Inductive Theorem Prover Based Verification of Concurrent Algorithms

Abdessamad Imine[1], **Yahya Slimani**[2] **and Sorin Stratulat**[3]

[1] Dépt. Informatique - Faculté des Sciences
Université des Sciences et de la Technologie d'Oran
B.P. 1505, El-M'naouar, 31000 Oran, ALGERIA.
e-mail: imine_a@mail.univ-usto.dz

[2] Dépt. Informatique - FST- Campus Universitaire - 1060 Tunis, TUNISIA.
e-mail: yahya.slimani@fst.rnu.tn

[3] INRIA. 2004 route des Lucioles – BP 93, 06902 Sophia Antipolis, FRANCE
e-mail: Sorin.Stratulat@inria.fr

Abstract

This paper describes a contribution to the area of mechanically theorem proving focusing especially on the automation of the invariance proof. In this respect, we present an initial version of a tool for automatically checking the invariance property of concurrent algorithms. Input to the tool consists of the soundness properties, expressed in TLA (Temporal Logic of Actions) that is an action-based linear-time temporal logic for specifying and verifying concurrent systems. The tool builds a transition system model expressing the semantics of TLA. A transition system consists of a set of system states and a set of mappings (actions) from system states to system states. This model is described in terms of conditional equations. As verification back-end to our tool we use SPIKE, an automated induction-based theorem prover, which is suitable for reasoning about theories where axioms are conditional equations. Indeed, the invariance properties of the transition system are checked as a SPIKE theorems. Moreover, the proved theorems can be used as lemmas while reasoning about other invariance properties. The utility of this tool is illustrated by an invariance proof for the Bakery mutual exclusion algorithm.

Keywords: Concurrent Algorithms, Inductive theorem prover, Invariance properties, Mechanical verification, Temporal Logic of Actions.

1 Introduction

Although computer programming is potentially an exact science, few programs are actually subjected to formal verification. Proofs of specification for even the simplest programs are often difficult to produce and unmanageably complicated. Hand-generated proofs may contain errors that are no easier to detect than programming errors. This shows the importance of automatic proof systems for reasoning formally about programs. The ultimate practicality of program verification will be determined by the extent to which the programmer can be relieved of the details of generating and checking correctness proofs through the application of mechanical proofs.

However, effective use of today's theorem provers typically requires expertise that is uncommon among software engineers. Detailed knowledge of prover's heuristics is often necessary in order to guide it successfully through a proof. Moreover, the classical logics underlying these systems are unsuitable for modelling some aspects of real programs, such as nondeterminism and concurrency. SPIKE [2, 3, 8], as described in Section 2, is an induction-based theorem prover, but nonetheless susceptible to these criticisms. This prover is suitable for theories whose axioms are conditional equations. Based on induction and rewriting for reasoning about algebraic specifications, this system provides no natural representation of concurrent systems.

Considerable research [6, 16, 15] has been devoted to the development of alternative formalisms for program specification and verification based on temporal logic and the state-transition program model. This approach allows natural representations of nondeterministic concurrent systems and their execution properties. For instance, TLA is a logic suited for reasoning about concurrent systems [14, 15]. Their behaviour and properties can be

expressed directly as formulas in the TLA logic, rather than programs in a separate programming language. However, reasoning about systems at the logic level involves a lot of tedious proof details. Hence, a step forward is done if the reasoning could be (partly) automated using a theorem prover.

Our goal is to combine the expressiveness of temporal logic with the power of inductive theorem proving in the design of a simple concurrent algorithm verification system. Our specification framework, as presented in Section 3, is a *simple*[1] TLA algorithms, grounded in the SPIKE logic.

In Section 4, we describe a scheme for formally representing algorithms and their execution states as terms in this logic. A first step consists in the definition of the semantics of TLA as a *transition system* model. A transition system consists of a set of system states and a set of mappings from system states to system states called *actions*. An action is selected and applied to the current state, producing a new state. Once the transition system is built, it is described in terms of conditional equations easy to specify in SPIKE. The invariance properties of the transition system are then checked as a SPIKE conjecture. Moreover, the proved conjectures can be used as lemmas while reasoning about other invariance properties.

We have designed a translator tool, as presented in Section 5, which makes easier reasoning about TLA specifications. Indeed, writing specifications and theorems directly in TLA avoids the errors that can be introduced when hand translating what one wants to prove into the SPIKE language.

In Section 6, we applies our encoding to the Bakery mutual exclusion algorithm for proving an invariance property. Finally, Section 7 concludes the paper with a comparison between different theorem provers able to reason on TLA specifications and possible directions for future work.

2 The Inductive Theorem Proving Paradigm

Theorem proving is a tool of developing and verifying mechanically mathematical proofs. The specification languages used (higher or first order logics) allow to define usual mathematical objects (sets, functions, ...) and can be generally understood as a mixture of predicate calculus, recursive definitions and inductively defined types. These languages are strong enough to model systems and express properties on them. Theorem provers provide an interactive environment for developing proofs using a set of tactics (elementary proof steps) and tacticals (combination of tactics). Possible tactics are implementations of either a deduction rule, rewriting rule, induction scheme or decision procedure [4].

