre}^0(\rho, \theta)$ would imply a similar precedence in $\mathbf{P}$ (due to Lemma 2 applied to θ): x would be cleared by θ between ρ and γ in $\mathbf{P}$, which would contradict the fact that ρ is the last statement instance writing x before τ.

Point 4 : At this point, we have proved that, given the recurrence hypothesis of semantic equivalence up to date $\tau - 1$ and a statement instance γ executed at τ in the parallel run $\mathbf{P}$ being considered, γ is also executed in $\mathbf{S}$ 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 $\mathbf{P}$ and $\mathbf{S}$.

This input equivalence implies that any output reference η of γ points to the same variable (denoted y) in both runs. Thus, in both runs, γ performs the same computations, on the same inputs, producing the same outputs. Therefore, the execution of γ in $\mathbf{P}$ produces no execution fault. To make sure that the semantic equivalence extends to the outputs of γ at date τ, it is sufficient to check that there is no conflict, i.e. no dependence relation, among the statement instances γ_i, say γ_1 and γ_2, coming to be executed at date τ in $\mathbf{P}$. Due to the semantic equivalence of all input references, input variables and output references of γ_1 and γ_2, such a conflict would also show up in $\mathbf{S}$.

In case the conflicting references would not be event references, such a conflict would translate into a dependence relation Dep, e.g. $\mathrm{Dep}(\gamma_1, \gamma_2)$, which, according to *(iv)*, would imply $\mathrm{Pre}^{\mathsf{s}}(\gamma_1, \gamma_2)$. Lemma 2 applies: having $\mathrm{Exe}^{\mathsf{s}}(\gamma_1)$ and $\mathrm{Pre}^{\mathsf{s}}(\gamma_1, \gamma_2)$ would imply $\mathrm{Pre}(\gamma_1, \gamma_2)$, which contradicts the fact that γ_1 and γ_2 execute at date τ.

In case the conflicting references refer to an event variable, first of all, γ_1 and γ_2 would not both be WAIT instances: at least one of them has to output an event (so that there is a conflict), i.e. be a CLEAR or a POST. Two cases have to be considered.

- The case when γ_1 and γ_2 are a POST and a WAIT, say in that order, involving an event variable ε (in both runs $\mathbf{S}$ and $\mathbf{P}$). Let π be the POST instance which writes ε for γ_2 in $\mathbf{S}$. We have $\mathrm{Sync}^{\mathsf{s}}(\pi, \gamma_2)$. We have noticed (point 3) that π executes before τ in $\mathbf{P}$. π is therefore different from γ_1. We will show that this leads to a contradiction. According to assumptions S1 and S2 (§3.4), either we would get $\mathrm{Pre}^0(\gamma_2, \gamma_1)$, or there would exist a CLEAR instance θ involving the event variable ε in $\mathbf{S}$, such that $\mathrm{Pre}^0(\gamma_1, \theta) \wedge \mathrm{Pre}^0(\theta, \gamma_2)$. Let us rule out these two cases.

  The case $\mathrm{Pre}^0(\gamma_2, \gamma_1)$. Since γ_1 would execute at date τ in $\mathbf{P}$, Lemma 2 applies: $\mathrm{Exe}^{\mathsf{s}}(\gamma_2) \wedge \mathrm{Pre}^0(\gamma_2, \gamma_1)$ would imply $\mathrm{Pre}(\gamma_2, \gamma_1)$, which would contradict the fact that γ_2 would execute at date τ, like γ_1, in $\mathbf{P}$.

  The case $\mathrm{Pre}^0(\gamma_1, \theta) \wedge \mathrm{Pre}^0(\theta, \gamma_2)$. According to lemma 2, having $\mathrm{Exe}^{\mathsf{s}}(\theta)$ and $\mathrm{Pre}^0(\theta, \gamma_2)$ would imply that θ would execute before τ in $\mathbf{P}$. Then, however, $\mathrm{Pre}^0(\gamma_1, \theta)$ would imply a similar precedence in $\mathbf{P}$ (lemma 2 applied to θ): γ_1 would execute before τ in $\mathbf{P}$, which contradicts the assumption that γ_1 would execute at τ.

- In the other cases, let us apply Assumption S1. We have a precedence Pre^0 between γ_1 and γ_2, e.g. $\mathrm{Pre}^0(\gamma_1, \gamma_2)$. According to lemma 2, having $\mathrm{Exe}^{\mathsf{s}}(\gamma_1)$ and $\mathrm{Pre}^0(\gamma_1, \gamma_2)$ would imply $\mathrm{Pre}(\gamma_1, \gamma_2)$, which would contradict the fact that γ_1 and γ_2 both execute at date τ.

Point 5 : We have thus proved that any statement instance executed in some parallel run $\mathbf{P}$ is also executed in $\mathbf{S}$, 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 $\mathbf{P}$.

