ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

**He Jifeng, Oxford University Computing Laboratory, UK, John Cooke,
Loughborough University, UK, and Peter Wallis, University of Bath, UK (Eds)**

BCS-FACS 7th Refinement Workshop

Proceedings of the BCS-FACS 7th Refinement Workshop, Bath, 3-5
July 1996

Separating Algorithm and Implementation in the Refinement of Parallel Program Specifications

Denis Roegel

Published in Collaboration with the
British Computer Society

Separating Algorithm and Implementation in the Refinement of Parallel Program Specifications

Denis Roegel
CRIN
Bâtiment LORIA
Campus scientifique — BP 239
54506 Vandœuvre-lès-Nancy cedex, France
roegel@loria.fr

Abstract

Correct concurrent programs can be obtained through the refinement of abstract specifications. In this paper, we explore a methodology, which we call *task separation*, in which we split the development of a program in two phases: a first stage where an algorithm is introduced from a TLA^+ specification, but where the data structures remain unconstrained; a second stage where the other aspects of the program are dealt with. The intermediate state can be represented in an object-oriented way, emphasizing its relationship with languages like C++. This notation, cTLA (concrete TLA) is defined semantically in TLA^+ and aims to provide a fair representation of an algorithm, compared to TLA^+ 's flat rendering. cTLA's types and values are the values in TLA^+ . cTLA's class instances are processes, and there are virtually an infinite number of processes, which can be activated. A numerical example, the block decomposition algorithm in the matrix multiplication, supports our views.

1 Introduction

It is a well known fact that parallel programming is difficult. There is a need for a framework allowing the formal derivation of parallel programs. First, because the combinatory explosion makes these programs uncheckable by a mere enumeration. Then, the new methods used are very time-consuming, cumbersome, and involve a lot of small and tedious details, all things to which a tool can bring a crucial help.

There are many ways of getting correct parallel programs. Starting with an abstract specification, we can transform it in such a way that the operations become more and more detailed, and such that there is a correctness relationship between a transformed specification and the previous specification. This *refinement process* makes it easier to check the correctness of a specification, for it is an *incremental verification*.

The refinement process can be carried on to some extent without caring much about the implementation. However, *modularity* is kept in mind, for it makes reusability easier. We will show that a detailed *object-oriented* specification can be obtained through this derivation. This seems a valuable intermediate target for further refinement towards *real parallel programs*, that is, programs for which there is a compiler.

The paper goes on as follows: we first introduce our design methodology, the *task separation*. Then, the next section reviews the TLA^+ specification language. Our notion of refinement is explained in the following section. Then, the cTLA language is sketched, first syntactically, then semantically. The matrix multiplication example is then developed, as well as its “implementation” in cTLA.

2 Task separation as a design methodology

Our aim is to write *correct* concurrent programs, possibly using specific features from the implementation. We begin with a specification, viz. an abstract representation of the problem and of its solution(s). Typically, this creative writing activity involves several operations:

- An *algorithm* satisfying the specification must be found; this often includes the removal of nondeterministic parts;
- The specification must be *structured* in order to exploit the parallelism; independent parts must be isolated;
- The *data structures* have to be refined;
- Tasks must be *distributed* over processes;
- The program should be *efficient*;
- Various *implementation details*, such as rounding problems, finite range of values, etc., must be handled;
- The program must be transformed in order to *improve readability*;
- etc.

We believe that it is unreasonable to try to achieve all these tasks at once. Actually, we propose to *gather the tasks* into two main groups:

1. *A group of steps introducing an algorithm and its structure.* In these steps, we don't worry about efficiency; our concern is to introduce an algorithm and to state why and how it is working;
2. *A group of steps handling data refinement, efficiency, implementation problems, tuning, etc.* In these steps, the algorithm is already defined, and our concern is *data*: *where* is data, *what* is data, *how* does data circulate, etc. The algorithm is no longer changed here.

In this paper, we concentrate on the first stage, namely the introduction of an algorithm. Furthermore, we *rewrite* the result of this stage in an object-oriented notation similar to C++, in order to emphasize the link between our notation and this object-oriented language.

Our design process can therefore be seen as a two-stage process, with an initial specification of a problem, with a real solution at the other end, and with the object-oriented notation cTLA in between, as shown on figure 1.

Figure 1: cTLA

Constraining intermediate stages is something quite natural. It is common practice in physics. In thermodynamics for instance [4], several special kinds of transformations are considered. These transformations are characterized by the fact that they keep some properties invariant. For instance, in an *adiabatic* transformation, there is no heat transfer between the body (gas or liquid) under consideration and the environment. There are so-called *diagrams* corresponding to the behavior of typical machines, or way to achieve things. An example is the Carnot cycle. These diagrams show how to decompose some thermodynamic transformation into several transformations each conserving some property.

We can view program transformation in the same fashion. For instance, we decide to first take the specification (the “gas”) and transform it, keeping the data structures (or more exactly the way to express the data structures) invariant. Instead, we change the algorithm. In a second step, the steam is reversed and the algorithm is kept invariant, but the data structures are changed. This procedure may not lead to the “best” program, but it is a safe way and it gives insight into how the program was made.

Figure 2: TLA formulæ and traces

3 The TLA^+ specification language

The validity of our methodology, namely, the fact that we ensure a correct program at the end of the derivation, relies heavily on the formal language in which we describe the problem, and within which we transform it. We do not know how to handle the correctness in non-formal frameworks, since such frameworks are obviously more ambiguous.

3.1 TLA

Our formal development technique is the specification language TLA^+ . This language was proposed by L. Lamport in 1991. It is built upon the Temporal Logic of Actions (TLA) [8]. TLA is a linear and discrete temporal logic. Its objects are traces and a TLA formula is a synthetic expression for a set of traces. A TLA formula is true for some traces and it is convenient to identify the TLA formula with the set of all traces for which this formula is true.

More precisely, given a trace

$$\sigma = \sigma_0, \sigma_1, \sigma_2, \dots$$

where σ_i is the state i , and a TLA formula $F \triangleq \text{Init} \wedge [\text{Next}]_v \wedge L$, we say that σ *satisfies* F , and we write it

$$\sigma \models F$$

(1) if the initial state σ_0 satisfies the predicate *Init*, (2) if each pair of consecutive states σ_i, σ_{i+1} is either a *stuttering step*, viz. no variables in v are changed, or is a step satisfying the relation *Next* and (3) if the whole trace satisfies the liveness constraint L . Each of the conjuncts reduces the relevant traces as shown in figure 2 where the traces are represented as crosses.

Relations between two consecutive states are called *actions*. An action refers to variables in a current state and to variables in a next state. The former are represented as usual, whereas the latter use the prime ($'$) notation. Here are some simple examples of actions and their intuitive meaning:

Action	Meaning
$x' = x$	x is unchanged
$x' = x + 1$	x gets increased by 1
$x' + y' = x + y$	the sum $x + y$ remains constant
$((x' = x + 1) \wedge (y' = y)) \vee ((y' = y + 1) \wedge (x' = x))$	either increase x by 1 and keep y unchanged, or do the opposite.

The liveness part is usually a conjunction of weak and strong fairness restrictions. If $L \triangleq \text{WF}_v(A)$, it means that we restrict the traces to those satisfying $\text{WF}_v(A)$, that is, those for which a step A changing v cannot be enabled for ever from some point on without ever being true. In other words, $\text{WF}_v(A)$ discards

 Figure 3: Example of TLA^+ module

the traces where a step A changing v is *always enabled from some state on*, and is never true. Strong fairness differs by the fact that the action A is supposed to be *enabled infinitely often from some state on*.

