# Semantic analysis of a control-parallel extension of Fortran

Gilbert Caplain René Lalement Thierry Salset

juin 1993

$N^o$ 93–18

# Semantic analysis of a control-parallel extension of Fortran

Gilbert Caplain René Lalement Thierry Salset

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

**Abstract**

*We study a correctness property of programs in a subset of Fortran X3H5, a control-parallel extension of Fortran. This property is an equivalence between the parallel program we analyze and its sequential version. Under some assumptions, we prove that this property follows from the preservation of dependences, defined on the sequential version, by the control flow and the synchronizations. To check this preservation, we propose an algorithm which builds a formula (an implication between systems of equations and inequations) from a new kind of block graph. In case the expressions are linear, the algorithm tries to prove that this formula is a tautology by means of the Omega test.*

# Analyse sémantique d'une extension à parallélisme de contrôle de Fortran[1]

**Résumé**

*Nous étudions une propriété d'un sous-ensemble de Fortran X3H5, une extension à parallélisme de contrôle de Fortran. Cette propriété est une équivalence entre le programme parallèle étudié et sa version séquentielle. Moyennant certaines hypothèses, nous prouvons que cette propriété résulte de la conservation des dépendances, définies sur la version séquentielle, par le flot de contrôle et les synchronisations. Pour vérifier cette conservation, nous proposons un algorithme qui construit une formule (une implication entre systèmes d'équations et d'inéquations) à partir d'un graphe de blocs. Dans le cas où les expressions sont linéaires, cet algorithme tente de prouver que cette formule est une tautologie à l'aide du test Omega.*

[1]Ce travail a bénéficié d'une aide du Ministère de la Recherche et de la Technologie, n$^o$ 90 S 0961

# Contents

# 1 Introduction

The main trend to give programmers access to parallel machines has been to provide them with parallelizing tools, either automatic or assisted ones. This was backed by the idea that parallel programming is too complex, and that the main needs were to port existing sequential codes to these new machines. However, some programmers like to design their algorithms in a genuine parallel way and are not satisfied with the output of existing parallelizers, not because of its quality, but because of the poorness of the understanding they provide. Others, trying to get the best performance from their costly supercomputers, must carefully parallelize their codes by hand.

Our aim is to give the programmer a tool to write parallel programs and to verify them. The kind of parallel programming this report addresses is "Control Parallelism — Single Program Multiple Data" (SPMD, Cf. [14]), and is presently used in most shared memory multiprocessors, either in the form of calls to libraries or through extensions to a sequential language, usually Fortran. Started in 1989 and supported by supercomputer manufacturers, the Parallel Computing Forum aimed at achieving the standardization of the syntax and semantics of "well understood parallel constructs". Following the Forum, an ANSI committee has been formed (X3H5), first to define a language independent model[9], then to bind it to Fortran-77[13], Fortran-90 and C. The language we study is the extension of Fortran-77 by the X3H5 parallel constructs.

The proposal defines what a "standard conforming" program is or is not. Unfortunately, many such properties are not decidable, unlike static properties such as type correctness. The usual conservative approach is to reject a program when a property may be unsatisfied and to accept it only if the property can be proved. To be able to analyze parallel programs without being too conservative, we restrict our study to a syntactic subset of the X3H5 Fortran language. A compliant program may be executed with an arbitrary number of processes and must complete execution without deadlock. A program must not assume a minimum number of processes in order to avoid a deadlock. In particular, a program must not deadlock when executed with a single process.

We begin with the definition of the language we study, borrowing its semantics from the "intuitive model of program execution" given in the X3H5 proposal. Then, we set up the correctness problem: a parallel program and its sequential version, that we define, must have the same observable behavior. This requirement has its origin in the need to consider the existing sequential programs as the reference. The next section defines several properties on the syntactic level which will be useful later on: textual ordering, loop nest of a statement, statement instances and iteration space of a statement, common loop nest and control type of two statements, variables read and written by a statement, scoping. Comparing the sequential run and any parallel run, we need to know if some statement may not be executed in both runs. While an execution formula can be defined in our language for the sequential version of a parallel program, this cannot be extended to the parallel program, for an execution may be non deterministic. A condition will be given that ensures that any parallel execution satisfies the sequential execution formula. We introduce the data dependences in a parallel program, with reference to the sequential order of execution: a sequential dependence is safe in the parallel program if it is preserved by what remains of the sequential control and the synchronizations. Next, a predicate stating a precedence relation between two statement instances is defined by combining the sequential control flow, the execution formula at specific statements, and the synchronization relation. We show how to compute this predicate, using a new kind of block graph so that a formula is associated to every path.

# 2 Language

## 2.1 A subset of ANSI X3H5

The language we study is a subset of Fortran–77 extended by a subset of the X3H5 parallel constructs. The extensions we include are the combined parallel and worksharing constructs PARALLEL DO and PARALLEL SECTIONS, the EVENT and ORDINAL explicit synchronization; inside a worksharing construct, the NEW statement may be used. We do not consider the more general PARALLEL regions, nor the mutual exclusion mechanisms (structured CRITICAL SECTION and unstructured LOCK synchronization), and the SCOMMON statement. On the sequential side, we have structured IF and DO loops, assignment, scalar and array variables, we exclude GOTO, procedure CALL, WHILE, DATA, EQUIVALENCE and SAVE statements. We allow for *parameters* under the form of variables that are not written. A *program instance* is obtained from a program by assigning constant values to the parameters. A specification of an abstract syntax for this subset is given in the following table.

Several restrictions are enforced.

- program parameters are not writable
- loop indices are not writable, and are not used outside of loops
- loop bounds only depend on indices of surrounding loops and program parameters
- subscript expressions in arrays only involve index variables and program parameters (staticity of references)
- ordinal statements only involve index variables, program parameters and ordinal variables

In case a statement is embedded in loops, a *statement instance* is associated to every possible value of the *loop index vector*, the vector of the index variables of loops within which this statement is embedded, ordered outermost left. The set of these integer vectors is the *iteration space* of the statement. If $a$ denotes a statement and $\boldsymbol{i}$ is in its iteration space, a statement instance will be denoted by $a(\boldsymbol{i})$. Some syntactic restrictions above are needed to define the statement instances. Furthermore, out of convenience, loops and ordinal variables will be normalized, i.e. their increment will be set to 1. Other restrictions will be added later, such as linearity of subscript and bound expressions and staticity of boolean expressions in IF statements, which are required by the algorithm we will introduce.

## 2.2 Execution model

The program execution begins with an initial process. A process has a data environment and at least one of the following attributes: a current statement pointer, a virtual processor. A process runs sequentially until a parallel construct is encountered; then it becomes the base process for this construct. A worksharing construct specifies a number of threads, i.e., of units of work, which may be assigned to processes running in parallel: each iteration of a PARALLEL DO and each section of a PARALLEL SECTIONS is a thread. As nested parallelism is allowed, these threads are not the smallest unit of work that can be passed to a process. When entering a parallel construct a *team* of processes is created (the base process may be a member of this team), then each worksharing

| | |
|---|---|
| PROGRAM | ::= DECLARATION$^+$ STMT$^+$ |
| DECLARATION | ::= NAME DIM$^*$ TYPE |
| DIM | ::= integer |
| TYPE | ::= *INTEGER* \| *REAL* \| *LOGICAL* \| *EVENT* \| *ORDINAL* |
| STMT | ::= ASSIGN \| IF \| DO \| PDO \| PSECTIONS \| E-POST \| E-WAIT \| O-SET \| O-POST \| O-WAIT |
| ASSIGN | ::= LHS RHS |
| LHS | ::= SCALAR-REF \| ARRAY-REF |
| RHS | ::= EXP |
| SCALAR-REF | ::= NAME |
| ARRAY-REF | ::= NAME EXPLIST |
| EXPLIST | ::= EXP$^+$ |
| NAME | ::= identifier |
| EXP | ::= ... |
| IF | ::= BEXP THEN ELSE ENDIF |
| BEXP | ::= EXP |
| THEN | ::= STMT$^*$ |
| ELSE | ::= STMT$^*$ |
| ENDIF | ::= void |
| DO | ::= HEAD BODY ENDDO |
| HEAD | ::= INDEX LB UB |
| INDEX | ::= NAME |
| LB | ::= EXP |
| UB | ::= EXP |
| BODY | ::= STMT$^+$ |
| ENDDO | ::= void |
| PDO | ::= HEAD NEW BODY ENDPDO |
| NEW | ::= NAME$^*$ |
| ENDPDO | ::= void |
| PSECTIONS | ::= NEW SECTION$^*$ ENDPSECTIONS |
| SECTION | ::= S-HEAD BODY S-END |
| S-HEAD | ::= NAME S-WAIT |
| S-END | ::= void |
| S-WAIT | ::= NAME$^*$ |
| ENDPSECTIONS | ::= void |
| E-POST | ::= LHS |
| E-WAIT | ::= LHS |
| O-SET | ::= LHS EXP |
| O-POST | ::= LHS EXP |
| O-WAIT | ::= LHS EXP |

Table 1: Abstract syntax of the language

thread is assigned to some team member until all work defined by the construct is completed. If the worksharing is ORDERED, this assignment and the beginning of each process must be done sequentially: in the index ordering for a PARALLEL DO, in the section ordering in a PARALLEL SECTIONS. Otherwise, the assignment may be done in any order. Team members which are not assigned work and members which have completed their work wait at the end of the construct until all work is completed.

Each team member shares its data environment with the base process, except for private objects specified by the parallel construct. The index variable of a PARALLEL DO is implicitly private, other objects are explicitly made private through the NEW statement, which defines a new scope for the names it refers to. When a process completes its assigned work, all updates to the shared data environment are made available to the other team members. When entering the parallel construct, a new instance of private objects is created in the data environment of each member of the team: here, creation means memory allocation, but neither initialization nor copy of the object from the global environment. Notice that the private attribute is relative to the process and not to the thread: all the threads executed by the same process share the same private object.

At the end of a parallel construct, the team is dissolved and its base process continues execution.

Two mechanisms for explicit synchronization are available: events and ordinals (or sequences).

An *event* has one of two defined values: *cleared* and *posted*. Undefined when created, an event gets a value by one of the atomic statements CLEAR and POST. The WAIT statement tests the value of the event, continues if this value is posted, and repeats this step at a later time otherwise. Each of these statements is a synchronization point where all shared objects are made consistent to the processes owning the event.

An *ordinal* is a counting semaphore used to communicate between loop iterations or distinct loops. It has integer values, is initialized to an integer by the SET statement, is incremented by POST and tested by WAIT. Increment and test are synchronization points where all shared objects are made consistent. Moreover, POST$(o, n)$ compares the value of ordinal $o$ to integer $n-1$; if this value is less than $n-1$, the statement is retried at a later time, and if it is equal to $n-1$, it proceeds incrementing ordinal $o$. WAIT$(o, n)$ compares the value of $o$ to $n$; if it is less than $n$, the statement is retried, and otherwise proceeds.

It is straightforward to prove that our language does not allow infinite loops: all programs come to an end, either a normal termination, or a deadlock. A *waiting statement* is a statement on which the control may come to a wait, till some condition is met. In our language subset, the waiting statements are the event and ordinal WAIT, the ordinal POST, the WAIT clause of a section, the END PARALLEL DO and END PARALLEL SECTIONS.

### 2.3 Serial semantics

With every (parallel) program is associated its sequential version, which is the result of the transformation of PARALLEL DO into DO, the deletion[1] of PARALLEL SECTIONS, SECTION, END PARALLEL SECTIONS and explicit synchronization statements, and the substitution of new names for names specified in NEW statements within their scope; this last point will be expanded later on.

[1] Instead of deleting these statements, it will be more convenient to replace them by ordinary CONTINUE statements in order to keep track of them.

The intended observable behavior of the parallel program is that of its sequential version, whose semantics is (admittedly) well-defined. The differences between the original program and its sequential version lie in two new features of the parallel constructs:

- the execution order of statements is not completely specified
- the data environment is not always the same, because of NEW statements

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

### 2.4 Correctness

Our aim is to prove the correctness of a parallel program. But correctness against what? Usually a correctness proof is given for a specification or for some structural property, such as safety or liveness. In our case, the specification *is* the behavior of the sequential version of the program, and the structural properties are those of this version.

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.

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

## 3 Syntactic definitions

We begin with a set of definitions which will be needed in the correctness proofs below. With each statement $a$ or pair of statements $a, b$ we associate a loop nest $\mathfrak{N}(a)$, an iteration formula $\mathrm{Iter}_a(\boldsymbol{p}, \boldsymbol{x})$, a common loop nest $\mathfrak{N}(a, b)$, a common control type $\mathfrak{C}(a, b)$, and sets of read and written references $\mathfrak{In}(a)$ and $\mathfrak{Out}(a)$.

### 3.1 Textual ordering

For two statements $a$ and $b$, the relation $a \ll b$ states that $a$ precedes $b$ in the textual order. Its easy definition is omitted.

### 3.2 Loop nest of a statement and iteration space

For a statement $a$, $\mathfrak{N}(a)$, called the loop nest of $a$, is the list of the surrounding loop headers, ordered outside-in, and written as 4-tuples $(C, i, l, u)$, where $C$ is DO or PDO, $i$ is an index name, and $l$ and $u$ are expressions. We suppose that all index names are different and that each expression depends only on previous indices and program parameters. If $N$ is a loop nest, its length is $|N|$; the loop depth of statement $a$ is $|\mathfrak{N}(a)|$, the number of (sequential or parallel) loops surrounding

it. In the following, we use the symbol :: to concatenate two lists or add an element to either end of a list. The definition of the loop nest is given by the rules:

- If $p$ is a PROGRAM statement, and $p.\text{STMT}^+ = \ldots a \ldots$, then $\mathfrak{N}(a) = [\,]$
- If $a$ is a DO statement, and $a.\text{BODY} = \ldots b \ldots$, then $\mathfrak{N}(b) = \mathfrak{N}(a) :: (\text{DO}, a.\text{HEAD})$, $\mathfrak{N}(a.\text{HEAD}) = \mathfrak{N}(a.\text{ENDDO}) = \mathfrak{N}(a)$
- If $a$ is a PARALLEL DO statement, and $a.\text{BODY} = \ldots b \ldots$, then $\mathfrak{N}(b) = \mathfrak{N}(a) :: (\text{PDO}, a.\text{HEAD})$, $\mathfrak{N}(a.\text{HEAD}) = \mathfrak{N}(a.\text{ENDPDO}) = \mathfrak{N}(a)$
- If $a$ is an IF statement, then $\mathfrak{N}(a.\text{BEXP}) = \mathfrak{N}(a.\text{THEN}) = \mathfrak{N}(a.\text{ELSE}) = \mathfrak{N}(a)$
- If $a$ is a PARALLEL SECTIONS statement, and $a.\text{SECTION}* = \ldots b \ldots$, then $\mathfrak{N}(b) = \mathfrak{N}(a)$
- If $a$ is a SECTION statement, and $a.\text{STMT}^+ = \ldots b \ldots$ then $\mathfrak{N}(b) = \mathfrak{N}(a)$

Every expression *exp* in a statement $a$ is "bound" by the loop nest $\mathfrak{N}(a)$, so it must be considered as written $[\mathfrak{N}(a)]exp$. For instance, given the loop

```
do I = 1, N
 do J = 1, I
  do K = I-J, I
   A(I,J,K) = 0
  end do
 end do
end do
```

the assignment `A(I,J,K) = 0` is bound by the three surrounding loop heads, and could be written as:

$$[1 \leqslant I \leqslant N][1 \leqslant J \leqslant I][I - J \leqslant K \leqslant I]A(I, J, K) = 0$$

As we may consider some ending sublist of $\mathfrak{N}(a)$, say $N_2$ when $\mathfrak{N}(a) = N_1 :: N_2$, we must also write $[N_1]N_2$ instead of $N_2$ to stress in what syntactic context this sublist is located. Alternatively (but less accurately), we can write $N_2(I_1, \ldots, I_p)$ if $I_1$, ..., $I_p$ are the index names in $N_1$.