As an immediate consequence, this parallel run **P** cannot enter an infinite loop: indeed, if some WHILE construct looped indefinitely in **P**, the semantic equivalence along **P** would imply a similar infinite loop in **S**, which is ruled out by *(iii)*. That same semantic equivalence along **P** also ensures that **P** produces no execution fault.

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

Let us apply Lemma 1 to the execution of γ is **S**. According to hypotheses *(ii)* et *(iii)*, there is no *persistently waiting* nor *persistently pending* statement instance in **S**. Therefore, Lemma 1 implies that $\mathrm{Exe}^{\mathsf{s}}(\gamma)$ is dependent on statement instances β which precede γ Pre^0-wise and are all executed in **S**, before γ. By definition of γ, these β are executed in **P**, with semantic equivalence, as shown previously. Therefore, Lemma 1 and the non execution of γ in **P** imply that, in **P**, either γ is persistently pending, or γ is a persistently waiting WAIT. Let us examine these two possibilities.

In the absence of infinite loops and execution faults in **P**, the sequential assignment assumption (§2.2) implies that a statement instance can be persistently pending only if some WAIT instance ω, ranking before it in the sequential order, is reached and deadlocks in **P**. Being reached in **P**, ω is reached too, and executes, in **S**. Its deadlock in **P**, and its preceding γ in **S**, would contradict the definition of γ.

Finally, there remains the case when γ is a reached and deadlocking WAIT. Let ε_γ be the event γ involves in **S**. First of all, we must show that γ involves the same event in **P**.

Let x be a variable other than ε_γ, if any, involved as an input of γ in **S** (x exists in case of dynamic reference). By definition of γ, all instances preceding γ in **S** are executed, with semantic equivalence, in **P**. It is therefore the case for the instance β_x which computes x for γ in **S** (for any such x, β_x exists, due to the determinacy condition (§2.3)).

Is there a possibility that, in **P**, x is rewritten after β_x ? If this were so, it would be through an instance denoted δ_x. Then, δ_x executes in **S**, with semantic equivalence, therefore after γ. We would then get $\mathrm{Dep}(\gamma, \delta_x)$ (dependence associated to x), $\mathrm{Exe}^{\mathsf{s}}(\gamma)$, $\mathrm{Exe}^{\mathsf{s}}(\delta_x)$, therefore $\mathrm{Pre}^{\mathsf{s}}(\gamma, \delta_x)$ by *(iv)*. According to Lemma 2 applied to δ_x, having $\mathrm{Exe}^{\mathsf{s}}(\gamma)$ and $\mathrm{Pre}^{\mathsf{s}}(\gamma, \delta_x)$ would imply $\mathrm{Exe}(\gamma)$, which would contradict the deadlock on γ in **P**. Hence, x is not written again after β_x in **P**.

Here we will use a lemma:

Lemma 3 *We assume the semantic equivalence along* **P**. *Let* γ *be a* WAIT *instance, deadlocking in* **P**. *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* $\mathrm{Pre}(\alpha, \gamma)$.

The proof of this lemma is provided in Appendix B, after the proof of Lemma 2.

For any such variable x, we have $\mathrm{Exe}^{\mathsf{s}}(\beta_x)$ and $\mathrm{Dep}(\beta_x, \gamma)$. We have $\mathrm{Exe}^{\mathsf{s}}(\gamma)$. So, according to *(iv)*, we have $\mathrm{Pre}^{\mathsf{s}}(\beta_x, \gamma)$. Then, lemma 3 gives $\mathrm{Pre}(\beta_x, \gamma)$.

Together with the fact that x is not rewritten in **P** after β_x, this precedence $\mathrm{Pre}(\beta_x, \gamma)$, from an executed instance to a deadlocking WAIT, implies (§3.3) that, *if* x is an input of γ in **P** (which we do not know yet, at this point!), the value of x input by γ is (and remains) the one computed by β_x in **P** as well as in **S**. Checking that x is indeed an input of γ in **P** is performed through a

recurrence on the indirection order of x, similar to the one we used in Point 1. It therefore turns out that the event involved in γ in $\mathbf{P}$ indeed is (and remains) ε_γ, the same as in $\mathbf{S}$.

By definition of γ, all instances previous to γ in $\mathbf{S}$ are executed, with semantic equivalence, in $\mathbf{P}$. This is the case, therefore, for the POST instance π which sets value *posted* to ε_γ for γ in $\mathbf{S}$. So, the deadlock on γ would imply that, in $\mathbf{P}$, some CLEAR statement instance θ executes and clears ε_γ after π, and before γ used it. θ also executes in $\mathbf{S}$ and clears the same event (as shown above, about the semantic equivalence extended to all instances executed in $\mathbf{P}$).

