From a Specification to an Equivalence Proof in Object-Oriented Parallelism

Isabelle Attali, Denis Caromel, Sylvain Lippi

INRIA Sophia Antipolis, CNRS - I3S - Univ. Nice Sophia Antipolis,
BP 93, 06902 Sophia Antipolis Cedex - France
First.Last@inria.fr

Abstract.
We investigate the use of a TLA specification for modeling and proving parallelization within an object-oriented language.
Our model is based on Eiffel// a parallel extension of Eiffel, where sequential programs can be reused for parallel or concurrent programming with very minor changes. We want to prove that both versions of a given program (sequential and parallel) are equivalent with respect to some properties.
This article presents a description in TLA+ that captures the general Eiffel// execution model, and, as a case-study, specifies a program (a binary search tree) in both its sequential and parallel form. We then prove a property that demonstrates a behavioral equivalence for the two versions.

1 Introduction

In this article, we investigate the use of a specification in TLA [14] and TLA+ [15, 16] for concurrent and parallel object-oriented programming. Our model is based on the Eiffel// language [5, 7], a parallel extension of Eiffel [19].

One interesting aspect of the Eiffel// language is that existing sequential programs can be re-used when designing and programming parallel system: given a sequential system, using program transformations, one can manually generate an equivalent system running in parallel. We then would like to prove that both versions of the system (sequential and parallel) are equivalent.

With a formal definition of the concurrent model, our goal is to describe formally (and prove) parallelization techniques and algorithms for the Eiffel// framework. We want to define formally program transformations, especially parallelizations, from Eiffel to Eiffel// . Thus, we need indeed to prove that transformations preserve the meaning (the semantics, the behavior) of programs.

We already have a formal semantics of Eiffel// [2] based on a transition system whose states represent global configurations of a given system of objects. Objects are either passive or active. The semantics of a program is given by a transition system which represents all possible executions. Using this approach, the major drawback is the tremendous number of configurations, so we are currently studying a method for reducing the number of states, using partial order techniques.

We also investigated the use of π-calculus [20] to model an Eiffel// system [23, 3] (a translation from Eiffel and Eiffel// to π-calculus has been written in Natural Semantics). The π-calculus models the system as a quite low-level, with complexity manifested in a large number of channels and a loss of structure, making it difficult to understand and manipulate the π-terms. In that framework, the next step is to use bisimulation techniques [1] to prove the behavioral equivalence between a sequential and a parallel behavior.

We present here a description in TLA+ that captures in a more natural way the richness of the execution model. From the specification of two versions of a single program (one is sequential, one is parallel), we are able to prove that the two versions are equivalent with respect to some properties. The example that serves our purpose is the management of a binary tree (e.g. for a symbol table) with primitives for insertion and search.

Compared to [18], where a binary tree example is also used, the Eiffel// model allows use of the same code for both sequential and parallel versions; no `commit` instruction is needed in Eiffel// . This feature is critical for reuse, and is also beneficial for the proof since the actions modeling the program are identical in both cases.

Compared to the work on the $\pi o \beta \lambda$ design language [12], the Eiffel// model features synchronous and asynchronous calls through polymorphism and automatic continuations, which allow to achieve parallelization without changing the source code ($\pi o \beta \lambda$ for instance changes the place of the *return* statement, or needs to add a *delegate* statement). Another major difference seems to be the absence of future-based synchronizations, such as the wait-by-necessity. However, it is worth mentioning that the *island* concept is related to the subsystem notion, but seems different (since objects within an Eiffel// subsystem can perfectly reference the root of another subsystem). Finally, to our knowledge, there is no model and proof of the language within the TLA framework; a comparison would be interesting.

The next section of this article is a brief overview of the concurrent model. Section 3 introduces the main features of TLA and TLA+. In Section 4, we present our description of the Binary Tree example using TLA+. The proof of behavioral equivalence is sketched in Section 5. Finally, Section 6 discusses our contribution and outlines future work.

2 The Eiffel// language

In this section, we give a quick overview of a concurrent language, Eiffel// [7], defined as an extension of Eiffel [19] to support programming of parallel applications. These extensions are not concerned with syntax, but are purely semantic. Both Eiffel and Eiffel// are strongly typed, statically-checked class-based languages. Our purpose here is not to discuss the rationale of Eiffel// (see for instance [2] for a formal specification), and issues related to object-based concur-

rent languages (the reader can refer to a recent and related design (C++// [8]), and to related research [21, 24, 4, 22, 25]). We present the concurrent features of the model and illustrate it with an example which will serve throughout the paper.

2.1 Features

The Eiffel// model uses the following principles:

- heterogeneous model with both passive and active objects;
- polymorphism between passive and active objects;
- sequential active objects;
- unified syntax between message passing and inter-process communication;
- systematic asynchronous communications towards active objects;
- wait-by-necessity (automatic and transparent futures);
- automatic continuations;
- no shared passive objects (call-by-value between active objects);
- centralized and explicit control by default.

As in Eiffel, the text of an Eiffel// program (or system) is a set of classes, with a distinguished class, the root class. Parallelism is introduced via a particular class named `Process`. Instances of classes inheriting (directly or not) from the `Process` class are active objects. Active objects inherit a default behavior (through the **live** routine, which ensures that requests are treated in a *fifo* order), but this behavior can be redefined by overriding the **live** routine. All other objects are passive. Polymorphism between passive and active objects is possible: an entity which is not declared of an active object type can dynamically refer to an active object heir. In that case, thanks to dynamic binding, a feature-call dynamically becomes an asynchronous communication between active objects (*Inter-Process Communication*).