When we create a mathematical formula from a Fortran expression, we will have to substitute variables for index variables; we will use the notation $[\boldsymbol{x} \setminus \mathfrak{N}(a)]$ for this substitution. For instance $[(x_1, x_2, x_3) \setminus \mathfrak{N}(a)]\texttt{I+J-K} = x_1 + x_2 - x_3$.

Usually, the iteration space of a statement $a$ is the set of integer vectors, that the index vector specified by $\mathfrak{N}(a)$ may take as values. In order to deal with program parameters, we replace this set by a formula $\text{Iter}_a(\boldsymbol{p}, \boldsymbol{x}) = I(\mathfrak{N}(a))$ involving these parameters, $\boldsymbol{p}$, and a list of variables $\boldsymbol{x}$ associated to the index names in $\mathfrak{N}(a)$, which is the conjunction of the loop headers:

$$\begin{aligned} I([\,]) &= \text{true} \\ I((C, i, l, u) :: N) &= (l \leqslant i \leqslant u) \wedge I(N) \end{aligned}$$

This formula does not depend on the loop types ($C$ is DO or PDO). When the program parameters are given integer values $\boldsymbol{p}$, the formula $\text{Iter}_a$ determines a subset $\text{Iter}_a(\boldsymbol{p})$ of $\mathbf{N}^d$ (where $d$ is the length of $\mathfrak{N}(a)$), called the iteration space of $a$, for the given program instance.

### 3.3 Common loop nest and common control type of two statements

For each pair of statements $a$, $b$, the common loop nest $\mathfrak{N}(a,b)$ is defined by

$$\mathfrak{N}(a,b) = \mathrm{merge}(\mathfrak{N}(a), \mathfrak{N}(b)),$$

using the auxiliary function $\mathrm{merge} : \mathrm{list} \times \mathrm{list} \to \mathrm{list}$

$$\begin{aligned}
\mathrm{merge}([\,], N) &= [\,] \\
\mathrm{merge}(N, [\,]) &= [\,] \\
\mathrm{merge}(h :: N_1, h :: N_2) &= h :: \mathrm{merge}(N_1, N_2) \\
\mathrm{merge}(h_1 :: N_1, h_2 :: N_2) &= [\,] \quad \text{if } h_1 \neq h_2
\end{aligned}$$

For any pair of statements $a$, $b$, we introduce the *common iteration formula* $\mathcal{CIS}(a,b) = I(\mathfrak{N}(a,b))$.

For each pair of statements $a$, $b$, the *common control type* $\mathfrak{C}(a,b)$ is defined as one of the symbols ';', '||' or '?', according to the least common ancestor of $a$ and $b$, in the abstract syntax tree of the program: it is ';', meaning a sequential control relation, if this ancestor is a sequence of statements (body), it is '||', meaning a parallel control relation, if the ancestor is PARALLEL SECTIONS; it is '?' if the ancestor is an IF; moreover, it is also ';' when $a$ is a boolean expression of an IF, and $b$ is a statement in one of its branches. We write "$a'$ in $a$" when $a'$ is a son of $a$ in the abstract syntax tree.

$$\frac{p \text{ is a program} \quad a,b \in p.\textsc{stmt}^+}{\mathfrak{C}(a,b) = ;}$$

$$\frac{p \text{ is a DO} \quad a,b \in p.\textsc{body}}{\mathfrak{C}(a,b) = ;}$$

$$\frac{p \text{ is a PARALLEL DO} \quad a,b \in p.\textsc{body}}{\mathfrak{C}(a,b) = ;}$$

$$\frac{p \text{ is a SECTION} \quad a,b \in p.\textsc{body}}{\mathfrak{C}(a,b) = ;}$$

$$\frac{\begin{array}{c}p \text{ is a PARALLEL SECTIONS} \\ a \neq b \in p.\textsc{section}* \qquad a.\textsc{s-head.name} \in b.\textsc{s-head.s-wait}\end{array}}{\mathfrak{C}(a,b) = ;}$$

$$\frac{c \text{ is an IF statement} \quad a \text{ in c.THEN}}{\mathfrak{C}(c.\textsc{bexp}, a) = ;}$$

$$\frac{c \text{ is an IF statement} \quad b \text{ in c.ELSE}}{\mathfrak{C}(c.\textsc{bexp}, b) = ;}$$

$$\frac{a \neq b \quad a' \text{ in a} \quad b' \text{ in b} \quad \mathfrak{C}(a,b) = ;}{\mathfrak{C}(a',b') = ;}$$

$$\frac{\begin{array}{c}p \text{ is a PARALLEL SECTIONS} \qquad a.\textsc{s-head.name} \notin b.\textsc{s-head.s-wait} \\ a \neq b \in p.\textsc{section}* \qquad b.\textsc{s-head.name} \notin a.\textsc{s-head.s-wait}\end{array}}{\mathfrak{C}(a',b') = ||}$$

$$\frac{a' \text{ in a} \quad b' \text{ in b} \quad \mathfrak{C}(a,b) = ||}{\mathfrak{C}(a',b') = ||}$$

$$\frac{c \text{ is an IF statement} \quad a \text{ in c.THEN} \quad b \text{ in c.ELSE}}{\mathfrak{C}(a,b) = ?}$$

$$\frac{a' \text{ in a} \quad b' \text{ in b} \quad \mathfrak{C}(a,b) = ?}{\mathfrak{C}(a',b') = ?}$$

Figure 1 shows the abstract syntax tree of a program, and the loop nest and common control type of two statements.

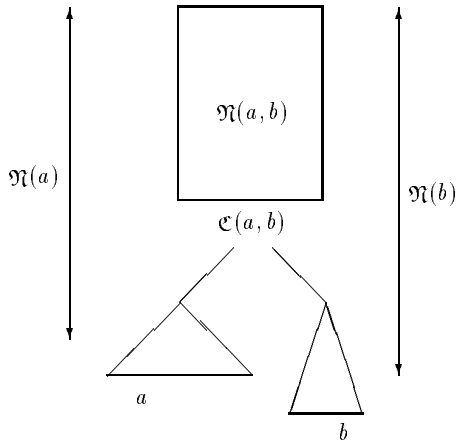


Figure 1: Loop nest and common control type of two statements

### 3.4 $\mathfrak{In}$ and $\mathfrak{Out}$ sets

The $\mathfrak{In}$ set of an expression or a statement is the set of all scalar and array references which are used (or read) by it. The $\mathfrak{Out}$ set of a statement is the empty set or the singleton set of the reference which is written by the statement. These sets are needed to compute the dependence relation between two statements.

#### 3.4.1 $\mathfrak{In}$ set for an expression

$$\begin{aligned}
\mathfrak{In}(constant) &= \varnothing \\
\mathfrak{In}(E1\ op\ E2) &= \mathfrak{In}(E1) \cup \mathfrak{In}(E2) \\
\mathfrak{In}(op\ E1) &= \mathfrak{In}(E1) \\
\mathfrak{In}(name) &= \{name\} \\
\mathfrak{In}(name(E_1, \ldots E_n)) &= \{name(E_1, \ldots E_n)\} \cup \mathfrak{In}(E_1) \cup \ldots \cup \mathfrak{In}(E_n)
\end{aligned}$$

This definition is naturally extended to lists of expressions.

### 3.4.2 $\mathfrak{In}$ and $\mathfrak{Out}$ sets for a statement

In what follows, a scalar reference will be treated as an array reference with an empty list of subscript expressions.

- $a$ is an assign statement

$$\begin{aligned} \mathfrak{In}(a) &= \mathfrak{In}(a.\text{RHS}) \cup \mathfrak{In}(a.\text{LHS.EXPLIST}) \\ \mathfrak{Out}(a) &= \{a.\text{LHS}\} \end{aligned}$$

- $a$ is an event POST statement

$$\begin{aligned} \mathfrak{In}(a) &= \mathfrak{In}(a.\text{LHS.EXPLIST}) \\ \mathfrak{Out}(a) &= \{a.\text{LHS}\} \end{aligned}$$

- $a$ is an event WAIT statement

$$\begin{aligned} \mathfrak{In}(a) &= \mathfrak{In}(a.\text{LHS}) \\ \mathfrak{Out}(a) &= \varnothing \end{aligned}$$

- $a$ is an ordinal SET statement

$$\begin{aligned} \mathfrak{In}(a) &= \mathfrak{In}(a.\text{EXP}) \cup \mathfrak{In}(a.\text{LHS.EXPLIST}) \\ \mathfrak{Out}(a) &= \{a.\text{LHS}\} \end{aligned}$$

- $a$ is an ordinal POST statement

$$\begin{aligned} \mathfrak{In}(a) &= \{a.\text{LHS}\} \cup \mathfrak{In}(a.\text{EXP}) \cup \mathfrak{In}(a.\text{LHS.EXPLIST}) \\ \mathfrak{Out}(a) &= \{a.\text{LHS}\} \end{aligned}$$

- $a$ is an ordinal WAIT statement

$$\begin{aligned} \mathfrak{In}(a) &= \{a.\text{LHS}\} \cup \mathfrak{In}(a.\text{EXP}) \cup \mathfrak{In}(a.\text{LHS.EXPLIST}) \\ \mathfrak{Out}(a) &= \varnothing \end{aligned}$$

Let strip be the mapping from left-hand sides (scalar or array references) to names, stripping off the list of expressions from every array reference: $\text{strip}(a) = \{a.\text{NAME}\}$, if $a$ is a left hand side. For instance, $\text{strip}(A(I-1, J)) = A$, $\text{strip}(I) = I$. As for expressions, any left-hand side name occurrence in a statement $a$ is bound by $\mathfrak{N}(a)$. Let $\mathfrak{OutNames}(a)$, resp. $\mathfrak{InNames}(a)$ be the set of names resulting from the application of the strip mapping to all elements of $\mathfrak{Out}(a)$, resp. $\mathfrak{In}(a)$: for instance, if $\mathfrak{In}(a) = \{A(I+1), A(I-1), I\}$, then $\mathfrak{InNames}(a) = \{A, I\}$. The strip mapping is naturally extended to mappings $\mathfrak{Out}(a) \rightarrow \mathfrak{OutNames}(a)$ and $\mathfrak{In}(a) \rightarrow \mathfrak{InNames}(a)$.

### 3.5 Scoping and sharing

In addition to program units, a parallel construct (PARALLEL DO and PARALLEL SECTIONS) defines a new scope for names. The scope of the index variable of a DO contained in a parallel construct is this construct; the same applies to a PARALLEL DO index variable. The scope of a name specified in a NEW statement is the parallel construct containing this statement.

The scoping mechanism of the NEW statement introduces a strong difference between the parallel program and its sequential version. In order to consider that the NEW statement must be deleted to get the sequential version, it shall be required that this statement introduce fresh names instead of reusing the same name in a new scope. Doing so, a write, out of the parallel construct, on a variable made private inside has the same effect in the parallel program and its sequential version. Moreover, an array promotion of these variables will be performed, as will be argued later.

## 4 Execution predicate

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

For every statement $a$, we would like to use a predicate $\mathrm{Exe}(a(\boldsymbol{i}))$ meaning that the statement instance $a(\boldsymbol{i})$ is executed. In the case of a waiting statement (e.g. a WAIT), "being executed" will mean "being passed". An important fact must be pointed out: whereas we will define $\mathrm{Dep}_{a,b}$ as a formula involving only the indices $\boldsymbol{i}$ (of $a$) and $\boldsymbol{j}$ (of $b$) and the program parameters, there is no such formula for Exe, and the predicate itself depends on the specific run of the program. It will be used in the following proofs where it is sufficient to know that it is "run-time defined", more accurately run-time valued.

In order to avoid this problem, we will be interested in the condition $\mathrm{Exe}^{\mathsf{S}}$ for a statement to be executed in the *sequential* version. It is well-defined and its expression is rather straightforward for our language, whose control structures have been drastically restricted: it involves the boolean expressions of the embedding IF statements and the bounds of the embedding loops. As bound loops depend only on index variables of the embedding loops and on program parameters, the value of a loop bound is determined only by the iteration vector. However, arbitrary variables may occur in a boolean expression, so that its value does not depend only on the iteration vector, but on the whole data environment, $\rho$. In turn, we can derive from a standard dynamic semantics of our language a partial function associating to the initial environment, a statement $a$ and an iteration vector $\boldsymbol{i}$, the environment $\mathit{curr\_env}@a(\boldsymbol{i})$ in which the execution of $a(\boldsymbol{i})$ takes place; this environment is only defined when instance $a(\boldsymbol{i})$ is to be executed. So, for an arbitrary expression $\mathit{exp}$ whose evaluation is required by the execution of statement $a$, its value $[\![\mathit{exp}]\!]@a(\boldsymbol{i})$ is defined as $[\![\mathit{exp}]\!](\mathit{curr\_env}@a(\boldsymbol{i}))$.

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

- $a$ is not contained in a loop nor an IF: then, $\mathrm{Exe}^{\mathsf{S}}(a) = \mathit{true}$.

