Logic Abstract Modules: A new TLA-based model for Specifying and Verifying Concurrent Programs

Yahya Slimani and El Habib Daho†
Dept. Informatique - Faculté des Sciences de Tunis
Campus Universitaire - 1060 Tunis, Tunisia - Fax:+216-1-885190
e-mail:yahya.slimani@fst.rnu.tn
† USTO University (Algeria) - Fax:+213-6-410874

Abstract

A method for specifying and proving concurrent systems is described. It is based on a combination of Abstract Data Types (ADT) and Temporal Logic of Actions (TLA). The key concept is that the specification and the proof of a system uses an hybrid formalism: whilst the specification is described in a ADT-based module, the proof is expressed in Temporal Logic of Actions. Our proposal is illustrated via the well known dining philosophers problem.

1 INTRODUCTION

Specifications are means to define formally the behavior of a system or a system component. In particular, the specification of a program is the expression of the requirements that a user imposes on a program he wishes to use [2, 7].

Formal specification techniques have been promulgated as a means for making communication about the functionality of a precise system and less prone to errors due to misinterpretations [7].

Many specification methods have been proposed for sequential programs. They generally specify a program in terms of the values returned by its procedures and/or functions. However, this approach is inadequate for a concurrent program, for two major reasons: (i) several components of a concurrent program (i.e., processes) may be executed in parallel; (ii) a concurrent program has particular behaviors, like waiting and deadlock, that cannot exist in sequential one. Hence, it is not possible to specify a concurrent program in terms of the values returned, but it must be specified in terms of its behavior [9, 10, 12].

Structurally speaking, most concurrent programs can easily be separated into two parts: a *synchronization* part, which enforces the necessary constraints on the relative timing of the execution of the different processes and a *functional* part, which manipulates the data and performs the computation required of the program. For example, the part of a concurrent program that ensures mutual exclusion between sections of code is in the synchronization part of that program, whereas the code that is made mutually exclusive is in the functional part.

This structural separation suggests to use two complementary methods for specifying a concurrent program: one for its functional part, and the other for its behavior part. As part of this viewpoint, we suggest, in this paper, to combine two formalisms for specifying concurrent program: the Abstract Data Type (ADT) [3, 4, 8] to specify its functional part, and the Temporal Logic of Actions (TLA) [13] to specify its behavior.

The remainder of the paper is organized as follows: section 2 presents a brief overview of ADT and TLA. Section 3 gives the main elements of the proposed model for specifying and verifying concurrent programs. Section 4 illustrates the practicability of the proposed model through a classical concurrency model. Section 5 outlines the methodology used to proof some properties of concurrent programs. Section 6 discusses the proof of safety property. In section 7, the liveness property is discussed. Section 8 concludes the paper.

2 OVERVIEW on ADT and TLA

The concept of Abstract Data Types has got much attention in the field of specification of software [3, 4, 8, 11]. Such types allow to formally specify operations of a system in a non-algorithmic, *axiomatic* manner.

TLA, the Temporal Logic of Actions, is a logic for specifying and reasoning about concurrent systems [13]. Systems and their properties are described by logical formulas; the TLA formula $\Pi \Rightarrow \Phi$ asserts that the system represented by Π satisfies the property, or implements the system, represented by Φ. TLA is a linear-time temporal logic that can express liveness (eventuality) as well as safety (invariance) properties. Since TLA is a formal logic, it allows completely rigorous reasoning.

For the purposes of this paper, we can consider TLA to be ordinary predicate logic, except with two classes of variables called *rigid variables* and *flexible variables*, extended with the two operators $'$ (prime) and $\Box$. Quantification is over rigid variables only. (Full TLA also includes quantification over flexible variables, which serves as a hiding operator.) We often refer to flexible variables simply as variables.

The semantics of TLA is based on the concept of a *state*, which is an assignment of values to (flexible) variables. The meaning of a TLA formula is a set of *behaviors*, where a behavior is a sequence of states [13]. The operator $\Box$ is the standard temporal-logic "always" operator; the prime is a "next-state" operator. The complete semantics of TLA can be found elsewhere [13]; here we explain TLA informally through simple examples.

As a first example, we write a TLA formula specifying a program that starts with the variable x equal to any natural number, and keeps incrementing x by 1 forever. Letting Nat denote the set of natural numbers, the obvious way to express this with the prime and $\Box$ operators is

$$(x \in Nat) \ \wedge \ \Box(x' = x + 1) \tag{1}$$

The predicate $x \in Nat$ asserts that the value of x in the initial state is an element of Nat; the action $x' = x + 1$ asserts that the value of x in the next state is always 1 greater than its value in the current state, and the temporal formula $\Box(x' = x + 1)$ asserts that this is true for all steps—that is, for all pairs of successive states.