In addition to these three conjuncts, variables can be *hidden* through quantification. A general TLA formula is therefore something like

$$\exists x : Init \wedge [Next]_v \wedge L$$

where x is hidden. $\exists$ is the existential quantification for flexible variables.

3.2 TLA^+

TLA^+ (see for instance [9]) adds structure to TLA. Definitions are organized in predicates, actions, temporal formulas, theorems, etc., and the values of the variables are sets in Zermelo-Fraenkel's set theory [14]. The definitions are grouped in modules which can be imported by other modules. A very simple TLA^+ module is shown in figure 3.

This module defines two variables, x and y . These variables are parameters and might therefore be instantiated by an other module which would import module M . This module also defines a predicate $Init$ and an action A . Finally, the temporal formula $Spec$ sums up the relevant behaviors we want to describe. There is nothing special about the name $Spec$ and actually a module may well contain several (possibly incompatible) temporal formulas. There is nothing to execute in a module. On the contrary, a module must be seen as a convenient way of gathering information about possible executions. (However, if there are theorems and assumptions, the correctness of the module is defined as the validity of the implication of the theorems by the assumptions.) Using the previous module, we can now speak of the set of traces satisfying $Spec$. Properties of $Spec$ can be proven, that is, the validity of expressions such as $Spec \Rightarrow F$, where F is some TLA formula, can be ascertained.

One interesting feature about TLA^+ is that it is a specification language, and therefore convenient for the expression of algorithms, but the transformation of these specifications, viz. the refinement of specifications, can be described within TLA too, as it corresponds to the implication between the TLA formulas. Moreover, refinements, and properties of a specification can be proven using the TLA proof system and rules (described in [8]).

4 Refinement

The correctness is achieved by a stepwise-refinement process. At the beginning, we have an abstract specification with only the minimum informations needed to check that an implementation is correct. Usually, this abstract specification describes the relationship between the input data and the output data. The less

informations we give, the easier it will be to check that an implementation meets the specification, and also the easier it will be to understand the specification.

It is very important that the first specification be short and precise, for this specification is the beginning and there will only be human confidence in it. The first specification must be *obviously correct*.

A TLA^+ specification is describing infinite behaviors. Here, however, we are not interested in the whole behaviors, but only in some features of the behaviors. We are interested in particular phases of the computation. These phases are usually made up of a small set of TLA^+ actions. The initial specification only gives us the minimal informations upon these actions. New specifications will tell us more and show the structure of the actions. In order for the new specification to meet the requirements of the former, we demand that new actions *simulate* the old ones. This simulation is expressed by implication in TLA^+ . For instance, if the first specification has an action a , and if we found a way of satisfying a with a new action b , it is necessary that $b \Rightarrow a$. If such is the case, all occurrences of the old action a in the first specification shall be replaced by the new action b in the new specification, including in fairness conditions, if there are some.

We call such a refinement, where actions are replaced by other more detailed actions, an *action refinement*.

The stepwise refinement process gives us a chain of specifications, as shown in figure 4.

Figure 4: Refinement chain

5 Emphasizing the structure through an OO *notation*: cTLA

Ultimately, our design process should produce real programs, ready to run. However, the question is: what programs should we produce? We have chosen to emphasize a feature of the specification language, namely its modularity. A parallel object-oriented language would seem to be a good choice. There are indeed a certain number of parallel extensions of object-oriented languages such as CC++ [16], pC++ [7] or $\mu\text{C}++$ [1]. In these languages, processes are instances of classes and communication goes through member function (aka methods) calls.

However, our purpose is not — at least not yet — to produce actual C++ code, or code of one of its parallel extension. As a matter of fact, our interest in correctness supersedes the others, and producing correct C++ code would commit us in expressing C++’s semantics within the TLA^+ specification language. We might have decided to chose a subset of C++ and express it within TLA^+ . Yet, this seemed unadapted. We not only wanted a subset of C++, we also wanted to simplify several things. We wanted a language close to the specification language, in order to express its semantics easily. But we also wanted a language which might be transformed in CC++. Hence, we needed an *intermediate language* between TLA^+ and CC++. The purpose of this language was to give an *object-oriented view* of a TLA^+ specification, but also to show what should still be done in order to transform the specification into a real program. The semantics of a subset of a real programming language will anyway have to be expressed in TLA^+ in order to ensure the correctness.

We called our intermediate language cTLA, which stands for “concrete TLA”, but one can also read it “TLA in C”. cTLA is above all an *object-oriented notation*. We will summarize here its main features as well as some less important features, but which are useful for the understanding of the cTLA excerpts which can be found later in the paper. We will mainly explain how to read TLA^+ behind cTLA.

The central idea of cTLA is that it is a language whose control structures are those of CC++, but whose variables can have the same values as in TLA^+ . This abstractness of the data structures really frees us for the expression of the algorithms.

5.1 cTLA's types

All values in cTLA are sets, as in TLA^+ . Sets can be built from other sets with standard operators. For instance, $[S \rightarrow T]$ is the set of all functions whose domain is S and whose codomain is a subset of T . There are some provisions for records, tuples, strings, etc., as well as in TLA^+ .

cTLA is a typed language, but the types are just sets and so they are not essentially different from values. New sets can be defined using the `set` command. For instance, one might write

```
set F = [S -> F];
```

which is really an extension of C's initialization of variables at declaration time (e.g. `int i=0;`).

Types can be parameterized through an accessible parameter. An accessible parameter is either a global variable or a class variable. In figure 8 in the matrix multiplication example, we define a set of square matrices parameterized by the set of values in the matrix.

5.2 Variables and assignments

Variables in cTLA can be local or global. They can be local to a member function, local to a class or global to the program (local to the `main` function). The value of a variable can be any set definable in Zermelo-Fraenkel's set theory, with the addition of the *CHOOSE* operator (Hilbert's ϵ), which we don't use in this paper. Variables can be modified through assignments, as in C. It is required that all assignments in cTLA are *deterministic*.

5.3 Synchronization variables

The idea of synchronization variables is borrowed to C++ [16] and to its forerunner PCN [5]. These variables have initially a special undefined state and can only receive a value once. Since such variables can be shared, this allows for synchronization. For instance, one part (thread) of a program might write

```
a=b;
```

where `b` is initially undefined, and an other part might write

```
b=...;
```

As soon as `b` is defined, the first thread can resume. Synchronization variables are defined using the keyword `sync` and an example appears in figure 30.

5.4 Classes

The main structure in cTLA is the class structure. This structure embodies variables and member functions, as in C++. Member functions can be either private or public, and class variables are always private. They can only be accessed through member functions.

5.5 Member functions

Member functions correspond to TLA^+ actions or groups of TLA^+ actions. A member function is either of type `void` (default type), of a predefined type such as `Nat` or of a type built with `set`. The parameters are passed by value. Local variables to member functions correspond to quantified variables in actions. Member functions can be `atomic`, in which case they cannot be interleaved with other member functions of the same instance of the same class. A member function can not modify variables which are exterior to the class.

5.6 Processes

Each class can have a `Spec` member function, which describes the “proper behavior” of a class instance, that is, the behavior which is not triggered by other processes. The examples of this paper do not have such a function, for the computation are always triggered by an other process.

An instance of a class is a process. Processes can be defined as *groups of actions*. More detailed examples will appear elsewhere. Processes are numbered from 0 on, and created with `newp`. The first argument of `newp` is a constructor of a class (possibly with arguments), and the second is the number assigned to the new process. The `main` function is in process 0. Examples of creation and calls of functions in other processes appear for instance in figure 25.