Assumption S1 (§3.4) implies that there is a relation Pre^0 between θ and π, and between θ and γ. We cannot have $\mathrm{Pre}^0(\theta, \pi)$ because this would imply (Lemma 2 applied to π) that θ executes before π in $\mathbf{P}$. So, we have $\mathrm{Pre}^0(\pi, \theta)$. We cannot have $\mathrm{Pre}^0(\theta, \gamma)$ because, together with $\mathrm{Pre}^0(\pi, \theta)$, this would imply an execution order in $\mathbf{S}$: π before θ before γ, and π would not post for γ, as it is supposed to. So, we have $\mathrm{Pre}^0(\gamma, \theta)$.

θ executes in $\mathbf{P}$, so Lemma 2 applies to θ: γ executes in $\mathbf{S}$ and we have $\mathrm{Pre}^0(\gamma, \theta)$; so, γ executes in $\mathbf{P}$ before θ, which contradicts the deadlock on γ.

This ends the derivation of our theorem.

◀

Deadlocks and infinite loops

Along the lines of the above derivation, it is straightforward to see what may happen whenever the ordered single process run $\mathbf{S}$ produces a waiting deadlock, contrarily to hypothesis *(ii)*, or infinitely loops, contrarily to hypothesis *(iii)* – keeping in mind that both circumstances are *unwanted* in our framework.

In case $\mathbf{S}$ is endless, the statement instances executing in $\mathbf{S}$ execute too in any parallel run $\mathbf{P}$ – hence an infinite loop in $\mathbf{P}$ – but, in case the infinite loop is nested in a parallel construct, some units of work in this parallel construct may execute in $\mathbf{P}$, although they are not reached in $\mathbf{S}$.

In case $\mathbf{S}$ produces a waiting deadlock, the statement instances executing in $\mathbf{S}$ execute too in any parallel run $\mathbf{P}$, but, in the (usual) case when the deadlocking WAIT instance is located in a parallel construct, it may happen that some units of work in this parallel construct execute in $\mathbf{P}$, although they are not reached in $\mathbf{S}$. It may even occur that the event corresponding to the deadlocking WAIT instance happen to get *posted* thus, due to a POST instance located after the WAIT instance in the sequential order. Such an occurrence will then arise *randomly*, essentially depending on the number of available processes and the loading of units of work on these processes, two aspects of the program execution the user is not supposed to have any control on.

6 An example; an incrementality property

We will illustrate the possible applications of our theorem on a little example, together with highlighting an interesting property of *incrementality* of a check-and-repair procedure inspired by our result.

Let us consider a parallel program instance in which two statement instances α and β, both executed and in a dependence relation $\mathrm{Dep}(\alpha, \beta)$, happen not to be in a precedence relation $\mathrm{Pre}^{\mathsf{s}}(\alpha, \beta)$.

We consider a check-and-repair procedure which, in presence of such an unpreserved dependence, aims at *reinforcing* the precedence $\mathrm{Pre}^{\mathsf{s}}$ so that the dependence being considered becomes preserved

by the precedence thus reinforced. We must indeed obtain a *reinforcement*, in the sense that, for the whole program, we must get $\mathrm{Pre}^{\mathsf{s}}_{before} \Rightarrow \mathrm{Pre}^{\mathsf{s}}_{after}$ where $\mathrm{Pre}^{\mathsf{s}}_{before}$ and $\mathrm{Pre}^{\mathsf{s}}_{after}$ denote the precedences $\mathrm{Pre}^{\mathsf{s}}$ before and after the repair, respectively. (If this reinforcement of $\mathrm{Pre}^{\mathsf{s}}$ is performed by adding or displacing synchronizations, it must be checked that assumptions S1 and S2 dealing with synchronizations (§3.4) remain true after the repair.) Thus, the implications $\mathrm{Dep} \Rightarrow \mathrm{Pre}^{\mathsf{s}}$ already checked at this point, will remain afterwards.

Here, the important point to notice is that this *repair* of this lack of dependence preservation *does not compromise* the other verifications already performed at this point; the dependences which have already been checked to be preserved at this point will remain preserved after the repair: there is no need to *revisit* the whole program source at this point. This **incremental** aspect of the check-and-repair procedure is a direct consequence of the reference of our theorem to the sequential semantics; it would not be ensured if our theorem had referred to the parallel semantics in some essential way (because such a reinforcement of the precedence may indeed radically change the behavior of the parallel program).

Let us study an example. Figure 8 shows a portion of program. The sequential version (which provides the reference for the required semantics) is on the right.

On this portion of program, we assume that the undisplayed statements (represented by "...") do not contain references $A()$ or P, nor synchronizations. Besides, we assume that this portion of program is not itself nested within another parallel construct.

This program is *incorrect* in our sense, since it exhibits memory conflicts involving variable P: dependences from `c1` and `c2`, to `d1`, `a` and `p`, are not preserved. This will have to be fixed, but, due to the *incrementality* property, this must not prevent us from checking whether dependences involving variables $A()$ are preserved.