- The innermost nesting of $a$ is in an IF statement $c$, of boolean expression $c.\text{BEXP}$, and let $\boldsymbol{i}$ be an iteration vector of $c$ and $a$:
  - if $a$ is in the THEN branch, $\text{Exe}^{\mathsf{S}}(a(\boldsymbol{i})) = \text{Exe}^{\mathsf{S}}(c(\boldsymbol{i})) \wedge [\![c.\text{BEXP}]\!]@c(\boldsymbol{i})$
  - if $a$ is in the ELSE branch, $\text{Exe}^{\mathsf{S}}(a(\boldsymbol{i})) = \text{Exe}^{\mathsf{S}}(c(\boldsymbol{i})) \wedge \neg([\![c.\text{BEXP}]\!]@c(\boldsymbol{i}))$
- The innermost nesting of $a$ is in a loop $c$ of lower and upper bound expressions $c.\text{LB}$ and $c.\text{UB}$ respectively. Let $\boldsymbol{i}$ be an iteration vector of $c$ and $j$ be an integer taken as value by the index defined by $c$. We then have: $\text{Exe}^{\mathsf{S}}(a(\boldsymbol{i} :: j)) = \text{Exe}^{\mathsf{S}}(c(\boldsymbol{i})) \wedge ([\![c.\text{LB}]\!]@c(\boldsymbol{i}) \leq j \leq [\![c.\text{UB}]\!]@c(\boldsymbol{i}))$

We will show later that under adequate assumptions, the Exe predicate is actually well-defined and equal to $\text{Exe}^{\mathsf{S}}$. In case no variable occurs in the boolean expressions, the above definitions are simplified so that the predicate $\text{Exe}^{\mathsf{S}}(a(\boldsymbol{i}))$ is defined by a formula $\text{Exe}^{\mathsf{S}}$; the previous definitions for IF and loops can be changed into:

$$\begin{aligned}
\text{Exe}^{\mathsf{S}}_a(\boldsymbol{x}) &= \text{Exe}^{\mathsf{S}}_c(\boldsymbol{x}) \wedge c.\text{BEXP}[\boldsymbol{x} \setminus \boldsymbol{i}] \\
\text{Exe}^{\mathsf{S}}_a(\boldsymbol{x}) &= \text{Exe}^{\mathsf{S}}_c(\boldsymbol{x}) \wedge (\neg c.\text{BEXP})[\boldsymbol{x} \setminus \boldsymbol{i}] \\
\text{Exe}^{\mathsf{S}}_a(\boldsymbol{x} :: y) &= \text{Exe}^{\mathsf{S}}_c(\boldsymbol{x}) \wedge (c.\text{LB} \leq y \leq c.\text{UB})[\boldsymbol{x} \setminus \boldsymbol{i}]
\end{aligned}$$

# 5 Precedences

Sequencing of parallel programs is specified by the sequential subset of the language and by the synchronization primitives. As this specification is only partial, we define a "precedence" predicate instead of the more familiar sequencing predicate, and we split its definition into a sequential precedence and a synchronization precedence.

## 5.1 Sequential precedence

For each pair of statements $a$ and $b$, we have defined the common loop nest $\mathfrak{N}(a,b)$ and the common control type $\mathfrak{C}(a,b)$. If $N$ is a loop nest and $C$ is ';' or '||', we now define a formula $\prec_{N,C}$ of two lists of variables, the lengths of which are not smaller than the length of $N$. Let $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ be lists of variables whose length is that of $N$, and $\boldsymbol{y}_1$, $\boldsymbol{z}_2$ be any lists of variables.

$$\begin{aligned}
\prec_{N,;}(\boldsymbol{x}_1 :: \boldsymbol{y}_1, \boldsymbol{x}_2 :: \boldsymbol{z}_2) &= \prec_N(\boldsymbol{x}_1, \boldsymbol{x}_2) \vee (\boldsymbol{x}_1 = \boldsymbol{x}_2) && \text{if } a \ll b \\
\prec_{N,;}(\boldsymbol{x}_1 :: \boldsymbol{y}_1, \boldsymbol{x}_2 :: \boldsymbol{z}_2) &= \prec_N(\boldsymbol{x}_1, \boldsymbol{x}_2) && \text{if } b \ll a \text{ or } a = b \\
\prec_{N,||}(\boldsymbol{x}_1 :: \boldsymbol{y}_1, \boldsymbol{x}_2 :: \boldsymbol{z}_2) &= \prec_N(\boldsymbol{x}_1, \boldsymbol{x}_2)
\end{aligned}$$

In particular, $\prec_{N,C}$ specializes into $\prec_{\mathfrak{N}(a,b),\mathfrak{C}(a,b)}$ which will be convenient to abbreviate as $\text{Pre}^0_{a,b}$, the variables of which are two lists of length $|\,\mathfrak{N}(a)|$ and $|\,\mathfrak{N}(b)|$. This formula expresses the precedence between execution instances of $a$ and $b$ in the parallel program. For integer vectors $\boldsymbol{p}$, $\boldsymbol{i}$ and $\boldsymbol{j}$ such that $\text{Iter}_a(\boldsymbol{p},\boldsymbol{i}) \wedge \text{Iter}_b(\boldsymbol{p},\boldsymbol{j})$ is true, $\text{Pre}^0_{a,b}(\boldsymbol{i},\boldsymbol{j})$ means that if instances $a(\boldsymbol{p},\boldsymbol{i})$ and $b(\boldsymbol{p},\boldsymbol{j})$ are both executed, then $a(\boldsymbol{p},\boldsymbol{i})$ is executed before $b(\boldsymbol{p},\boldsymbol{j})$.

The sequential order $\prec\!\prec_{N,C}$ is obtained from the definition of the precedence order $\prec$ in transforming every PARALLEL DO into a DO statement and deleting all the PARALLEL SECTIONS and

SECTION statements. This is not the execution ordering with a single process, since the synchronizations are not taken into account. Let $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ be lists of variables whose length is that of $N$, and $\boldsymbol{j}_1$, $\boldsymbol{z}_2$ be any lists of variables.

$$\begin{array}{lcll} \prec\prec_{N,;}(\boldsymbol{x}_1 :: \boldsymbol{j}_1, \boldsymbol{x}_2 :: \boldsymbol{z}_2) & = & \prec\prec_N(\boldsymbol{x}_1, \boldsymbol{x}_2) \vee (\boldsymbol{x}_1 = \boldsymbol{x}_2) & \text{if } a \ll b \\ \prec\prec_{N,;}(\boldsymbol{x}_1 :: \boldsymbol{j}_1, \boldsymbol{x}_2 :: \boldsymbol{z}_2) & = & \prec\prec_N(\boldsymbol{x}_1, \boldsymbol{x}_2) & \text{if } b \ll a \text{ or } a = b \\ \prec\prec_{N,||}(\boldsymbol{x}_1 :: \boldsymbol{j}_1, \boldsymbol{x}_2 :: \boldsymbol{z}_2) & = & \prec\prec_{N,;}(\boldsymbol{x}_1 :: \boldsymbol{j}_1, \boldsymbol{x}_2 :: \boldsymbol{z}_2) & \end{array}$$

As above, $\prec\prec_{\mathfrak{N}(a,b),\mathfrak{C}(a,b)}$ is abbreviated into $a \prec\prec b$, or $\mathrm{Seq}_{a,b}$

## 5.2 Synchronization precedence

We define synchronization pairs, and for each such pair $u$, a synchronization formula $\mathrm{Sync}_u(\boldsymbol{p}, \boldsymbol{x}, \boldsymbol{y})$.

### 5.2.1 Event synchronization

Let $a$, $b$ be event POST and WAIT statements on the same event (scalar or array) variable: we have $\mathfrak{OutNames}(a) \cap \mathfrak{InNames}(b) = \{n\}$, $\mathfrak{Out}(a) = \{r_p\}$, $r_w \in \mathfrak{In}(b)$, and $\mathrm{strip}(r_w) = \mathrm{strip}(r_p) = n$. The pair $([\mathfrak{N}(a)]r_p, [\mathfrak{N}(b)]r_w)$ is called an event synchronization pair.

### 5.2.2 Ordinal synchronization

Let $a$, $b$ be ordinal POST and WAIT statements on the same ordinal variable (of increment 1, as for loops): $\mathfrak{OutNames}(a) \cap \mathfrak{InNames}(b) = \{n\}$, $\mathfrak{Out}(a) = \{r_p\}$, $r_w \in \mathfrak{In}(b)$, and $\mathrm{strip}(r_w) = n$. The pair $([\mathfrak{N}(a)]r_p, [\mathfrak{N}(b)]r_w)$ is called an explicit ordinal synchronization pair. Let $a$ be an ordinal POST statement, and $\mathfrak{Out}(a) = \{r_p\}$. The reference $[\mathfrak{N}(a)]r_p$ is called an implicit ordinal synchronization reference. Note that if $u$ is a synchronization pair, then the opposite $\hat{u}$ cannot be a synchronization pair.

### 5.2.3 Synchronization formula

With every synchronization pair $u = ([\mathfrak{N}(a)]r_p, [\mathfrak{N}(b)]r_w)$ based on $(a, b)$, we associate a synchronization formula $\mathrm{Sync}_u : a \to b$. By definition, $\mathrm{strip}(r_p) = \mathrm{strip}(r_w) = n$, and both $r_p$, $r_w$ are scalar references or array references of the same dimension, say $d$. Let $r_p$.EXPLIST $= [e_1, \ldots, e_d]$, and $r_w$.EXPLIST $= [f_1, \ldots, f_d]$. These subscript expressions $e_i$ and $f_i$ may contain the program parameters $\boldsymbol{p}$. For an event synchronization pair, the formula $\mathrm{Sync}_u$ is the conjunction of the equations and inequations of the following system:

$$\left\{ \begin{array}{l} [\boldsymbol{x} \setminus \mathfrak{N}(a)]e_i = [\boldsymbol{y} \setminus \mathfrak{N}(b)]f_i \quad i = 1, \ldots, d \\ \quad \mathrm{Iter}_a(\boldsymbol{p}, \boldsymbol{x}) \\ \quad \mathrm{Iter}_b(\boldsymbol{p}, \boldsymbol{y}) \end{array} \right.$$

where $\boldsymbol{x}$ (resp. $\boldsymbol{y}$) is a tuple of integer variables of length $|\,\mathfrak{N}(a)|$ (resp. $|\,\mathfrak{N}(b)|$).

For an explicit ordinal synchronization pair, the synchronization formula is the conjunction of:

$$\left\{ \begin{array}{l} \quad [\boldsymbol{x} \setminus \mathfrak{N}(a)]e_i = [\boldsymbol{y} \setminus \mathfrak{N}(b)]f_i \qquad i = 1, \ldots, d \\ [\boldsymbol{x} \setminus \mathfrak{N}(a)]r_p.\mathrm{EXP} = [\boldsymbol{y} \setminus \mathfrak{N}(b)]r_w.\mathrm{EXP} \\ \quad \mathrm{Iter}_a(\boldsymbol{p}, \boldsymbol{x}) \\ \quad \mathrm{Iter}_b(\boldsymbol{p}, \boldsymbol{y}) \end{array} \right.$$

where $\boldsymbol{x}$ and $\boldsymbol{y}$ are as above.

Let $[\mathfrak{N}(a)]r_p$ be an implicit ordinal synchronization reference. We associate with it a synchronization formula $\mathrm{Sync}_u : a \to a$, which is the conjunction of:

$$\begin{cases} [\boldsymbol{x} \setminus \mathfrak{N}(a)]r_p.\text{EXP} + 1 = [\boldsymbol{y} \setminus \mathfrak{N}(a)]r_p.\text{EXP} \\ \mathrm{Iter}_a(\boldsymbol{p}, \boldsymbol{x}) \\ \mathrm{Iter}_a(\boldsymbol{p}, \boldsymbol{y}) \end{cases}$$

where $\boldsymbol{x}$ and $\boldsymbol{y}$ are tuples of integer variables of length $|\mathfrak{N}(a)|$.

### 5.3 Precedence formula

We would like to use a predicate $\mathrm{Pre}(a(\boldsymbol{i}), b(\boldsymbol{j}))$, expressing that: "If $a(\boldsymbol{i})$ and $b(\boldsymbol{j})$ are both executed, then the overall parallel program structure (sequential control and synchronization) implies that $a(\boldsymbol{i})$ is executed before $b(\boldsymbol{j})$ in any run of the parallel version".

In order to obtain Pre from $\mathrm{Pre}^0$, we have to compose the sequential control flow $\mathrm{Pre}^0$ with the synchronization flow Sync. This composition is not exactly a transitive closure, as might be expected; $\mathrm{Pre}(a(\boldsymbol{i}), b(\boldsymbol{j}))$ and $\mathrm{Pre}(b(\boldsymbol{j}), c(\boldsymbol{k}))$ do not imply $\mathrm{Pre}(a(\boldsymbol{i}), c(\boldsymbol{k}))$. For instance:

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

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

So, instead of transitivity of Pre, we have "transitivity modulo Exe":

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

The relation Pre would therefore be obtained, through this transitive closure modulo Exe along paths, and by disjunction between alternate paths (in a "conjunction in series, disjunction in parallel" manner), from the previously calculated relation $\mathrm{Pre}^0$ and the synchronization precedence Sync. However, the Exe predicate is not well-defined, so we replace it by $\mathrm{Exe}^{\mathsf{S}}$ and we will prove that this replacement is sound. We define the $\mathrm{Pre}^{\mathsf{S}}$ predicate as the transitive closure of $\mathrm{Pre}^0$ and Sync modulo $\mathrm{Exe}^{\mathsf{S}}$.

The transitive closure of $\mathrm{Pre}^0$ is taken care of by the above calculation of $\mathrm{Pre}^0$. Therefore, it is straightforward to show that the precedence paths to consider between statements $a$ and $b$ are

paths of the following form (considering only event synchronizations):

$$a \rightarrow p_1 \rightsquigarrow w_1 \rightarrow p_2 \rightsquigarrow w_2 \rightarrow \ldots \rightarrow p_n \rightsquigarrow w_n \rightarrow b,$$

where $\rightarrow$ denotes the $\mathrm{Pre}^0$ relation, $p_i$ denotes a POST, $w_i$ denotes a WAIT, and $\rightsquigarrow$ denotes the synchronization relation Sync. The corresponding calculation of $\mathrm{Pre}^{\mathsf{S}}$ will be realized through relations such as:

$$\mathrm{Pre}^0_{a,p_1} \wedge \mathrm{Exe}^{\mathsf{S}}_{p_1} \wedge \mathrm{Sync}_{p_1,w_1} \wedge \mathrm{Exe}^{\mathsf{S}}_{w_1} \wedge \mathrm{Pre}^0_{w_1,p_2} \wedge \ldots \wedge \mathrm{Exe}^{\mathsf{S}}_{w_n} \wedge \mathrm{Pre}^0_{w_n,b} \Rightarrow \mathrm{Pre}^{\mathsf{S}}_{a,b}$$

Ordinal synchronizations work along similar lines.

# 6 Data dependences

When several parallel processes share a memory location, if one of them writes it, and others read or write it, the resulting effect is unpredictable, because of a race between these processes. This is considered to be an error in a parallel program, at least in our case, because the results are different from the sequential version.

In a sequential program, as well as in a single process execution, there is no memory conflict, but each memory location induces a *data dependence* between statement instances which sequentially access it, from write to read (a flow dependence), from read to write (an antidependence), or from write to write (an output dependence). This data dependence generates a partial order on statement instances which is included in the total execution order. Starting with a parallel program, the data dependence order defined for its sequential version must be preserved by any multiple process execution, so that no memory conflict can occur.

Dependences computation does not play the same role for parallelization and for verification of parallel programs. Dependences in sequential programs are allowed and are obstructions to restructuring transformations. Dependences in a parallel program can be *unsafe* and must be forbidden if they incur memory conflicts; on the other hand, dependences that are shown to be safe are allowed. Although this safety problem is dynamic, we want to search statically for a possible occurrence of memory conflicts. Doing so, we point out pairs of statements some instances of which may create memory conflicts.

## 6.1 Dependence pairs

Let $a$, $b$ be IF boolean expressions, or assignments. We define dependence pairs based on $(a, b)$ as pairs of the form $([\mathfrak{N}(a)]r, [\mathfrak{N}(b)]s)$, where $r$, resp. $s$ is a reference in $a$, resp. $b$, as follows.