`last_process()` is a function whose value is the number of a process such that there is no active process with a number equal or greater than the one returned by this function.

5.7 Dropped C(C)++ features

The cTLA notation originated from a simplification of CC++ [16]. Not all features of C++ or CC++ have been kept. Among those that have been dropped, there are pointers (all calls are by value), templates, operator overloading and inheritance. Some of these features might be added in a future version of cTLA, but the current version appears already rich enough for common applications.

6 Semantics of cTLA

The semantics of a cTLA program is defined in that we associate a TLA^+ specification to it. An excerpt of such a specification appears in figure 5. It corresponds to the function shown in figure 21. Several operators used in figure 5 are defined in the module *comdefs* shown in figure 6. In figure 5, the *Restrict* action corresponds to the entrance of the external `parfor`: the program counter is split and the parallelism depth is increased. Action *Restrict2* is the entrance in the inner `parfor` loops. Action *Restrict3* are the assignments. *Restrict4* are the actions leaving the inner `parfor` loop and *Restrict5* are those leaving the outer loop. *Restrict6* is the action returning the result.

The specification adds various variables, such as a program counter (*pc*), local variables (*loc*), function parameters (*var*). The exclusive access of a function is achieved through semaphores (*sem*). The relation between cTLA programs and their equivalent TLA^+ specification is detailed at length in [13].

7 The matrix multiplication example

We will now develop our methodology on a small example, and show in practice how cTLA is handled.

We consider square matrices with a number of rows and columns equal to a power of 2. Such a matrix will be split in four square submatrices (figure 7) and will lead to independent computations which can be done in parallel.

We will first define a specification for the multiplication and eventually derive two cTLA programs, one corresponding to a very tight implementation, and the other to a loose implementation. This vocabulary shall be explained later.

7.1 Initial TLA^+ modules

We do first define a few useful sets (figure 8): the set of positive numbers Nat^+ and the set $sqm[E]$ of square matrices whose size is a power of 2 and whose values are in E . A square matrix is a triple whose first element is a function giving the elements of the matrix. Eventually, this function would probably be mapped (implemented) onto an array. The set of square matrices is parameterized by the set of values in the matrices.

The first description of the multiplication is given in the *Matrix_product* module (figure 9). It uses the *compatible_matrix* predicate and the *DotProd* function. *compatible_matrix* is a predicate verifying that two matrices can be multiplied. It suffices to check that the sizes are the same and that the image sets are

module <i>Matrix_product_ctla1_restrict</i>	
import <i>comdefs, abbreviations</i>	
actions	
<i>Restrict1</i>	$\triangleq \wedge (Pc(restrict, \langle \rangle) = \langle \text{"a"}, 1 \rangle) \wedge Ok(restrict) \wedge (interleave = 1)$ $\wedge (\forall i \in Nat, 0 \leq i < Ar(restrict, \text{"r2"}) - Ar(restrict, \text{"r1"}) + 1 :$ $Pc'(restrict, \langle i \rangle) = \langle \text{"b"}, 2 \rangle)$ $\wedge (Pc'(restrict, \langle \rangle) = \langle \text{"a"}, 2 \rangle) \wedge CO(\{Pc(restrict, \langle i \rangle)\}, v)$
<i>Restrict2(i)</i>	$\triangleq \wedge Pc(restrict, \langle i \rangle) = \langle \text{"b"}, 2 \rangle$ $\wedge (\forall j \in Nat, 0 \leq j < Ar(restrict, \text{"c2"}) - Ar(restrict, \text{"c1"}) + 1 :$ $Pc'(restrict, \langle i, j \rangle) = \langle \text{"c"}, 3 \rangle)$ $\wedge (Pc'(restrict, \langle i \rangle) = \langle \text{"b"}, 3 \rangle) \wedge CO(\{Pc(restrict, \langle i, j \rangle)\}, v)$
<i>Restrict3(i, j)</i>	$\triangleq \wedge (Pc(restrict, \langle i, j \rangle) = \langle \text{"c"}, 3 \rangle) \wedge (Pc'(restrict, \langle i, j \rangle) = \langle \text{"d"}, 3 \rangle)$ $\wedge (loc'[restrict][\text{"f"}][i][j] = (Ar(restrict, \text{"A"})[1])[i] + Ar(restrict, \text{"r1"}) - 1][j + Ar(restrict, \text{"c1"}) - 1])$ $\wedge CO(\{Pc(restrict, \langle i, j \rangle), loc[restrict][\text{"f"}]\}, v)$
<i>Restrict4(i)</i>	$\triangleq \wedge (\forall j \in Nat, 0 \leq j < (Ar(restrict, \text{"c2"}) - Ar(restrict, \text{"c1"}) + 1 : Pc(restrict, \langle i, j \rangle) = \langle \text{"d"}, 3 \rangle)$ $\wedge Pc'(restrict, \langle i \rangle) = \langle \text{"d"}, 2 \rangle$ $\wedge CO(\{Pc(restrict, \langle i \rangle)\}, v)$
<i>Restrict5</i>	$\triangleq \wedge (\forall i \in Nat, 0 \leq i < (Ar(restrict, \text{"r2"}) - Ar(restrict, \text{"r1"}) + 1 : Pc(restrict, \langle i \rangle) = \langle \text{"d"}, 2 \rangle)$ $\wedge Pc'(restrict, \langle \rangle) = \langle \text{"d"}, 1 \rangle$ $\wedge CO(\{Pc(restrict, \langle \rangle)\}, v)$
<i>Restrict6</i>	$\triangleq \wedge Pc(restrict, \langle \rangle) = \langle \text{"d"}, 1 \rangle$ $\wedge Res'(restrict) = \langle loc[restrict][\text{"f"}], Ar(restrict, \text{"r2"}) - Ar(restrict, \text{"r1"}) + 1, Ar(restrict, \text{"A"})[3] \rangle$ $\wedge (Pc'(restrict, \langle \rangle) = \langle \text{"e"}, 1 \rangle) \wedge SetDone(restrict) \wedge CO(\{Pc(restrict, \langle \rangle), Res(restrict), done[this][restrict]\}, v)$
<i>Next_restrict</i>	$\triangleq \vee Restrict1 \vee (\exists i : (i \geq 0) \wedge (i < (Ar(restrict, \text{"c2"}) - Ar(restrict, \text{"c1"}) + 1)) \wedge Restrict2(i))$ $\vee (\exists i, j \in Nat : \wedge (i \geq 0) \wedge (i < (Ar(f, \text{"c2"}) - Ar(f, \text{"c1"}) + 1)) \wedge (j \geq 0)$ $\wedge (j < (Ar(restrict, \text{"r2"}) - Ar(restrict, \text{"r1"}) + 1)) \wedge Restrict3(i, j))$ $\vee (\exists i \in Nat : (i \geq 0) \wedge (i < (Ar(restrict, \text{"c2"}) - Ar(restrict, \text{"c1"}) + 1)) \wedge Restrict4(i))$ $\vee Restrict5 \vee Restrict6$

 Figure 5: Semantics of **restrict** function (see figure 21)

identical. The components of a tuple are accessed through the notation ‘.’. The multiplication is defined using the dot product. The dot product itself is defined recursively using

$$\sum_{k=0}^n a_{ik} \cdot b_{kj} = \left(\sum_{k=0}^{n-1} a_{ik} \cdot b_{kj} \right) + a_{in} \cdot b_{nj}$$

The *Matrix_product* module provides the action *mult_a*¹ which can be used to define more complex specifications. However, this action is not yet ready for the implementation, and especially not for a parallel implementation. Indeed, it tells us nothing on the *order of the operations*.