Inductive theorem proving is the paradigm based on *inductive reasoning*. This one is simply a method of performing inferences in domains where there exists a well-founded relation on the objects. It is fundamental when proving properties of numbers, data-structures, or programs axiomatized by a set of equations and/or conditional axioms. As opposed to deductive theorems, inductive theorems are usually valid only in some particular models of the axioms [2]. As a classical example, consider the data-structure of nonnegative integers built up using the constant 0 and the successor function S. Every element of this structure can be represented by a variable-free (*ground*) term that involves 0 and S only. Suppose now we define the addition operation $+$ by the following axioms (implicitly universally quantified): i) $x + 0 = x$, and ii) $x + S(y) = S(x + y)$.

Clearly, adding two integers, using the above equations, yields a nonnegative integer. For instance, $(0+S(0))+0$ equates to $S(0)$ by deductive reasoning: apply just once the second axioms and twice the first. However, $0+x = x$ is not a deductive consequence of the above equations. This is a typical example of an identity whose proof requires some kind of induction. To prove it, we can use the following induction scheme:

Basic Case: $0 + 0 = 0$

Induction Step: $0 + x = x$ implies $0 + S(x) = S(x)$

The proof of Basic Case is trivial, and the proof of Induction Step follows immediately from the defining axioms and the induction hypothesis $0 + x = x$.

2.1 SPIKE

In this section, we describe the main features of SPIKE[2] [8], the automated deduction system we have used for reasoning on TLA specifications.

The specification language of SPIKE allows to define functions axiomatically. Proofs are performed in a theory where axioms are built from first-order conditional equations and goals to be proved are equational clauses. The

[1] We do not consider the hiding variables, the fairness concept and the refinement mappings.

[2] The binaries of SPIKE are accessible at
`http://www.loria.fr/~stratula/Spike_distr/spike.tgz`.

equations are oriented as rewrite rules using a well-founded order. This order serves also for supporting inductive reasoning. The signature of the theory is assumed to be many-sorted and partitioned into *constructors* and defined symbols (*operators*). When for all possible ground arguments the result of the defined operators can be expressed only in terms of constructors and variables, we say that these operators are completely defined *w.r.t* the constructors and that the specification is *sufficiently complete*. This requirement is very natural when building specifications in a structured way. Moreover when the evaluation of any ground expression terminates and produces a unique result, irrespective of the computation strategy, we say that the specification is *ground convergent*.

SPIKE has a deduction mechanism based on *induction*. Its inference rules transform a conjecture from the current state of the proof to a (potentially empty) set of new conjectures. Two main steps to transform a conjecture can be performed: i) (**Generate** rules) the instantiation of a subset of its variables (called *induction variables*) with elements describing their sorts (called *test sets*), and ii) (**Simplify** rules) its simplification or elimination by axioms and (instances of) other 'not yet proved' conjectures, which play the role of induction hypotheses. The treated conjecture may be stored for its later use to further computations. During the proof, it is not required any hierarchy between the conjectures and the proof method allows for *simultaneous induction* [2, 3]. SPIKE has been developed on this principle and incorporates optimizations such as powerful simplification techniques based on conditional rewriting and decision procedures. The system can be also disprove non-trivial false conjectures.

Proofs are controlled with the help of *strategies*. Strategies in SPIKE allow to choose the application order of the inference rules on the treated conjecture, and also the application parameters of these rules. The way the rules are applied is critical for the success of a proof.

Recently, in [20, 1] it was proposed an integration schema of a linear arithmetic decision procedure in SPIKE [20]. This permits the reduction of the search space without the use of some explicitly stated lemmas such as the transitivity axioms. Thanks to it, SPIKE begins to be applied to the verification of complex examples [19].

3 Overview on TLA

The Temporal Logic of Actions (TLA) is a logic for reasoning about concurrent algorithms [14, 15]. One of the main ideas of TLA is that algorithms are expressed directly in the logic, rather in a separate programming language. TLA is based on a simple temporal logic, with $\Box$ ("always") as the only primitive temporal operator.

An *action* is a boolean expression that states how initial and final states are related. As an example:

$$(x' = x + 1) \wedge (y' = y) \tag{1}$$

is an action that increments x by 1 and leaves y unchanged. Thus primed variables always stand for final states.

Predicates can always be interpreted as actions (as actions, they put no restrictions on the final states). Similarly, both predicates and actions can be interpreted as temporal formulas. Indeed, a predicate asserts something about the state at time 0 while an action relates the states at time 0 and time 1.

If $\mathcal{A}$ is an action and f is a state function, then $[\mathcal{A}]_f$ is the action $\mathcal{A} \vee (f = f')$ where f' is the same as f but with all occurrences of program variables primed. As example, if $\mathcal{A}$ is the formula (1) then the action $[\mathcal{A}]_{(x,y)}$ enables arbitrary changes of all other variables than x and y. In this way stuttering can be modelled.

TLA assumes that there is infinite supply of program variables, though a given algorithm always mentions only a finite number of them. The values that variables can take are not organized in types. Instead, TLA assumes that there exists only one single set of values, and a variable can take any value from this set.

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

$$Init \wedge \Box[\mathcal{N}]_v \wedge L \tag{2}$$

where $Init$ is a predicate describing the initial states, $\mathcal{N}$ is the disjunction of all the actions of the algorithm expressing how the variables may change, and L is a liveness condition. Note that safety and liveness are treated within a single framework. The safety part of (2) is $\Box[\mathcal{N}]_v$ which states that every step of the program must be permitted by $\mathcal{N}$ or it must leave v unchanged. The liveness part L asserts that infinitely many steps (ones that do change v) occur. The reader is referred to [14, 15] for an explanation of the definition of L.