- If $\mathfrak{OutNames}(a) \cap \mathfrak{InNames}(b) \neq \varnothing$, let $n$ be the name such that $\mathfrak{OutNames}(a) \cap \mathfrak{InNames}(b) = \mathfrak{OutNames}(a) = \{n\}$, and $r_o$ such that $\mathfrak{Out}(a) = \{r_o\}$, and $\mathrm{strip}(r_o) = n$; for every reference $r \in \mathfrak{In}(b)$ such that $\mathrm{strip}(r) = n$, let us consider the out-in dependence pair $([\mathfrak{N}(a)]r_o, [\mathfrak{N}(b)]r)$
- if $\mathfrak{OutNames}(b) \cap \mathfrak{InNames}(a) \neq \varnothing$, then let $n$ be such that $\mathfrak{OutNames}(b) \cap \mathfrak{InNames}(a) = \mathfrak{OutNames}(b) = \{n\}$, and $r_o$ such that $\mathfrak{Out}(b) = \{r_o\}$, and $\mathrm{strip}(r_o) = n$; for every reference $r \in \mathfrak{In}(a)$ such that $\mathrm{strip}(r) = n$, let us consider the in-out dependence pair $([\mathfrak{N}(a)]r, [\mathfrak{N}(b)]r_o)$

- if $\mathfrak{OutNames}(a) \cap \mathfrak{OutNames}(b) \neq \emptyset$, then let $n$ be the name such that $\mathfrak{OutNames}(a) \cap \mathfrak{OutNames}(b) = \mathfrak{OutNames}(a) = \mathfrak{OutNames}(b) = \{n\}$, and $r_o$, $s_o$ the references such that $\mathfrak{Out}(a) = \{r_o\}$, $\mathfrak{Out}(b) = \{s_o\}$, and $\mathrm{strip}(r_0) = \mathrm{strip}(s_0) = n$; let us consider the out-out dependence pair $([\mathfrak{N}(a)]r_o, [\mathfrak{N}(b)]s_o)$.

Note that if $u = ([\mathfrak{N}(a)]r_o, [\mathfrak{N}(b)]r)$ is an out-in pair, then $\hat{u} = ([\mathfrak{N}(b)]r, [\mathfrak{N}(a)]r_o)$ is an in-out pair, if $u = ([\mathfrak{N}(a)]r, [\mathfrak{N}(b)]r_o)$ is an in-out pair, then $\hat{u} = ([\mathfrak{N}(b)]r_o, [\mathfrak{N}(a)]r)$ is an out-in pair, and that if $u = ([\mathfrak{N}(a)]r_o, [\mathfrak{N}(b)]s_o)$ is an out-out pair, so is $\hat{u} = ([\mathfrak{N}(b)]s_o, [\mathfrak{N}(a)]r_o)$. The pair $\hat{u}$ is called the opposite dependence pair.

**Remark** For the computation of dependences, there is no evaluation problem caused by possible conflicts, since the subscript expressions only involve parameters and index variables and we supposed that none of them is writable: the value of a subscript expression is therefore well-defined.

## 6.2 Sharing and dependences

For the computation of dependences, a process $p$ (resp. $q$) executes $a$ (resp. $b$), and a memory location is accessible in the data environment of $p$ and $q$. If only shared variables were involved, there would be only one data environment, as in the case of sequential programs. Private variables occur in two forms: implicit (loop indices) and explicit (statement NEW). Let us remind that the private attribute is relative to the process and not to the thread (see Section 2.2).

There is no dependence between loop indices since we have supposed that they are not writable (and not because they are private). The situation is more intricate for explicitly private variables.

If a variable is explicitly private with respect to a parallel construct, there is no dependence if processes $p$ and $q$ are not the same, but we do not know which process is involved in the execution, so dependence is intrinsically dynamic. However, such independence may not preserve the serial interpretation. For instance, in

```
parallel sections
 new T
 section
  T = 0
 section
  A = T
end parallel sections
```

the conflict between `T=0` and `A=T` is removed if the sections are executed by different processes, leaving shared variable `A` undefined, whereas it remains if only one process is involved, the result depending on the scheduling of the sections. If the PARALLEL SECTIONS is ORDERED, the behavior is the same as in the sequential version; otherwise, variable `A` is undefined on completion of this construct. Now, before `A=T`, put an arbitrary sequential body in the second section; variable `A` is still defined in the sequential version, whereas in the parallel program, the definition of `A` depends only on the definition of `T` in this *sequential* body. Since we want to investigate only the parallel constructs, we should then accept programs that lead to variables with undefined values while their sequential versions lead to defined values. Therefore, we must modify the definition of the sequential version of a program.

In the fragment we study, making an object private to a parallel construct is normally useful in order to use it as a temporary variable, local to one thread[2]. This is the case for:

```
parallel do I = 1, 100
new T
 T = A(I) ** 2
 B(I) = T * (T - 1)
end parallel do
```

where the use of variable `T` allows `A(I)**2` to be computed only once. (Actually, it should be the compiler's task to make such optimization, and not the programmer's.) On the contrary, in the loop

```
parallel do I=1, 100
new T
 T = T + 1
 A(I) = T
end parallel do
```

`T` is uninitialized when each process executes the assignment for the first time, leading to an array `A` with random values. If an initialization `T = 0` is inserted just after `new T`, then every `A(I)` gets 1. The same behavior appears in the sequential version, in both cases, if we suppose that the NEW statement is converted into renaming in the sequential version. This program is incorrect for misuse of sequential, not parallel constructs.

In order to preserve the serial semantics, the simplest idea is to require that the value of a private variable used in a thread has been computed in the same thread, and therefore to promote a private variable to an array in case of PARALLEL DO, or to duplicate it as one distinct variable per section, in case of PARALLEL SECTIONS. This will faithfully represent the case of maximum parallelism (one process per thread). Even if this interpretation is not that of X3H5, because the full language does need a privatization mechanism based on processes, we believe that the intended meaning of private variable in the fragment we study is relative only to parallel threads instead of processes. Hence, in the definition of the sequential version of a parallel program, the NEW $n$ statement is transformed into

1. a new name in the scope
2. an array promotion: if statement NEW $n$ is specified in a parallel loop of header $h$, then a dimension is added to name $n$, whose length is the number of iterations in $h$.

Note that array promotion for private variables in parallel loops should need dynamic allocation, since the number of threads is the result of loop bounds evaluation; so the NEW statement should be replaced by a DIMENSION statement just before the loop, within its loop nest, at a place where it must not be found in Fortran 77 (if this nest is not empty). For instance, given a loop nest

[2]Note that the PARALLEL region of X3H5 allows for more general ways of using a private variable in the scope of the parallel construct, where initialization and use of the value computed by the work distributed by the worksharing constructs can be done within the parallel construct; we do not study this form.

```
parallel do I = 1, 100
 do J = 1, I
  parallel do K = J, I
  new T
   .....
```

its sequential version should be something like

```
do I = 1, 100
 do J = 1, I
  dimension T(J:I)
  do K = J, I
   .....
```

A static hull of the array dimension can be defined, giving in this example `dimension T(1:100)` instead of `dimension T(J:I)`, so that the sequential version we consider does remain within the framework of Fortran 77. The computation of this static hull is made possible by our assumption that loop bounds depend only on parameters and indices of surrounding loops.

A few examples are shown below, with the sequential versions at the right of the parallel program. With simple renaming

```
parallel sections
new T
 section
  T = 0                                Tps1 = 0
 section
  A = T                                A = Tps2
end parallel sections
```

there is no dependence, since the names are made different. With array promotion, in

```
parallel do I = 3, 100                 dimension Tdo(3:100)
new T                                  do I = 3, 100
 T = A(I) ** 2                          Tdo(I) = A(I) ** 2
 B(I) = T * (T - 1)                     B(I) = Tdo(I) * (Tdo(I) - 1)
end parallel do                        end do
```

or, in the sequentially incorrect program

```
parallel do I=1, 100                   dimension Tdo(1:100)
new T                                  do I=1, 100
 T = T + 1                              Tdo(I) = Tdo(I) + 1
 A(I) = T                               A(I) = Tdo(I)
end parallel do                        end do
```

there is a dependence pair involving the variable `Tdo`.

The scalar variable `T` is private for the PARALLEL DO and shared for the PARALLEL SECTIONS in the following fragment:

```
parallel do I = 1, 100                 dimension Tdo(1:100)
new T                                  do I = 1, 100
 parallel sections (ordered)
```

```
      section
       T = A(I)                              Tdo(I) = A(I)
       post(E(I))
      section
       wait(E(I))
       B(I) = T                              B(I) = Tdo(I)
     end parallel sections
    end parallel do                         end do
```

The value of T used in the $i$th parallel iteration has been computed in the same iteration, but in the other thread of the PARALLEL SECTIONS.

With this completed definition of the sequential version, there will be no dependence involving private variables at the structure level where they have been so declared.

### 6.3 Dependence formula

Let $u = ([\mathfrak{N}(a)]r, [\mathfrak{N}(b)]s)$ be any dependence pair based on $(a, b)$. By definition, $\text{strip}(r) = \text{strip}(s) = n$, and both $r$, $s$ are scalar references or array references of the same dimension, say $d$. Let $r.\text{EXPLIST} = [e_1, \ldots, e_d]$, and $s.\text{EXPLIST} = [f_1, \ldots, f_d]$. These subscript expressions $e_i$ and $f_i$ may contain as variables the program parameters $\boldsymbol{p}$.

We associate to the dependence pair $u$ a dependence formula $\text{Dep}_u : a \to b$, which is the conjunction of the equations and inequations of the following system:

$$\begin{cases} [\boldsymbol{x} \setminus \mathfrak{N}(a)]e_i = [\boldsymbol{y} \setminus \mathfrak{N}(b)]f_i \quad i = 1, \ldots, d \\ (a \prec\prec b)(\boldsymbol{x}, \boldsymbol{y}) \\ \text{Iter}_a(\boldsymbol{p}, \boldsymbol{x}) \\ \text{Iter}_b(\boldsymbol{p}, \boldsymbol{y}) \end{cases}$$

where $\boldsymbol{x}$ (resp. $\boldsymbol{y}$) is a tuple of integer variables of length $|\mathfrak{N}(a)|$ (resp. $|\mathfrak{N}(b)|$), and $\prec\prec$ is the sequential order. This dependence formula $\text{Dep}_u$ is named out-in, in-out or out-out according to the dependence pair from which it is built. We say that $u$ *is a dependence* between the references $r$ and $s$ if the dependence formula $\text{Dep}_u(\boldsymbol{p}, \boldsymbol{x}, \boldsymbol{y}) : a \to b$ is satisfiable, i.e, has a solution $(\boldsymbol{p}, \boldsymbol{i}, \boldsymbol{j})$. Further, it is called a true dependence, an anti-dependence, or an output dependence, when $\text{Dep}_u$ is respectively an out-in, in-out or out-out formula.

As the relation $a \prec\prec b$ is anti-symmetric, every tuple $(\boldsymbol{i}, \boldsymbol{j})$ can be solution of at most one of the formulas $\text{Dep}_u$ and $\text{Dep}_{\hat{u}}$, where $\hat{u}$ is the opposite of $u$. However, both $\text{Dep}_u$ and $\text{Dep}_{\hat{u}}$ may have (different) solutions; this is the case for the loop

```
      do I = 1, 10
a:     A(I) = ...
b:     ... = A(3*I-5)
      end do
```

For $u = ([1 \leqslant I \leqslant 10]A(I), [1 \leqslant I \leqslant 10]A(3 * I - 5))$, $\text{Dep}_u$ is $x = 3y - 5 \wedge x \leqslant y \wedge 1 \leqslant x \leqslant 10 \wedge 1 \leqslant y \leqslant 10$, and $\text{Dep}_{\hat{u}}$ is $x = 3y - 5 \wedge y < x \wedge 1 \leqslant x \leqslant 10 \wedge 1 \leqslant y \leqslant 10$. The pair $(1, 2)$ is a solution of $\text{Dep}_u$, whereas $(4, 3)$ is a solution of $\text{Dep}_{\hat{u}}$: both formulas are satisfiable.

Note that if $n$ is the name of a scalar reference, then the dependence equation on $n$ reduces to $n = n$, and the set of solutions is the graph of the relation $\prec\prec$ in the product $\text{Iter}_a \times \text{Iter}_b$.

### 6.4 Depth and support

Let $a$, $b$ be IF boolean expressions, or assignments. The dependence vectors are vectors in the common iteration space. The notation $\boldsymbol{j}_{\restriction N}$ stands for the $|N|$ first components of $\boldsymbol{j}$. If $(\boldsymbol{i},\boldsymbol{j}) \in \mathrm{Iter}_a \times \mathrm{Iter}_b$ satisfies a dependence formula $\mathrm{Dep}_u : a \rightarrow b$, then the difference $\boldsymbol{j}_{\restriction \mathfrak{N}(a,b)} - \boldsymbol{i}_{\restriction \mathfrak{N}(a,b)} \in \mathcal{CIS}(a,b)$ is called a *dependence vector*. Let $\mathrm{Vec}_u$ be their set. The dependence is *uniform* if the set $\mathrm{Vec}_u$ is reduced to one element: this is a very usual case, but using formulas allows us also to deal with non uniform dependences. When the dependence is loop independent (it occurs within the same iteration of embedding loops), $\mathrm{Vec}_u = \mathrm{Vec}_{\hat{u}} = \{(0,\ldots,0)\}$.

The *depth* of a dependence $\mathrm{Dep}_u : a \rightarrow b$ is the least rank $k$ such that there exists a solution $(\boldsymbol{p},\boldsymbol{i},\boldsymbol{j})$ of $\mathrm{Dep}_u$ with a non-zero $k$th component of $\boldsymbol{j}_{\restriction \mathfrak{N}(a,b)} - \boldsymbol{i}_{\restriction \mathfrak{N}(a,b)}$.

The *support* of a non-zero dependence is the binding loop in the loop nest whose depth is equal to the dependence depth: it is *serial* if it is a DO, and *parallel* if it is a PARALLEL DO statement. The support index is the index of the support loop statement.

For instance, in the loop

```
parallel do I
 do J
  A(I,J) = ...
  ... = A(I-1,J+1)
 end do
end parallel do
```

the dependence is of depth 1, the support loop is the `parallel do I`, which is parallel, the support index is `I`.

The support of a zero dependence is the common control type of the pair of statements; it is *serial* if the common control type is ';' and *parallel* if it is '||'.