Formula (1) asserts that the value of x is incremented in each step of a behavior. For reasons explained in [13], we want also to allow steps that leave x unchanged. Letting $[\mathcal{A}]_{\{}$ denote $\mathcal{A} \vee (\{' = \{)$, this is expressed by the TLA formula

$$(x \in Nat) \ \wedge \ \Box[x' = x + 1]_x \ \wedge \ \mathrm{WF_x}(\mathrm{x'} = \mathrm{x} + 1) \tag{2}$$

The conjunct $\Box[x' = x + 1]_x$ asserts that every step of the program either increments x by 1 or leaves it unchanged. It allows behaviors in which x remains unchanged forever. The conjunct $\mathrm{WF_x}(\mathrm{x'} = \mathrm{x} + 1)$ expresses the liveness property that infinitely many $x' = x + 1$ steps (ones that do increment x) occur. The reader is referred to [5, 13] for an explanation of the WF operator and its definition in terms of $'$ and $\Box$.

In general, the canonical form of a TLA formula describing an algorithm is

$$Init \;\wedge\; \Box[\mathcal{A}]_{\sqsubseteq} \;\wedge\; \mathcal{F} \tag{3}$$

where *Init* is a predicate describing the initial state, $\mathcal{A}$ is an action describing how the variables may change, v is the tuple of all program variables, and F is a liveness condition. The conjunct $\Box[\mathcal{A}]_{\sqsubseteq}$ asserts that every step is either an $\mathcal{A}$ step or else leaves all variables unchanged (since a tuple is unchanged iff every component is unchanged).

3 THE MODEL

3.1 Motivations

As a specification model, we follow the same approach contained in [11, 12]. In this approach, the purely functional aspect of an abstract data type is turned on a more *procedural* one, leading to an *abstract module*, i.e. a set of *procedures* applicable to a concrete object of that type [11]. This modification in the structure of a pure ADT is needed in order to take into account some particular properties of concurrent programs. In effect, a pure ADT defines only the *functionality* of its operations: their domains and ranges, and the functional effects of their applications. When this definition is widely sufficient in sequential programs, it remains inadequate to describe some operational rules of concurrent actions. A simple example of such rules is rules about the mutual exclusive applications of access operations, that are more common in concurrent programs. Hence, it is needed to enhance the pure ADT model by an another formalism to describe such operational rules [11]. Among these potential formalisms, Temporal Logic [12] has proved itself as an appropriate tool for describing such operational or -in reality- temporal relationships. In this paper, we want to introduce a new form of Temporal Logic, namely the Temporal Logic of Actions developed by Lamport [13], into the general concept of axiomatic specification in the sense of abstract data types [8].

3.2 Logic ADT definition

The proposed model is structurally based on two elements: a *module* and a *type*. Even though a module describes the behavior of a concurrent program, a type defines data types using by that program, and specifies its operations in an axiomatic manner. The general form of these elements are as follows:

MODULE ⟨ Name ⟩ : ⟨ Name of Procedures or Actions ⟩
 Based On : ⟨ Other Types and Modules ⟩
 Methods : ⟨ List of procedures with their formal parameters ⟩
 Variables : ⟨ Local and Global variables for message passing ⟩
 Predicates : ⟨ Predicates of n-arity (n$\geq$ 0)⟩
 State Functions : ⟨ Description of the initial state of variables ⟩
 INIT$_{\mathcal{M}}$
 Actions : ⟨ TLA specification of the procedure behaviors ⟩
 Local formulas :
 $\Phi \triangleq \mathrm{INIT}_{\mathcal{M}} \wedge \Box\ [\mathcal{A}\_\mathrm{proc}_1(\cdots)\ \vee \ldots \vee \mathcal{A}\_\mathrm{proc}_k(\cdots)]_{\langle f_{\mathcal{M}} \rangle}$
 $\mathrm{f}_{\mathcal{M}} \triangleq$ ⟨ Tuples of states variables or predicates ⟩
ENDMODULE.

TYPE ⟨ Name ⟩ **Is**
  Based On : ⟨ Names of predefined Types ⟩
  SORT : ⟨ List of Set of Items ⟩
  OPNS : ⟨ List of Operation Symbols of n-arity (n$\geq$ 0) ⟩
  VAR : ⟨ List of variables ⟩
  AXIOMS : ⟨ Characteristics of OPNS symbols ⟩
ENDTYPE.

4 APPLICATION

In order to illustrate our proposed method, we take as example, the well known dining philosophers problem.

We can simulate the operating of this problem by two modules: a module which represents the behavior of philosophers according to their associated conditions, and a module which represents the state of forks during the activities of philosophers.

Type Philo **Is**
  SORT Philo's.
  OPNS
    $P_0, P_1, \ldots, P_4 : \rightarrow$ Philo's.
    NextD_ : Philo's $\rightarrow$ Philo's.
  AXIOMS
    NextD(P_0) = P_1 NextD(P_3) = P_4
    NextD(P_1) = P_2 NextD(P_4) = P_0
    NextD(P_2) = P_3
EndType.

Type Fork **Is**
  SORT fk's.
  OPNS
    $F_0, F_1, \ldots, F_4 : \rightarrow$ fk's.
EndType.

Type Phil_Steps **Is**
  SORT Phil_Phases.
  OPNS
    Eat,Think: $\rightarrow$ Phil_Phases.
EndType.

Type Fork_Steps **Is**
  SORT Fork_State.
  OPNS
    free,busy: $\rightarrow$ Fork_State.
EndType.

Type PhiloFork **Is**
  Based On
    Philo,Fork.
  OPNS

Phil_R_, Phil_L_: fk's $\rightarrow$ Philo's.
Fork_R_,Fork_L_: Philo's $\rightarrow$ fk's.