As a proof system, TLA has only a small set of basic proof rules. In addition to this, simple temporal reasoning is used, as well as ordinary mathematical reasoning about actions. The TLA formula $\Pi \Longrightarrow \Phi$ asserts that the system represented by Π satisfies the property, or implements the system, represented by Φ.

Example 3.1 *As example we consider a simple algorithm, which increments variables x and y indefinitely. The initial state is expressed by the predicate*[3]*:*

$$Init \triangleq (x = 0) \wedge (y = 0)$$

The incrementation is defined by the following two actions:

$$\mathcal{M}_1 \triangleq (x' = x + 1) \wedge (y' = y) \quad and \quad \mathcal{M}_2 \triangleq (y' = y + 1) \wedge (x' = x)$$

So the algorithm is described by the following formula:

$$\Pi \triangleq Init \wedge \Box[\mathcal{M}_1 \vee \mathcal{M}_2]_{(x,y)} \wedge L$$

4 Encoding TLA in SPIKE

It is not a trivial to task to find a proper encoding of the TLA logic for use in an inductive theorem prover, and there probably isn't any such thing as the right formalism. In this section, we present our formalization for expressing TLA specifications and their execution in SPIKE. In the sequel, we assume that all the formulas are universally quantified.

4.1 States and Variables

The model proposed by TLA is traditional in the sense that we have a state-transition system with named variables to express the states and actions to express the state transitions. The simplicity of the model comes from the absence of communication constructs.

As opposed to TLA, in our formalization, we assume that every variable has associated a well-defined sort, *i.e* it contains at least one element. The states are represented as tuples where every component corresponds to one variable of the state. A state is fully characterized by the values of these variables.

Let consider the example 3.1. If we assume that the variables x and y range over natural numbers, then the corresponding state space is represented in SPIKE as follows:

$$Tpl : Nat \times Nat \longrightarrow State,$$

where Nat is the sort of the natural numbers.

The values of variables may be changed between two states. In this respect, we represent these variables as functions defined over $State$ that return the value of the corresponding tuple component. Therefore, the precedent variables x and y are defined as

$$\begin{array}{lcll} X & : & State \longrightarrow Nat, & X(Tpl(x_1, x_2)) = x_1 \\ Y & : & State \longrightarrow Nat, & Y(Tpl(x_1, x_2)) = x_2 \end{array}$$

4.2 Actions

In TLA, an action is a boolean expression containing variables, primed variables and values. This expression states how the initial and the final states are related. In fact, it means how the execution of an action from a given state st_1 (current state) generates a new state st_2 (next state). For example, the TLA actions may express the instructions of a given algorithm (as we will see in Section 6). So, if this algorithm uses the variables $v_1, v_2, \ldots, v_n$ then an action can be considered as follows:

$$Cond \wedge \bigwedge_{i=1}^{n} v_i' = Exp_i$$

$Cond$ is a predicate over $v_1, v_2, \ldots, v_n$ expressing the condition under which the action can be executed and Exp_i is an expression containing unprimed variables and/or values.

[3] The symbol $\triangleq$ means equals by definition.

An action is *enabled* (*disabled*) if its condition in st_1 is true (false). A disabled action leaves the current state unchanged.

We consider that all the actions of the system are of sort *Action*. The function *Exec*, defined as:

$$Exec : Action \times State \longrightarrow State,$$

implements the transition between the states when applying an action.

For example, the following TLA action, composed with the following two equations:

$$\mathcal{A}_1 \triangleq (x' = x + 1) \wedge (y' = y),$$

is formalized as follows:

$$Exec(\mathcal{A}_1, st) = Tpl(X(st) + 1, Y(st))$$

This means that the execution of $\mathcal{A}_1$ from a given state st produces a new state whose first component is the first component of st incremented by 1, while the second one remains unchanged. The action $\mathcal{A}_1$ is always enabled.

Let consider another TLA action:

$$\mathcal{A}_2 \triangleq (x = 2) \wedge (x' = x + 1) \wedge (y' = y)$$

is encoded by a conditional equation:

$$X(st) = 2 \Longrightarrow Exec(\mathcal{A}_2, st) = Tpl(X(st) + 1, Y(st))$$

This means that the execution of $\mathcal{A}_2$ from a given state st produces the same state as $\mathcal{A}_1$, provided the first component of st is 2. Here, *Exec* is not complete because it is defined only when the action $\mathcal{A}_2$ is enabled. To complete it, we add the following conditional equation:

$$X(st) \neq 2 \Longrightarrow Exec(\mathcal{A}_2, st) = st$$

corresponding to the case when $\mathcal{A}_2$ is disabled.

We recall that the canonical form (2) contains an action that allows "stuttering" steps, *i.e.* it leaves all variables of system unchanged. We call this action *Stutt*, and we define it as follows:

$$Exec(Stutt, st) = st$$

4.3 Behaviors

In TLA, the operational semantics of concurrency is based on the transition model. A transition system is given by a set of actions. It is assumed that all the actions are *atomic*; that is, each action causes a single transition. Hence, the evolution of the system can be simulated by *interleaving* atomic actions. The *behavior* of the system can be represented by means of sequences of actions called *traces*. A trace in which every action may occur an unbounded number of times is called *fair* trace (if every action is not guaranteed to happen infinitely often, liveness properties could be not proved).