## 7 Correctness of parallel programs

The *dependence preservation* requirement (introduced in [5]) can be expressed as follows:

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

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

$$\mathrm{Exe}_a^{\mathsf{S}} \wedge \mathrm{Exe}_b^{\mathsf{S}} \wedge \mathrm{Dep}_{a,b} \Rightarrow \mathrm{Pre}_{a,b}^{\mathsf{S}}$$

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

Moreover, we could then replace Dep and $\mathrm{Pre}^{\mathsf{S}}$ by approximations, from above for Dep, from below for $\mathrm{Pre}^{\mathsf{S}}$ : it is sufficient to use predicates $\mathrm{Dep}_\star$ and $\mathrm{Pre}_\star^{\mathsf{S}}$ such that $\mathrm{Dep} \Rightarrow \mathrm{Dep}_\star$ and $\mathrm{Pre}_\star^{\mathsf{S}} \Rightarrow \mathrm{Pre}^{\mathsf{S}}$, respectively meaning that "a dependence may exist" and "a precedence must exist".

In the following results we do not consider ordinal synchronizations, but only event synchronizations. These results could be rather straightforwardly extended so as to include ordinals.

### 7.1 Single process execution

The ANSI X3H5 proposal requires that a compliant program not deadlock whatever the number of available processes. Therefore, a program should not deadlock in a single process execution. On the other hand, we wish to ensure the semantic equivalence between the sequential version and any single process execution. This will be taken care of by a result we will first prove.

Before stating Assumption A1, let us recall that, under the X3H5 proposal, the ORDERED condition in a worksharing construct means that its parallel threads will be assigned to the available processes, in the sequential order. In other words, no parallel thread shall be assigned to an available process as long as another parallel thread preceding it in the sequential order, has not been assigned yet.

**Assumption A1** All parallel constructs in which there are synchronization statements (i.e. POST, WAIT, CLEAR statements) are ORDERED.

***Comment*** This assumption, though not a requirement of the proposal, is suggested in it, as a hint towards "good" programming. It can easily be checked at compilation time. Throughout our study, we will consider only programs meeting this assumption.

**Theorem 1** *Under the following hypotheses:*

- *Assumption A1*
- *There is no dependence between two parallel threads within any non-*ORDERED *parallel construct;*
- *Some single process execution does not deadlock;*

*the sequential version and any single process execution are semantically equivalent (in the sense defined above).*

*Proof :* We consider a program instance. Let us first restrict ourselves to the case when all parallel constructs are ORDERED. In that case, by definition of the ORDERED clause, there is exactly one single process run, and the execution order of the statement instances, if executed in both runs, is the same in both runs.

For this program instance, let us compare the execution of its sequential version and a single process execution of its parallel version. The only possible differences between the two versions, then, are consecutive to the fact that the synchronization statements are disabled in the sequential version.

Therefore, considering runs of the two programs, the first difference arising, if any, involves a synchronization statement. For everything occurring before that moment, both versions are semantically equivalent. Thus, the event involved in this synchronization statement is the same in both runs. This synchronization statement cannot be a POST, nor a CLEAR, because no difference would thus occur. It has to be a WAIT involving an event which is not *posted* at that point, hence a deadlock in the single process run, whereas the sequential run proceeds.

Therefore, either such a deadlock occurs — in which case, there is a semantic equivalence up to the deadlock point — or there is no difference arising between the two runs.

Let us consider now the case when there are non-ORDERED parallel constructs. Then, according to Assumption A1, these constructs contain no synchronization statement. Now, still considering a program instance in its sequential and its single process aspects, an extra difference must be considered: in every non-ORDERED parallel construct, the parallel threads may be assigned to the process in any order, which gives place to several possible single process runs.

We assume that there is no dependence between any two such threads. Therefore, no difference is brought, in the values computed, by inverting any two threads. Thus, any permutation of threads brings no such difference: the threads are "execution order independent". Hence we obtain the same result as in the case when all parallel constructs are ORDERED.

◀

## 7.2 Deadlock analysis

It is straightforward to express the no-deadlock condition for single process execution, in any program instance, under Assumption A1. According to the previous result, we have to express that, for any WAIT statement instance, any single process run does not deadlock precisely on that statement instance. So, for any given WAIT instance $\omega$, we wish to express that, either $\omega$ is not reached in the sequential version, or, if it is, and assuming that no deadlock occurred before $\omega$ in the single process run we are considering – hence the semantic equivalence up to $\omega$ – the event that $\omega$ involves is *posted* at that point. The semantic equivalence up to $\omega$ is crucial here, allowing us to consider expression evaluation *in the sequential version* up to $\omega$.

Let $W$ be the finite set of WAIT statements in the program, each $w \in W$ with an iteration vector $\boldsymbol{i}_w$. For every $w$, the event expression involved is $w.\text{LHS}$, and its value computed by instance $w(\boldsymbol{i})$ in a run equivalent to the sequential execution up to itself is $[\![w.\text{LHS}]\!]@w(\boldsymbol{i})$.

$$\text{nodeadlock} = \bigwedge_{w \in W} \forall \boldsymbol{y}_w (\text{Exe}^{\mathsf{S}}_w(\boldsymbol{y}_w) \Rightarrow [\![w.\text{LHS}]\!]@w(\boldsymbol{y}_w) = \textit{posted})$$

An event is *posted* at some point, iff this event has been previously posted by a POST and not cleared by a CLEAR meanwhile. Let $P$ (resp. $C$) be the set of POST (resp. CLEAR) statements in the program.

$$\begin{aligned}([\![w.\text{LHS}]\!]@w(\boldsymbol{y}_w) = \textit{posted}) \quad = \quad & \bigvee_{p \in P} \exists \boldsymbol{x}_p \\ & \text{Seq}_{p,w}(\boldsymbol{x}_p, \boldsymbol{y}_w) \wedge \text{Exe}^{\mathsf{S}}_p(\boldsymbol{x}_p) \wedge \\ & (p.\text{LHS.NAME}) = w.\text{LHS.NAME}) \wedge \\ & (p.\text{LHS.EXPLIST}[\boldsymbol{i}_p \setminus \mathfrak{N}(p)] = w.\text{LHS.EXPLIST}[\boldsymbol{i}_w \setminus \mathfrak{N}(w)]) \wedge \\ & \text{noclear}_{p,w}(\boldsymbol{x}_p, \boldsymbol{y}_w)\end{aligned}$$

The noclear predicate expresses that no CLEAR has cleared the event posted by a POST before a WAIT:

$$\begin{aligned}\text{noclear}_{p,w}(\boldsymbol{x}_p, \boldsymbol{y}_w) \quad = \quad & \bigwedge_{c \in C} \forall \boldsymbol{z}_c \\ & \neg[\text{Seq}_{p,c}(\boldsymbol{x}_p, \boldsymbol{z}_c) \wedge \text{Seq}_{c,w}(\boldsymbol{z}_c, \boldsymbol{y}_w) \wedge \text{Exe}^{\mathsf{S}}_c(\boldsymbol{z}_c) \wedge \\ & (c.\text{LHS.NAME} = p.\text{LHS.NAME}) \wedge \\ & (c.\text{LHS.EXPLIST}[\boldsymbol{z}_c \setminus \mathfrak{N}(c)] = p.\text{LHS.EXPLIST}[\boldsymbol{x}_p \setminus \mathfrak{N}(p)])]\end{aligned}$$

### 7.3 Main result

**Assumption S1 (weak version)** A CLEAR statement, and a POST or WAIT statement involving the same event variable, must be related in a precedence relation, or else must be mutually exclusive (lying in alternative branches of an IF). In other words, there must be no race condition between a POST or WAIT statement, and a CLEAR statement involving the same event.

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

**Assumption S1 (strong version)** A CLEAR statement, and a POST or WAIT statement involving the same event must not be present within the same parallel construct.

***Comment*** This strong version, that we will assume in our algorithms, seems reasonable too; moreover, it can be checked quite straightforwardly.

**Assumption S2 (Ensured precedence from POST to WAIT)** For any WAIT statement, for any event variable this statement may involve, all POST statements susceptible to trigger this WAIT by posting the corresponding event, if there are several, are mutually exclusive (only one may be executed in a given execution).

***Comment*** This assumption will be necessary, in our context, in order to be able to use $\mathrm{Sync}(p, w)$ as a precedence, when a POST statement instance $p$ posts the event that the WAIT statement instance $w$ is waiting for. Otherwise, in case several non mutually exclusive POST statement instances post the same event, it is no longer possible to state that any of them, if executed, is executed before the corresponding WAIT. Notice that we do not require a symmetric condition: one POST may post towards several non mutually exclusive WAITs.

Under Assumptions S1(weak) and S2, together with the *static event reference* assumption (the restriction that the event arguments depend only on parameters and indices, not variables), the relation Sync is well-defined, in the sense that it can be expressed statically.

**Theorem 2** *Under the following hypotheses:*

*i. Assumptions S1 (weak version) and S2;*

*ii. No deadlock in any single process run;*

*iii. For all statements a and b,* $\mathrm{Exe}_a^{\mathsf{S}} \wedge \mathrm{Exe}_b^{\mathsf{S}} \wedge \mathrm{Dep}_{a,b} \Rightarrow \mathrm{Pre}_{a,b}^{\mathsf{S}}$*;*

*the execution predicate* Exe *is well-defined and equals* $\mathrm{Exe}^{\mathsf{S}}$*, and the parallel program is semantically equivalent to its sequential version.*

We will first prove a lemma.

**Lemma 1** *For any statement instance* $\alpha$*, except the program start and* WAIT *statements,*

- *either* $\mathrm{Exe}(\alpha)$ *(resp.* $\mathrm{Exe}^{\mathsf{S}}(\alpha)$*) equals the conjunction of* $\mathrm{Exe}(\beta)$ *(resp.* $\mathrm{Exe}^{\mathsf{S}}(\beta)$*) for one or several statement instances* $\beta$ *such that* $\mathrm{Pre}^0(\beta, \alpha)$*;*

- *or there exists a statement instance* $\beta$ *such that* $\mathrm{Pre}^0(\beta, \alpha)$ *and that* $\mathrm{Exe}(\alpha)$ *(resp.* $\mathrm{Exe}^{\mathsf{S}}(\alpha)$*) depends only on the result of the execution of* $\beta$ *on the data and control environment; moreover,* $\beta$ *has to be executed in order for* $\alpha$ *to be executed.*

*Moreover, for any* WAIT *instance* $\omega$*, the condition denoted* $\psi(\omega)$ *(resp.* $\psi^S(\omega)$*) for this instance to be reached (not meaning that it is passed) similarly depends on statement instances* $\beta$ *such that* $\mathrm{Pre}^0(\beta, \omega)$*.*

*Proof (of the lemma).* For any statement instance $\alpha$, except the program start and WAITs, we are in one of the following cases, for the statement $a$ of which $\alpha$ is an instance:

- $a$ is the first statement in the THEN or ELSE branch of an IF, or the first statement in a loop body: then, we are in the second case of the lemma, where $\beta$ corresponds to the IF statement or the loop head, respectively.

- $a$ is a END PARALLEL DO or END PARALLEL SECTIONS, or a section WAIT: then we are in the first case of the lemma, the execution of the statement instance depends on the termination of every parallel thread it concludes, or (for a section WAIT) of some thread which precedes it (in the sense of $\mathrm{Pre}^0$).

- In all other cases (remember we rule out the program start and the WAITs), there is a statement $b$ immediately preceding $a$ (in an obvious sense): we are again in the first case of the lemma, with one $\beta$: the corresponding instance of $b$.

As for the condition for a WAIT instance to be reached, the same proof is obtained by fictively inserting a CONTINUE statement just before the WAIT statement, and applying the previous derivation to this CONTINUE.

◀

Notice that this result is consistent with the previously mentioned fact that the precedences expressed by $\mathrm{Pre}^0$ are independent of the specific parallel run considered – a fact that we will keep in mind in the following proof of the theorem.

*Proof (of the theorem).* We consider a program instance, by giving values to the parameters. Then, there is only one run of the sequential version of this program instance, whereas there are generally several possible runs of the parallel version. We consider one of them.

1) We will make use of a notion of *time step*. In order to remain as general as possible, we will assume three properties for the set of time steps considered here: the execution of a statement instance is achieved in one time step; for every time step, there is (at least potentially) a next one; two statement instances which are in a precedence relation cannot execute in the same time step.

Let $t$ be a time step such that the following recurrence assumption holds:

> *Semantic equivalence up to time step t:* for any statement instance $\alpha$ executed strictly before $t$ in this parallel run, $\alpha$ is also executed in the sequential version; moreover, all variables involved in $\alpha$ (as input or output) underwent the same computations, due to the same statement instances, in both runs up to the point reached, in the parallel run, just before time step $t$.

We wish to prove that this semantic equivalence extends to time step $t$.

2) Let us consider the set $E_t$ of all statement instances $\gamma$ *eligible for execution* at time step $t$, i.e. which are executed at time step $t$ by *some* parallel run, knowing the current run up to time step $t$. All or some of them will be executed at time step $t$ by the existing processes, while the rest of them will be postponed to the set $E_{t'}$ of statement instances eligible for execution at the next time step $t'$ – if still eligible. We first show that for any $\gamma \in E_t$, $\mathrm{Exe}^{\mathsf{S}}(\gamma)$ holds.

For $\gamma \in E_t$, in case $\gamma$ is not a WAIT statement instance, the above lemma applies: $\mathrm{Exe}(\gamma)$ depends only on some statement instance(s) $\beta$ such that $\mathrm{Pre}^0(\beta, \gamma)$, and such that $\beta$ has been executed before $t$. The recurrence hypothesis then implies that the result of the execution of $\beta$ on the environment is identical in the sequential run. Therefore, $\gamma$ is also executed in the sequential version: we have $\mathrm{Exe}^{\mathsf{S}}(\gamma)$.

In case $\gamma$ is a WAIT statement instance, its executability implies that it has been reached and that the event it involves is posted at this point; hence, this event has been previously posted and not previously cleared afterwards. Due to the recurrence hypothesis and the above lemma, this statement is therefore reached (and executed, because it is disabled) in the sequential version: therefore, we have $\mathrm{Exe}^{\mathsf{S}}(\gamma)$ too.

Notice that the executability of $\gamma$ at time step $t$ cannot be overridden afterwards in case $\gamma$ is postponed. This results from the above lemma in case $\gamma$ is not a WAIT; when it is, this results from the weak version of Assumption S1, which rules out the possibility that the event it involves be *cleared* randomly.

3) We show now that the hypothesis of semantic equivalence propagates to all instances in $E_t$ executed at $t$ by the given run. Let us consider an instance $\gamma \in E_t$, and some variable $x$ used by $\gamma$ as input. In order to ensure the semantic equivalence for $\gamma$, since we assume the semantic equivalence up to time step $t$, we just need to rule out the case that the value of $x$ used by $\gamma$ as input in the sequential version would be computed by some instance $\beta$ *not executed before time* $t$ in this parallel run. If this were the case, we would have $\mathrm{Exe}^{\mathsf{S}}(\beta)$ because it is $\beta$ which computes $x$ in the sequential version; we have $\mathrm{Exe}^{\mathsf{S}}(\gamma)$ as we saw in 2); we would have $\mathrm{Dep}(\beta, \gamma)$; therefore we would get $\mathrm{Pre}^{\mathsf{S}}(\beta, \gamma)$. This is not sufficient for immediately excluding that $\gamma$ be executable at time step $t$, because we must remember that $\mathrm{Pre}^{\mathsf{S}}$ is defined referring to the sequential version and the values acquired by various expressions in that version, and we do not know, at this point, what happens after time step $t$. However, a closer look will show us that $\mathrm{Pre}^{\mathsf{S}}(\beta, \gamma)$ indeed rules out the hypothesis that $\gamma \in E_t$, hence the contradiction we are looking for.

