Separating Algorithm and Implementation in the Refinement of Parallel Program Specifications
Denis Roegel
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 into 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 combinatorial 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](image)

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.
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

\[ \sigma = \sigma_0, \sigma_1, \sigma_2, \ldots \]

where \( \sigma_i \) is the state \( i \), and a TLA formula \( F \triangleq \text{Init} \land [\text{Next}]_v \land L \), we say that \( \sigma \) satisfies \( F \), and we write

\[ \sigma \models F \]

(1) if the initial state \( \sigma_0 \) satisfies the predicate \( \text{Init} \), (2) if each pair of consecutive states \( \sigma_i, \sigma_{i+1} \) is either a \textit{stuttering step}, viz. no variables in \( v \) are changed, or is a step satisfying the relation \( \text{Next} \) 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 \textit{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:

<table>
<thead>
<tr>
<th>Action</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x'=x )</td>
<td>( x ) is unchanged</td>
</tr>
<tr>
<td>( x'=x+1 )</td>
<td>( x ) gets increased by 1</td>
</tr>
<tr>
<td>( x'+y'=x+y )</td>
<td>the sum ( x+y ) remains constant</td>
</tr>
<tr>
<td>( ((x'=x+1) \land (y'=y)) \lor ((y'=y+1) \land (x'=x)) )</td>
<td>either increase ( x ) by 1 and keep ( y ) unchanged, or do the opposite.</td>
</tr>
</tbody>
</table>

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...
Separating Algorithm and Implementation in the Refinement of Parallel Program Specifications

---

module M

parameters
  x, y : VARIABLE

predicate
  Init ≡ x > 0

action
  A ≡ (x > y) ∧ (x' = x − y)

temporal
  Spec ≡ Init ∧ □[A]_x ∧ WF_x(A)

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

∃ x : Init ∧ [Next]_x ∧ L

where x is hidden. ∃ 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 ⇒ 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.

\[
\Pi_0 \xrightarrow{\text{implies}} \Pi_1 \xrightarrow{\text{is refined in}} \cdots \xrightarrow{\text{cTLA}} \text{"Program"}
\]

*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 \( \mu \)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 C++, 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 \texttt{set} command. For instance, one might write

\[
\texttt{set } F = [S \rightarrow F];
\]

which is really an extension of C’s initialization of variables at declaration time (e.g. \texttt{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 \texttt{main} function). The value of a variable can be any set definable in Zermelo-Fraenkel’s set theory, with the addition of the \texttt{CHOOSE} operator (Hilbert’s \(\varepsilon\)), 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 \textit{deterministic}.

5.3 Synchronization variables

The idea of synchronization variables is borrowed to CC++ [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

\[
b=\ldots;
\]

As soon as \(b\) is defined, the first thread can resume. Synchronization variables are defined using the keyword \texttt{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 \texttt{void} (default type), of a predefined type such as \texttt{Nat} or of a type built with \texttt{set}. The parameters are passed by value. Local variables to member functions correspond to quantified variables in actions. Member functions can be \texttt{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 an 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 sfm[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
identical. The components of a tuple are accessed through the notation \( (\cdot) \). 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 \textit{Matrix.product} module provides the action \texttt{mult.a} 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 \textit{order of the operations}.

The \textit{Matrix.product} module shows two related objects: the \texttt{mult} function and the \texttt{mult.a} action.\footnote{In order to clearly distinguish the actions from similar TLA+ functions or definitions, we give names ending with \texttt{a} to all the actions.} 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 \textit{measure} function, as is done in the PVS system.\footnote{The function definition uses square brackets \texttt{[]} and \texttt{1}, whereas the operator definitions (including the actions) use simple parentheses.}

\section{7.2 Refined modules}

The \textit{Matrix.product} module is refined into the \textit{Matrix.product2} module (figure 10). Among other things, this new module defines several useful state functions:

\begin{itemize}
  \item \texttt{restrict} (see figure 21)
\end{itemize}
module comdefs

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

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

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

transition functions
  Arg[f, proc, n] ≜ var[proc][f][n] /* function argument */
  Result[f, proc] ≜ Arg[f, proc, 0] /* function result */
  Res[f] ≜ Result[f, this]
  Ar[f, v] ≜ Arg[f, this, v]
  Pc[f, t] ≜ pc[this][f][t]