The *Matrix_product* module shows two related objects: the *mult* function and the *mult_a* action.² The latter is a definition and will be expanded. A definition cannot be recursive. This avoids having actions which are not clearly defined. One might overcome this limitation in that he/she ensures that the definition expansion stops. This would be similar to providing a *measure* function, as is done in the PVS system.[3]

7.2 Refined modules

The *Matrix_product* module is refined into the *Matrix_product2* module (figure 10). Among other things, this new module defines several useful state functions:

¹In order to clearly distinguish the actions from similar TLA⁺ functions or definitions, we give names ending with ‘_a’ to all the actions.

²The function definition uses square brackets [and], whereas the operator definitions (including the actions) use simple parentheses.

```

module comdefs
parameters
    semNP, done, sem, active, var, last_process : VARIABLE

predicate
    Active(proc)  $\triangleq$  active[proc] = "yes"          /* process activity */
    OkC(f, proc)  $\triangleq$  (sem[proc][f] = 0)  $\wedge$  NotDone(f, proc)  $\wedge$  Active(proc)    /* constructor's guard */
    Ok(f)  $\triangleq$  OkC(f, this)          /* guard of an other function */
    Done(f, proc)  $\triangleq$  done[proc][f] = 1          /* guard of an end execution wait */
    NotDone(f, proc)  $\triangleq$  done[proc][f] = 0

actions
    SetDoneC(f, proc)  $\triangleq$  done'[proc][f] = 1
    SetDone(f)  $\triangleq$  SetDoneC(f, this)          /* end of constructor */
    SetNotDone(f, proc)  $\triangleq$  done'[proc][f] = 0    /* green fire for execution */
    P(f, proc, id)  $\triangleq$  /* semaphore */
         $\wedge$  (sem[proc][f][0] > 0)  $\wedge$  (sem'[proc][f][0] = sem[proc][f][0] - 1)
         $\wedge$  SetNotDone(f, proc)  $\wedge$  (sem'[proc][f][1] = id)
    V(f, proc)  $\triangleq$  (sem'[proc][f] = sem[proc][f] + 1)  $\wedge$  Done(f, proc)    /* semaphore */
    PsemNP(id)  $\triangleq$  /* process creation semaphore */
        (semNP[0] > 0)  $\wedge$  (semNP[0]' = semNP[0] - 1)  $\wedge$  (semNP[1]' = id)
    VsemNP  $\triangleq$  semNP' = semNP + 1    /* process creation semaphore */
    SetActive(proc)  $\triangleq$  active'[proc] = "yes"    /* process creation */

transition functions
    Arg(f, proc, n)  $\triangleq$  var[proc][f][n]          /* function argument */
    Result(f, proc)  $\triangleq$  Arg(f, proc, 0)          /* function result */
    Res(f)  $\triangleq$  Result(f, this)
    Ar(f, v)  $\triangleq$  Arg(f, this, v)
    Pc(f, t)  $\triangleq$  pc[this][f][t]
    
```

Figure 6: Definition of operators

$$\left(\frac{A_{11} \mid A_{12}}{A_{21} \mid A_{22}} \right) \times \left(\frac{B_{11} \mid B_{12}}{B_{21} \mid B_{22}} \right) = \left(\frac{A_{11} \times B_{11} + A_{12} \times B_{21} \mid A_{11} \times B_{12} + A_{12} \times B_{22}}{A_{21} \times B_{11} + A_{22} \times B_{21} \mid A_{21} \times B_{12} + A_{22} \times B_{22}} \right)$$

Figure 7: Matrix decomposition in blocks

```

module Matrix
import Real real numbers and operations on them

    Nat+  $\triangleq$  Nat \ {0}
    sqm[E]  $\triangleq$  {⟨f, n, E⟩ : (f ∈ [Nat+ → [Nat+ → E]])  $\wedge$  (n ∈ Nat+)  $\wedge$  (∃ p ∈ Nat : n = 2p)}
```

 Figure 8: Definition of the set of square matrices of 2^p lines.

```

module Matrix_product
import Matrix

parameters
  E : CONSTANT

predicate
  compatible_matrix(A, B)  $\triangleq$ 
     $\wedge (A \in sqm[E]) \vee (B \in sqm[E])$ 
     $\wedge A.2 = B.2$ 
     $\wedge A.3 = Real$ 

state function
  DotProd[i  $\in Nat^+$ ][j  $\in Nat^+$ ][k  $\in Nat^+$ ][A  $\in sqm[E]$ ][B  $\in sqm[E]$ ]  $\triangleq$ 
    if k = 0 then 0
    else DotProd[i][j][k - 1][A][B] + A[i][k] * B[k][j]
  mult[A  $\in sqm[E]$ ][B  $\in sqm[E]$ ]  $\triangleq$  let g[i][j]  $\triangleq$  DotProd[i][j][A.2][A][B]
    in  $\langle g, A.2, A.3 \rangle$ 

actions
  mult_a(A, B, C)  $\triangleq$ 
     $\wedge compatible\_matrix(A, B)$ 
     $\wedge C' = mult[A][B]$ 
    
```

 Figure 9: *Matrix_product* module

- the *add* function computes the sum of two compatible square matrices; this is merely defined as being the matrix whose first component is the sum of the first components of the two matrices;
- the *restrict* function returns a sub-block of a given matrix, from lines r_1 to r_2 and from column c_1 to column c_2 ;
- the *build* function takes four blocks and gathers them in order to build a square matrix twice as big.

All these operations consist merely in the definition of the appropriate function giving the elements. For instance, the *restrict* function does essentially *shift* the function giving the matrix elements.

The *mult* function is then redefined in order to use the previous definitions to express the matrix multiplication as a composition of multiplications of smaller matrices. The eight submatrices are defined, and they are rearranged according to the formula given in figure 7. The *mult_a* action is the operational equivalent of the *mult* function. However, it does only express the *instantaneous* computation of the product, and the assignment of its result to the new value of *C*.

Comparing to the *Matrix_product* module, all what we have done was to give an equivalent but more detailed definition of the *mult* function. The new *mult* definition is recursive and corresponds to the block decomposition of the matrix.

The next step is the splitting of the *mult_a* action in order to *mirror* the splitting of the *mult* function. That is, we want to provide a *time structure* to the multiplication, or in other words, we want the multiplication to be performed by TLA^+ actions and not by functions.

The *Matrix_product2* module is therefore refined into the *Matrix_product3* module (figure 11). Local variables are introduced in the *mult_a* action, in that they are quantified. This step shows the equivalence in TLA^+ between the **LET** construction and the existential quantification. The multiplication is however still performed in a flat way. That is, the multiplication is still defined only by functions.

The *Matrix_product3* module is then refined into the *Matrix_product4* module (figure 12) which contains merely new actions. The *mult_a* action has not changed and only *action-equivalents* of the *add*, *build* and *restrict* functions are introduced. The functions *add*, *build* and *restrict* are still needed in the definition of the *mult_a* action.