Var

f: fk's; p,q: Philo's.

AXIOMS

Phil_R (F_0) = P_0	Fork_R (P_0) = F_1
Phil_L (F_0) = P_4	Fork_L (P_0) = F_0
Phil_R (F_1) = P_1	Fork_R (P_1) = F_2
Phil_L (F_1) = P_0	Fork_L (P_1) = F_1
Phil_R (F_2) = P_2	Fork_R (P_2) = F_3
Phil_L (F_2) = P_1	Fork_L (P_2) = F_2
Phil_R (F_3) = P_3	Fork_R (P_3) = F_4
Phil_L (F_3) = P_2	Fork_L (P_3) = F_3
Phil_R (F_4) = P_4	Fork_R (P_4) = F_0
Phil_L (F_4) = P_3	Fork_L (P_4) = F_4

Phil_R (f) = NextD (Phil_L (f))
Fork_R (p) = Fork_L (NextD (p))
Phil_R (Fork_L (q)) = q
Phil_L (Fork_R (q)) = q

EndType.

MODULE Philosopher: goeat,gothink.

Based On

fk's,PhiloFork,Philo,Fork.

Methods

goeat (p:Philo's);
gothink (p:Philo's);

Variables

P_Phase : Phil_Phases;

Predicates

Phil_E(p) $\triangleq$ (P_Phase = Eat)
Phil_T(p) $\triangleq$ (P_Phase = Think)
Next(p,q) $\triangleq$ (Fork_R(p) = Fork_L(q) $\wedge$ q = NextD(p))
$\vee$ (Fork_L(p) = Fork_R(q) $\wedge$ p = NextD(q))

State Functions

F$_p$ $\triangleq$ $\langle P\_Phase, Phil\_E(p), Phil\_T(p)\rangle$
INIT$_{philo}$ $\triangleq$ $\wedge$ $\forall$ p $\in$ Philo's: Phil_T(p)
$\wedge$ $\forall$ p,q $\in$ Philo's: Next(p,q)
$\Rightarrow$ Phil_T(p) $\wedge$ Phil_T(q)

Actions

goeat(p) $\triangleq$ $\wedge$ Phil_T(p)
$\wedge$ Takeup(Fork_L(p)) $\wedge$ Takeup(Fork_R(p))
$\wedge$ $\neg$ Phil_T$'$(p) $\wedge$ Phil_E$'$(p)
$\wedge$ $\forall$ q $\in$ Philo's: Next(p,q) $\Rightarrow$
$\wedge$ $\neg$ Phil_E$'$(Phil_L(Fork_L(p)))
$\wedge$ $\neg$ Phil_E$'$(Phil_R(Fork_R(p)))

gothink(p) $\triangleq$ $\wedge$ Phil_E(p)

$\wedge$ Putdown(Fork_L(p)) $\wedge$ Putdown(Fork_R(p))
$\wedge \neg$ Phil_E$'$(p)
$\wedge$ Phil_T$'$(p)
$\wedge\ \forall$ q $\in$ Philo's: Next(p,q) $\Rightarrow$
  $\wedge \neg$ Phil_E$'$(Phil_L(Fork_L(p)))
  $\wedge \neg$ Phil_E$'$(Phil_R(Fork_R(p)))

ACT_Philo(p) $\triangleq$ goeat(p) $\vee$ gothink(p)

Temporal Formula

$\Phi_{philo} \triangleq$ INIT$_{philo}$ $\wedge$
$\Box$ [ACT_Philo(p)]$_{\langle P\_Phase, Phil\_E(p), Phil\_T(p)\rangle}$

ENDMODULE.

MODULE forks: Putdown,Takeup.

Based On

Fork_Steps,Fork.

Methods

Putdown(f:fk's);
Takeup(f:fk's);

Variables

ForkPhase:Fork_State;

Predicates

Forkdispo(f) $\triangleq$ (ForkPhase = dispo)
Forkbusy(f) $\triangleq$ (ForkPhase = busy)

State Functions

F$_f$ $\triangleq$ $\langle ForkPhase, Forkdispo(f), Forkbusy(f)\rangle$
INIT$_{fork}$ $\triangleq$ $\forall$ f $\in$ fk's: Forkdispo(f)

Actions

Takeup(f) $\triangleq$ $\wedge$ Forkdispo(f)
$\wedge$ Forkbusy$'$(f)
$\wedge \neg$ Forkdispo$'$(f)

Putdown(f) $\triangleq$ $\wedge$ Forkbusy(f)
$\wedge$ Forkdispo$'$(f)
$\wedge \neg$ Forkbusy$'$(f)

ACT_Fork(f) $\triangleq$ Takeup(f) $\vee$ Putdown(f)

Temporal Formula

f$_{fork}$ $\triangleq$ $\langle$ Forkdispo(f) , Forkbusy(f) $\rangle$
Φ_{fork} $\triangleq$ INIT$_{fork}$ $\wedge \Box$ [ACT_Fork(f)]$_{\langle F_f\rangle}$

ENDMODULE.