Figure 6: Definition of operators

\[
\begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix}
\times
\begin{pmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{pmatrix}
= \begin{pmatrix}
A_{11} \times B_{11} + A_{12} \times B_{21} & A_{11} \times B_{12} + A_{12} \times B_{22} \\
A_{21} \times B_{11} + A_{22} \times B_{21} & A_{21} \times B_{12} + A_{22} \times B_{22}
\end{pmatrix}
\]

Figure 7: Matrix decomposition in blocks

module Matrix

import Real real numbers and operations on them

Nat^+ ≜ Nat \ {0}

sqm[E] ≜ \{(f, n, E) : (f \in \text{Nat}^+ \rightarrow \text{Nat}^+ \rightarrow E) \land (n \in \text{Nat}^+) \land (\exists p \in \text{Nat} : n = 2^p)\}

Figure 8: Definition of the set of square matrices of 2^p lines.
\begin{verbatim}
import Matrix

parameters
E : constant

predicate
compatible_matrix(A, B) \triangleq \land (A \in sqm[E]) \lor (B \in sqm[E])
\land A.2 = B.2
\land A.3 = Real

state function
DotProd[s \in Nat^+][j \in Nat^+][k \in Nat^+][A \in sqm[E]][B \in sqm[E]] \triangleq
if k = 0 then 0
else DotProd[s][j][k - 1][A][B] + A[s][k] \star B[k][j]
mult[A \in sqm[E]][B \in sqm[E]] \triangleq let
\phi[s][j] \triangleq DotProd[s][j][A.2][A][B]
in (\phi, A.2, A.3)

actions
mult_a(A, B, C) \triangleq \land compatible_matrix(A, B)
\land C = \text{mult}[A][B]
\end{verbatim}

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 multiplica-
tion 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 func-
tion. 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 \textsc{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
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 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 x1, ..., xn are all the free variables occurring primed in A or non-
primed in B, then A • B is equal to ⋄ $x_1, \ldots, x_n : A ∧ B$, where $\hat{A}$ is A with each $x'_i$ replaced by $\$x_i$, and $\hat{B}$ is B with each $x_i$ replaced by $\$x_i$.
For instance, \((x' = x + 1) \bullet (x' = x + 1) \triangleq \exists x : (x = x + 1) \land (x' = x + 1)\). 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, \ldots, may change, but the result C is the same, and this is what we seek. The \(-\bullet\) 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\(^+\)).
Separating Algorithm and Implementation in the Refinement of Parallel Program Specifications

Figure 13: Dependences in the \textit{mult}\_\textit{a} action of module \textit{Matrix\_product4}

\begin{verbatim}
... module Matrix_product5
  actions
    add\_a(A, B, C) \triangleq \ldots
    restrict\_a(A, r_1, r_2, c_1, c_2, C) \triangleq \ldots
    build\_a(A, B, C, D, M) \triangleq \ldots
    mult\_a(A, B, C) \triangleq \wedge \text{compatible_matrix}(A, B)
      \wedge (A:2 = B:2 = 1) \wedge (\exists f : \wedge f[i][j] = ((A:1[i][1]) * (B:1[i][1]))
      \wedge C = \{f, 1, A:3\})
      \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}, A_{A1}, A_{A2}, A_{A3}, A_{A4}:
        \bullet \wedge \text{restrict\_a}(A, 1, [A:2]/2, 1, (A:2)/2, A_{11})
        \wedge \text{restrict\_a}(A, 1, [A:2]/2, [A:2]/2 + 1, A:2, A_{12})
        \wedge \text{restrict\_a}(A, [A:2]/2 + 1, A:2, 1, (A:2)/2, A_{21})
        \wedge \text{restrict\_a}(A, [A:2]/2 + 1, A:2, [A:2]/2 + 1, A:2, A_{22})
        \wedge \text{restrict\_a}(B, 1, [B:2]/2, 1, [B:2]/2, B_{11})
        \wedge \text{restrict\_a}(B, 1, [B:2]/2, [B:2]/2 + 1, B:2, B_{12})
        \wedge \text{restrict\_a}(B, [B:2]/2 + 1, B:2, 1, [B:2]/2, B_{21})
        \wedge \text{restrict\_a}(B, [B:2]/2 + 1, B:2, [B:2]/2 + 1, B:2, B_{22})
        \bullet \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})
        \bullet \wedge (\text{add\_a}(M_{1111}, M_{1221}, A_{A1}) \wedge (\text{add\_a}(M_{1112}, M_{1222}, A_{A2}))
        \wedge (\text{add\_a}(M_{2111}, M_{2221}, A_{A3}) \wedge (\text{add\_a}(M_{2112}, M_{2222}, A_{A4}))
        \bullet C' = \text{build}(A_{A1}, A_{A2}, A_{A3}, A_{A4})
\end{verbatim}

Figure 14: \textit{Matrix\_product5} module

BCS-FACS 7th Refinement Workshop
then we can use a call to an action as a shortcut.