The next refinement, from module *Matrix_product4* to module *Matrix_product5* (figure 14) is the first to introduce *action structure* in the *mult_a* action. First, the new actions *add_a*, *restrict_a* and *build_a* are

```

module Matrix_product2
import Matrix

parameters
    E : CONSTANT

...

state function
    add[A, B]  $\triangleq$  let s[i][j]  $\triangleq$  (A.1)[i][j] + (B.1)[i][j]
                     in  $\langle s, A.2, A.3 \rangle$ 

    restrict[A, r1, r2, c1, c2]  $\triangleq$  let f[i][j]  $\triangleq$  (A.1)[i + r1 - 1][j + c1 - 1]
                     in  $\langle f, r_2 - r_1 + 1, A.3 \rangle$ 

    build[A, B, C, D]  $\triangleq$ 
        let f[i][j]  $\triangleq$  if i ≤ A.2 then if j ≤ A.2 then (A.1)[i][j]
                               else (B.1)[i][j - A.2 + 1]
                               else if j ≤ A.2 then (C.1)[i - A.2 + 1][j]
                               else (D.1)[i - A.2 + 1][j - A.2 + 1]

        in  $\langle f, 2 * A.2, A.3 \rangle$ 

    mult[A ∈ sqm[E]][B ∈ sqm[E]]  $\triangleq$ 
        if A.2 = B.2 = 1
        then let f[i][j]  $\triangleq$  ((A.1)[1][1]) * ((B.1)[1][1])
             in  $\langle f, 1, A.3 \rangle$ 
        else let A11  $\triangleq$  restrict[A, 1, (A.2)/2, 1, (A.2)/2]
                A12  $\triangleq$  restrict[A, 1, (A.2)/2, (A.2)/2 + 1, A.2]
                A21  $\triangleq$  restrict[A, (A.2)/2 + 1, A.2, 1, (A.2)/2]
                A22  $\triangleq$  restrict[A, (A.2)/2 + 1, A.2, (A.2)/2 + 1, A.2]
                B11  $\triangleq$  restrict[B, 1, (B.2)/2, 1, (B.2)/2]
                B12  $\triangleq$  restrict[B, 1, (B.2)/2, (B.2)/2 + 1, B.2]
                B21  $\triangleq$  restrict[B, (B.2)/2 + 1, B.2, 1, (B.2)/2]
                B22  $\triangleq$  restrict[B, (B.2)/2 + 1, B.2, (B.2)/2 + 1, B.2]
        in build[add[mult[A11][B11], mult[A12][B21]],
                add[mult[A11][B12], mult[A12][B22]],
                add[mult[A21][B11], mult[A22][B21]],
                add[mult[A21][B12], mult[A22][B22]]]

actions
    mult_a(A, B, C)  $\triangleq$   $\wedge$  compatible_matrix(A, B)
                         $\wedge$  C' = mult[A][B]
    
```

 Figure 10: *Matrix_product2* module

introduced in replacement of the function calls *add*, *restrict*, *build*. However, whereas the function calls didn't need a special ordering in the *Matrix_product4* module, since everything was flat, it is now different. There is now a notion of ordering. The ordering is a consequence of the dependencies in the *Matrix_product4* module. The dependencies are summed up in figure 13.

So, we must express that the *restrict_a* actions occur first, then that the *mult* functions are called (we avoid recursion up to now), then that the *add_a* actions and eventually the *build* function are called.

Sequentiality can be expressed in TLA⁺ by means of ancillary variables and guards. We can also impose a very tight structure, using the action composition operator '•'. Basically, if *A* and *B* are actions, *A* • *B* is a new action, whose meaning is the same as if *A* and *B* had occurred in sequence. The precise definition is the following:

Definition (Action composition) *If* $x_1, \dots, x_n$ *are all the free variables occurring primed in* *A* *or non-primed in* *B*, *then* $A \bullet B$ *is equal to* $\exists \$x_1, \dots, \$x_n : \hat{A} \wedge \tilde{B}$, *where* $\hat{A}$ *is* *A* *with each* x'_i *replaced by* $\$x_i$, *and* $\tilde{B}$ *is* *B* *with each* x_i *replaced by* $\$x_i$.

```

module Matrix_product3
...
actions
  mult_a(A, B, C)  $\triangleq$ 
     $\wedge$  compatible_matrix(A, B)
     $\wedge \vee (A.2 = B.2 = 1) \wedge (\exists f : \wedge f[i][j] = ((A.1)[1][1]) * ((B.1)[1][1])$ 
       $\wedge C' = \langle f, 1, A.3 \rangle)$ 
     $\vee \exists A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, B_{12}, B_{21}, B_{22},$ 
       $M_{1111}, M_{1221}, M_{1112}, M_{1222}, M_{2111}, M_{2221}, M_{2112}, M_{2222},$ 
       $AA_1, AA_2, AA_3, AA_4 :$ 
       $\wedge A_{11} = \text{restrict}[A, 1, (A.2)/2, 1, (A.2)/2]$ 
       $\wedge A_{12} = \text{restrict}[A, 1, (A.2)/2, (A.2)/2 + 1, A.2]$ 
       $\wedge A_{21} = \text{restrict}[A, (A.2)/2 + 1, A.2, 1, (A.2)/2]$ 
       $\wedge A_{22} = \text{restrict}[A, (A.2)/2 + 1, A.2, (A.2)/2 + 1, A.2]$ 
       $\wedge B_{11} = \text{restrict}[B, 1, (B.2)/2, 1, (B.2)/2]$ 
       $\wedge B_{12} = \text{restrict}[B, 1, (B.2)/2, (B.2)/2 + 1, B.2]$ 
       $\wedge B_{21} = \text{restrict}[B, (B.2)/2 + 1, B.2, 1, (B.2)/2]$ 
       $\wedge B_{22} = \text{restrict}[B, (B.2)/2 + 1, B.2, (B.2)/2 + 1, B.2]$ 
       $\wedge (M_{1111} = \text{mult}[A_{11}][B_{11}]) \wedge (M_{1221} = \text{mult}[A_{12}][B_{21}])$ 
       $\wedge (M_{1112} = \text{mult}[A_{11}][B_{12}]) \wedge (M_{1222} = \text{mult}[A_{12}][B_{22}])$ 
       $\wedge (M_{2111} = \text{mult}[A_{21}][B_{11}]) \wedge (M_{2221} = \text{mult}[A_{22}][B_{21}])$ 
       $\wedge (M_{2112} = \text{mult}[A_{21}][B_{12}]) \wedge (M_{2222} = \text{mult}[A_{22}][B_{22}])$ 
       $\wedge (AA_1 = \text{add}[M_{1111}, M_{1221}]) \wedge (AA_2 = \text{add}[M_{1112}, M_{1222}])$ 
       $\wedge (AA_3 = \text{add}[M_{2111}, M_{2221}]) \wedge (AA_4 = \text{add}[M_{2112}, M_{2222}])$ 
       $\wedge C' = \text{build}[AA_1, AA_2, AA_3, AA_4]$ 

```

 Figure 11: *Matrix_product3* module

```

module Matrix_product4
...
actions
  add_a(A, B, C)  $\triangleq$   $\wedge$  compatible_matrix(A, B)
     $\wedge \exists s : ((s[i][j] = (A.1)[i][j] + (B.1)[i][j]) \wedge (C' = \langle s, A.2, A.3 \rangle))$ 
  restrict_a(A, r1, r2, c1, c2, C)  $\triangleq$   $\exists f : \wedge f[i][j] = (A.1)[i + r_1 - 1][j + c_1 - 1]$ 
     $\wedge C' = \langle f, r_2 - r_1 + 1, A.3 \rangle$ 
  build_a(A, B, C, D, M)  $\triangleq$ 
    let f[i][j]  $\triangleq$  if i ≤ A.2 then if j ≤ A.2 then (A.1)[i][j]
      else (B.1)[i][j - A.2 + 1]
      else if j ≤ A.2 then (C.1)[i - A.2 + 1][j]
      else (D.1)[i - A.2 + 1][j - A.2 + 1]
    in M' =  $\langle f, 2 * A.2, A.3 \rangle$ 
  mult_a(A, B, C)  $\triangleq$  ...