5 PROOF METHODOLOGY

In the two following sections, we describe how safety and liveness properties are proved in our model. To achieve such as proofs, we use the method of structuring proofs introduced in [5]. A hierarchical structured proof is a sequence of steps, each with a proof. The proof of a step is either a short paragraph or calculation, or else a hierarchically structured proof. The idea is to make the unstructured "leaf" proofs sufficiently easy to check that they are highly unlikely to be wrong. We indicate how proofs of our theorems are carried down to the level at which leaf proof consists of simple expansion of definitions and propositional logic.

For example, in the section devocated to liveness property, we highlight two different kinds of reasoning in TLA proofs: steps 1, 2 and 3 illustrate the *action reasoning*, not involving temporal operators; step 4 illustrates *temporal reasoning*. Action reasoning is almost always the longest and most difficult part of a proof, so we decide, like in [5], to use separate encoding for the action and temporal reasoning.

6 SAFETY PROPERTY

Having expressed the Dining Philosophers problem as a Logic Abstract Data Type, we now consider how to express and prove properties of such a problem. A property is expressed by a TLA formula F. The assertion "Program Φ has property F" is expressed in TLA by the validity of the formula $\Phi \Rightarrow F$, which asserts that every behavior satisfying Φ satisfies F. In concurrent program, two popular classes of properties are generally considered: *invariance* and *eventuality*. In this section, we discuss the first one.

An invariance property is expressed by a TLA formula $\Box P$, where P is a predicate. Among the invariance properties, we consider the deadlock freedom: P asserts that the program is not deadlocked. As a case of deadlock freedom, we consider the following property (P1):

"Two neighbor philosophers not eat simultaneously".

We can rewrite this property as a TLA formula:

$$\forall p, q \in \text{ Philo's: Next(p,q) } \Rightarrow \neg \text{ (Phil\_E(p) } \wedge \text{ Phil\_E(q))}$$

Proof of this invariance property comes to proof the following theorem:

$$\Phi_{philo} \Rightarrow \Box(\forall p, q \in \text{ Philo's: Next(p,q) } \Rightarrow \neg \text{ (Phil\_E(p) } \wedge \text{ Phil\_E(q))}$$

Proof:

Let p and q two philosophers; we pose:

$$\text{INV(p,q)} \triangleq \text{(Next(p,q) } \Rightarrow \neg \text{ (Phil\_E(p) } \wedge \text{ Phil\_E(q))}$$

The following rule INV1 of TLA:

$$\frac{I \wedge [\mathcal{N}]_f \Rightarrow I'}{I \wedge \Box[\mathcal{N}]_f \Rightarrow \Box I}$$

tells us that we must prove:

$$\text{INIT}_{philo} \Rightarrow \text{INV(p,q)} \tag{4}$$

$$\text{INV(p,q)} \wedge [ACT\_Philo(p)] \wedge [ACT\_Philo(q)] \Rightarrow \text{INV}'\text{(p,q)} \tag{5}$$

First, we consider the proof of (4): $\text{INIT}_{philo} \Rightarrow \text{INV(p,q)}$.

In a first time, we expand the definition of INIT_{philo}, then we give the proof:

$$
\begin{aligned}
\text{INIT}_{philo} \quad &\equiv \quad \wedge\ \forall\ \text{p} \in \text{Philo's : Phil\_T(p)} \qquad\qquad \text{(By definition)}\\
&\qquad \wedge\ \forall\ \text{p,q} \in \text{Philo's: Next(p,q)} \Rightarrow \text{Phil\_T(p)} \wedge \text{Phil\_T(q)}\\
&\Rightarrow \quad \forall\ \text{p,q} \in \text{Philo's: Next(p,q)} \Rightarrow \text{Phil\_T(p)} \wedge \text{Phil\_T(q)}
\end{aligned}
$$

Using the definition of predicates Phil_T(p) and Phil_E(p) in Philosopher module: (Phil_T(p) $\Rightarrow \neg$ Phil_E(p)), and propositional logic, we obtain:

$$
\begin{aligned}
&\Rightarrow \quad \forall\ \text{p,q} \in \text{Philo's: Next(p,q)} \Rightarrow \neg\ \text{Phil\_E(p)} \wedge \neg\ \text{Phil\_E(q)}\\
&\Rightarrow \quad \forall\ \text{p,q} \in \text{Philo's: Next(p,q)} \Rightarrow \neg\ \text{Phil\_E(p)} \vee \neg\ \text{Phil\_E(q)}
\end{aligned}
$$

Second, we consider the proof of (5):

$$\text{INV(p,q)} \wedge [\text{goeat(p)} \vee \text{gothink(p)}] \wedge [\text{goeat(q)} \vee \text{gothink(q)}] \Rightarrow \text{INV}'\text{(p,q)}$$

By definition, we have:

$$\text{INV}'\text{(p,q)} \triangleq \text{Next(p,q)} \Rightarrow \neg\ [\text{Phil\_E}'\text{(p)} \wedge \text{Phil\_E}'\text{(q)}]$$

To prove INV'(p,q), we consider the following assumptions:

$$
\begin{aligned}
&\wedge\ INV(p,q)\\
&\wedge\ [\text{goeat(p)} \vee \text{gothink(p)}]\\
&\wedge\ [\text{goeat(q)} \vee \text{gothink(q)}]
\end{aligned}
$$