The splitting of the \textit{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 \textit{Matrix\_product6} (figure 15).

\begin{figure}[h]
\begin{center}
\begin{verbatim}
module Matrix_product6
  ...
actions
  add_a(A, B, C) \triangleq \ldots
  restrict_a(A, r_1, r_2, e_1, e_2, C) \triangleq \ldots
  build_a(A, B, C, D, M) \triangleq \ldots
rest_a(A, B, A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, B_{12}, B_{21}, B_{22}) \triangleq
  \land \text{restrict}_a(A, 1, [A_{21}] / 2, 1, [A_{2}] / 2, A_{11})
  \land \text{restrict}_a(A, 1, [A_{2}] / 2, [A_{2}] / 2 + 1, A_{2}, A_{12})
  \land \text{restrict}_a(A, [A_{2}] / 2 + 1, A_{2}, [A_{2}] / 2 + 1, A_{2}, A_{22})
  \land \text{restrict}_a(B, 1, [B_{2}] / 2, 1, [B_{2}] / 2, B_{11})
  \land \text{restrict}_a(B, [B_{2}] / 2, [B_{2}] / 2 + 1, B_{2}, B_{12})
  \land \text{restrict}_a(B, [B_{2}] / 2 + 1, B_{2}, 1, [B_{2}] / 2, B_{21})
  \land \text{restrict}_a(B, [B_{2}] / 2 + 1, B_{2}, [B_{2}] / 2 + 1, B_{2}, B_{22})

mult_blocks_a(A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, B_{12}, B_{21}, B_{22}, M_1, M_2, M_3, M_4, M_5, M_6, M_7, M_8) \triangleq
  \land [M'_1 = \text{mult}_a(A_{11}, [B_{11}]) \land M'_4 = \text{mult}_a(A_{12}, [B_{11}])]
  \land [M'_3 = \text{mult}_a(A_{12}, [B_{12}]) \land M'_6 = \text{mult}_a(A_{11}, [B_{12}])]
  \land [M'_0 = \text{mult}_a(A_{11}, [B_{21}]) \land M'_3 = \text{mult}_a(A_{12}, [B_{21}])]
  \land [M'_2 = \text{mult}_a(A_{11}, [B_{22}]) \land M'_5 = \text{mult}_a(A_{12}, [B_{22}])]

add_blocks_a(M_1, M_2, M_3, M_4, M_5, M_6, M_7, M_8, R_1, R_2, R_3, R_4) \triangleq
  \land [\text{add}_a(M_1, M_2, R_1) \land [\text{add}_a(M_3, M_4, R_2)]
  \land [\text{add}_a(M_5, M_6, R_3) \land [\text{add}_a(M_7, M_8, R_4)]

mult_one_one_a(A, B, C) \triangleq
  (\exists j : j[i][j] = (\{A_{11}\}[i][1]) \land \land C' = (\{B.1\}[1][1])

mult_a(A, B, C) \triangleq
  \land \text{compatible_matrix}(A, B)
  \lor (A_{2} = B_{2} = 1) \land \text{mult_one_one_a}(A, B, C)
  \lor \exists A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, B_{12}, B_{21}, B_{22},
    M_1, M_2, M_3, M_4, M_5, M_6, M_7, M_8, R_1, R_2, R_3, R_4 :
    \text{\textbullet\ rest}_a(A, B, A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, B_{12}, B_{21}, B_{22})
    \text{\textbullet\ mult_blocks_a(A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, B_{12}, B_{21}, B_{22}, M_1, M_2, M_3, M_4, M_5, M_6, M_7, M_8, R_1, R_2, R_3, R_4)
    \text{\textbullet\ add_blocks_a(M_1, M_2, M_3, M_4, M_5, M_6, M_7, M_8, R_1, R_2, R_3, R_4)
    \text{\textbullet\ build_a(R_1, R_2, R_3, R_4, C)]}
\end{verbatim}
\end{center}
\caption{Matrix\_product6 module}
\end{figure}