Cohabitation of active and passive objects leads to an organization in subsystems. Each subsystem contains an active object (root) and the passive objects it references. Within a subsystem, the execution is sequential and communication is synchronous: the target object immediately serves the request and the caller waits for the return of the result. Between subsystems, execution is parallel and communication is asynchronous: the target object (an active object) stores the request in a list of pending requests, and the caller carries on execution.

Synchronization is handled via *wait-by-necessity*, a data-driven mechanism which automatically triggers a wait when an object attempts to use the result of an awaited value (transparent future). The wait-by-necessity, by automatically adding some synchronization, tends to maintain the behavior of a sequential program when doing the parallelization. Finally, the automatic continuation, by automatically taking care of delegation behaviors without blocking the current object, allows reuse of such sequential code in parallel subsystems.

```
class Binary_Tree
  export   insert, search
  feature
    key, info : INTEGER;
    left, right : BINARY_TREE;
    insert (k : INTEGER; i : INTEGER) is
       do -- inserts information i with key k
         if key = 0 then key := k; info := i;
         left.create; right.create;
         elsif k = key then info := i;
         elsif k < key then left.insert(k,i);
         else right.insert(k,i);
         end;
       end;
    search (k : INTEGER) : INTEGER is
       do -- searches for the value of key k
         if key = 0 then result := <notfound>;
         elsif k = key then result := info;
         elsif k < key then result := left.search(k);
         else result := right.search(k);       --(AC)
         end;
       end;
end -- Binary_Tree

class P_Binary_Tree
  repeat Binary_Tree
  inherit Process;
          Binary_Tree redefine left, right;
  feature
    left, right : P_Binary_Tree;
end -- P_Binary_Tree
```

(a) Sequential and Parallel Binary Trees

```
class Client
  feature
    v: INTEGER;
    sequential_behavior is
      local bt: Binary_Tree;
        do
          bt.create;
          bt.insert(3, 6);
          bt.insert(4, 8);
          bt.insert(2, 4);
          bt.insert(6, 12);
          bt.insert(1, 2);
          v := bt.search(4);
          v := v + v
        end; -- sequential_behavior
    parallel_behavior is
      local bt: P_Binary_Tree;
        do
          bt.create;
          bt.insert(3, 6);
          bt.insert(4, 8);
          bt.insert(2, 4);
          bt.insert(6, 12);
          bt.insert(1, 2);
          v := bt.search(4);
          v := v + v      -- (WBN)
        end; -- parallel_behavior
end -- Client
```

(b) Reusing code for a parallel behavior

Fig. 1. An Eiffel// system

2.2 An example

As an illustration, Figure 1 presents an Eiffel// system introduced in [6]. It provides a parallel version of the sequential class `Binary_Tree`, which describes the management of a sorted binary tree with two routines `insert` and `search`: each node of the tree has two children (`left` and `right`), an information field (`info`) and an associated key (`key`); keys of the left (resp. right) subtree of a node are smaller (resp. greater) than the key of this node. A zero value for the key denotes a leaf of the tree.

To parallelize the binary tree we define the `P_Binary_Tree` class. It inherits from the `Process` class and the `Binary_Tree` class; no other programming is necessary; the full version of the class is actually shown in the Figure 1.a. A client can use the two classes (sequential and parallel binary trees) for both sequential and parallel behaviors, which makes it possible to reuse existing sequential code (using an object `bt` of class `P_Binary_Tree` instead of `Binary_Tree`). In that example, the default *fifo* behavior and wait-by-necessity (labeled in the example with (WBN)) ensure that all insertions are handled in a correct order, and before the search; this allows to preserve the semantics of the sequential system in the parallel execution. Automatic continuation (labeled in the example with (AC)) allows not to block the current node when forwarding the `search` request to a child, and still ensures that the final result will properly be sent to the objects in which it is needed.

Of course, this equivalence only holds in the case of a single client object; if several clients in different active objects (processes) send `insert` and `search` requests, the behavior can be different from the sequential version.

3 The Temporal Logic of Actions

TLA [14] (Temporal Logic of Actions) is a linear-time temporal logic meant to model and reason on concurrent systems. Systems and their properties are described by logical formulas. TLA has been used in a number of varied applications: concurrent algorithms [9] as well as hybrid systems [13].

3.1 TLA

TLA formulas are interpreted as predicates on behaviors where a behavior is an infinite list of states and a state is an assignment of values to variables. Here is the TLA formula defining our "binary tree" system :

$$\Phi \triangleq \mathit{InitEiffel}// \wedge \mathit{InitClient} \wedge \Box[\mathbf{updatePendings} \vee \mathbf{send} \vee \mathbf{insert} \vee \mathbf{search}]_w$$

We can distinguish two parts[1]:

[1] We could have added a third part called *a fairness condition* which is used to prove liveness properties.

- *InitEiffel*// $\wedge$ *InitClient*
  This first part specifies the first state of the accepted behavior. For example, on the first state, we have only one object of the Client class;
- $\Box[$**updatePendings** $\vee$ **send** $\vee$ **insert** $\vee$ **search**$]_w$
  This second part specifies the different possible steps of the accepted behaviors. A step is made of two states (an initial state and a final state). The predicates **updatePendings**, **send**, **insert**, and **search** are actions on steps. They are built with unprimed and primed variables. Primed variables represent the value of the variable in the final state of the step. For example, the action x'=x+1 accepts all steps where x is incremented by one.
  The symbol $\Box$ is the temporal symbol "always" which says that every step is an **updatePendings**, a **send**, an **insert** or a **search** step. The index `w` is the tuples of all the variables we need. Those variables are called *flexible* because their value can change during the execution of the system. We also use some rigid variables (or constants). For example, **initBinary** specifies the value of an object of class Binary_Tree when it has just been created.