Let T = INV(p,q) $\wedge$ [goeat(p) $\vee$ gothink(p)] $\wedge$ [goeat(q) $\vee$ gothink(q)].
How do we prove INV'(p,q)? To find the answer, we use a proof by case analysis:

$$
\begin{aligned}
\text{INV(p,q)} \wedge \text{goeat(p)} &\Rightarrow \text{INV}'\text{(p,q)} && (6)\\
\text{INV(p,q)} \wedge \text{gothink(p)} &\Rightarrow \text{INV}'\text{(p,q)} && (7)\\
\text{INV(p,q)} \wedge \text{goeat(q)} &\Rightarrow \text{INV}'\text{(p,q)} && (8)\\
\text{INV(p,q)} \wedge \text{gothink(q)} &\Rightarrow \text{INV}'\text{(p,q)} && (9)\\
\text{INV(p,q)} \wedge &\\
\langle \text{Phil\_T(p),Phil\_E(p)} \rangle' = \langle \text{Phil\_T(p),Phil\_E(p)} \rangle &\Rightarrow \text{INV}'\text{(p,q)} && (10)
\end{aligned}
$$

In the following, we give the proof of (6) and (7); the others are equally equivalent.
First, we consider the proof of (6):T=INV(p,q) $\wedge$ goeat(p) $\Rightarrow$ INV'(p,q).

The proof becomes:

$$
\begin{aligned}
T \quad \equiv \quad &\wedge\ (\neg Phil\_E(p) \vee \neg Phil\_E(q)) \qquad\qquad \text{(By hypothesis of INV(p,q))}\\
&\wedge\ Phil\_T(p) \qquad \text{(By definition of goeat(p))}\\
&\wedge\ Takeup(Fork\_L(p))\ \wedge\ Takeup(Fork\_R(p))\\
&\wedge\ \neg\ Phil\_T'(p)\ \wedge\ \neg\ Phil\_E'(p)\\
&\wedge\ \forall \text{q} \in \text{Philo's: Next(p,q)} \Rightarrow\\
&\qquad \wedge\ \neg\ Phil\_E'(Phil\_L(Fork\_L(p)))
\end{aligned}
$$

$$\wedge\neg\ Phil\_E^{'}(Phil\_R(Fork\_R(p)))$$

$$\Rightarrow\ \wedge\neg\ Phil\_E(p))$$
$$\wedge\ Phil\_E^{'}(p)$$
$$\wedge\ \forall \text{q} \in \text{Philo's: Next(p,q)} \Rightarrow$$
$$\wedge\neg\ Phil\_E^{'}(Phil\_L(Fork\_L(p)))$$
$$\wedge\neg\ Phil\_E^{'}(Phil\_R(Fork\_R(p)))$$

By hypothesis, we have Next(p,q) $\equiv$ True; if we suppose the case where:

$$\wedge\ \text{q} = \text{NextD(p)} \tag{11}$$
$$\wedge\ \text{Fork\_R(p)} = \text{Fork\_L(q)} \tag{12}$$

Then, we obtain:

$$\Rightarrow\ \wedge\neg\ \text{Phil\_E(p)}$$
$$\wedge\ \text{Phil\_E}^{'}\text{(p)}$$
$$\wedge\neg\ \text{Phil\_E}^{'}\text{(Phil\_L(Fork\_L(p)))}$$
$$\wedge\neg\ \text{Phil\_E}^{'}\text{(Phil\_R(Fork\_R(p)))}$$
$$\Rightarrow\ \neg\ \text{Phil\_E}^{'}\text{(Phil\_R(Fork\_L(q)))}$$
$$\Rightarrow\ \wedge\neg\ \text{Phil\_E}^{'}\text{(q)}$$

If we change the assumptions (11 and 12) of previous case, by:

$$\wedge\ \text{p} = \text{NextD(q)}$$
$$\wedge\ \text{Fork\_L(p)} = \text{Fork\_R(q)}$$

the proof gives as final result the following formula:

$$\Rightarrow\neg\ \text{Phil\_E}^{'}$$

Now, we give the proof of (7): $T' = INV(p,q) \wedge gothink(p) \Rightarrow INV^{'}(p,q)$.

The proof becomes:

$$T^{"}\ \equiv\ \wedge\ (\neg\ \text{Phil\_E(p)} \vee \neg\ \text{Phil\_E(q)})\qquad \text{(By hypothesis of INV(p,q))}$$
$$\wedge\ \text{Phil\_E(p)}\qquad \text{(By definition of gothink(p))}$$
$$\wedge\ \text{Putdown(Fork\_L(p))}\ \wedge\ \text{Putdown(Fork\_R(p))}$$
$$\wedge\ \text{Phil\_E}^{'}\text{(p)}$$
$$\wedge\ \text{Phil\_T}^{'}\text{(p)}$$
$$\wedge\ \forall\ \text{q} \in \text{Philo's: Next(p,q)} \Rightarrow$$
$$\wedge\neg\ \text{Phil\_E}^{'}\text{(Phil\_L(Fork\_L(p)))}$$
$$\wedge\neg\ \text{Phil\_E}^{'}\text{(Phil\_R(Fork\_R(p)))}$$

$$\Rightarrow\ \wedge\neg\ \text{Phil\_E(q)}$$

(Because Phil_E(p)=True by definition and $\neg$ Phil_E(p)=false)
$\wedge \neg$ Phil_E$'$(p)
$\wedge\ \forall$ q $\in$ Philo's: Next(p,q) $\Rightarrow$
$\wedge \neg$ Phil_E$'$(Phil_L(Fork_L(p)))
$\wedge \neg$ Phil_E$'$(Phil_R(Fork_R(p)))