```

 Figure 12: *Matrix_product4* module

For instance, $(x' = x + 1) \bullet (x' = x + 1) \triangleq \exists \$x : (\$x = x + 1) \wedge (x' = \$x + 1)$. Hence, $(x' = x + 1) \bullet (x' = x + 1) \Rightarrow (x' = x + 2)$.

The definition of *mult_a* in the *Matrix_product5* module implies clearly the definition of *mult_a* in the *Matrix_product4* module, hence, is a refinement of it. The values of *A*, *B*, ..., may change, but the result *C* is the same, and this is what we seek. The ‘•’ operation can be seen as a way to guarantee the dependencies.

Now, if one stares at the *Matrix_product5* module, it becomes clear that the call to *mult* should be replaced by a call to *mult_a*. However, a problem arises here: in TLA^+ , it is not possible to have recursive definitions, and hence no recursive actions. Recursive actions are forbidden in TLA^+ because there is no guaranty that their expansion will terminate. If we can guarantee that the expansion terminates, and that the call to an action is equivalent to a similar call to a function (recursive functions are allowed in TLA^+),

 Figure 13: Dependences in the *mult_a* action of module *Matrix_product4*

 Figure 14: *Matrix_product5* module

then we can use a call to an action as a shortcut.

The splitting of the *mult_a* action in similar pieces, occurring in fractions of TLA⁺ steps bears of course similarities to a fractal approach.

Before reaching that step, we can however break the specification a little better, and this is shown in module *Matrix_product6* (figure 15).

```

----- module Matrix_product6 -----
...
actions
  add_a(A, B, C)  $\triangleq$  ...
  restrict_a(A, r1, r2, c1, c2, C)  $\triangleq$  ...
  build_a(A, B, C, D, M)  $\triangleq$  ...
  rest_a(A, B, A11, A12, A21, A22, B11, B12, B21, B22)  $\triangleq$ 
     $\wedge$  restrict_a(A, 1, (A.2)/2, 1, (A.2)/2, A11)
     $\wedge$  restrict_a(A, 1, (A.2)/2, (A.2)/2 + 1, A.2, A12)
     $\wedge$  restrict_a(A, (A.2)/2 + 1, A.2, 1, (A.2)/2, A21)
     $\wedge$  restrict_a(A, (A.2)/2 + 1, A.2, (A.2)/2 + 1, A.2, A22)
     $\wedge$  restrict_a(B, 1, (B.2)/2, 1, (B.2)/2, B11)
     $\wedge$  restrict_a(B, 1, (B.2)/2, (B.2)/2 + 1, B.2, B12)
     $\wedge$  restrict_a(B, (B.2)/2 + 1, B.2, 1, (B.2)/2, B21)
     $\wedge$  restrict_a(B, (B.2)/2 + 1, B.2, (B.2)/2 + 1, B.2, B22)
  mult_blocks_a(A11, A12, A21, A22, B11, B12, B21, B22, M1, M2, M3, M4, M5, M6, M7, M8)  $\triangleq$ 
     $\wedge$  (M'_1 = mult[A11][B11])  $\wedge$  (M'_2 = mult[A12][B21])
     $\wedge$  (M'_3 = mult[A11][B12])  $\wedge$  (M'_4 = mult[A12][B22])
     $\wedge$  (M'_5 = mult[A21][B11])  $\wedge$  (M'_6 = mult[A22][B21])
     $\wedge$  (M'_7 = mult[A21][B12])  $\wedge$  (M'_8 = mult[A22][B22])
  add_blocks_a(M1, M2, M3, M4, M5, M6, M7, M8, R1, R2, R3, R4)  $\triangleq$ 
     $\wedge$  (add_a(M1, M2, R1))  $\wedge$  (add_a(M3, M4, R2))
     $\wedge$  (add_a(M5, M6, R3))  $\wedge$  (add_a(M7, M8, R4))
  mult_one_one_a(A, B, C)  $\triangleq$ 
     $(\exists f : \wedge f[i][j] = ((A.1)[1][1]) * ((B.1)[1][1])$ 
     $\wedge C' = \langle f, 1, A.3 \rangle)$ 
  mult_a(A, B, C)  $\triangleq$ 
     $\wedge$  compatible_matrix(A, B)
     $\wedge \vee (A.2 = B.2 = 1) \wedge$  mult_one_one_a(A, B, C)
     $\vee \exists A11, A12, A21, A22, B11, B12, B21, B22,$ 
      M1, M2, M3, M4, M5, M6, M7, M8, R1, R2, R3, R4 :
      

- rest_a(A, B, A11, A12, A21, A22, B11, B12, B21, B22)
- mult_blocks_a(A11, A12, A21, A22, B11, B12, B21, B22, M1, M2, M3, M4, M5, M6, M7, M8)
- add_blocks_a(M1, M2, M3, M4, M5, M6, M7, M8, R1, R2, R3, R4)
- build_a(R1, R2, R3, R4, C)

```

Figure 15: *Matrix_product6* module

There, we would of course like to write instead

```

mult_blocks_a(A11, A12, A21, A22, B11, B12, B21, B22, M1, M2, M3, M4, M5, M6, M7, M8)  $\triangleq$ 
   $\wedge$  mult_a(A11, B11, M1)  $\wedge$  mult_a(A12, B21, M2)
   $\wedge$  mult_a(A11, B12, M3)  $\wedge$  mult_a(A12, B22, M4)
   $\wedge$  mult_a(A21, B11, M5)  $\wedge$  mult_a(A22, B21, M6)
   $\wedge$  mult_a(A21, B12, M7)  $\wedge$  mult_a(A22, B22, M8)

```

Once the definitions are written that way, it is straightforward to convert them into cTLA. This will be shown in the next sections.

8 cTLA code

8.1 Introduction to the two proposed implementations

The analysis of the block decomposition for matrix multiplication shows many independent parts. This is best emphasized by the dependencies diagram of figure 13. For instance, the computation of matrix M_{1111} from the matrices A_{11} and B_{11} is independent from the computation of matrix M_{2222} from matrices A_{22} and B_{22} . In reality, for various reasons, one computation might be slower than the other one.

The first implementation that we propose ignores this fact and first computes all the restrictions (A_{11} , A_{12} , etc.), then all the multiplications of submatrices, then all the additions, then builds the result. The implementation is a systolic one, following a wave.

The second implementation does not impose such rendez-vous, and frees each thread. The dependencies must however be preserved and this is done through synchronization variables.

8.2 cTLA code: a tight implementation

We now want to refine the *Matrix_product6* module (figure 15) into a cTLA class. First, here is an elementary implementation of the Nat^+ type:

```
set Natp = {x :in: Nat : x > 0}
```

`Natp` is the set of all positive integers. This corresponds to the Nat^+ definition in module *Matrix*. Figure 16 shows the definition of the parameterized type `sqm`.

```
set sqm = [E :in: {Nat,Real} |-> {<<f,n,E>> : (f :in: [Natp -> [Natp -> E]])
                                     /\ (n :in: Natp) /\ (:E: p :in: Nat : n=2^p)}]];
```

Figure 16: `sqm[E]` set

```
set N3 = [Nat -> [Nat -> Nat]];
```

Figure 17: `N3` set

```
set tupsqm=<<E,n>> :in: {Nat,Real} :times: Nat
|-> [{i : (i :in: Nat) /\ (i>0) /\ (i :leq: n)} -> sqm[E]]];
```

Figure 18: `tupsqm[E,n]` set

All the TLA^+ definitions seen up to now are only type definitions and had a simple correspondence in $cTLA^3$. The *Matrix_product6* module is however different in that it contains actions. Such a module will correspond to a cTLA class. This class is sketched in figure 19 and following. We will comment each of the components of the class.