3.2 TLA+

Our specification uses TLA+ [15, 16], a complete language based upon TLA. We are using the module organization of TLA specifications: our module structure is straightforward: each class is modelled as a module (Client, Binary_Tree) and we add an Eiffel// module for the actions and state predicates common to all the specifications of Eiffel// programs.

We also use the "Sequences" module, defined in [17], which defines usual operators on sequences (Head, Tail, concatenation, and length). For example, Seq(S) represents the set of all the sequences of values belonging to S.

Here are some notations used to represent functions and records (as presented in [16]):

- $[A \rightarrow B]$ is the set of all the functions from A into B.
- the notation: $[1 \mapsto "a", 2 \mapsto "b", 3 \mapsto "c"]$ denotes a function from the set {1,2,3} into {"a","b","c"}.
- if f is a function, we note:

  f[i], the application of f to i;

  [f EXCEPT ![i] = j], the function which is equal to f except in i where the returned value is j;

  [f EXCEPT ![i] = @+j] the function which is equal to f except in i where the returned value is $f[i] + j$ (@ denotes $f[i]$).
- records are particular functions which associate a value to a string.

If rec is a record, we note $rec.field1$ the value of $rec["field1"]$.

4 The Specification

We choose to model the Eiffel (and Eiffel//) programs in three parts using TLA+; some are specific to the binary tree example (definition and use in a sequential

or a parallel setting) but one is abstract and generic enough to be used for any example: it captures the features of the Eiffel// programming model. Namely, our specification is expressed in three modules (see Figure 2):

- Eiffel// : state predicates and actions common to all specifications of Eiffel and Eiffel// programs;
- Binary_Tree: actions related to objects of class `Binary_Tree`;
- Client: actions related to the object of class `Client`.

All the general and modular aspects of the translation are in the Eiffel// module. It captures the specific semantics of the underlying model, such as asynchronous calls, futures, and wait-by-necessity. All the actions correspond to the management of requests, reply, and automatic continuations.

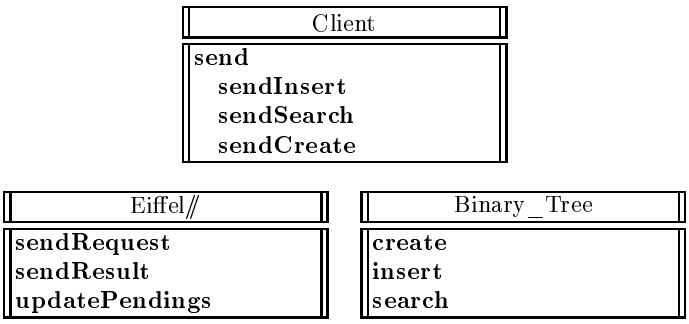


Fig. 2. The modules of the TLA+ specification

The Client and the Binary_Tree modules are specific to this case study. However, each action of the Binary_Tree module corresponds to one instruction of the corresponding class, when viewed from the perspective of a translation from Eiffel// code to TLA. Finally, each action of the Client module represents a call to a feature of Binary_Tree (**sendCreate**, **sendInsert**, **sendSearch**).

Please note that the Eiffel// module specifies both the sequential and the parallel cases. For instance, this module specifies synchronous calls in the case of a standard object, or asynchronous one with wait-by-necessity if the target object of a communication is active. As a consequence, the client and Binary_Tree specifications can be the same for both sequential and parallel versions.

We give a brief description of the flexible variables and then explain the state predicates and actions of the three modules.

4.1 Definition of flexible variables

The state function w gives the tuple of all variables in the description. We cannot detail all of them but instead give an intuition of their role in the specification:

$$w \triangleq \langle object, active, requests, curReq, future, pendings, result, synchro, pc \rangle$$

- objects : *object*, *active*
  The *object* variable is a function from object identifiers to objects (records that are object attributes). For instance, a Binary_Tree instance is a record of the form: $[key \mapsto 2, info \mapsto 4, left \mapsto 5, right \mapsto nil]$. The *active* variable is a boolean function that states whenever an object is active or passive.
- list of requests : *requests*, *curReq*
  Each call to an object is modeled by a request; every object has a list of pending requests. The *requests* variable is a function from object identifiers to lists of requests; each request includes the name of the target routine, a list of effective parameters, and a future for the result (see below).
  The *curReq* variable gives the request an object is currently handling.
- sending requests and results: *future*, *pendings*, *result*

  $future \in [futureIDs \rightarrow values \cup \{?\}]$
  Future stores the awaited value corresponding to a given future ID ; "?" means that the result is not yet arrived.

  $pendings \in [objectIDs \rightarrow Seq(futureIDs \times futureIDs)]$
  Pendings is used when the result of a request is itself a future ID, allowing to specify automatic continuations.

  $result \in [objectIDs \rightarrow values \cup futureIDs]$
  Result corresponds to the result Eiffel variable to be returned to the caller at the end of a routine.
- synchronous calls: *synchro*
  The *synchro* variable allows specification of synchronizations arising from synchronous and asynchronous calls. It is a function from object identifiers to two possible sets of values:
  - the value "OK" in case of an asynchronous call;
  - in the case of a synchronous call, a "future" identifier is used to block the calling object (in other words, we introduce an artificial data dependency, specifying an immediate wait on a future that is not yet needed).
- program counter: *pc*
  This is a standard way of ordering instructions in an object (execution is sequential within a subsystem).