We clearly would not have $\mathrm{Pre}^0(\beta, \gamma)$, since otherwise, $\beta$ would necessarily have been executed before time step $t$; so $\mathrm{Pre}^{\mathsf{S}}(\beta, \gamma)$ would be realized through synchronizations, i.e., as previously explained, through one or several paths of the form:

$$\beta \rightarrow p_1 \text{ or } \beta = p_1$$

$$p_1 \rightsquigarrow w_1 \rightarrow p_2 \rightsquigarrow w_2 \rightarrow \cdots \rightarrow p_n \rightsquigarrow w_n$$

$$w_n \rightarrow \gamma \text{ or } w_n = \gamma$$

where, again, $\rightarrow$ denotes a $\mathrm{Pre}^0$ relation, $p_i$ denotes a POST, $w_i$ denotes a WAIT, and $\rightsquigarrow$ denotes a synchronization link Sync.

The executability of $\gamma$ at time step $t$ implies that, either some WAIT instance $w_n$ has been executed before $t$, or – in case $\gamma = w_n$ – the event involved in $w_n$ has been *posted*. Then, necessarily,

this event has been previously posted by $p_n$, according to Assumption S2. Continuing upstream through the above path, we find that the executability of $\gamma$ at time step $t$ implies that $\beta$ must have been executed before, hence the contradiction.

We have thus proved that any statement instance $\alpha$ executed in some parallel run is also executed in the sequential version, and that any variable involved in this statement instance undergoes the same calculations (and therefore receives the same values) in both runs. This shows $\mathrm{Exe} \Rightarrow \mathrm{Exe}^{\mathsf{S}}$ and the semantic equivalence restricted to all variables effectively involved in some parallel run.

4) For the converse, $\mathrm{Exe}^{\mathsf{S}} \Rightarrow \mathrm{Exe}$, suppose by contradiction that there are statement instances which are executed in the sequential version and not in some parallel run, and let $\alpha$ be the earliest one in the sequential order, for some given run.

In case $\alpha$ is not a WAIT, according to the lemma, the non-execution of $\alpha$ in this parallel run is dependent on previous (in the sense of $\mathrm{Pre}^0$) statement instances; by definition of $\alpha$, these instances are executed in this parallel run, with the same effects on the control and data environment, as shown previously. This contradicts the non-execution of $\alpha$ in this parallel run.

In case $\alpha$ is a WAIT – hence a deadlock in this parallel run – let us consider the single process run matching the sequential version (this is the single process run obtained by considering the non-ORDERED parallel constructs as if they were ORDERED). Then, due to the semantic equivalence up to $\alpha$, as above, the event involved in $\alpha$ in both runs is the same, and it is similarly *uninitialized* or *cleared* in both runs. This would imply a deadlock in this single process run, a possibility which we rule out in hypothesis (ii) of our theorem.

◀

This theorem, together with Theorem 1, straightforwardly implies the following result:

**Corollary 1** *Under the following hypotheses :*

- *Assumption A1, S1(weak) and S2;*
- *Some single process run does not deadlock;*
- *There is no dependence between two parallel threads within any non-*ORDERED *parallel construct;*
- *For all statements a and b,* $\mathrm{Exe}_a^{\mathsf{S}} \wedge \mathrm{Exe}_b^{\mathsf{S}} \wedge \mathrm{Dep}_{a,b} \Rightarrow \mathrm{Pre}_{a,b}^{\mathsf{S}}$ *;*

*the parallel program and its sequential version are semantically equivalent; especially, no parallel run can deadlock.*

# 8 An algorithm to check the correctness of a program

The aim of this algorithm is to check the condition *iii* of Theorem 2. No other checking is to be done : the syntactic restrictions, assumptions A1, S1(strong) and S2, and the no-deadlock condition must be checked elsewhere. Besides, the algorithm presently relies on the possibility of statically defining the relevant predicates by formulas. Therefore, we assume that no variable occurs in boolean conditions of IF statements.

### 8.1 Easy cases

For every dependence pair $u = (r, s)$ based on $a, b$, the following steps are realized:

1. if $\mathfrak{N}(a,b)$ does not contain any PARALLEL DO and if $\mathfrak{C}(a,b) \neq ||$, then $a$ and $b$ cannot be concurrently executed, so $u$ may only be a safe dependence, and therefore has not to be considered

2. otherwise, as condition $\mathrm{Seq}_{a,b}(\boldsymbol{x}, \boldsymbol{y})$ is a disjunction, each member of this disjunction, of the form $(x_1 = y_1) \wedge \ldots \wedge (x_{k-1} = y_{k-1}) \wedge (x_k < y_k)$, gives a system (a conjunction of equations and inequations) which is tested in turn for each value of $k$, from 1 to $|\mathfrak{N}(a,b)|$

3. if no system has solutions, then $u$ is not a dependence and has not to be considered

4. the first (i.e. corresponding to the least $k$) system which has a solution yields the depth $k$ and support of the dependence

5. if this support is sequential, which means a DO loop, then this dependence is preserved

6. if this support is parallel, then the dependence may not be preserved, and synchronizations must be analyzed

7. other systems (i.e. corresponding to larger $k$'s) must be solved too, looking for other dependences

The algorithm which is used for these systems is the Omega test, based on Fourier-Motzkin's variable elimination, recently developed by W. Pugh[10] at the University of Maryland, and part of Wolf's *tiny* parallelization tool (Cf. [12]). As this test applies only to *linear* systems, we can deal only with programs whose bound and subscript expressions are linear. As for other tests, such as those built upon the simplex algorithm, it is a conservative test that can only ensure non-existence of solutions.

The interest of the Omega test lies in its ability to deal with integer unknowns and to realize symbolic projections $\pi_{\boldsymbol{x}}$ in order to eliminate the variables not in $\boldsymbol{x}$, which give reduced forms of the constraints. A reduced form is an important information which may be used to fix an incorrect program; however, we know how to use this reduced form only when projection does not splinter a problem into several ones.

This test solves more general problems than systems of equations and inequations; it can compute a *gist* formula defined so that $(\mathrm{gist}\, p \,\mathrm{given}\, q) \wedge q$ is equivalent to $p \wedge q$ ([11]). In order to prove that a formula $\forall \boldsymbol{x}\boldsymbol{y}\,(D \Rightarrow \exists \boldsymbol{z} P)$ is a tautology, we compute $\mathrm{gist}\, \pi_{\boldsymbol{x},\boldsymbol{y}}(P) \,\mathrm{given}\, D$, and check that it is true.

### 8.2 Non locality of synchronization

The following example shows that a synchronization may have a global effect.

```
parallel sections (ordered)
 section
  post(E(1))
  parallel do (ordered) I = 2,N
```

```
      a
      post(E(I))
      wait(F(I-1))
      b
     end parallel do
    section
     do I = 1,N
      wait(E(I))
      post(F(I))
     end do
   end parallel sections
```

The precedence between $a(i)$ and $b(i+1)$, for $i \geqslant 2$ is caused by the synchronizations, not by the control flow inside the PARALLEL DO; the precedence chain is $a(i) \rightarrow postE(i) \rightsquigarrow wait(E(i)) \rightarrow post(F(i)) \rightsquigarrow wait(F(i)) \rightarrow b(i+1)$.

Therefore, to compute precedences within a construct, the synchronizations force us to consider a whole program unit. This non-locality, in the same flavor as other non-structured control primitives (the infamous goto, the locks), is of course an unpleasant feature of parallel programming in Fortran X3H5.

### 8.3 Block graph

To compute precedences in a program unit, a block graph is built. This is a graph whose nodes are *blocks*, a generalization of basic blocks (Cf. [1]); a similar structure is proposed in [5]. Blocks are of five kinds: fork blocks, synchronization blocks, conditional blocks, join blocks and ordinary blocks.

- a fork block is a PARALLEL SECTIONS block
- a synchronization block is
  - either a sequence of statements, the first of which is a (event or ordinal) WAIT, or the last of which is an event POST (but both synchronization statements are nor allowed in the same block), and the others do not contain any synchronization statement
  - or made of one ordinal POST
- a conditional block block contains the boolean expression of an IF
- a join block is an END PARALLEL SECTIONS or END IF block
- an ordinary block is a list of statements where no synchronization statement occurs.

Note that statements in a block are not only simple ones (assignments, as in basic blocks, and synchronizations) but may contain any structured control, sequential or parallel, which does not involve synchronization. Once this decomposition into blocks is done, a structured statement can be seen as made up from a sequence of blocks instead of a sequence of statements. All statements in a block share the same iteration vector (to which other components may be added inside for

loops) and the same execution formula (to which other formulas may be conjuncted inside loops and IF branches).

The decomposition is done so that, first, the synchronization blocks are maximal, second the ordinary blocks are maximal. For instance, there exists a conditional block only when at least one branch contains a synchronization; two ordinary blocks are not consecutive.

Precedences within a block are computed by the $Pre^0$ relation. This relation is extended to hold between blocks, or between statements and blocks, considering a block as a statement.

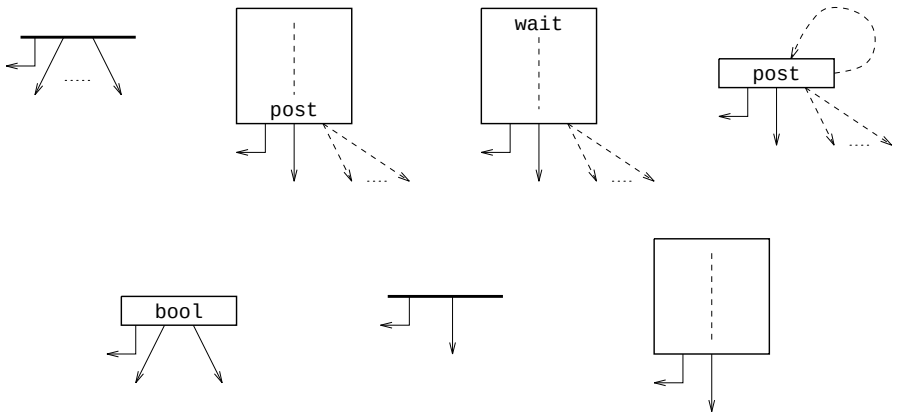


Figure 2: Blocks

Blocks are implemented as structures, with links as shown in Figure 2, which could use outgoing fields `succ` to the next block in a sequence, the sections in a PARALLEL SECTIONS or the branches of an IF, `back` to the first block in a DO body, `sync` to the WAIT blocks related to a POST block.

Graph edges are of two kinds:

- synchronization edges between a POST-block and a WAIT-block for events or ordinals of same name, and from an ordinal POST-block to itself
- control edges:
  - from an IF block to the first blocks of its branches
  - from the last blocks in branches of an IF to the END IF block
  - from a block to the following block in the sequence of a structured statement
  - from the last block in a DO loop to its first block
  - from the PARALLEL SECTIONS block to the first block in every section
  - from the last block of every section to the END PARALLEL SECTIONS block
  - from the last block of a section to the first block of a section waiting for it (through a WAIT clause)

Once built, as shown on the following section, the graph is simplified, by removing join and empty blocks.

**Procedure** $graph([s_1, \ldots, s_m], l_i, l_o, l_d)$

**input** a sequence of statements $s_1, \ldots, s_m$, an input link $l_i$

**output** a next link $l_o$, a DO loop link $l_d$

**side-effect** creation of a list of blocks

**method**

```
l : current input link, initialized to l_i
k : current block
for each statement s_i do
  if no synchronization occurs in s_i
    if (k = 0)
      CreateBlock(l, k)
    AddToBlock(k, s_i, l)
  if s_i is a WAIT or an ordinal POST
    if (k ≠ 0)
      CloseBlock(k)
    CreateBlock(l, k)
    AddToBlock(k, s_i, l)
    if s_i is an ordinal POST
      CloseBlock(k)
  if s_i is an event POST
    if (k = 0)
      CreateBlock(l, k)
    if CheckWait(k)
      CloseBlock(k)
      CreateBlock(l, k)
    AddToBlock(k, s_i, l)
    CloseBlock(k)
  if s_i is an IF, at least one branch of which contains a synchro
    if (k ≠ 0)
      CloseBlock(k)
    CreateIfBlock(l, s_i.BEXP, k, l_1, l_2)
    graph(s_i.THEN, l_1, l'_1, _)
    graph(s_i.ELSE, l_2, l'_2, _)
    CreateEndIfBlock(l'_1, l'_2, k, l)
  if s_i is a DO whose body contains a synchro
    if (k ≠ 0)
      CloseBlock(k)
    graph(s_i.BODY, l, l', l_d)
    CloseDo(l, l_d)
  if s_i is a PARALLEL DO whose body contains a synchro
    if (k ≠ 0)
      CloseBlock(k)
    graph(s_i.BODY, l, l', l_d)
  if s_i is a PARALLEL SECTIONS at least one of the n sections of which contains a synchro
    if (k ≠ 0)
      CloseBlock(k)
    CreateForkBlock(l, n, k, l')
    for j = 1, ..., n do
      graph(s_i.SECTION*[j], l'[j], l''[j], _)
    AddWaitEdges(k, l', l'')
    CreateJoinBlock(l'', k, l)
```

## 8.4 Construction of the block graph

The algorithm takes as input a sequence of statements and outputs a sequence of blocks; it uses the following procedures:

*CreateBlock*$(l, k)$: allocates a new block $k$, and assigns it to $l$

*CreateIfBlock*$(l, exp, k, l_1, l_2)$: allocates a new IF block $k$, puts *exp* into its expression field, assigns it to $l$, and makes $l_1$ and $l_2$ refer to its outgoing pointer fields

*CreateForkBlock*$(l, n, k, l')$: allocates a new PARALLEL SECTIONS block $k$ for $n$ sections, assigns it to $l$, and makes pointer array variable $l'$ refer to its outgoing pointer field

*CreateJoinBlock*$(l', n, k, l)$: allocates a new END PARALLEL SECTIONS block $k$ for $n$ sections, assigns $l'[j]$ to $k$, and makes $l$ refer to its outgoing pointer field

*AddToBlock*$(k, s, l)$: add statement $s$ at the end of block $k$, and makes $l$ refer to the outgoing pointer field of $k$

*CloseBlock*$(k)$: assigns the null pointer to $k$

*CloseDo*$(l, l_d)$: assigns the block referred to by $l$ to $l_d$

*CreateEndIfBlock*$(l'_1, l'_2, k, l)$: allocates a new END IF block $k$, assigns it to $l'_1$ and $l'_2$ and makes $l$ refer to the outgoing pointer field of $k$

*CheckWait*$(k)$: returns *true* if first statement in $k$ is a WAIT, and *false* otherwise

*AddWaitEdges*$(k, l', l'')$: add edges corresponding to section WAIT clauses, updates $l'$ and $l''$ to remove edges from the fork block to waiting sections, and from waited sections to the join block.