All the member functions relevant to the matrix multiplication will be grouped in a cTLA class (figure 19). These are all the functions shown in the *Matrix_product6* module. In addition to these functions, the class contains a variable `e` which is the set of possible values in the matrices. Some of the member functions are `private`, the others are `public`. A `private` member function cannot be called from outside its class.

The correspondence of the cTLA functions `add`, `restrict`, `build`, `rest`, `add_blocks`, `mult_blocks` and `mult_one_one` to the definitions in module *Matrix_product6* is straightforward. Some of the initial assignments are now done with `parfor` which does initiate a parallel loop. A `parfor` loop terminates when *all* its

³Because cTLA data structures are essentially those of TLA^+ .


```

class Matrix_product_ctla1
{
private:
    set e;
public:
    void Matrix_product_ctla1(set E){e=E}          // constructor
    int compatible_matrix(sqm[e],sqm[e]);
    sqm[e] add(sqm[e],sqm[e]);                      // prototypes
    sqm[e] restrict(sqm[e],Nat,Nat,Nat,Nat);
    sqm[e] build(sqm[e],sqm[e],sqm[e],sqm[e]);
    tupsqm[e,8] rest(sqm[e] A,sqm[e] B);
    tupsqm[e,8] mult_blocks(sqm[e],sqm[e],sqm[e],sqm[e],
                           sqm[e],sqm[e],sqm[e],sqm[e]);
    tupsqm[e,4] add_blocks(sqm[e],sqm[e],sqm[e],sqm[e],
                           sqm[e],sqm[e],sqm[e],sqm[e]);
    sqm[e] mult_one_one(sqm[e],sqm[e]);
    sqm[e] mult(sqm[e],sqm[e]);
}
    
```

Figure 19: Matrix_product class

threads have completed. Similarly, a `par` loop finishes when all its threads have completed. The difference between a `par` and a `parfor` loop are that the `parfor` construction defines a local constant variable for each of its threads.

The proof of the correspondance involves proving that the TLA^+ actions corresponding to a `cTLA` function have the same net effect than the action in the *Matrix_product6* module.

```

sqm[e] Matrix_product_ctla1::add(sqm[e] A,sqm[e] B)
{
    N3 s;
    if (compatible_matrix(A,B))
    {
        parfor(Nat i=0; i<A[2];i++)
            parfor(Nat j=0; j<A[2];j++)
                s[i][j] = (A[1])[i][j] + (B[1])[i][j];
        return <<s,A[2],A[3]>>;
    }
}
    
```

Figure 20: Matrix_product class, add function

```

sqm[e] Matrix_product_ctla1::restrict(sqm[e] A,Nat r1,Nat r2,Nat c1,Nat c2)
{
    N3 f;
    parfor(Nat i=0;i<r2-r1+1;i++)
        parfor(Nat j=0;j<c2-c1+1;j++)
            f[i][j] = (A[1])[i+r1-1][j+c1-1];
    return <<f,r2-r1+1,A[3]>>;
}
    
```

Figure 21: Matrix_product class, restrict function

The computation of a matrix product is shown on figure 28. *A* and *B* are supposed to be two matrix variables, of type `sqm[E]`.

First, a matrix variable *C* is defined, of the same type as *A* and *B*. Then, an instance of the class `Matrix_product_ctla1` is defined as process 1. This instance is initialized with the set *E*. The multiplication of *A* and *B* is then simply performed by the call of the `mult` function of process 1, with arguments *A*

```

sqm[e] Matrix_product_ctla1::build(sqm[e] A,sqm[e] B,sqm[e] C,sqm[e] D)
{
    N3 f;
    parfor(Nat i=0;i<2*A[2];i++)
        parfor(Nat j=0;j<2*A[2];j++)
            {
                if (i <= A[2]){
                    if (j <= A[2])
                        f[i][j] = (A[1])[i][j];
                    else
                        f[i][j] = (B[1])[i][j-A[2]+1];
                }
                else{
                    if (j <= A[2])
                        f[i][j] = (C[1])[i-A[2]+1][j];
                    else
                        f[i][j] = (D[1])[i-A[2]+1][j-A[2]+1];
                }
            }
    return <<f,2*A[2],A[3]>>
};

```

Figure 22: Matrix_product class, build function

```

tupsqm[e,8] Matrix_product_ctla1::rest(sqm[e] A,sqm[e] B)
{
    sqm[e] A11,A12,A21,A22,B11,B12,B21,B22;
    int lp;
    lp=last_process();
    newp(Matrix_product_ctla1(e),lp+1);
    ...
    newp(Matrix_product_ctla1(e),lp+8);
    par {
        A11=process(lp+1).restrict(A,1,(A[2])/2,1,(A[2])/2);
        ...
        B22=process(lp+8).restrict(B,(B[2])/2+1,B[2],(B[2])/2+1,B[2])
    }
    return <<A11,A12,A21,A22,B11,B12,B21,B22>>;
};

```

Figure 23: Matrix_product class, rest function

and B. The result is put in C.

8.3 cTLA code : a loose implementation

The previous implementation remained very symmetrical. We did not allow ourselves to take advantage of the slowness of some computations. But there are obviously many non-necessary meeting points. For instance, this implementation requires that *no* block be multiplied before *all* **restrict** operations are completed.

An other approach is to let the final compiler worry itself about all of this, and to implement the dependencies only by means of synchronization variables **sync**. Figure 29 shows the **Matrix_product** class, its variables and its member functions.

Figure 30 shows the main **mult** function. The difference with the previous cTLA implementation is the organization of the operations. Actually, the multiplication algorithm remains the same. However, all the actions are flattened. There is no longer an obvious ordering between the function calls. What is making all the difference? It is the fact that the function calls get ordered by themselves because of the use of synchronization variables. It is assumed that A and B are two defined variables at the beginning of the **mult** function call. Each of the **restrict** function calls puts its result in A11, A12, etc. These variables, as well as M1, M2, etc., are initially undefined. Hence, the function call **mult**(A11,B11) will suspend if either A11 or

```

tupsqm[e,4] Matrix_product_ctla1::add_blocks(sqm[e] M1,sqm[e] M2,sqm[e] M3,sqm[e] M4,
                                             sqm[e] M5,sqm[e] M6,sqm[e] M7,sqm[e] M8)
{
    sqm[e] R1,R2,R3,R4;
    int lp;
    lp=last_process();
    newp(Matrix_product_ctla1(e),lp+1);newp(Matrix_product_ctla1(e),lp+2);
    newp(Matrix_product_ctla1(e),lp+3);newp(Matrix_product_ctla1(e),lp+4);
    par {
        R1=process(lp+1).add(M1,M2);
        R2=process(lp+2).add(M3,M4);
        R3=process(lp+3).add(M5,M6);
        R4=process(lp+4).add(M7,M8);
    }
    return <<R1,R2,R3,R4>>;
};
    
```

Figure 24: Matrix_product class, block addition functions

```

tupsqm[e,8] Matrix_product_ctla1::mult_blocks(sqm[e] A11,sqm[e] A12,sqm[e] A21,sqm[e] A22,
                                              sqm[e] B11,sqm[e] B12,sqm[e] B21,sqm[e] B22)
{
    sqm[e] M1,M2,M3,M4,M5,M6,M7,M8;
    int lp;
    lp=last_process();
    newp(Matrix_product_ctla1(e),lp+1);
    ...
    newp(Matrix_product_ctla1(e),lp+8);
    par {
        M1=process(lp+1).mult(A11,B11);
        ...
        M8=process(lp+8).mult(A22,B22);
    }
    return <<M1,M2,M3,M4,M5,M6,M7,M8>>;
};
    
```

Figure 25: Matrix_product class, block multiplication functions

```

sqm[e] Matrix_product_ctla1::mult_one_one(sqm[e] A,sqm[e] B)
{
    N3 f;
    f[1][1] = ((A[1])[1][1])*((B[1])[1][1]);
    return <<f,1,A[3]>>;
};
    
```

Figure 26: Matrix_product class, degenerated case

B11 have not yet been extracted. The dependencies are therefore still there, but the use of synchronization variables makes it much easier to handle threads with different speeds.