By hypothesis of:

$$\wedge \quad Next(p,q)$$
$$\wedge \text{ q} = \text{NextD(p)} \ \wedge \ \text{Fork\_R(p)} = \text{Fork\_L(q)}$$

The proof becomes:

$$\Rightarrow \quad \begin{array}{c} \wedge \neg \text{ Phil\_E(q)} \\ \wedge \neg \text{ Phil\_E}'\text{(p)} \\ \wedge \neg \text{ Phil\_E}'\text{(Phil\_L(Fork\_L(p)))} \\ \wedge \neg \text{ Phil\_E}'\text{(Phil\_R(Fork\_R(p)))} \end{array}$$

$$\Rightarrow \quad \begin{array}{c} \wedge \neg \text{ Phil\_E(q)} \\ \wedge \neg \text{ Phil\_E}'\text{(p)} \\ \wedge \neg \text{ Phil\_E}'\text{(Phil\_L(Fork\_L(p)))} \\ \wedge \neg \text{ Phil\_E}'\text{(q)} \end{array}$$

$$\Rightarrow \quad \wedge \neg \text{ Phil\_E}'\text{(p)} \wedge \ \neg \text{ Phil\_E}'\text{(q)}$$

The proofs of (8), (9) and (10) are equally similar..

7 LIVENESS PROPERTY

We now describe how a liveness property is proved in our model. We try to prove the fact that if a philosopher thinks and want to eat, it shall be eat. Such as property can be described by the syntactic formula: $\forall p \in Philo's : Phil\_T(p) \Rightarrow \Diamond Phil\_E(p)$. Here, we remark that the temporal formula contained in the Philosopher module specifies only safety properties. Hence, we enhance this formula by a weak fairness condition for the *goeat(p)* action. Then, we obtain the following formula:

$$WF_{\langle P\_Phase, Phil\_T(p), Phil\_E(p)\rangle}\ (goeat(p)\ \triangleq$$
$$(\Box \Diamond \neg\ Enabled(goeat(p))\ \vee\ \Box \Diamond\ (goeat(p))$$

that can be rewritten as:

$$WF_{\langle Phil\_T(p), Phil\_E(p)\rangle}\ (goeat(p))\ \triangleq$$
$$(\Box \Diamond \neg\ Enabled(goeat(p))\ \Rightarrow\ \Box \Diamond\ (goeat(p))$$

Hence, the new temporal formula of Philosopher module is:

$$\begin{array}{rcl} \Phi_{philo} & \triangleq & INIT_{philo} \\ & & \wedge\ \Box\ [ACT\_Philo(p)]_{\langle P\_Phase, Phil\_E(p), Phil\_T(p)\rangle} \\ & & \wedge\ WF_{\langle Phil\_T(p), Phil\_E(p)\rangle} \end{array}$$

Now, we are in position to present the complete proof.

Theorem: $\Phi_p \Rightarrow (Phil\_T(p) \leadsto Phil\_E(p))$

1. $Phil\_T(p) \wedge [ACT\_Philo(p)]_{\langle Phil\_E(p), Phil\_T(p)\rangle} \Rightarrow Phil\_T\,'(p) \vee Phil\_E'(p)$

 Proof : We assume (1) $Phil\_T(p) \wedge [ACT\_Philo(p)]_{\langle Phil\_E(p), Phil\_T(p)\rangle}$ and prove $Phil\_T'(p) \vee Phil\_E'(p)$. If we pose $f \triangleq \langle Phil\_E(p), Phil\_T(p)\rangle$, the assumption (1) can be rewritten as three conjunctions:
 (C1) $Phil\_T(p) \wedge$ goeat(p); (C2) $Phil\_T(p) \wedge$ gothink(p); and (C3) $Phil\_T(p) \wedge$ f′=f.

 1.1. (C1) $Phil\_T(p) \wedge$ goeat(p) $\Rightarrow Phil\_T'(p) \vee Phil\_E'(p)$

 Proof : By definition of goeat(p) we have:

$$
\begin{aligned}
Phil\_T(p) \wedge goeat(p) \;\equiv\;& Phil\_T(p) \wedge Phil\_T(p) \\
& \wedge Takeup(Fork\_L(p)) \wedge Takeup(Fork\_R(p)) \\
& \wedge \neg\ Phil\_T'(p) \\
& \wedge\ Phil\_E'(p) \\
& \wedge \forall\ q\ \in\ Philo's : Next(p,q) \Rightarrow \\
& \qquad \wedge \neg\ Phil\_E'(Phil\_L(Fork\_L(p))) \\
& \qquad \wedge \neg\ Phil\_E'(Phil\_R(Fork\_R(p)))
\end{aligned}
$$

 qed $\Rightarrow$ goeat(p) $\Rightarrow Phil\_E'(p)$
 $\Rightarrow Phil\_T'(p) \vee Phil\_E'(p)$