In this paper, we restrict only to unfair traces because we are interested here in proving safety properties. To represent the behaviors of the system, we give a recursive function Fs which takes a trace and an initial state and returns the final state. This function[4] $Fs : Action^* \times State \longrightarrow State$ is recursively defined as follows:

$$Fs(tr, s) = \begin{cases} st & \text{if} \quad tr = \langle\rangle \\ Fs(tr', Exec(a, st)) & \text{if} \quad tr = \langle a \circ tr' \rangle \end{cases} \tag{3}$$

where $\langle\rangle$ is an empty sequence and $\langle a \circ tr' \rangle$ is sequence whose first action is a followed by the sequence tr'.

[4] X^* is the set of sequences of elements of X.

4.4 Invariance Properties

We will not state all the proof rules of TLA. Instead, we consider one typical rule in some detail. This rule is called **INV1** and is used to prove invariance properties:

$$\textbf{INV1}.\frac{I \wedge [\mathcal{N}]_f \Longrightarrow I'}{I \wedge \Box[\mathcal{N}]_f \Longrightarrow \Box I}$$

INV1 expresses the fact that whenever all the actions preserve I then $\Box I$ holds, provided that I holds initially [14].

We now describe how **INV1** is derived directly from our transition system model, presented in the section 4.3. For this, we consider a property to be proved as a boolean function over *State*, declared as:

$$Goods : State \longrightarrow Bool$$

where *Bool* is the sort of booleans.

A property is an invariant if whenever it holds at some state of the behavior, it also holds for all the subsequent states. Stated in terms of Fs and *Goods*, the invariance properties have the following form:

$$Goods(st) \Longrightarrow Goods(Fs(tr, st)) \tag{4}$$

That is, for the sequence tr, if the property *Goods* holds on a state, then it will hold on the final state given by Fs. Since st and tr are universally quantified, *Goods* is invariant over all the behaviors.

When using the invariance proof rule to prove an invariance property, one must prove that *Goods* is preserved by all the actions a in any state st:

$$Goods(st) \Longrightarrow Goods(Exec(a, st)) \tag{5}$$

In our formalization, **INV1** is implemented by the following proof rule **PR1**:

$$\frac{\begin{array}{l}\mathbf{I}_1.\ Goods(st_0)\\ \mathbf{I}_2.\ Goods(st) \Longrightarrow Goods(Exec(a, st))\end{array}}{Goods(st) \Longrightarrow Goods(Fs(tr, st))}$$

where premise $\mathbf{I}_1$ means that *Goods* holds on an initial state st_0. By an induction on the trace tr, one can easily prove that the formula (4) is deduced from (5).

5 Translator Tool

In this section, we give a presentation of our tool that translates "humanly readable" TLA specifications into SPIKE language. Their readability makes proofs easier to maintain when the specifications change than they would be if written directly in SPIKE. We first present how systems are described in this tool and how the verification process works. We also show how both specification and verification are connected with the SPIKE system.

5.1 A Specification Formalism

In our tool, systems are described in formalism close to TLA's language. In fact, a system is given as a set of actions defining conditional data transformations, where program variables are of any sort definable in SPIKE and allowed value expressions are any expressions definable in SPIKE. Figure 1 shows the grammar defining this specification formalism. This grammar is presented using the conventions of [10], where n^{*NL} means a possibly empty list of n's, n^{+COMMA} means a nonempty list of n's separated by ',', and n^{+SPC} means a nonempty list of n's separated by blanks. $\{\ldots|\ldots\}$ denotes alternatives and $[\ldots]$ denotes optional parts.

The input specification to our tool begins with a declaration of the name of the studied system, followed by a directive to read the files containing predefined SPIKE theories such that natural numbers, booleans, lists, ... The next two key words enable the declaration of (flexible) variables whose sorts can be defined in SPIKE, and values of some nullary sorts, *i.e.* their set of variable-free (*ground*) terms is finite [2]. As the grammar of the figure 1 uses some non-terminals (of the form <*spike_...*>) of the grammar of the SPIKE language, the user can then define functions axiomatically in SPIKE style by means of *defined functions* and *axioms* key words. These functions will be used in the declaration of either *Init* predicate, *actions* or the invariance property. The rest of the specification is the same as TLA specification except for *Invariant* where we must indicate the property to be proved.

specif	::=	$item^{*NL}$
item	::=	*name* | *directive* | *declaration* | *function* | *definition*
name	::=	**System** *ident*
directive	::=	**Use** $ident^{+SPC}$
declaration	::=	**Variables** var_dec^{*NL} | **Values** val_dec^{*NL}
var_dec	::=	$ident^{+COMMA}$ **:** *ident* **;**
val_dec	::=	*ident* **:** $ident^{+COMMA}$ **;**
function	::=	**Defined functions** <*spike_declarations*>
		Axioms <*spike_horn_clauses*>
definition	::=	**Init** *def* | **Actions** def^{*NL} | **Invariant** *def*
def	::=	*ident* == *indent_exp*
indent_exp	::=	*expr* | $pre\_op\_exp^{*NL}$
pre_op_exp	::=	/\ {*expr* | *pre_op_exp*} | \/ {*expr* | *pre_op_exp*}
expr	::=	*expr* /\ *expr* | *expr* \/ *expr* | ~ *expr* | *ident*['] = *expr*
		| *expr infix_indent expr* | <*spike_expressions*>
infix_indent	::=	{+ | - | * | / | > | < | >= | <=}$^{+}$

Figure 1: The grammar of the translator input.