9 Proof of the implementation

Figure 31 shows how cTLA is related to the other specifications: through a refinement process, we arrive at the specification shown left. This specification is transformed through a mapping into a cTLA program. This mapping is not (yet) proved. The cTLA program is actually a *different view* on the TLA⁺ specification expressing its semantics, and shown at the bottom. This TLA⁺ specification is obtained mechanically from the cTLA program. Proving the implementation is therefore proving that the bottom specification implies

```

sqm[e] Matrix_product_ctla1::mult (sqm[e] A, sqm[e] B)
{
    tupsqm[e,8] R,M;
    tupsqm[e,4] S;
    if (compatible_matrix(A,B))
    {
        if (A[2]==B[2]==1)
            return mult_one_one(A,B);
        else
        {
            R=rest(A,B);
            M=mult_blocks(R.1,R.2,R.3,R.4,R.5,R.6,R.7,R.8);
            S=add_blocks(M.1,M.2,M.3,M.4,M.5,M.6,M.7,M.8);
            return build(S.1,S.2,S.3,S.4);
        }
    }
    else
        return :emptyset;;
};

```

Figure 27: Matrix_product class, mult function

```

main()
{
    set E=Nat;
    sqm[E] A,B,C;
    ...           /* filling of A and B */
    newp(Matrix_product_ctla1(E),1);
    C=process(1).mult(A,B);
}

```

Figure 28: Matrix_product call

```

class Matrix_product_ctla2
{
private:
    set e;
public:
    void Matrix_product_ctla2(set E){e=E} // constructeur
    int compatible_matrix(sqm[e],sqm[e]);
    sqm[e] add(sqm[e],sqm[e]);           // prototypes
    sqm[e] restrict(sqm[e],Nat,Nat,Nat,Nat);
    sqm[e] build(sqm[e],sqm[e],sqm[e],sqm[e]);
    sqm[e] mult (sqm[e],sqm[e]);
}

```

Figure 29: Matrix_product class: loose implementation

the left specification. This should be done for a number of patterns in the initial specification, so that the mappings don't have to be rechecked each time.

10 Conclusion

We have shown in this paper a methodology which appears very interesting and fruitful when combined with an abstract specification language such as TLA^+ . The refinements were carried on, and the algorithm was introduced in TLA^+ . This was done to such an extent that it was easily possible to rewrite the result in a C++-like way. Not only does the result bear similarities with C++, it does also intuitively behave naturally.

```

sqm[e] Matrix_product_ctla2::mult(sqm[e] A,sqm[e] B)
{
    sync sqm[e] A11,A12,A21,A22,B11,B12,B21,B22,M1,M2,M3,M4,M5,M6,M7,M8,
                AA1,AA2,AA3,AA4;

    N3 f;
    int lp;
    lp=last_process();
    if (compatible_matrix(A,B)){
    if (A[2]==B[2]==1){
        f[1][1] = ((A[1])[1][1])*((B[1])[1][1]);
        return <<f,1,A[3]>>;
    }
    else{
        par{
            newp(matrix_product_ctla2(e),lp+1);
            ...
            newp(matrix_product_ctla2(e),lp+20);
            A11=process(lp+1).restrict(A,1,(A[2])/2,1,(A[2])/2);
            ...
            B22=process(lp+8).restrict(B,(B[2])/2+1,B[2],(B[2])/2+1,B[2]);
            M1=process(lp+9).mult(A11,B11);
            ...
            M8=process(lp+16).mult(A22,B22);
            AA1=process(lp+17).add(M1,M2);
            ...
            AA4=process(lp+20).add(M7,M8);
            return build(AA1,AA2,AA3,AA4);
        }
    }
}
    
```

Figure 30: Matrix_product class: loose implementation, multiplication

Figure 31: Proof of the implementation

However, cTLA is not C++, nor one of its concurrent extensions. There is still some work in order to get real parallel programs, but we believe that this intermediate state shows that it is possible to come easily close to a real language and still have a well-defined semantics.

References

- [1] Peter A. Buhr and Richard A. Strooboscher. *μC++ Annotated Reference Manual*, 1995. Version 4.2.
- [2] K. Mani Chandy and Carl Kesselman. CC++: A Declarative Concurrent Object Oriented Programming Notation, 1993.

- [3] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A Tutorial Introduction to PVS, June 1995. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995.
- [4] Richard P. Feynman, Robert B. Leighton, and Matthew L. Sands. *The Feynman lectures on physics*. Addison-Wesley, 1963–65.
- [5] I. Foster and S. Tuecke. Parallel Programming with PCN. Technical report, Argonne National Laboratory, University of Chicago, January 1993. Version 2.0.
- [6] Ian T. Foster. *Designing and Building Parallel Programs – Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Publishing Company, 1995.
- [7] Dennis Gannon, Shelby X. Yang, and Peter Beckman. User Guide for a Portable Parallel C++ Programming System, pC++. Technical report, Department of Computer Science & CICA, Indiana University, Bloomington, Indiana, USA, September 1994.
- [8] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
- [9] Leslie Lamport. TLA⁺. Available at <http://www.research.digital.com/SRC/tla/tla.html>, August 1994.
- [10] Leslie Lamport. TLA WWW page: <http://www.research.digital.com/SRC/personal/LeslieLamport/tla/tla.html>, 1995.
- [11] Leslie Lamport and Stephan Merz. Specifying and Verifying Fault-Tolerant Systems. In *Proceedings of the Third International Symposium on Formal Techniques in Real Time and Fault Tolerant Systems, 19–23 September 1994, Lübeck, Germany*, 1994.
- [12] Bertrand Meyer. Systematic Concurrent Object-Oriented Programming. *Communications of the ACM*, 36(9):56–80, 1993.
- [13] Denis Roegel. *Étude de la sémantique de programmes parallèles « réels » en TLA*. PhD thesis, Université Henri Poincaré, Nancy, France, 1996.
- [14] J. R. Shoenfield. Axioms of Set Theory. In Jon Barwise, editor, *Handbook of Mathematical Logic*, chapter B.1, pages 321–344. North-Holland Publishing Company, 1977.
- [15] Paolo A. G. Sivilotti. A Verified Integration of Parallel Programming Paradigms in CC++. In H.J. Siegel, editor, *8th International Parallel Processing Symposium (IPPS 94), Cancun*, pages 44–50. IEEE, April 1994.
- [16] Paul A. G. Sivilotti and Peter A. Carlin. A Tutorial for CC++. Technical Report Caltech CS-TR-94-02, Compositional Systems Research Group, Department of Computer Science, California Institute of Technology, Caltech Mail Stop 256-80, Pasadena, California 91125, 1994.