4.2 The Binary_Tree module

The Binary_Tree module exports three main actions, each of them corresponds to a routine of the class Binary_Tree (`create`, `insert`, `search`). These actions are a translation of Eiffel// instructions into the TLA specification. The translation seems to be straightforward enough to permit some automatic translation in the future.

The action **create**(objID, attr) creates an instance of the class Binary_Tree, generates a new identifier `id` for this object, updates the value of the attribute attr of the object objID (creation site, this corresponds to the statement `bt.create`) with this new reference. Note that `id` is active whenever objID is.

$$\begin{array}{l}\textbf{create}(objID, attr) \triangleq \exists\, id \in objectIDs \\ \quad \wedge\ object[id] = \text{FREE} \\ \quad \wedge\ object' = [object \text{ EXCEPT } ![objID].attr = id, \\ \qquad\qquad\qquad\qquad\quad ![id] = initBinary \quad] \\ \quad \wedge\ active' = [active \text{ EXCEPT } ![id] = active[objID]] \\ \quad \wedge \text{ UNCHANGED } \langle future, synchro, requests, curReq, pendings, result \rangle \\ \\ initBinary \triangleq [key \mapsto 0, info \mapsto 0, left \mapsto nil, right \mapsto nil]\end{array}$$

The action **insert** (resp. **search**) is a disjunction of actions corresponding to the different instructions of the routine `insert` (resp. `search`). In both cases, sequentiality is managed with the *pc* variable.
For instance, action $\textbf{insert}_4$ corresponds to the statement `left.create` :

$$\begin{array}{lll}\textbf{insert}_4(objID) & \triangleq & \wedge\ pc[objID] = 4 \\ & & \wedge\ pc' = [pc \text{ EXCEPT } ![objID] = 5] \\ & & \wedge\ \textbf{create}(objID, left)\end{array}$$

Finally, here is the first action $\textbf{search}_1$ which corresponds to the test expression `if key=0` :

$$\begin{array}{lll}\textbf{search}_1(objID) & \triangleq & \wedge\ pc[objID] = 1 \\ & & \wedge\ \vee\ \wedge\ object[objID].key = 0 \\ & & \qquad \wedge\ pc' = [pc \text{ EXCEPT } ![objID] = 2] \\ & & \quad\ \vee\ \wedge\ \neg(object[objID].key = 0) \\ & & \qquad \wedge\ pc' = [pc \text{ EXCEPT } ![objID] = 3] \\ & & \wedge \text{ UNCHANGED } \langle future, synchro, requests, curReq, \\ & & \qquad\qquad\qquad\qquad active, object, pendings, result \rangle\end{array}$$

4.3 The Client module

The client object simply sends a "create" request followed by some "insert" requests with finally some "search" requests to the object representing the root of the binary tree. This list of requests which are sent by the client during execution is contained in the *extRequests* rigid variable. The Client module is made of one main action **send** which is a conjunction of actions modeling the possible requests (**sendInsert**, **sendSearch**, and **sendCreate**). The state predicate *notWaiting* checks whenever an object is blocked by a synchronous call.

$$\begin{array}{lll}\textbf{send} & \triangleq & \wedge\ notWaiting(clientID) \\ & & \wedge\ pc[clientID] \in 1..Len(extRequests) \\ & & \wedge\ pc' = [pc \text{ EXCEPT } ![clientID] = @ + 1] \\ & & \wedge\ \textbf{sendInsert} \vee \textbf{sendSearch} \vee \textbf{sendCreate}\end{array}$$

The Client module imports the two other modules and define the following formula, corresponding to the accepted behavior:

$$\Phi \triangleq InitEiffel// \wedge InitClient \wedge \Box[\textbf{updatePendings} \vee \textbf{send} \vee \textbf{insert} \vee \textbf{search}]_w$$

The parallel behavior is specified with

$$\Phi \wedge active[root] = \text{“true”}$$

while

$$\Phi \wedge active[root] = \text{“false”}$$

specifies a sequential behavior. So every property holding for Φ is true for both behaviors (sequential and parallel).

4.4 The Eiffel// module

The Eiffel// module is the generic part of the specification: it captures all the parallel semantics of the model, and is to be reused when modeling other programs.

Exported actions represent transmission of requests and transmission of results to the caller. These mechanisms, formally described in [2] in Natural Semantics, are expressed in TLA using three main actions: **sendRequest**, **sendResult**, and **updatePendings**.

The action **sendRequest**$(orig, dest, req)$ sends the request req from object $orig$ to object $dest$:

1. the request is first added to the list of requests of object dest;
2. the future value is set to "?" (i.e. unknown, not yet arrived);
3. the synchro value of the caller is set to:
   - OK in case of an asynchronous call,
   - the future itself for a synchronous call, in order to block the caller immediately until the value is returned, as discussed earlier.

This action is expressed in TLA as follows:

$$\begin{array}{l}
\textbf{sendRequest}(orig, dest, req) \triangleq \\
\quad \wedge\ requests' = [requests \text{ EXCEPT } ![dest] = @ \circ \langle req \rangle] \\
\quad \wedge\ future' = [future \text{ EXCEPT } ![req.future] = ?] \\
\quad \wedge\ \vee\ active[dest] = \text{“true”} \wedge synchro' = [synchro \text{ EXCEPT } ![orig] = \text{OK}] \\
\qquad \vee\ active[dest] = \text{“false”} \wedge synchro' = [synchro \text{ EXCEPT } ![orig] = req.future] \\
\quad \wedge\ \text{UNCHANGED } < active, pendings, curReq >
\end{array}$$

The action **sendResult** is performed by an object which sends back the result of the request it is responding to. There are two cases:

- the result is an effective value (not a future): in this case, the function *future* is updated with the value;
- the result is a future identifier id; in this case, the list *pendings* is updated with this identifier coupled with the awaited future.