Let us focus our attention on the dependence between statements `b` and `a`, involving references $A()$. We notice that this dependence, as well as the corresponding precedences that we will find, depend on variable P. Since the sequential version sets the semantic reference here (in virtue of our theorem), we have not to bother (at this point) with the memory conflicts mentioned above involving P, and we consider the "sequential" value of P (which equals 2 or 3) at `b` and `a`. Since $P > 0$, we have a dependence from `a` to `b` involving the variable array $A()$, which will have to be preserved so that the values of $A()$ computed by `a` be, in case of need, those used by `b`, P iterations later.

We choose to include the execution conditions $\mathrm{Exe}^{\mathsf{s}}$ in our predicates Dep. We obtain:

$$\mathrm{Dep}(a_x, b_y) = (1 \leq x \leq N) \wedge (1 \leq y \leq N) \wedge (y > x) \wedge (y = x + [\![P]\!]@a_x)$$

Let us explain this relation. a_x (resp. b_y) denotes the instance of `a` (resp. `b`) corresponding to iteration x (resp. y). The four terms of this conjunction denote, respectively: the condition of execution of `a`; the condition of execution of `b`; the condition for a_x to execute before b_y in the sequential version; and the condition that the two instances a_x and b_y access to the same variable in the sequential version. Let us remind that $[\![P]\!]@a_x$ denotes the value of P read by instance a_x in the sequential version[13].

[13] As a simplification, we assumed here that N is a parameter. In case it is a variable, in the expressions displayed here, "N" should be replaced by "$[\![N]\!]@d2$".

```
      B=...                                B=...
      ...                                  ...
      psections
        section
          ...                              ...
          if(B) then                       if(B) then
c1:         P=2                              P=2
          else                             else
c2:         P=3                              P=3
          endif                            endif
f:        ...                              ...
        section
d1:       do J=1,P
q:          post E(J)
          enddo
          ...                              ...
d2:       pdo I=1,N                        do I=1,N
            ...                              ...
w:          wait E(I)
b:          ...=A(I)                         ...=A(I)
a:          A(I+P)=...                       A(I+P)=...
p:          post E(I+P)
          endpdo                           enddo
      endpsections
      ...                                  ...
```

Figure 8: An example of program checking. The sequential version is on the right.

```
      B=...
      ...
      CLEAR F                          addendum
      psections
        section
          ...
          if(B) then
c1:         P=2
          else
c2:         P=3
          endif
          POST F                       addendum
f:        ...
        section
          WAIT F                       addendum
d1:       do J=1,P
q:          post E(J)
          enddo
          ...
d2:       pdo I=1,N
            ...
w:          wait E(I)
b:          ...=A(I)
a:          A(I+P)=...
p:          post E(I+P)
          endpdo
      endpsections
      CLEAR F                          addendum
      ...
```

Figure 9: The previous example, with addenda

Let us now look for precedences which may stand from `a` to `b`. It is easy to detect that the control precedence Pre^0 is not sufficient, and that synchronization relations $\mathrm{Sync}^{\mathtt{s}}$ have to be involved. More precisely, let us consider a path:

$$a_x \to p_u \rightsquigarrow w_z \to b_y$$

(We use notations introduced in §3.3). Let us express the corresponding precedence relation:

$$\mathrm{Pre}^{\mathtt{s}}(a_x, b_y) = \mathrm{Pre}^0(a_x, p_u) \wedge \mathrm{Sync}^{\mathtt{s}}(p_u, w_z) \wedge \mathrm{Pre}^0(w_z, b_y)$$

As regards the synchronization $\mathrm{Sync}^{\mathtt{s}}$, we assume that it has been checked that a synchronization indeed stands from `p` to `w`, in the sense that, under the sequential semantics, the events $E()$ are not *posted* yet when the control enters the portion of program considered here.

We get:

$$\mathrm{Pre}^{\mathtt{s}}(a_x, b_y) = (x = u) \wedge (1 \leq u \leq N) \wedge (u + [\![P]\!]@p_u = z) \wedge (1 \leq z \leq N) \wedge (z = y)$$

We have to check whether there is a precedence path of this form such that $\mathrm{Dep}(a_x, b_y) \Rightarrow \mathrm{Pre}^{\mathtt{s}}(a_x, b_y)$, i.e. whether there is an instance p_u of `p` and an instance w_z of `w` such that we get this implication. In the above expression of $\mathrm{Pre}^{\mathtt{s}}$, we try to eliminate u and z[14]:

- Eliminate u by $(x = u)$:

$$\mathrm{Pre}^{\mathtt{s}}(a_x, b_y) = (1 \leq x \leq N) \wedge (x + [\![P]\!]@p_x = z) \wedge (1 \leq z \leq N) \wedge (z = y)$$

- Eliminate z by $(z = y)$:

$$\mathrm{Pre}^{\mathtt{s}}(a_x, b_y) = (1 \leq x \leq N) \wedge (x + [\![P]\!]@p_x = y) \wedge (1 \leq y \leq N)$$

to be compared with:

$$\mathrm{Dep}(a_x, b_y) = (1 \leq x \leq N) \wedge (1 \leq y \leq N) \wedge (y > x) \wedge (y = x + [\![P]\!]@a_x)$$

We indeed get $\mathrm{Dep}(a_x, b_y) \Rightarrow \mathrm{Pre}^{\mathtt{s}}(a_x, b_y)$, provided that $[\![P]\!]@p_x = [\![P]\!]@a_x$, which is verified as soon as we check that P is not rewritten between the two readings *in the sequential version*, which is what we have assumed.