5.2 Tool Structure

The tool we have designed for proving invariance properties is written in Caml language [5]. It consists of three modules: the lexer, the parser and the converter. The translator input is a text file – typed by the user – containing the specification written in TLA style. The checking of lexical correctness (*Lexer*) allows to transform the input file into a list of tokens identifiable by the syntactic analyzer (*Parser*). This one enables first to verify the compatibility of the succession of tokens against the grammar's abstract tree (see figure 1), then it stocks the input text into a standard data structure. On the basis of this data structure, the converter module constructs the SPIKE specification. The user can modify this specification – by adding useful informations – before to submit it to prover.

5.3 Methodology

Letting Θ the algebraic specification corresponding to TLA specification. Our methodology for reasoning about concurrent systems described by TLA with the deductive methods based on induction and rewriting is the following:

1. Corresponding to each variable a well-defined sort. These sorts are combined for constructing the sort *State*, that is the basic element of the TLA operational semantics.
2. Defining to each variable a function playing the rôle of *accessor* in system state.
3. Every action is encoded in a set of rewrite rules; these ones express when an action is or not enabled. We add also a rewrite rule that formalizes the action allowing "stuttering" steps.
4. The recursive function Fs simulates the system behaviors. As restriction, Fs is based on unfair traces that are sufficient for proving invariance properties.
5. The steps 1-4 allow the construction of Θ. The user can define auxiliary functions, for example, the addition over naturals.
6. To prove an invariance property we must verify that it is an inductive consequence of Θ, by using the proof rule **PR1**.

6 Bakery's Example

As example, we prove a mutual exclusion property for the well-known algorithm of *Bakery* [16]. It is called the Bakery algorithm, since it is based on the idea that customers, as they enter, pick numbers that form an ascending

sequence. Then a customer with a lower number has higher priority in accessing its critical section, which in this case is the control location "c". Indeed, this algorithm consists of two processes that execute the same program code do not access simultaneously a particular part of the code, considered as critical.

6.1 TLA Specification

An intuitive transition system and the TLA specification (in the translator's input language) of this algorithm are given in (a) and (b), respectively of figure 2.

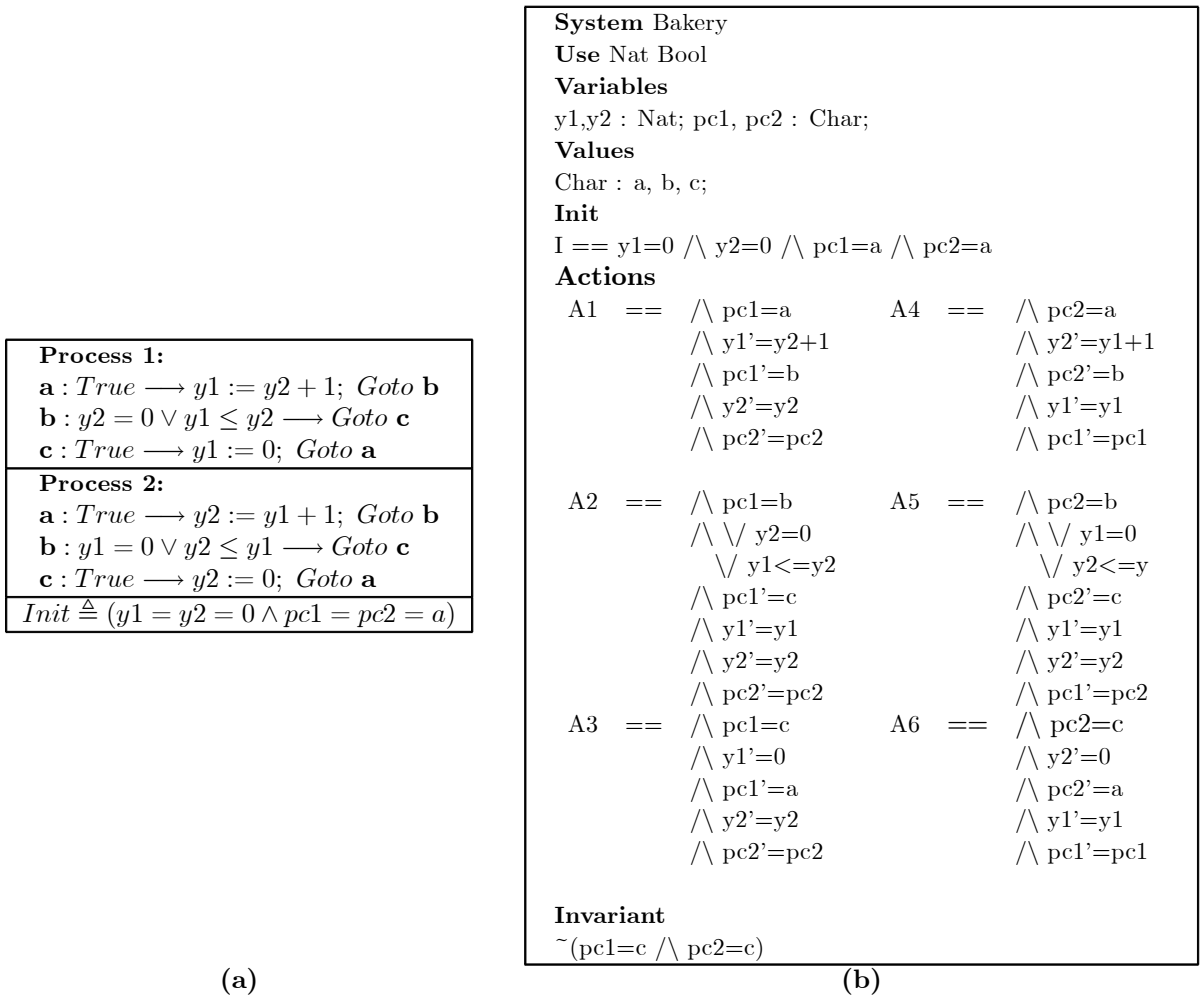

Figure 2: The Bakery algorithm.