 1.2. (C2) $Phil\_T(p) \wedge$ gothink(p) $\Rightarrow Phil\_T'(p) \vee Phil\_E'(p)$

 Proof : By definition of gothink(p) we have:

$$
\begin{aligned}
Phil\_T(p) \wedge gothink(p) \;\equiv\;& Phil\_T(p) \wedge Phil\_E(p) \\
& \wedge\ Putdown(Fork\_L(p)) \wedge Putdown(Fork\_R(p)) \\
& \wedge \neg\ Phil\_E'(p) \\
& \wedge\ Phil\_T'(p) \\
& \wedge \forall\ q\ \in\ Philo's : Next(p,q) \Rightarrow \\
& \qquad \wedge \neg\ Phil\_E'(Phil\_L(Fork\_L(p))) \\
& \qquad \wedge \neg\ Phil\_E'(Phil\_R(Fork\_R(p)))
\end{aligned}
$$

Assumption (C2) and formula $Phil\_T(p) \wedge Phil\_E(p)$ = False lead to a contradiction. To avoid this, we take the negation of (C2) as new assumption: $\neg$ $Phil\_T(p)$ $\vee$ $\neg$ gothink(p). Since $Phil\_T(p)$=True, we consider only the second term of the above negation. This term gives:

$$
\begin{aligned}
\neg\, gothink(p) \;\equiv\; & \vee \neg\, Phil\_E(p) \\
& \vee \neg\, (Putdown(Fork\_L(p)) \wedge Putdown(Fork\_R(p))) \\
& \vee\, Phil\_E'(p) \\
& \vee \neg\, Phil\_T'(p) \\
& \vee \neg\, (\forall\, q\; \in\; Philo's : Next(p,q) \Rightarrow \\
& \qquad \wedge \neg\, Phil\_E'(Phil\_L(Fork\_L(p))) \\
& \qquad \wedge \neg\, Phil\_E'(Phil\_R(Fork\_R(p))))
\end{aligned}
$$

qed $\Rightarrow$ $Phil\_E'(p)$.

1.3. (C3) $Phil\_T(p) \wedge f'=f \Rightarrow Phil\_T'(p) \vee Phil\_E'(p)$

Proof : We obtain:

$$
\begin{aligned}
Phil\_T(p) \wedge f' = f \;\equiv\; & \wedge\, Phil\_T(p) \\
& \wedge\, \langle\, Phil\_E(p), Phil\_T(p)\rangle' = \langle\, Phil\_E(p), Phil\_T(p)\rangle \\
\equiv\; & \wedge\, Phil\_T(p) \\
& \wedge\, Phil\_T'(p) = Phil\_T(p) \\
& \wedge\, Phil\_E'(p) = Phil\_E(p)
\end{aligned}
$$

By hypothesis, $Phil\_T(p)$=True; consequently, $Phil\_E(p)$=False. Then, we obtain:

$\Rightarrow$ $Phil\_T(p) \wedge Phil\_T'(p) \wedge \neg\, Phil\_E'(p)$
$\Rightarrow$ $Phil\_T'(p)$
qed $\Rightarrow$ $Phil\_T'(p) \vee Phil\_E'(p)$.

2. $Phil\_T(p) \wedge [ACT\_Philo(p) \wedge \text{goeat(p)}]_{\langle Phil\_E(p), Phil\_T(p)\rangle} \Rightarrow Phil\_E'(p)$

Proof :

$$\begin{aligned}
Phil_T(p) \wedge goeat(p) \;\equiv\;& Phil_T(p) \wedge goeat(p) \\
& \wedge \langle Phil_E(p), Phil_T(p) \rangle' \\
& \wedge \neg\; \langle Phil_E(p), Phil_T(p) \rangle \\
\equiv\;& \wedge\; Phil_T(p) \wedge goeat(p) \\
& \wedge\;\; \vee\; Phil_E'(p) \neq Phil_E(p) \wedge Phil_T'(p) \neq Phil_T(p) \\
& \quad \vee\; Phil_E'(p) \neq Phil_E(p) \wedge Phil_T'(p) = Phil_T(p) \\
& \quad \vee\; Phil_E'(p) = Phil_E(p) \wedge Phil_T'(p) \neq Phil_T(p)
\end{aligned}$$

Since *Phil_T(p)*=True by hypothesis, the above proof can be simplified as follows:

$$\begin{aligned}
\equiv\;& \wedge\; Phil_T(p) \wedge goeat(p) \\
& \wedge \vee\; Phil_E'(p) \wedge \neg\; Phil_T'(p) \quad \{= True\} \\
& \quad \vee Phil_E'(p) \wedge Phil_T'(p) \quad \{= False\} \\
& \quad \vee\; \neg\; Phil_E'(p) \wedge \neg\; Phil_T'(p) \quad \{= False\}
\end{aligned}$$

$$\Rightarrow \neg\; Phil\_T'(p) \wedge Phil\_E'(p)$$

qed *Phil_E'(p).*

3. *Phil_T(p)* $\Rightarrow$ Enabled(goeat(p))$_{\langle Phil\_E(p), Phil\_T(p) \rangle}$
Predicate Enabled(goeat(p))$_{\langle Phil\_E(p), Phil\_T(p) \rangle}$ stipulates that is possible to execute the operation corresponding to the action goeat(p)$_{\langle Phil\_E(p), Phil\_T(p) \rangle}$. This possibility can be written as follows:

Proof :

$$\begin{aligned}
Enabled(goeat(p))_{\langle Phil_E(p), Phil_T(p) \rangle} \;\equiv\;& \vee\; Phil_T(p) \\
& \quad \wedge\; Takeup(Fork_L(p)) \wedge Takeup(Fork_R(p)) \\
& \vee\; Phil_T(p) \\
& \quad \wedge \forall\; q \;\in\; Philo's : Next(p, q) \Rightarrow Phil_T(q) \\
& \vee\; Phil_E(p) \\
& \quad \wedge \forall\; q \;\in\; Philo's : Next(p, q) \Rightarrow Phil_T(q) \\
& \vee\; Phil_T(p)
\end{aligned}$$

Intuitively speaking, the third disjunction means that a philosopher p eats, while these neighbors are waiting (the forks). Then, p terminates but it not takedown the forks; instead of takedowning the forks, it just looks of his neighbors and pick up its eating. In this case, we can use the simplication rule: $A \vee (A \cup B) \equiv A$.

$$\begin{aligned}
Enabled(goeat(p))_{\langle Phil_E(p), Phil_T(p) \rangle} \;\equiv\;& \\
& Phil_T(p) \vee (Phil_E(p) \\
& \wedge \forall\; q \;\in\; Philo's : \\
& \quad Next((p, q) \Rightarrow Phil_T(q)
\end{aligned}$$

qed *Phil_T(p)* $\Rightarrow$ Enabled(goeat(p))$_{\langle Phil\_E(p), Phil\_T(p) \rangle}$ = True

4. $\Box[ACT\_Philo(p)]_{\langle Phil\_E(p), Phil\_T(p)\rangle} \wedge WF_{F_{\langle goeat(p)\rangle}} \Rightarrow$
$(Phil\_T(p) \leadsto Phil\_E(p))$

Proof :By steps 1,2 and 3.

qed The theorem follows: $\Phi_p \Rightarrow \Box(Phil\_T(p) \Rightarrow \Diamond Phil\_E(p))$.

8 Conclusion

In this paper, we presented an hybrid specification model that combines both an algebraic model for specification (the ADT model) and a temporal model for verification (the TLA logic). We used the well known example of eating philosophers to illustrate the practical use of our model. We give, in turn, the specification of this example in terms of Logic Abstract Modules, then we verify two important properties, namely the safety and liveness properties.

References

[1] E.A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10:110-135, February 1975.

[2] J. Desharnais, A. Mili, R. Mili, J. Mullins, and Y. Slimani., Semantics of Concurrent Programming. In *Parallel & Distributed Computing Handbook, Albert Y.H. Zomaya (Ed.)*, Mc Graw-Hill Series on Computer Engineering, 1996.

[3] H. Ehrig, B. Mahr., Fundamentals of Algebraic Specification 1. Equations and Initial Semantics. *EATCS Monographs on Theoretical Computer Science*, 6, Springer-Verlag, 1985.

[4] H. Ehrig, B. Mahr., Fundamentals of Algebraic Specification 2. Module Specifications and Constraints. *EATCS Monographs on Theoretical Computer Science*, 21, Springer-Verlag, 1990.

[5] U. Engberg, P. Gronning and L. Lamport. Mechanical verification of concurrent systems with TLA. In *Computer-Aided Verification*, Lecture Notes in Computer Science, Springer-Verlag, 1992.

[6] S.J. Garland and J.V Guttag. An overview of LP, the Larch Prover. In N. Dershowitz, editor, *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, 355:137-151, Springer-Verlag, 1989.

[7] N. Gehany and A.D. McGettrick (Eds)., Software specification techniques. *Addison-Wesley*, 1986.

[8] A. Gogen, J.W Thatcher, E.G. Wagner., An initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types. In *Current Trends in Programming Methodology, R. Yeh (Ed.)*, 4:80-149, Prentice-Hall, 1978.

[9] B. Jonsson., Compositional Specification and Verification of Distributed Systems. *ACM Trans. of Prog. Lang. and Systems*, 16:259-303, March 1994.

[10] S. Kaplan., Algebraic specification of concurrent systems. *Theoretical Computer Science*, 69:69-115, 1989.

[11] F. Kröger., Abstract modules: Combining algebraic and temporal logic specification means. *Techniques et Sciences Informatiques (TSI)*, 6:559-573, 1987.

[12] L. Lamport., Specifying Concurrent Program Modules. *ACM Trans. on Prog. Lang. and Systems*, 5:190-222, 1983.

[13] L. Lamport., The Temporal Logic of Actions. *ACM Trans. on Prog. Lang. and Systems*, 16:872-923, 1994.

[14] Y. Slimani, E.H Daho. A TLA-based semantic model for specifying and verifying message-passing programs. Technical Report, Computer Science Institute, Univ. USTO Oran (Algeria), September 1997.