This action is expressed in TLA as follows (with $\textbf{sendResult}_1$ and $\textbf{sendResult}_2$ for the two cases):

$$
\begin{array}{lll}
\textbf{sendResult}(objID) & \triangleq & \land\ pc' = [pc \text{ EXCEPT } ![objID] = 1] \\
& & \land\ curReq' = [curReq \text{ EXCEPT } ![objID] = @ + 1] \\
& & \land\ result' = [result \text{ EXCEPT } ![objID] = \text{OK}] \\
& & \land \lor \textbf{sendResult}_1(objID) \\
& & \quad \lor \textbf{sendResult}_2(objID)
\end{array}
$$

$$
\begin{array}{lll}
\textbf{sendResult}_1(objID) & \triangleq & \land\ result[objID] \notin futureIDs \\
& & \land\ future' = [future \text{ EXCEPT } ! \\
& & \qquad [requests[objID][curReq[objID]].future] = result[objID]] \\
& & \land \text{ UNCHANGED } \langle synchro, requests, active, object, pendings \rangle
\end{array}
$$

$$
\begin{array}{lll}
\textbf{sendResult}_2(objID) & \triangleq & \land\ result[objID] \in futureIDs \\
& & \land\ pendings' = [pendings \text{ EXCEPT } ![objID] = \\
& & \qquad @ \circ \langle requests[objID][curReq[objID]].future, result[objID] \rangle] \\
& & \land \text{ UNCHANGED } \langle future, synchro, requests, active, object \rangle
\end{array}
$$

Finally, the action **updatePendings** is not performed by any object but by something that can be referred to as some runtime action. It specifies the semantics of an automatic continuation, allowing a caller to access the result of the request when several references to futures are chained together, all to receive the same value. For example, when the result of request1 equals the result of request2 which in turns equals the result of request3. The result of **updatePendings** is an update of both *future* and *pendings*:

$$
\begin{array}{l}
\textbf{updatePendings} \triangleq \\
\quad \land \exists\, objID \in objectIDs : \exists\, n \in 1..Len(pendings[objID]) \\
\qquad \land\ future[\ pendings[objID][n][2]\ ] \neq ? \\
\qquad \land\ future' = [future \text{ EXCEPT } ![pendings[objID][n][1]] = \\
\qquad\qquad\qquad future[\ pendings[objID][n][2]\ ]] \\
\qquad \land\ pendings' \triangleq [pendings \text{ EXCEPT } ![objID] = \\
\qquad\qquad [i \in 1..Len(@) - 1 \mapsto \textbf{if } i < n \textbf{ then } @[i] \textbf{ else } @[i+1]]\ ] \\
\quad \land \text{ UNCHANGED } \langle synchro, requests, curReq, active, object, pc, result \rangle
\end{array}
$$

5 The behavioral equivalence proof

Intuitively, we would like to prove that, for a single client, or more generally several clients being in the same subsystem[2] (whether it is the `Binary_Tree` or the `P_Binary_Tree` class), when a result is returned after a search, it is the same in both cases.

Note that this is a partial equivalence. In the general case, it cannot be otherwise due to the non-strict aspect of the Eiffel// model; in particular, the wait-by-necessity implies that a sequential Eiffel system might not terminate when the Eiffel// does. Also note that we do not have a proof of the form *im-*

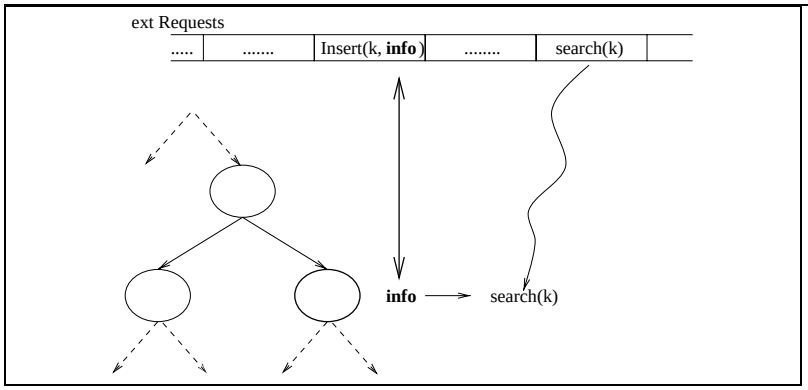
[2] See Section 2.1 for the definition of subsystems.

plementation => specification since none of the two versions (sequential and parallel) implies the other.

More specifically, we will prove that, given a list of `insert` and `search` requests that is sent by the `client` object (this list is represented by the *extRequests* rigid variable) and a `search` request, when an object responds to this request, it gives the same result in the sequential and parallel cases. Figure 3 gives such TLA state predicate that we note E. It expresses the fact that upon a `search`, when the key is found, the value being returned always corresponds to the value of the last `insert` request within *extRequests* with the same key.

To simplify the property, we assume that `insert` and `search` requests are not "mixed" in the *extRequests* list; we first have all the `insert` requests and then all the `search` requests. So, we can easily express that when an object responds to a `search(k)` request, the result is the `info` argument of the last `insert` request with a `key` argument equal to k.