The program increments a variable, local to each process, and is made of three instructions. The status of the concurrent system consisting of the two process can be characterized by the variables $y1$, $y2$, $pc1$ et $pc2$. The variables $pc1$ et $pc2$ are program counters, taking one of the values a, b and c. We are interested in the mutual exclusion property of this algorithm, *i.e.* the property that the algorithm never reaches a situation where $pc1 = pc2 = c$ holds. This is shown by proving in TLA the formula $\Theta \Longrightarrow \Box I$ where:

$$I \triangleq \neg(pc1 = c \wedge pc2 = c) \tag{6}$$

and Θ est the canonical form of the concurrent system from 2.(b).

6.2 Algebraic specification

In the following, we build a formal description of the mutual exclusion example using an algebraic specification language based on conditional rewrite rules according to the methodology presented in Section 4.[5]

The axioms are conditional rewriting rules of the form: $(\bigwedge_{i=1}^{n} C_i) \Longrightarrow lhs \longrightarrow rhs$, where C_i are predicates, lhs and rhs are terms. $(\bigwedge_{i=1}^{n} C_i)$ and $lhs \longrightarrow rhs$ represent the *condition* and the *conclusion* respectively of the rewrite rule. For readability, we write them in a sequent notation:

$$\frac{lhs}{rhs}\ \langle rule\_name, (\bigwedge_{i=1}^{n} C_i)\rangle$$

where $rule\_name$ is the rule identifier.

6.2.1 Sort Constructors

The most important sorts in our algebraic specification correspond to:

- *Space states*: $State$, with the constructor set $\{Tpl\}$.
- *Counter values*: $Char$; with the constructor set {"a","b","c"}.
- *Action identifiers*: $Action$, with the constructor set $\{a1,a2,a3,a4,a5,a6,Stutt\}$, where $Stutt$ is the identifier of action allowing "stuttering steps".
- *Booleans*: $Bool$, defined by $\{T,F\}$.
- *Natural numbers*: Nat, with the constructor set $\{0,s\}$.
- *Traces*: $Action^*$, with the constructor set $\{\langle\rangle,\circ\}$.

We denote the equality (disequality) predicates for the sorts $Char$ and Nat by $=_c$ ($\neq_c$) and $=_n$ ($\neq_n$), respectively.

Since the mutual exclusion algorithm uses four variables ($y1$ and $y2$ are natural numbers, and $pc1$ and $pc2$ take counter values), the state constructor is declared as:

$$Tpl : Nat \times Nat \times Char \times Char \longrightarrow State$$

6.2.2 Rewrite Rules

The accessor functions to the to state components are:

$$\begin{aligned} Y1,\ Y2 &: State \longrightarrow Nat \\ Pc1,\ Pc2 &: State \longrightarrow Char \end{aligned}$$

The corresponding rewrite rules of these functions are:

$$\frac{Y1(Tpl(c_1,c_2,c_3,c_5))}{c_1}\ \langle R_1,(True)\rangle$$
$$\frac{Y2(Tpl(c_1,c_2,c_3,c_4))}{c_2}\ \langle R_2,(True)\rangle$$
$$\frac{Pc1(Tpl(c_1,c_2,c_3,c_4))}{c_4}\ \langle R_3,(True)\rangle$$
$$\frac{Pc2(Tpl(c_1,c_2,c_3,c_4))}{c_5}\ \langle R_4,(True)\rangle$$

We denote with $True$ a rewrite rule condition that is always valid.

[5] The full SPIKE specification can be accessed at `http://www-sop.inria.fr/lemme/Sorin.Stratulat/tmp/bakery.spike`.

There are seven actions in the mutual exclusion algorithm. We now express the action *A1* (see figure 2.(b)) as rewrite rules. The rewrite rule

$$\frac{Exec(a1, st)}{Tpl(Y2(st)+1, Y2(st), \text{“}b\text{”}, Pc2(st))} \ \langle R_1 a1, cond\rangle$$

is applied when the condition $cond = (Pc1(st) =_c \text{“}a\text{”})$ is satisfied (the action $a1$ is executed when it is enabled). The case in which $a1$ is disabled is given by the following rewrite rule:

$$\frac{Exec(\alpha_1, st)}{st} \ \langle R_2\alpha_1, (Pc_1(st) \neq_c \text{“}a\text{”})\rangle$$

The other actions *A2*, ...,*A6* are expressed in the same way as *A1*, i.e. each action is encoded in a set of rewrite rules indicating when it is or not enabled.