Now we must take care of variable P. We have mentioned that dependences involving P, from `c1` and `c2` to `d1`, `a` and `p` are not preserved. A somewhat radical way to fix this problem, is to "resequentialize" the PSECTIONS – but this may be costly if the execution of statements `f` is time-consuming. Rather, a more subtle fix consists in introducing a new synchronization from the end of P's computation to the beginning of P's use. This synchronization should not involve an event $E()$ already in use in loops `d1` and `d2`. A possibility is to use another event (here F), duly reinitialized before and after this use in case it already exists in the rest of the program. Under this condition, all the previously checked dependence preservations (e.g. those involving $A()$ here) still stand. A possible fix of our portion of program is shown in Figure 9.

[14] These principles of algorithmic treatment of dependences and precedences are explained in [26, 6, 7].

7 Conclusion

We have studied a property of *correctness* of parallel programs in the shared-memory programming model. This model is widely used in scientific computing and implementable on many parallel machines, including distributed-memory ones.

We have considered a parallel language obtained by adding a few parallel constructs (parallel loops, parallel sections and event synchronizations) within a fairly standard sequential imperative language (we do not make *static control* assumptions often considered in the literature). We are interested in a property of **sequential correctness** defined as a *semantic equivalence* between a parallel program and its *sequential version*, that we define. In this framework, a parallel program is viewed as the result of a *parallelization* of some given sequential program, and it is required that the results of any run of the parallel program be identical to those of this sequential program; the improvement sought through the parallelization lies only in the ability to obtain these results faster.

The main object of this paper is to present and derive a theorem which states sufficient conditions for this sequential correctness property. The important aspect of this result is the fact that these sufficient conditions (mainly preservation of dependences) refer to the semantics of the sequential version only: they do not refer in any way to presupposed properties of some specific parallel run of the program. In other words, due to this result, checking that *any possible run* of some parallel program being considered will meet the desired correctness requirements boils down to checking some predicates pertaining to some *sequential* program. Especially, this reference to a sequential semantics allows to use all resources of *dataflow analysis* usually applied to the study of sequential programs, in the process of checking these predicates.

The derivation of the theorem makes use of a notion of *execution date*. In a run of a parallel program, every execution of a statement is attributed a date, such that the outputs of computations made "at some date" are not available as inputs before "the next date". This execution date feature can be described as a *causality-preserving time discretization*. The main part of the proof makes use of a recurrence on the date, in order to derive that the semantic equivalence between the parallel program being considered and its sequential version, propagates along any possible parallel execution. The requirement to refer to the *sequential* semantics, whereas the recurrence proceeds along a *parallel* run, explains the intricacy of the proof.

A preliminary application of this theorem, dealing with a subset of our language, is developed in [7, 26]. There are prospects that this result could be applied through tools to verify a wide range of programs.

Appendix A: Proof of Lemma 1

Lemma 1, introduced in §4.2, specifies 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 $\mathrm{Pre}^0(\beta, \alpha)$. (Let us remind that $\psi(\alpha)$ denotes the condition for α to be executed or persistently waiting or persistently pending.)

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, $\psi(\alpha)$ 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. If at least one of these β produces an execution fault, this implies $\psi(\alpha) = false$.

We have $\mathrm{Exe}(\alpha) = \psi(\alpha)$ *except in the three following cases:*

- *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 persistently 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 (or in an infinite loop situation in the latter case).*

- *α is an instance of the first statement in a* PDO *body or in a* SECTION *of a* PSECTIONS*, without being an instance of a* WAIT*: then, $\psi(\alpha)$ expresses the condition for α to be executed or persistently pending; the latter possibility occurs in case of a deadlock or infinite loop.*

- *α is an instance of a* ENDWHILE*: then, $\psi(\alpha)$ expresses the condition for α to be executed or persistently pending; the latter possibility occurs in case the* WHILE *infinitely loops, or in case of a deadlock or infinite loop within an iteration.*

Proof : We will refer to the previously specified execution model (§2.2). We will successively examine all possible cases in our language.

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. In the following cases, we assume that a is not a WAIT.