Figure 3 also gives a graphical overview of the property. For a given search, when the key is found at a node, the information to be returned is equal to the `info` of the last `insert` with the same key. Because *extRequests* is a rigid variable, the value returned is always the same, for both the sequential and the parallel cases.



$$
\begin{array}{l}
E \triangleq \forall\, id \in objectIDs \\
\quad \wedge \textsc{Enabled}\,(search_8(id)) \\
\quad \wedge\ result[id] \in Nat^* \\
\quad \Rightarrow \\
\quad \exists\, n \in 1..Len(extRequests) \\
\qquad \wedge\ extRequests[n].routine = \text{“insert”} \\
\qquad \wedge\ extRequests[n].args[1] = object[id].key \\
\qquad \wedge\ extRequests[n].args[2] = result[id] \\
\qquad \wedge \forall\, m \in n{+}1..Len(extRequests) \\
\qquad\quad \vee\ extRequests[m].routine \neq \text{“insert”} \\
\qquad\quad \vee\ extRequests[m].args[1] \neq object[id].key
\end{array}
$$

Fig. 3. A behavioral equivalence property

To some extent, the property can be related to bisimulation techniques as used in a π-calculus framework. Please note that we cannot prove direct equivalence between sequential and parallel behaviors, or even that one implies the other.

We do not give the complete proof but rather a scheme. The proof is based on an invariant I, and the principle is the following:

1. find an intermediate invariant I;
2. prove that the initial state verifies I: $\mathit{InitEiffel}// \wedge \mathit{InitClient} \Rightarrow I$
3. prove that every action preserves I;
4. finally prove that $I \Rightarrow E$.

5.1 The invariant

We first describe the invariant (a state predicate) and then prove that it holds during the execution of the Binary Tree system. This invariant is presented as the conjunction of 20 state predicates. We group those predicates into four sets depending on what kind of property they express (typing, data control and value, structures of data, request flow):

$$
\begin{array}{lcl}
I & \triangleq & \wedge\ \mathit{invTyping} \\
 & & \wedge\ \mathit{invDataValue} \\
 & & \wedge\ \mathit{invStructure} \\
 & & \wedge\ \mathit{invRequest}
\end{array}
$$

All the sets being defined as:

$$
\begin{array}{lcl}
\mathit{invTyping} & \triangleq & \wedge\ \mathit{searchResultType} \\
 & & \wedge\ \mathit{clientType} \\
 & & \wedge\ \mathit{insertArgType} \\
 & & \wedge\ \mathit{searchArgType} \\
 & & \wedge\ \mathit{responderType}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{invDataValue} & \triangleq & \wedge\ \mathit{pcValue} \\
 & & \wedge\ \mathit{curReqMax} \\
 & & \wedge\ \mathit{curReqValue} \\
 & & \wedge\ \mathit{leaf} \\
 & & \wedge\ \mathit{rightValue} \\
 & & \wedge\ \mathit{keyValue} \\
 & & \wedge\ \mathit{insertArgValue} \\
 & & \wedge\ \mathit{requestsValue} \\
 & & \wedge\ \mathit{emptyRequests} \\
 & & \wedge\ \mathit{infoValue}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{invStructure} & \triangleq & \wedge\ \mathit{descendantOfRoot} \\
 & & \wedge\ \mathit{onlyOneFather} \\
 & & \wedge\ \mathit{notFatherOfRoot}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{invRequest} & \triangleq & \wedge\ \mathit{requestFlow} \\
 & & \wedge\ \mathit{rootRequests}
\end{array}
$$

As an illustration, we mention and detail a few invariants that will be used in the sequel of the paper.

The invariant *descendantOfRoot* states that every `binary` object is a descendant of the root of the tree (*rootID* object) and has one and only one father. More formally, the corresponding TLA predicate is:

$$
\begin{array}{lll}
descendantOfRoot & \triangleq & \forall\, id \in objectIDs \\
& & \quad \wedge\ object[id] \in binaryClass \\
& & \quad \wedge\ id \neq rootID \\
& & \quad \Rightarrow \\
& & \quad \exists\, l \in Seq(objectIDs) \\
& & \qquad \wedge\ Head(l) = rootID \\
& & \qquad \wedge\ Last(l) = id \\
& & \qquad \wedge\ \forall\, n \in 1..Len(l)-1 \\
& & \qquad\quad \wedge\ object[l[n]] \in binaryClass \\
& & \qquad\quad \wedge\ \vee\ object[l[n]].left = l[n+1] \\
& & \qquad\qquad \vee\ object[l[n]].right = l[n+1]
\end{array}
$$

Two invariants capture the tree structure of the list of objects at execution. The invariant *onlyOneFather* denotes the fact that each `binary` object (except the root one) is referenced by a unique object, while *notFatherOfRoot* states that none of them reference the root object.

The invariant *requestFlow* is also central for the proof. It states that each `binary` object has a request list that is a prefix of its father request list.

The informal statement of the invariant *rootRequests* is that the request list at the root object (requests[rootID]) is a sub-list of *extRequests*, starting at the second element (the first one being a special request that create the first node), and finishing at the last element that was sent by the client (pc[clientID]-1).

$$
rootRequests \quad \triangleq \quad requests[rootID] = SubSeq(extRequests, 2, pc[clientID] - 1)
$$

We prove the invariant I by following the method given in [14]. We have to prove $\Psi \Rightarrow \Box I$ where Ψ is the specification of the Binary Tree system. We use rule $INV1$ (from [14]):

$$
\frac{I \wedge [\mathcal{A}]_w \Rightarrow I'}{I \wedge \Box[\mathcal{A}]_w \Rightarrow \Box I}
$$

$\mathcal{A}$ is the disjunction of all possible actions of the specification. We have to prove $I \wedge [\mathcal{A}]_w \Rightarrow I'$ by showing that all the formulae of this form: $I \wedge [\mathcal{A}_i]_w \Rightarrow I'_j$ are valid.
All possible actions are:

- **updatePendings**: 1 case;
- **send**: 3 cases;
- **insert**: 8 cases;
- **search**: 11 cases.