## 8.5 Computation of precedence formulas

Let $a$, $b$ be statements that can be involved in a dependence relation, between which we must compute a precedence relation. Let $K_a$ and $K_b$ be the blocks containing $a$ and $b$, and $u$ a path from $K_a$ to $K_b$.

- decompose $u$ as a product $u_1 \dots u_n$ where each $u_i$ is, either a control path (i.e., a product of control edges), or a synchronization edge, so that any two consecutive $u_i$ are not both control paths; if $u$ is empty, then form $\mathrm{Pre}^0_{a,b}$
- if $u_i$ is a control path from block $K_p$ to block $K_q$ and $K_p \neq K_q$, then form $\mathrm{Pre}^0_{K_p,K_q}$;
- if $u_i$ is a control path from block $K_p$ to itself, and $n = 1$, then form $\mathrm{Pre}^0_{a,b}$;
- if $u_i$ is a synchronization edge and the statements involved are $p$ and $w$, form $\mathrm{Exe}^{\mathsf{S}}_{K_p} \wedge \mathrm{Sync}_{K_p,K_w} \wedge \mathrm{Exe}^{\mathsf{S}}_{K_w}$; for $u_1$, form $\mathrm{Pre}^0_{a,p} \wedge \mathrm{Exe}^{\mathsf{S}}_{K_p} \wedge \mathrm{Sync}_{K_p,K_w} \wedge \mathrm{Exe}^{\mathsf{S}}_{K_w}$; for $u_n$ form $\mathrm{Exe}^{\mathsf{S}}_{K_p} \wedge \mathrm{Sync}_{K_p,K_w} \wedge \mathrm{Exe}^{\mathsf{S}}_{K_w} \wedge \mathrm{Pre}^0_{w,b}$
- conjunct these $n$ formulas to obtain $\mathrm{Pre}^{\mathsf{S}}_u$.

## 8.6 The formula to be proved

We have to prove, for a path $u$, that the formula

$$\forall \boldsymbol{x} \forall \boldsymbol{y} \exists \boldsymbol{z} \, (\mathrm{Exe}_a^{\mathsf{S}} \wedge \mathrm{Exe}_b^{\mathsf{S}} \wedge \mathrm{Dep}_{a,b} \Rightarrow \mathrm{Pre}_u^{\mathsf{S}})$$

is valid. Aside from free program parameters, quantified variables are:

- iteration variables $\boldsymbol{x}$ of $a$ and $\boldsymbol{y}$ of $b$
- iteration variables $\boldsymbol{z}$ for blocks on path $u$, occurring only on the right hand side of the implication.

$\mathrm{Dep}_{a,b}$ is a disjunction $D_1 \vee \ldots \vee D_m$, because of its component $\mathrm{Seq}_{a,b}$; every $D_i$ is a system of equations and inequations. Only terms $D_i$ which have not be proven unsatisfiable by the Omega test must be processed here.

$\mathrm{Exe}^{\mathsf{S}}$ is a conjunction of IF conditions and inequations derived from loop bounds, involving only parameters and iteration variables.

$\mathrm{Pre}_u^{\mathsf{S}}$ is a conjunction (along $u$) of formulas $\mathrm{Exe}^{\mathsf{S}}$, $\mathrm{Pre}^0$ and Sync; $\mathrm{Pre}^0$ is a disjunction of equations and inequations, like *Seq*; Sync is a conjunction of equations and inequations on loop bounds. Transforming $\mathrm{Pre}_u^{\mathsf{S}}$ into disjunctive normal form, i.e. into a disjunction of systems $P_j$, we must show that for all $i$ and all $\boldsymbol{x}, \boldsymbol{y}$, there exists a system $P_j$ such that

$$D_i(\boldsymbol{x}, \boldsymbol{y}) \Rightarrow P_j(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$$

has at least one solution in $\boldsymbol{z}$.

It is sufficient to prove that, for all $i$, there exists a system $P_j$ such that

$$\forall \boldsymbol{x} \forall \boldsymbol{y} \exists \boldsymbol{z} \; D_i(\boldsymbol{x}, \boldsymbol{y}) \Rightarrow P_j(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$$

is valid. This is only a sufficient condition.

## 8.7 Examples

### Example 1

```
      A(0) = ....
      post(E(0))
      parallel do (ordered) I=1,N
w:     wait(E(I-1))
b:     .... = A(I-1)
a:     A(I) = .....
p:     post(E(I))
      end parallel do
```

Statements $a$ and $b$ satisfy:

- $\mathfrak{N}(a, b) = [(\mathrm{PDO}, I, 1, N)]$
- $\mathfrak{C}(a, b) = ;$

- $\mathfrak{In}(b) = \{A(I-1)\}$
- $\mathfrak{Out}(a) = \{A(I)\}$
- $\mathfrak{OutNames}(a) \cap \mathfrak{InNames}(b) = \{A\}$

Therefore, the dependence formula must be built:

$$\mathrm{Dep}_{a,b}(x,y) = (1 \leqslant x \leqslant N) \wedge (1 \leqslant y \leqslant N) \wedge (x < y) \wedge (x = y - 1)$$

As this formula is satisfiable, precedences must be computed. Control precedences are:

$$\mathrm{Pre}^0_{a,b}(x,y) = \mathit{false}.$$

As $\mathrm{Pre}^0_{a,b}(x,y)$ does not follow from $\mathrm{Dep}_{a,b}(x,y)$, synchronizations must be considered, and the block graph must be built. It is displayed on Figure 3. We have $\mathrm{Exe}^{\mathsf{S}}_a(x) = (1 \leqslant x \leqslant N)$ and $\mathrm{Exe}^{\mathsf{S}}_b(y) =$

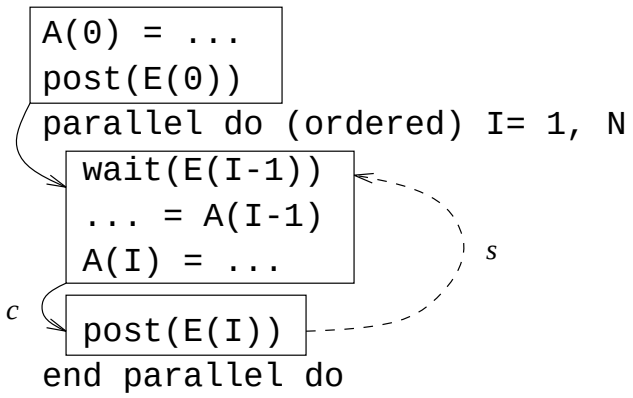


Figure 3: Block graph: example 1

$(1 \leqslant y \leqslant N)$. Statements $a$ and $b$ occur in the same wait block $k_w$, and there is a synchronization edge $s$ from the post block $k_p$ to $k_w$ with the formula $\mathrm{Sync}_{p,w}(x,y) = (1 \leqslant x \leqslant N) \wedge (1 \leqslant y \leqslant N) \wedge (x = y - 1)$, and a control edge $c$ from $k_w$ to $k_p$; paths from $k_p$ to $k_w$ are the $s(cs)^n$ for $n \geqslant 0$. The formula $\mathrm{Pre}_s(x,y,z_1,z_2)$ is $\mathrm{Pre}^0_{a,p}(x,z_1) \wedge \mathrm{Exe}^{\mathsf{S}}_p(z_1) \wedge \mathrm{Sync}_{p,w}(z_1,z_2) \wedge \mathrm{Exe}^{\mathsf{S}}_w(z_2) \wedge \mathrm{Pre}^0_{w,b}(z_2,y)$. We have to prove:

$$\forall x \forall y \exists z_1 \exists z_2 \, (\mathrm{Dep}_{a,b}(x,y) \wedge \mathrm{Exe}^{\mathsf{S}}_a(x) \wedge \mathrm{Exe}^{\mathsf{S}}_b(y) \Rightarrow \mathrm{Pre}^{\mathsf{S}}_s(x,y,z_1,z_2))$$

i.e.,

$$\begin{array}{l} \forall x \forall y \exists z_1 \exists z_2 \\ ((1 \leqslant x \leqslant N) \wedge (1 \leqslant y \leqslant N) \wedge (x < y) \wedge (x = y - 1) \\ \Rightarrow \\ (x = z_1) \wedge (1 \leqslant z_1 \leqslant N) \wedge (z_1 = z_2 - 1) \wedge (1 \leqslant z_2 \leqslant N) \wedge (z_2 = y)) \end{array}$$

This problem is passed to the Omega test, which computes a *gist* which is true (empty system); this is sufficient to prove the program correctness. The following trace is obtained:

```
parameter list = (n)

------- conflict:
- conflict on name = a
- statement A =
      a(i166) = 1
- statement B =
      x(i166) = a(i166-1)
- type = wr
- head operator = seq
- A << B = false
- common nest =
paralleldo i166
- dependence equation =
i166 = -1+i166
- iteration space for statement A =
1 <= i166
i166 <= n
- iteration space for statement B =
1 <= i166
i166 <= n

allpaths=((& & &) (& & & & &) (& & & & & & &) (& & & & & & & & &))
testing path of length 3
path=(() pre0 0)

alternative sys:
precedence problem:
- Eq:
0) y0 = z1
1) z0 = x0
2) z1 = 1+z0
- GEq:
0) 1 <= z0
1) z0 <= n
2) 1 <= z1
3) z1 <= n
dependence system:
0) y0 = 1+x0
0) 1+x0 <= y0
1) y0 <= n
2) 1 <= y0
3) x0 <= n
4) 1 <= x0
projected precedence system:
5) y0 <= 1+x0
6) 1+x0 <= y0
7) 1 <= x0
8) 1+x0 <= n
gist: ()
```

If we substitute `E(I+1)` for `E(I-1)`, the computed gist is instead $1 + y \leqslant x$, which is not a valid formula, as variables $x$ and $y$ are universally quantified: this result is a serious hint for the program

to be incorrect.

**Example 2: a non uniform dependence**

```
      parallel sections (ordered)
       section
        do I=1,L
a:       A(I) = ...
p:       post(E(I))
        end do
       section
        do J=1,M
         do K=1,N
w:        wait(E(J+K))
b:        ... = A(J+K)
         end do
        end do
      end parallel sections
```

Statements $a$ and $b$ satisfy:

- $\mathfrak{N}(a,b) = ()$
- $\mathfrak{C}(a,b) = ||$
- $\mathfrak{In}(b) = \{A(J+K)\}$
- $\mathfrak{Out}(a) = \{A(I)\}$
- $\mathfrak{OutNames}(a) \cap \mathfrak{InNames}(b) = \{A\}$.

The dependence formula is:

$$\mathrm{Dep}_{a,b}(x,y,z) = (1 \leqslant x \leqslant L) \wedge (1 \leqslant y \leqslant M) \wedge (1 \leqslant z \leqslant N) \wedge (x = y + z)$$

The precedence graph is displayed on Figure 4. The shortest path between the blocks containing $a$ and $b$ yields a synchronization precedence formula, which is the same as $\mathrm{Dep}_{a,b}(x,y,z)$. Checking "Dep $\Rightarrow$ $\mathrm{Pre}^{\mathsf{S}}$" is therefore straightforward, as shown in the following output, by the third alternative system.

```
parameter list = (n m l)

------- conflict:
- conflict on name = a
- statement A =
      a(i173) = x(i173)
- statement B =
      y(j172+k171) = a(j172+k171)
- type = wr
- head operator = par
- A << B = true
```

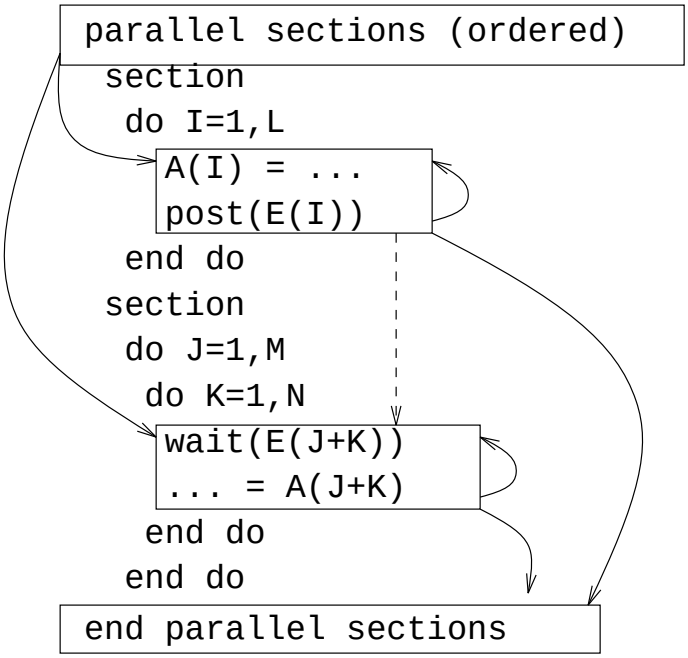


Figure 4: Block graph: example 2

```
- common nest =
- dependence equation =
i173 = j172+k171
- iteration space for statement A =
1 <= i173
i173 <= l
- iteration space for statement B =
1 <= j172
j172 <= m
1 <= k171
k171 <= n
allpaths=((& &) (& & &))
testing path of length 2
path=(() 0)

alternative sys:
path problem:
- Eq:
0) z0 = x0
1) z1+z2 = z0
- GEq:
0) 1+z1 <= y0
1) 1 <= z0
2) z0 <= l
3) 1 <= z1
4) z1 <= m
5) 1 <= z2
```

```
6) z2 <= n
dependence system:
0) y0+y1 = x0
0) y1 <= n
1) 1 <= y1
2) y0 <= m
3) 1 <= y0
4) x0 <= l
5) 1 <= x0
projected precedence system:
6) x0 <= n+m
7) 2 <= x0
8) x0 <= l
9) 1+x0 <= n+y0
10) 2 <= y0
11) 1 <= m
12) 1 <= n
gist: (10 9)
2 <= y0
1+x0 <= n+y0

alternative sys:
path problem:
- Eq:
0) y0 = z1
1) z0 = x0
2) z1+z2 = z0
- GEq:
0) 1+z2 <= y1
1) 1 <= z0
2) z0 <= l
3) 1 <= z1
4) z1 <= m
5) 1 <= z2
6) z2 <= n
dependence system:
0) y0+y1 = x0
0) y1 <= n
1) 1 <= y1
2) y0 <= m
3) 1 <= y0
4) x0 <= l
5) 1 <= x0
projected precedence system:
6) 1+x0 <= y0+y1
7) x0 <= n+y0
8) x0 <= l
9) 1 <= y0
10) y0 <= m
11) 1+y0 <= x0
gist: (6)
1+x0 <= y0+y1

alternative sys:
path problem:
```

```
- Eq:
0) y1 = z2
1) y0 = z1
2) z0 = x0
3) z1+z2 = z0
- GEq:
0) 1 <= z0
1) z0 <= l
2) 1 <= z1
3) z1 <= m
4) 1 <= z2
5) z2 <= n
dependence system:
0) y0+y1 = x0
0) y1 <= n
1) 1 <= y1
2) y0 <= m
3) 1 <= y0
4) x0 <= l
5) 1 <= x0
projected precedence system:
6) y0+y1 <= x0
7) x0 <= y0+y1
8) x0 <= n+y0
9) x0 <= l
10) 1 <= y0
11) y0 <= m
12) 1+y0 <= x0
gist: ()
```

This is a case of non uniform dependence, where usual parallelization methods which begin with the computation of dependences cannot compute a dependence distance; here, the formal processing of equations allows to complete the correctness proof.