- a is the first statement in a PDO body: then, let c be the loop header; let $\mathbf{j}$ be the (possibly empty) index vector of c and $\mathbf{j}::k$ be the index vector of a. For any instance $\alpha = a(\mathbf{j}::k)$ to be executed, it is necessary that the corresponding instance $c(\mathbf{j})$ be executed without fault; conversely, the execution of $c(\mathbf{j})$, through the evaluation of its loop bounds, fully characterizes which instances $a(\mathbf{j}::k)$ are *reached or persistently pending*. Thus, $\psi(a(\mathbf{j}::k))$ is fully characterized by the execution of $c(\mathbf{j})$; and we have $\mathrm{Pre}^0(c(\mathbf{j}), a(\mathbf{j}::k))$. *However*, an instance $a(\mathbf{j}::k)$ such that $\psi(a(\mathbf{j}::k))$ may be persistently pending instead of finally executing, in deadlock or infinite loop situations.

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

- a is a ENDWHILE: then, let c be the corresponding WHILE header; let $\mathbf{j}$ be the (possibly empty) index vector of a and c. For any instance $\alpha = a(\mathbf{j})$ to be *reached or persistently pending*, it is necessary and sufficient that $c(\mathbf{j})$ be executed. So, $\psi(\alpha)$ fully depends on the execution

of $c(\mathbf{j})$, which precedes $a(\mathbf{j})$ Pre^0-wise. However, there again, α may be persistently pending whenever, either the WHILE construct infinitely loops, or there is a deadlock or infinite loop within an iteration.

At this point, we have just examined the four 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 persistently pending* – not meaning that it will finally be executed –; for an initial instance in a unit of work which happens to be a WAIT, to be *persistently pending or reached* – not meaning that it will finally get reached, or executed; and for a ENDWHILE to be *executed or persistently pending*.

Now, let us examine the other cases.

- a is a ENDPSECTIONS. Then, let c be the corresponding head of the PSECTIONS, and $\mathbf{j}$ the index vector common to c and a. The execution of $a(\mathbf{j})$ is fully determined by the execution (without fault) of the corresponding statement instances which end all the units of work in this PSECTIONS. All these instances precede $a(\mathbf{j})$ Pre^0-wise. They are specified independently of the run.

- a is a ENDPDO. Then, let c be the corresponding loop head and $\mathbf{j}$ be the index vector common to c and a. Then, the condition for $a(\mathbf{j})$ to be executed is that $c(\mathbf{j})$ be executed without fault and that, in case the index range is not empty (a circumstance determined by the execution of $c(\mathbf{j})$), the statement instances which end all the parallel units of work be executed without fault. All these instances precede $a(\mathbf{j})$ Pre^0-wise, and are specified by the execution of $c(\mathbf{j})$.

- a is the first statement in a DO loop body. Let c be the loop head, $\mathbf{j}$ be the index vector of c and $\mathbf{j} :: k$ be the index vector of a. Then, the execution without fault of $c(\mathbf{j})$ fully determines the range of values of k which will be considered, and for any of these values k, the condition for $a(\mathbf{j} :: k)$ to be executed is the execution without fault of $c(\mathbf{j})$ 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 $a(\mathbf{j} :: k)$ Pre^0-wise; the former is specified independently of the run, and specifies whether the latter is involved.

- a is a ENDDO. Then, let c be the corresponding loop head and $\mathbf{j}$ be the index vector common to c and a. Then, the condition for $a(\mathbf{j})$ to be executed is that $c(\mathbf{j})$ be executed without fault and that, in case the index range is not empty (a circumstance determined by the execution of $c(\mathbf{j})$), the last loop body instance (specified by the execution of $c(\mathbf{j})$) be executed without fault. Both statement instances precede $a(\mathbf{j})$ 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 without fault of the corresponding instance of c.

- a is a ENDIF. Let c be the corresponding IF, and $\mathbf{j}$ be the index vector common to a and c. The condition for $a(\mathbf{j})$ to be executed is the execution without fault of $c(\mathbf{j})$ and of some instance – ending the THEN part or the ELSE part – specified by the execution of $c(\mathbf{j})$. These two instances precede $a(\mathbf{j})$ Pre^0-wise.

- a is the first statement in a WHILE body, i.e. the test of the boolean condition. Let $\mathbf{j}$ be the index vector of the WHILE head c and $\mathbf{j}::k$ be the index vector of a. The condition for $a(\mathbf{j}::k)$ to be executed is the execution of $c(\mathbf{j})$ and (for the iterations other than the first) the execution without fault of the last statement instance in the WHILE body corresponding to the previous iteration. Both instances precede $a(\mathbf{j}::k)$ Pre^0-wise, and are specified independently of the run.

- a is the second statement in a WHILE body, i.e. the statement following the test statement we have just considered, here denoted b. Let $\mathbf{i}$ be the index vector of b and a. The execution of $a(\mathbf{i})$ is fully determined by the execution without fault of $b(\mathbf{i})$ (and the result of the test done by $b(\mathbf{i})$)

- The remaining case is the most straightforward: a has an immediate predecessor b, of same index vector $\mathbf{j}$, and the condition for $a(\mathbf{j})$ to be executed is exactly that $b(\mathbf{j})$ be executed without fault.