6.2.3 Proving Invariance Property

The property I (see formula (6)) can be expressed by the functions $Pc1$ and $Pc2$ as follows:

$$\rceil(Pc1(st) =_c \text{“}c\text{”} \wedge Pc2(st) =_c \text{“}c\text{”})$$

Unfortunately, we can't use our proof rule **PR1** for proving the property I because it is not an inductive invariant. As in TLA [14], we must find another property (stronger invariant), *Goods*, satisfying the three conditions:

$$Goods(st_0) \tag{7}$$

$$Goods(st) \Longrightarrow \rceil(Pc1(st) =_c \text{“}c\text{”} \wedge Pc2(st) =_c \text{“}c\text{”}) \tag{8}$$

$$Goods(st) \Longrightarrow Goods(Exec(act, st)) \tag{9}$$

where the initial state st_0 is $Tpl(0, 0, \text{“}a\text{”}, \text{“}a\text{”})$.
The invariance property $Goods : State \longrightarrow Bool$ is:

$$\begin{array}{ll}
\bigwedge & (Pc1(st) =_c \text{“}b\text{”} \wedge Pc2(st) =_c \text{“}c\text{”}) \Longrightarrow (Y1(st) =_n Y2(st)+1 \vee Y1(st) =_n 0 \vee Y2(st) < Y1(st)) \\
\bigwedge & (Pc1(st) =_c \text{“}a\text{”} \wedge Pc2(st) =_c \text{“}c\text{”}) \Longrightarrow (Y1(st) =_n 0 \vee Y2(st) < Y1(st)) \\
\bigwedge & (Pc1(st) =_c \text{“}c\text{”} \wedge Pc2(st) =_c \text{“}b\text{”}) \Longrightarrow (Y2(st) =_n 0 \vee Y1(st) <= Y2(st) \vee Y2(st) =_n Y1(st)+1) \\
\bigwedge & (Pc1(st) =_c \text{“}b\text{”} \wedge Pc2(st) =_c \text{“}b\text{”}) \Longrightarrow (Y1(st) =_n Y2(st)+1 \vee Y2(st) =_n Y1(st)+1) \\
\bigwedge & (Pc1(st) =_c \text{“}a\text{”} \wedge Pc2(st) =_c \text{“}b\text{”}) \Longrightarrow (Y1(st) =_n 0 \vee Y2(st) =_n Y1(st)+1) \\
\bigwedge & (Pc1(st) =_c \text{“}c\text{”} \wedge Pc2(st) =_c \text{“}a\text{”}) \Longrightarrow (Y2(st) =_n 0 \vee Y1(st) <= Y2(st)) \\
\bigwedge & (Pc1(st) =_c \text{“}b\text{”} \wedge Pc2(st) =_c \text{“}a\text{”}) \Longrightarrow (Y1(st) =_n Y2(st)+1 \vee Y2(st) =_n 0) \\
\bigwedge & \rceil(Pc1(st) =_c \text{“}c\text{”} \wedge Pc2(st) =_c \text{“}c\text{”}) \\
\bigwedge & (Pc2(st) =_c \text{“}b\text{”} \wedge (Y1(st) =_n 0 \vee Y2(st) < Y1(st))) \Longrightarrow \rceil(Pc1(st) =_c \text{“}c\text{”}) \\
\bigwedge & (Pc1(st) =_c \text{“}b\text{”} \wedge (Y2(st) =_n 0 \vee Y1(st) <= Y2(st))) \Longrightarrow \rceil(Pc2(st) =_c \text{“}c\text{”})
\end{array}$$

In SPIKE, the proofs of (7) and (8) are straightforward. The major effort was put while proving (9). In the following, we describe how SPIKE deals with this proof. Firstly, a **Generate** rule is applied and 567 instances are produced after the variables *act* and *st* are replaced respectively with the elements of the test set describing the sorts *Action*, *i.e.* {*a*1,*a*2,*a*3,*a*4, *a*5, *a*6, *Stutt*}, and *State*, i.e. $\{Tpl(x_1, x_2, x_3, x_4)\}$, where $x_1, x_2 \in \{0, 1, s(s(y))\}$ and $x_3, x_4 \in \{\text{“}a\text{”}, \text{“}b\text{”}, \text{“}c\text{”}\}$. In order to store the initial conjecture and to soundly use it in further computation, we need to simplify by rewriting or eliminate these instances.

As example, we consider the instance where a and st are substituted respectively by $a1$ and $Tpl(0, 0, \text{“}a\text{”}, \text{“}a\text{”})$. We obtain the following conjecture:

$$Goods(Tpl(0, 0, \text{“}a\text{”}, \text{“}a\text{”})) = T \Longrightarrow Goods(Exec(a1, Tpl(0, 0, \text{“}a\text{”}, \text{“}a\text{”}))) = T$$

whose condition, equivalent to (7), has been already shown to be true.

To simplify $Goods(Exec(a1, Tpl(0, 0, \text{“}a\text{”}, \text{“}a\text{”}))) = T$, a case rewriting is applied using the rewrite rules corresponding to α_1 (see section 6.2.2):

$$\text{“}a\text{”} =_c \text{“}a\text{”} \Longrightarrow Goods(Tpl(Y2(st)+1, Y2(st), \text{“}b\text{”}, Pc2(st))) = T$$
$$\text{“}a\text{”} \neq_c \text{“}a\text{”} \Longrightarrow Goods(Tpl(0, 0, \text{“}a\text{”}, \text{“}a\text{”})) = T$$

By rewriting and using the linear arithmetic decision procedure, these conjectures are eliminated since they are tautologies. The other instances are proved in similar way as above. The proof finishes when all the conjectures are eliminated. According to the proof rule **PR1**, the formula (4) is valid. On the other hand, SPIKE is able to prove it completely automatically. By (8), we conclude that $\neg(Pc1(st) =_c \text{“}c\text{”} \wedge Pc2(st) =_c \text{“}c\text{”})$ is true for any state st accessible from st_0 (for which *Goods* has been shown previously to be valid).