There, we would of course like to write instead
\[
\land \text{mult}_a(A_{11}, B_{11}, M_1) \land \text{mult}_a(A_{12}, B_{21}, M_2) \land \text{mult}_a(A_{11}, B_{12}, M_3) \land \text{mult}_a(A_{12}, B_{22}, M_4) \land \text{mult}_a(A_{21}, B_{11}, M_5) \land \text{mult}_a(A_{22}, B_{21}, M_6) \land \text{mult}_a(A_{21}, B_{12}, M_7) \land \text{mult}_a(A_{22}, B_{22}, M_8)
\]

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 $M_{111}$ from the matrices $A_{11}$ and $B_{11}$ is independent from the computation of matrix $M_{222}$ 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 $Matrix_{\text{product}}$ module (figure 15) into a cTLA class. First, here is an elementary implementation of the $Nat^+$ type:

```c
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 $sqm$.

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

Figure 16: $sqm[E]$ set

```c
set N3 = [Nat \rightarrow [Nat \rightarrow Nat]];.
```

Figure 17: $N3$ set

```c
set tupsqm={<f,n> :in: (Nat,Real) :times: Nat
\rightarrow [[i :in: (i :in: Nat) \land (i>0) \land (i :leq: n)] \rightarrow 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 $Matrix_{\text{product}}$ 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 $Matrix_{\text{product}}$ 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 $Matrix_{\text{product}}$ 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

$^3$Because cTLA data structures are essentially those of TLA$^+$.
class Matrix_product_ctla
{
    private:
    set E;
    public:
    void Matrix_product_ctla(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);   
    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.

{
    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[i1][i2][j1] + (B[i1][i2][j1];
    }
}

Figure 20: Matrix_product class, add function

sqm[e] Matrix_product_ctla::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[i][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_ctla is defined as process 1. This instance is initialized with the set E. The multipli-
cation of A and B is then simply performed by the call of the mult function of process 1, with arguments A
Separating Algorithm and Implementation in the Refinement of Parallel Program Specifications

```c
{
  int N;
  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
        else
          if (j < A[2])
          else
      }
};
```

Figure 22: Matrix_product class, build function

```c
tupsqm[e,8] Matrix_product_ctl1::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_ctl1(e),lp+1);
  ... newp(Matrix_product_ctl1(e),lp+8);
  par {
    ... M2=process(lp+8).restrict(b[1],(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.

Another 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
Separating Algorithm and Implementation in the Refinement of Parallel Program Specifications

```c
{
    sqm[e] M1, M2, M3, M4;
    int lp;
    lp=last_process();
    newp(Matrix_product_ctl1(e),lp+1);newp(Matrix_product_ctl1(e),lp+2);
    newp(Matrix_product_ctl1(e),lp+3);newp(Matrix_product_ctl1(e),lp+4);
    par {
        M1=process(lp+1).add(M1,M2);
        M2=process(lp+2).add(M3,M4);
        M3=process(lp+3).add(M5,M6);
        M4=process(lp+4).add(M7,M8);
    }
    return <<M1,M2,M3,M4>>;
};
```

Figure 24: Matrix_product class, block addition functions

```c
{
    sqm[e] M1, M2, M3, M4, M5, M6, M7, M8;
    int lp;
    lp=last_process();
    newp(Matrix_product_ctl1(e),lp+1);
    ....
    newp(Matrix_product_ctl1(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

```c
{
    M3 = f[
        f[1][1] = ((A[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
Separating Algorithm and Implementation in the Refinement of Parallel Program Specifications

```c
{
    tupsqm[e][8] R,M;
    tupsqm[e][4] S;
    if (compatible_matrix(A,B))
    {
            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

```c
main()
{
    set E=Nat;
    sqm[E][3] 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

```c
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);
    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.
Separating Algorithm and Implementation in the Refinement of Parallel Program Specifications

```c
{
    sync msq[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)){
            A[1][1] = ((A[1][1][1])*((B[1][1][1]));
        }
        else{
            par{
                newp(matrix_product_ctla2(e),lp+1);
                ...;
                newp(matrix_product_ctla2(e),lp+20);
                A11=process(lp+1).restrict(A,1,2,1,2);
                ...;
                B22=process(lp+8).restrict(B,1,2,1,2);
                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

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


BCS-FACS 7th Refinement Workshop
Separating Algorithm and Implementation in the Refinement of Parallel Program Specifications