◀

Appendix B: Proofs of Lemmas 2 and 3

Proof of Lemma 2

Lemma 2 is involved in the proof of Theorem 1 (Section 5), in point 1. We assume the hypotheses of Theorem 1.

Lemma 2 *We assume the hypothesis of semantic equivalence up to date $\tau - 1$, and a statement instance γ executed at τ in $\mathbf{P}$. 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* $\mathrm{Pre}(\alpha, \gamma)$.

This result also holds if we replace γ here by any statement instance executed before τ in $\mathbf{P}$.

Proof : We will prove the result involving γ; the latter extension will be straightforward. After giving a preliminary remark, we will prove the result in the restricted case when we have $\mathrm{Pre}^0(\alpha, \gamma)$; afterwards, we will derive the extension to the case when we have $\mathrm{Pre}^{\mathsf{s}}(\alpha, \gamma)$ and not $\mathrm{Pre}^0(\alpha, \gamma)$.

Preliminary remark. In a deadlock or infinite loop situation, let α be a persistently waiting or pending statement instance. No instance β such that $\mathrm{Pre}^0(\alpha, \beta)$ can be executed. This straightforwardly results from the execution model and the definition of Pre^0. (In other words, the execution flow cannot *by-pass* a deadlock nor an infinite loop.)

Considering Pre^0. So we have $\mathrm{Exe}^{\mathsf{s}}(\alpha)$ and $\mathrm{Pre}^0(\alpha, \gamma)$. We straightforwardly have $\mathrm{Pre}(\alpha, \gamma)$, because the control precedence Pre^0 is common to all runs (§3.3). Suppose that some instance α executed in $\mathbf{S}$ and such that $\mathrm{Pre}^0(\alpha, \gamma)$, is not executed in $\mathbf{P}$. (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 $\mathrm{Pre}^0(\alpha_1, \alpha)$ is also executed in $\mathbf{S}$ but not in $\mathbf{P}$, which will then lead to a contradiction.

Let us apply Lemma 1 to the execution of α in $\mathbf{S}$. According to hypotheses *(ii)* and *(iii)* of Theorem 1, there is no *persistently waiting or pending* statement instance in $\mathbf{S}$. Therefore, according to Lemma 1, $\mathrm{Exe}^{\mathsf{s}}(\alpha)$ depends on some instance(s) β_i executed in $\mathbf{S}$, and such that $\mathrm{Pre}^0(\beta_i, \alpha)$, hence $\mathrm{Pre}^0(\beta_i, \gamma)$. Therefore, if all these β_i were executed in $\mathbf{P}$, they would be executed before date τ,

hence the semantic equivalence, which would imply that α be reached or persistently pending in **P**. So, in **P**, 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 waiting or pending, thus participating in a deadlock or infinite loop situation. This possibility is ruled out by the above preliminary remark: having $\mathrm{Pre}^0(\alpha, \gamma)$ 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.

Thus, assuming that α is not executed in **P** implies that some other instance, α_1, preceding α Pre^0-wise, is not executed either in **P**, though it is in **S**.

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 **S** and preceded (Pre^0-wise) by the next one in the sequence. This contradicts the simple fact that there are a finite number of execution dates between the program start and any step it reaches, in any run[15].

Extending to $\mathrm{Pre}^{\mathsf{s}}$. Suppose that some instance α such that $\mathrm{Pre}^{\mathsf{s}}(\alpha, \gamma)$ and not $\mathrm{Pre}^0(\alpha, \gamma)$ is executed in **S**. $\mathrm{Pre}^{\mathsf{s}}(\alpha, \gamma)$ 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 $\mathrm{Sync}^{\mathsf{s}}$; moreover, all the π_i and ω_i are executed in **S** (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 have introduced (§3.3) in the definition of $\mathrm{Pre}^{\mathsf{s}}$.

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 date τ in **P**. By upward recurrence, we will prove that all π_i and ω_i, and finally α, are executed before τ in **P**. 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. to the event involved in ω_i, ε_{ω_i}: all computations of ε_{ω_i} performed before the execution of ω_i are identical in both runs and occurred in the same order. So, since we have $\mathrm{Sync}^{\mathsf{s}}(\pi_i, \omega_i)$, π_i was executed in **P** before ω_i and its execution made possible the execution of ω_i, in **P** as well as in **S**: we have $\mathrm{Sync}(\pi_i, \omega_i)$ (§3.4).

Let us now consider the case $i > 1$ and derive the execution of ω_{i-1}. Since we have $\mathrm{Pre}^0(\omega_{i-1}, \pi_i)$ and $\mathrm{Exe}^{\mathsf{s}}(\omega_{i-1})$, according to the first part of the lemma, ω_{i-1} is executed before π_i, hence before date τ, in **P**. Thus, we conclude that ω_1 is executed before τ in **P**. The above reasoning then ensures that π_1 too is executed before τ in **P**. 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 **P**; furthermore, we have $\mathrm{Pre}(\alpha, \gamma)$, by transitive closure modulo Exe.