The proof of the invariant is then made of 460 cases, more or less straightforward to prove. Of course, we can not give the proof here.

5.2 Proof of the property

Then, we deduce from this invariant the property E (as shown in Figure 3) and prove that $I \Rightarrow E$.

From the invariant *searchResultType*, we know that when an object replies to a "search" with an element of Nat^* as parameter, it gives the field *info*. So, from *infoValue* and *searchResultType*, when an object replies to a "search(k)" request with an element of Nat^* as parameter, it gives the field *info* of the last request `insert(k, i)` it received:

$$
\begin{array}{l}
\textit{infoValue} \wedge \textit{searchResultType} \Rightarrow \textit{searchResult} \\
\\
\text{where: } \textit{searchResult} \triangleq \forall\, id \in \textit{objectIDs} \\
\qquad \wedge\ \textsc{Enabled}\ (\textit{search}_8(id)) \\
\qquad \wedge\ \textit{result}[id] \in \textit{Nat}^* \\
\qquad \Rightarrow \\
\qquad \exists\, n \in 1..\textit{curReq}[id] \\
\qquad\quad \wedge\ \textit{requests}[id][n].\textit{routine} = \text{“insert”} \\
\qquad\quad \wedge\ \textit{requests}[id][n].\textit{args}[1] = \textit{object}[id].\textit{key} \\
\qquad\quad \wedge\ \textit{requests}[id][n].\textit{args}[2] = \textit{result}[id] \\
\qquad\quad \wedge\ \forall\, m \in n{+}1..\textit{curReq}[id] \\
\qquad\qquad \vee\ \textit{requests}[id][m].\textit{routine} \neq \text{“insert”} \\
\qquad\qquad \vee\ \textit{requests}[id][m].\textit{args}[1] \neq \textit{object}[id].\textit{key}
\end{array}
$$

From the invariant predicate *rootRequests*, $\forall k \in Nat$, we know that the request list of the object *rootID*, restricted to the requests which have k as first argument, is a prefix of the same restriction of the list *extRequests*.

From invariants *descendantOfRoot* and *requestFlow*, we can show that for every object of class `Binary_Tree`, and every $k \in Nat$, the request list of this object restricted to k is a prefix of the request list of the object *rootID* (also restricted to k). The proof is done by induction on the path length between the object under consideration and the *rootID* object.

This means that for every object of class `Binary_Tree`, and every $k \in Nat$, the request list of this object restricted to k is a prefix of the request list *extRequests* (also restricted to k). Figure 4 specifies the corresponding TLA state predicate, and also gives a graphical overview of the flow of requests within the binary tree.

From *searchResult* and *restrRequests*, when an object replies to a "search(k)" request with an element of Nat^* as parameter, it gives the field *info* of the last request `insert(k, i)` of the request list *extRequests*. Then:

$$\textit{searchResult} \wedge \textit{restrRequests} \Rightarrow E$$

We have proven the two formulae:

$$
\begin{array}{l}
\Phi \Rightarrow \Box I \\
I \Rightarrow T
\end{array}
$$

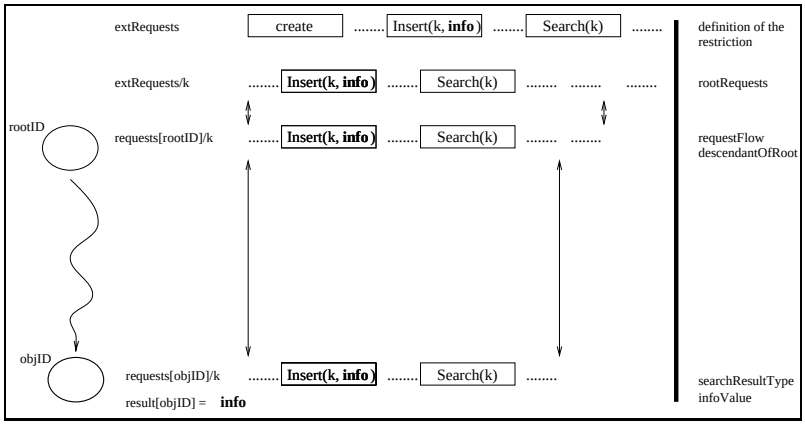
We use the STL4 rule (see page 17 in [14]):

$$\frac{I \Rightarrow T}{\Box I \Rightarrow \Box E} \quad STL4$$

Then, we have:

$\Phi \Rightarrow \Box E$

which terminates the proof.



$rootRequests \wedge descendantOfRoot \wedge requestFlow \Rightarrow restrRequests$

where: $restrRequests \triangleq$

$\forall\, id \in objectIDs : \forall\, k \in Nat : object[id] \in btObjects$

$\Rightarrow$

$\forall\, n \in 1..Len(\ restr(requests[id], k)\ )$

$\wedge\ restr(requests[id], k)[n].routine = restr(extRequests, k)[n].routine$

$\wedge\ restr(requests[id], k)[n].args = restr(extRequests, k)[n].args$

Fig. 4. Proof scheme of $restrRequests$

6 Conclusion

The specification we present in this paper is rather complex due to low-level primitives needed in TLA+ for expressing the sequence of instructions and the management of continuations. The specification includes a description of synchronous as well as asynchronous message passing and synchronization operations. Object-oriented features of the Eiffel// model (polymorphism between passive and active objects, reuse, inheritance for transforming sequential to parallel behaviors) are captured and used in the TLA specification (typed modules, modeling of dynamic types — reduced in that case to synchronous and asynchronous, reification of requests). The whole specification is made of 3 modules with a total of 30 actions. With this specification, we were able to prove a be-

havioral equivalence between the sequential version and the parallel version of the `Binary_Tree` example with a standard method using invariants.