7 Related Works and Conclusions

The mechanical verification of TLA proofs was previously performed by different theorem provers. Thus, TLP [10] is the first and perhaps the most widely used TLA prover. It was designed as a generic TLA front-end to various back-end provers. For this, it uses LP [11] , based on the first-order logic, as its main verification back-end. As this prover is considered as a proof assistant and not as an automatic prover then it requires some interactivity even for the invariance proof. PVS [7] is used for proving invariance properties of DisCo specifications, a language whose semantics is expressed in TLA [13]. This is the only work where more details of automating invariant proofs are given. Some higher-order theorem provers have been used for doing TLA proofs, like HOL in [21] and Isabelle in [12, 17], where reasoning can be slow and tedious because of the lack of predefined theories such as the linear arithmetics. Finally, Mokkedem and Ferguson have verified TLA specifications with the Eves prover [18] which is not a fully automatic verification system. So, an interactivity is needed in some invariance proofs for resembling hand proofs.

We have described our approach at formalizing TLA in SPIKE with the purpose of allowing mechanized reasoning of concurrent systems. Especially, we have defined the semantics of the basic concepts of TLA and proved the invariance proof rule of TLA as a SPIKE theorem. Once this was done, reasoning about systems could be done in SPIKE in a way which corresponds closely to the way reasoning is carried out on paper. To make this encoding easier, we have designed a translator which translates TLA specifications and theorems into the language of SPIKE.

SPIKE is relatively new system and its development is ongoing. The goal of this work was to examine the feasibility of using SPIKE for significant tasks. We have tested our encoding on many simple examples. The results of these experiments seem to indicate that it is certainly appropriate for modeling and verifying invariance properties of concurrent systems.

As future work, it is interesting to enhance our formalization of TLA. The first step will be to prove liveness proof rules. In other hand, we forecast to add quantification and refinement mappings. These will allow the mechanization of more complex examples.

References

[1] A. Armando, M. Rusinowitch and S. Stratulat. Incorporating decision procedures in implicit induction. In *Symposium on the Integration of Symbolic Computation and Mechanized Reasoning (CALCULEMUS 2001)*, June 2001. Siena, Italy.

[2] A. Bouhoula, E. Kounalis and M. Rusinowitch. Automated Mathematical Induction. *Journal of Logic and Computation*, 5(5):631-668, 1995.

[3] A. Bouhoula and M. Rusinowitch. Implicit induction in conditional theories. *Journal of Automated Reasoning*, 14(2):189-235, 1995.

[4] R.S Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press inc., Boston, 1988.

[5] Caml Homepage. The Caml Language. In http://caml.inria.fr/.

[6] K. M. Chandy and J. Misra. Parallel Program Design: A Foundation. Addison-Wesley, 1988.

[7] J. Crow, S. Owre, J. Rushby, N. Shankar and M. Srivas. A tutorial Introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, April 1995.

[8] G. Defourneaux. *Spike tutorial, inference system, user interface, user manual.* Technical Report 1109, LORIA, 1999.

[9] U. Engberg, P. Grönning and L. Lamport. Mechanical verification of concurrent systems with TLA. In *Computer-Aided Verification*, LNCS, Springer-Verlag, 1992.

[10] U. Engberg. *Reasoning in the Temporal Logic of Actions.* PhD Thesis, Aarhus University, Danmark, 1994.

[11] S. J. Garland and J. V. Guttag. An Overview of LP, the Larch Prover. In *Proceedings of the 3rd Conference on Rewriting Techniques and Applications*, pages 137-151, LNCS 355, Springer-Verlag, 1995.

[12] S. Kalvala. A formulation of TLA in Isabelle. Available at http://www.dcs.warwick.ac.uk/~sk/papers/tla.dvi, March 1995.

[13] P. Kellomäki. Mechanical Verification of DisCo Specifications. In *Binational Symposium on Specification, Development, and Verification of Concurrent Systems*, The Technion, Haifa, January 1996.

[14] L. Lamport. The Temporal Logic of Actions. Technical Report 79, DEC-SRC, Palo Alto, California, USA, 1991.

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

[16] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[17] S. Merz. An Encoding of TLA in Isabelle. Technical Report. Institut für Informatik. Universität München, Germany, December 1999.

[18] A. Mokkedem and M. Ferguson. Eves – A prover for TLA$^+$. Technical Report, INRS-Télécomunications, 1996.

[19] M. Rusinowitch, S. Stratulat and F. Klay. Mechanical verification of an ideal incremental ABR conformance algorithm. In *Proceedings of 12th International Conference on Computer-Aided Verification (CAV'2000)*, E. A. Emerson and A. P. Sistla editors, LNCS 1855, pages 344-357, Springer-Verlag, July 2000.

[20] S. Stratulat. *Preuves par récurrence avec ensembles couvrants contextuels. Applications à la vérification de logiciels de télécommunications.* PhD Thesis, Université Nancy I, November 2000.

[21] J. von Wright. Mechanising the temporal logic of actions in HOL. In Proceedings of the 1991 International Worshop on the HOL Theorem Proving System and its applications, pages 155-159, Davis, California, USA, August 1991.