◀

[15] In this reasoning, it is crucial to have $\mathrm{Exe}^{\mathsf{s}}(\alpha_i)$, together with $\mathrm{Pre}^0(\alpha_{i+1}, \alpha_i)$, to obtain the contradiction, since the ordering Pre^0 is not *well-founded* (because of the way DO and PDO loops generate sequences of statement instances which are infinite *on both sides*).

Proof of Lemma 3

Lemma 3 is involved in the proof of Theorem 1 (Section 5), in point 5. We assume the hypotheses of Theorem 1.

Lemma 3 *We assume the semantic equivalence along* **P**. *Let* γ *be a* WAIT *instance, deadlocking in* **P**. *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* $\mathrm{Pre}(\alpha, \gamma)$.

Proof : The derivation is quite similar to the one of Lemma 2 above. First, we bring an addendum to the preliminary remark in the proof of Lemma 2. We had observed this: α being a persistently waiting or pending statement instance, no instance β such that $\mathrm{Pre}^0(\alpha, \beta)$ can be executed. Now, such an instance β cannot be a reached and deadlocking WAIT instance either. This straightforwardly results from the execution model and the definition of Pre^0.

Considering this preliminary remark, the proof of lemma 2 can be easily adapted here.

◀

References

[1] A. Aho, R. Sethi, and J. D. Ullman. *Compilers*. Addison-Wesley, 1986.

[2] L. Bougé. Sémantiques du parallélisme : un tour d'horizon. Juillet 1988. http://www.ens-lyon.fr/~bouge/Research.

[3] 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.

[4] G. Caplain. *Propriétés de correction séquentielle dans un langage parallèle à mémoire partagée*. PhD thesis, Ecole Nationale des Ponts et Chaussées, septembre 1998. http://cermics.enpc.fr/theses/.

[5] G. Caplain. Checking sequential correctness in shared-memory parallel programs. In *The Eighth International Colloquium on Numerical Analysis and Computer Science with Applications*, Plovdiv, Bulgaria, August 1999. Proceedings to be published.

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

[7] G. Caplain, R. Lalement, and T. Salset. Checking the serial correctness of control-parallel programs. In *Parallel Architectures and Languages Europe*, pages 741–744, Athens, Greece, July 1994. Springer Verlag, LNCS 817.

[8] J.F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, July 1995.

[9] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[10] M.B. Dwyer. *Data Flow Analysis for Verifying Correctness Properties of Concurrent Programs.* PhD thesis, University of Massachusetts, Amherst, MA, September 1995.

[11] P. Feautrier. Dataflow analysis of array and scalar references. *Int. Journ. of Parallel Programming*, 20(1):23–53, Feb. 1991.

[12] P. B. Gibbons and E. Korach. Testing shared memories. *SIAM Journal on Computing*, 26(4):1208–1244, 1997.

[13] C.A. Gunter. *Semantics of Programming Languages.* MIT Press, 1992.

[14] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[15] K.L. Johnson, M.F. Kaashoek, and D.A. Wallach. CRL: High-performance all-software distributed shared memory. In *ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 213–228. ACM Press, December 1995.

[16] R. Lalement. *Logique, réduction, résolution.* Masson, 1990.

[17] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Comm. ACM*, 21:558–564, 1978.

[18] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, C–28:690–691, 1979.

[19] S.P. Masticola. *Static detection of deadlocks in polynomial time.* PhD thesis, Rutgers University, May 1993.

[20] S. Muchnick. *Advanced Compiler Design Implementation.* Morgan & Kaufman, 1998.

[21] S. Muchnick and N. Jones. *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981.

[22] OpenMP. A proposed industry standard API for shared memory programming. Technical report, October 1997. http://www.openmp.org/.

[23] C. Pancake. *Parallel Processing Model for High Level Programming Languages.* ANSI, March 1992. (Proposed Standard).

[24] C.H. Papadimitriou. The serializability of concurrent database updates. *J.ACM*, 26:631–653, 1979.

[25] M. Raynal. About logical clocks for distributed systems. *Publication interne IRISA n° 607*, octobre 1991.

[26] T. Salset. *Correction séquentielle de programmes parallèles dans le modèle asynchrone et mémoire partagée.* PhD thesis, Ecole Nationale des Ponts et Chaussées, juillet 1997. http://cermics.enpc.fr/theses/.

[27] J. Subhlok. *Analysis of Synchronization in a Parallel Programming Environment.* PhD thesis, Rice University, April 1998.

[28] G. Winskel. *The Formal Semantics of Programming Languages.* MIT Press, Cambridge, Mass., 1993.

[29] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, Cambridge, Mass., 1989.

[30] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.

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

[32] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, 1990.