The TLA model also provides means to prove other properties such as liveness, freedom from deadlock, or characterizations of sequential (or parallel) executions. We aim at investigating such properties in order to develop more knowledge about object-oriented concurrency and more specifically the Eiffel// model.

Compared a π-calculus description of Eiffel// [3], the TLA specification maintains some structure while π-calculus requires addition of many channels for modeling purposes only; moreover a lot of static typing is lost, due to channel names being sent across channels.

Compared to a specification of Eiffel// using operational semantics [2], the TLA specification appears to be of a lower level (e.g. a program counter has to be explicitly managed in TLA), while the granularity seems comparable (due to continuations). On the other hand, since the specification is usually based on a large number of actions, proofs require a tremendous number of cases for invariants; a mechanization of the proof using TLP [10] should be helpful and would also facilitate further modifications of the specification, within a computer-aided setting (as in [11]).

It still remains to generalize the property that was proved in this paper on a specific example. Intuitively, it seems that we should be able to state a general property that expresses some equivalence between a sequential system and its parallel version when the graph of objects at execution is a tree. The current structuring of the specification (the Eiffel// module contains all the concurrent features of the model itself) should help in that process. Moreover, the fact that the current proof relies so much on invariants that reflect the tree nature of the objects topology should be crucial for the generalization.

Acknowledgment:
The authors are grateful to Leslie Lamport and Andrew Wendelborn for helpful comments and discussions.

References

1. R. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. In *Proc. CONCUR '96*, volume 1119. LNCS, Springer-Verlag, 1996.
2. I. Attali, D. Caromel, S. Ehmety, and S. Lippi. Semantic-based visualization for parallel object-oriented programming. In *Proceedings of OOPSLA'96*, volume 31 of *ACM Sigplan Notices*, San Jose, CA, October 1996. ACM Press.
3. E. Bruneton. Implantation d'un traducteur eiffel// – π-calcul. Rapport de stage deuxième année, Ecole Polytechnique, juillet 1996.
4. *Concurrent Object-Oriented Programming.* Communications of the ACM, 36 (9), 1993. Special issue.

5. D. Caromel. Service, Asynchrony and Wait-by-necessity. *Journal of Object-Oriented Programming*, 2(4):12–22, November 1989.
6. D. Caromel. Concurrency and Reusability: From Sequential to Parallel. *Journal of Object-Oriented Programming*, 3(3):34–42, September 1990.
7. D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36(9):90–102, September 1993.
8. Denis Caromel, Fabrice Belloncle, and Yves Roudier. *The C++// Language*. MIT Press, 1996. ISBN 0-262-73118-5.
9. D. Doligez. *Conception, réalisation et certification d'un glaneur de cellules concurrent*. PhD thesis, Université Paris VII, 1995.
10. Urban Engberg. The tlp system. http://www.daimi.aau.dk/~urban/tlp/tlp.html.
11. G. Gonthier. Verifying the safety of a practical concurrent garbage collector. In *Proc. of the Eighth International Conference on Computer-Aided Verification*, ACM, New Brunswick, New Jersey, 1996.
12. S. Hodges and C. B. Jones. Non-interferance properties of a concurrent object-based language: Proofs based on an operational semantics. In *Object Orientation with Parallelism and Persistance*. Kluwer Academic Publishers, 1996. ISBN 0-7923-9770-3.
13. L. Lamport. Hybrid systems in tla$^+$. In *Hybrid Systems*, pages 77–102. Springer Verlag LNCS 736, 1993.
14. L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
15. L. Lamport. The Module Structure of TLA+. Technical Report 1996-002, DEC SRC, September 1996.
16. L. Lamport. The operators of TLA+. Technical Report 1997-006a, DEC SRC, April 1997.
17. L. Lamport and S. Merz. Specifying and verifying fault-tolerant systems. In H. Langmaack, W. P de Roever, and J. Vytopil, editors, *Formal Techniques in real-time and fault-tolerant systems*, volume 863 of *Lecture Notes in Computer Science*, pages 41–76. Springer-Verlag, 1994.
18. X. Liu and D. Walker. Confluence of processes and systems of objects. In *Proc. of Theory and Practice of Software Development (TAPSOFT'95) 6th International Joint Conference CAAP/FASE*, 1995.
19. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.
20. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, i and ii. *Journal of Information and Computation*, 100:1–77, 1992.
21. O. Nierstrasz. The next 700 concurrent object-oriented languages – reflections on the future of object-based concurrency. Object composition, Centre Universitaire d'Informatique, University of Geneva, June 1991.
22. O. Nierstrasz, P. Ciancarini, and A. Yonezawa, editors. *Rule-based Object Coordination*. LNCS 924. Springer-Verlag, 1995.
23. David Sagnol. π-calcul pour les langages à objets parallèles. Rapport de stage de DEA, Université de Nice - Sophia Antipolis, Juin 1995.
24. P. Wegner. Design issues for object-based concurrency. In *Proc. of the ECOOP '91 Workshop on Object-Based Concurrent Computing*, LNCS 612, 1992.
25. G. Wilson and P. Lu, editors. *Parallel Programing Using C++*. MIT Press, 1996. ISBN 0-262-73118-5.