**Example 3: synchronizations through loops**

```
      parallel sections (ordered)
       section
        do I=1,M
a:       A(2*I) = ...
p:       post(EA(2*I-1))
w:       wait(EB(I))
b:       ... = B(I)
        end do
       section
        do J=1,N
b':      B(3*J) = ...
p':      post(EB(3*J))
w':      wait(EA(J+1))
a':      ... = A(J)
```

```
 end do
end parallel sections
```

- $\mathfrak{N}(a, a') = ()$
- $\mathfrak{C}(a, a') = ||$
- $\mathfrak{In}(a') = \{A(J)\}$
- $\mathfrak{Out}(a) = \{A(2 * I)\}$
- $\mathfrak{OutNames}(a) \cap \mathfrak{InNames}(a') = \{A\}$.

The dependence formula between $a$ and $a'$ is:

$$\mathrm{Dep}_{a,a'}(x, y) = (1 \leqslant x \leqslant M) \wedge (1 \leqslant y \leqslant N) \wedge (2x = y)$$

The precedence graph is displayed on Figure 5. To the synchronization edge $s$ from the block

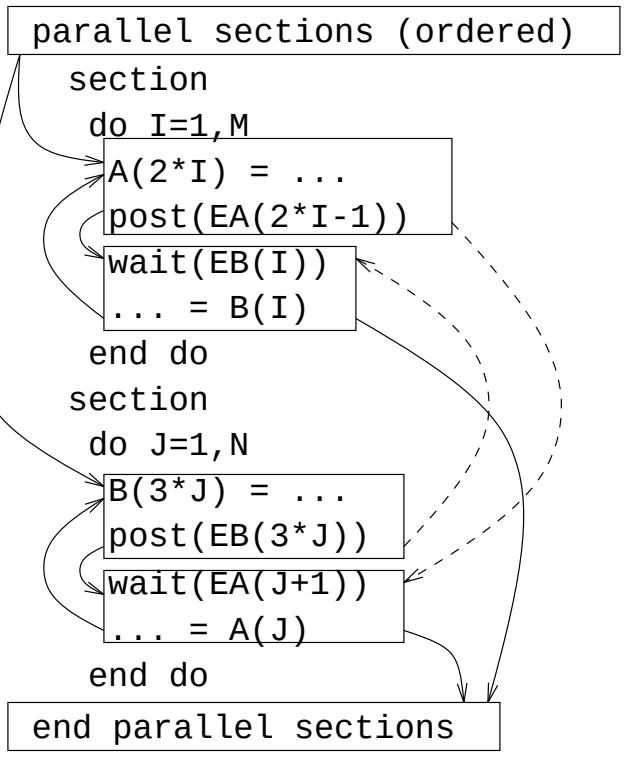


Figure 5: Block graph: example 3

containing $a$ to the block containing $a'$ is associated the synchronization formula

$$\mathrm{Sync}_{p,w'}(x, y) = (1 \leqslant x \leqslant M) \wedge (1 \leqslant y \leqslant N) \wedge (2x - 1 = y + 1);$$

we have the precedence formula

$$\mathrm{Pre}_s^{\mathsf{S}}(x,y,z_1,z_2) = \mathrm{Pre}^0_{a,p}(x,z_1) \wedge \mathrm{Sync}_{p,w'}(z_1,z_2) \wedge \mathrm{Pre}^0_{w',a'}(z_2,y).$$

i.e., the system

$$\begin{cases} 1 \leqslant x \leqslant m \\ 1 \leqslant z_1 \leqslant m \\ x \leqslant z_1 \\ 1 \leqslant z_2 \leqslant n \\ 2z_1 - 1 = z_2 + 1 \\ 1 \leqslant y \leqslant n \\ z_2 \leqslant y \end{cases}$$

Actually, this system is a disjunction of four alternative systems, because of the disjunctive form of the precedences. When the projection of each of these systems onto variables $x$, $y$, $m$, $n$ is done, the computed *gist* contains universal variables, so that none is true. It can be proved that the dependence is preserved if $2 \leqslant m$, but the system is not presently able to do it.

**Example 4**

```
do I=1,M
 A(I)=I
 post(E(I))
enddo
parallel sections (ordered)
 section
  parallel do (ordered) I=M+1,3*M
   wait(E(I-M))
   A(I)=I*A(I-M)
   post(E(I))
  end parallel do
 section
  parallel do J=1,M+1
   wait(E(3*M+1-J))
  end parallel do
  parallel do I=3*M+1,3*M+N
   A(I)=I*A(I-M)
  end parallel do
end parallel sections
```

There are four dependences, as shown on Figure 6 The precedence graph is displayed on Figure 7. The first two dependences (within the PARALLEL DO) are obviously not preserved, because there is no precedence path from the statement to itself.

```
------- conflict:
- conflict on name = a
- statement A =
      a(i162) = i162*a(i162-m)
```

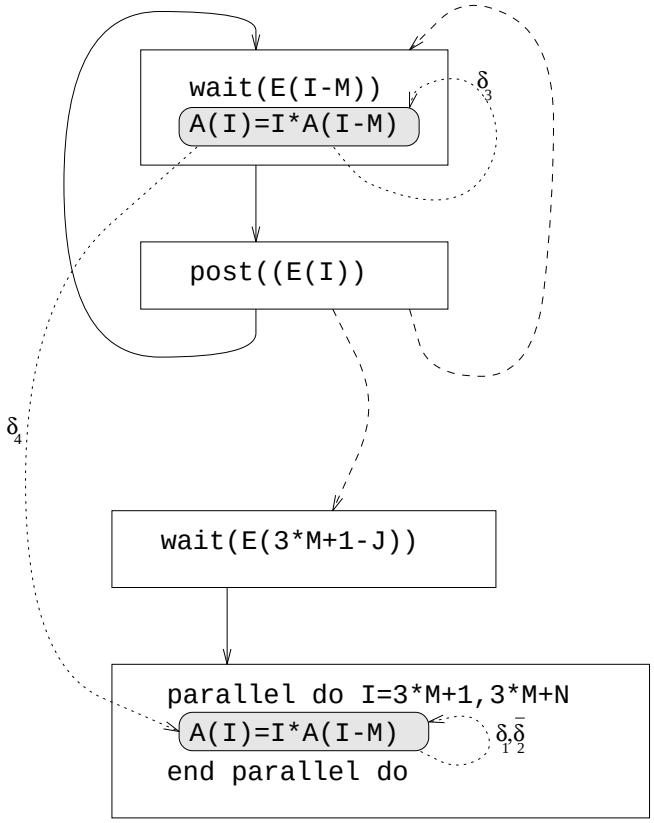


Figure 6: Dependences: example 4

```
- statement B =
      a(i162) = i162*a(i162-m)
- type = wr
- head operator = seq
- A << B = false
- common nest =
paralleldo i162
- dependence equation =
i162 = -m+i162
- iteration space for statement A =
1+ 3m <= i162
i162 <= 3m+n
- iteration space for statement B =
1+ 3m <= i162
i162 <= 3m+n

allpaths=()

------- conflict:
- conflict on name = a
- statement A =
```

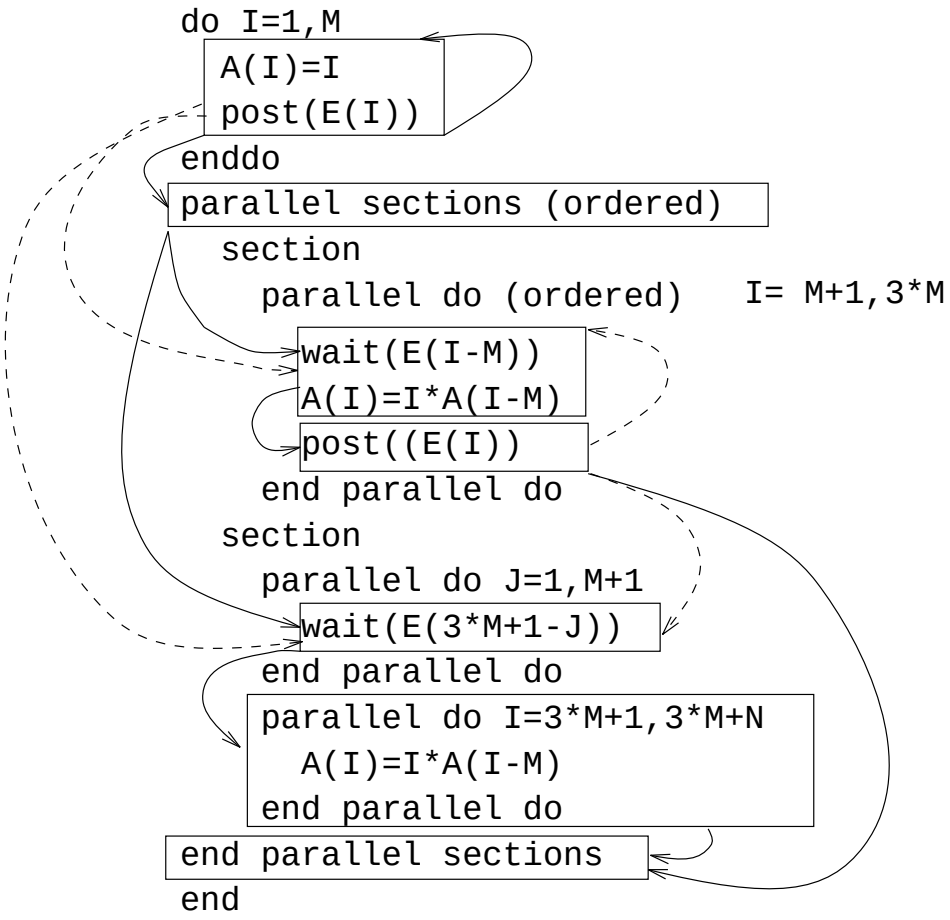


Figure 7: Block graph: example 4

```
      a(i162) = i162*a(i162-m)
- statement B =
      a(i162) = i162*a(i162-m)
- type = rw
- head operator = seq
- A << B = false
- common nest =
paralleldo i162
- dependence equation =
-m+i162 = i162
- iteration space for statement A =
1+ 3m <= i162
i162 <= 3m+n
- iteration space for statement B =
1+ 3m <= i162
i162 <= 3m+n

allpaths=()
```

The self-dependence is preserved through the first path, of length 3:

```
------- conflict:
- conflict on name = a
- statement A =
      a(i160) = i160*a(i160-m)
- statement B =
      a(i160) = i160*a(i160-m)
- type = wr
- head operator = seq
- A << B = false
- common nest =
paralleldo i160
- dependence equation =
i160 = -m+i160
- iteration space for statement A =
1+m <= i160
i160 <= 3m
- iteration space for statement B =
1+m <= i160
i160 <= 3m

allpaths=((& & &) (& & & & &) (& & & & & & &) (& & & & & & & & &))
testing path of length 3
path=(() pre0 0)

alternative sys:
path problem:
- Eq:
0) y0 = z1
1) z0 = x0
2) z1 = m+z0
- GEq:
0) 1+m <= z0
1) z0 <= 3m
2) 1+m <= z1
3) z1 <= 3m
dependence system:
0) y0 = m+x0
0) 1+x0 <= y0
1) y0 <= 3m
2) 1+m <= y0
3) x0 <= 3m
4) 1+m <= x0
projected precedence system:
5) y0 <= m+x0
6) m+x0 <= y0
7) 1+m <= x0
8) x0 <= 2m
gist: ()
```

The last one is also preserved.

# 9 Conclusions

In this report, we have presented a semantic analysis of a subset of Fortran X3H5. We believe that the emergence of a new standard will lead more programmers to design their own parallel programs, instead of relying on the automatic parallelization of sequential programs. However, it is not certain that control parallelism is the best choice, and the issue of the competition between Fortran X3H5 and data parallel extensions such as HPF is still unknown.

We tried to shed some light on the semantics of the control parallel constructs, using techniques such as dependence analysis, well-known in parallelization, but applied here in a different way. The correctness property we investigate is the semantic equivalence between a parallel program and its sequential version, which means that any (shared) memory location observes the same sequence of operations in any execution of the parallel program and in the execution of its sequential version. The problem is not to find all dependences in a program but to single out those which may be unsafe in a parallel execution. More accurately we must show that a dependence in the sequential version of the program is preserved by any parallel execution of the program. This preservation property comes from the work of Callahan, Kennedy and Subhlok ([5]). In case we cannot achieve such a proof, the dependence is considered as unsafe and reported as such to the programmer. To make the search for a proof possible, we had to strongly restrict the syntax of the subset we study. Our main result derives the semantic equivalence from several assumptions: syntactic restrictions, no deadlock in a single process execution, and the validity of a formula involving dependence, execution and precedence conditions. The construction and proof of this formula is the aim of the algorithm we propose.

This algorithm has been embedded in an interactive programming environment based on the Centaur[6] generic environment. It builds a graph whose nodes are blocks generalizing the basic blocks used in compiling, and whose edges hold the information of precedence (sequencing and synchronization). The syntactic restrictions allow for a formal processing of several predicates (dependence, iteration, control precedence, synchronization). With these predicates we build systems of linear equations and inequations with integer unknowns, and formulas which are quantified implications of such systems; such a formula is associated to each precedence path in the block graph between two nodes that may be in a dependence relation. In order to prove that such a formula is valid, we pass it to the Omega test ([10]). This processing, although rather heavy, gives results when more numerical algorithms fail, for instance where non uniform precedences are involved; moreover, it can output relations between program parameters which are sufficient conditions for a dependence to be preserved.

We look forward to release some of the assumptions of our main result, notably the staticity of references.

# References


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

[2] R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *ACM, Principles of Programming Languages.* ACM Press, 1987.

[3] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[4] U. Banerjee. An introduction to a formal theory of dependence analysis. *The Journal of Supercomputing*, 2:133–149, 1988.

[5] 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.

[6] D. Clément. GIPE: Generation of Interactive Programming Environments. *TSI*, 9(2):157–165, 1990.

[7] B. Dehbonei. *Génération de code et Analyse Interprocédurale au sein d'un Environnement de programmation Parallèle*. PhD thesis, Université Paris 6, December 1990.

[8] M.C. Giboulot, M. Loyer, G. Popovitch, and F. Thomasset. An interactive parallelizer under the Centaur environment. Technical report, ESPRIT-II, 1990.

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

[10] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[11] W. Pugh and D. Wonnacott. Eliminating false data dependences using the Omega test. Technical Report UMIACS-TR-92-114, Dept. of Computer Science, Univ. of Maryland, College Park, MD, dec. 1992.

[12] M. Wolfe. The *tiny* loop restructuring research tool. In *Proc. of the 1991 Internat. Conf. on Parallel Processing*, 1991.

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

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