Using Automated Induction-based Theorem Provers for Reasoning about Concurrent Systems

Abdessamad Imine* — **Yahya Slimani**** — **Sorin Stratulat*****

* *LORIA, INRIA*
Campus Scientifique
B.P. 239, 54506 VANDEOUVRE-LES-NANCY CEDEX
e-mail: Abdessamad.Imine@loria.fr

** *Dépt. Informatique - Faculté des Sciences de Tunis*
Campus Universitaire - 1060 Tunis, TUNISIA
e-mail: yahya.slimani@fst.rnu.tn

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

ABSTRACT. *We propose a combination of temporal logic and inductive reasoning for specifying and verifying concurrent systems. Indeed, we describe an attempt to mechanize the reasoning about concurrent systems, described as TLA (Temporal Logic of Actions) specifications. SPIKE, an induction-based automated theorem prover, is used to formalize the semantics of TLA, then to reason about algorithms in TLA. As a case study, it is illustrated an invariance proof for a mutual exclusion algorithm.*

RÉSUMÉ. *Nous proposons une combinaison de logique temporelle et du raisonnement inductif afin de spécifier et de vérifier des systèmes concurrents. Plus précisément, nous décrivons un essai d'automatiser le raisonnement sur des systèmes concurrents, décrits par des spécifications TLA (Temporal Logic of Actions). SPIKE, un démonstrateur automatique de théorèmes basé sur la récurrence, est utilisé pour formaliser la sémantique de TLA, puis de raisonner sur des algorithmes spécifiés en TLA. Comme étude de cas, nous illustrons une preuve d'invariance pour un algorithme d'exclusion mutuelle.*

KEYWORDS: *automated theorem proving, Temporal Logic of Actions, SPIKE, concurrent systems*

MOTS-CLÉS : *démonstration automatique, Temporal Logic of Actions, SPIKE, systèmes concurrents*

1. Introduction

In general, the software development requires a certain number of proof obligations. In particular cases, it may happen that some of these proofs be tedious, such that the manual verification process becomes rapidly unreliable, even impossible. This shows the importance of automatic proof systems, such as SPIKE [BOU 95a, BOU 95b], for eliminating a significant part of such computations. Indeed, SPIKE is an *automatic* theorem prover, suitable for theories whose axioms are conditional equations. Its main capabilities are i) the performance of proofs by *induction*, and ii) some help in the design of sound algebraic specifications. Moreover, SPIKE is meant to reduce the number of user interactions due to the automatization of numerous routine tasks.

TLA is a logic suited for reasoning about concurrent systems [LAM 94]. Their behavior 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.

The mechanical verification of TLA proofs was previously performed by different theorem provers. Thus, LP [GAR 95] has been used by Engberg for his TLP system [ENG 92, ENG 94]. PVS [CRO 93] can verify DisCo specifications, whose semantics is expressed in TLA [KEL 96]. Some higher-order theorem provers have been used for doing TLA proofs, like HOL in [WRI 91] and Isabelle in [KAL 95, MER 99]. Finally, Mokkedem and Ferguson have verified TLA specifications with the Eves prover [MOK 96].

This work is aimed to survey our experience on the use of automated induction-based theorem provers for reasoning about concurrent systems. Particularly, we consider TLA as framework for specification and SPIKE as tool for verification. By putting together TLA and SPIKE, one can perform interesting mechanical proofs of invariance properties over the behaviour of concurrent systems.

In this paper we describe an attempt to reason on *simple*[1] TLA. 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.

This paper is organized in the following way. In Section 2, we recall some basic concepts of TLA. Section 3 overviews briefly the features of SPIKE. In Section 4, we introduce the basic ingredients of our formalization of TLA in SPIKE. As a case

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

study, in Section 5 we prove an invariance property of a mutual exclusion algorithm. A more detailed comparison between different theorem provers able to reason on TLA specifications is included Section 6. The paper finishes with a brief conclusion and suggestions for future work.

2. Temporal Logic of Actions

Temporal Logic of Actions (TLA for short) is a logic for specifying and reasoning about concurrent systems [LAM 94]. Specifications and properties are described by logical formulas: the TLA formula $\Pi \implies \Phi$ 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 two basic kinds of properties of concurrent systems: *safety* (invariance), the property must always be true, and *liveness* (eventuality), the property must eventually be true. Since TLA is a formal logic, it allows completely rigorous reasoning.

For the purposes of this paper, we can consider TLA to be a first-order predicate logic, except with two classes of variables called *rigid variables* and *flexible variables*, extended with the two operators: $'$ (prime) and $\Box$. We often refer to flexible variables as variables.

The semantics of TLA is based on the concept of *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 [LAM 94]. The operator $\Box$ is the standard temporal-logic "always" operator; the prime is the "next-state" operator. The complete semantics of TLA can be found elsewhere [LAM 94]. Here we explain TLA informally through simple examples.

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

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

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 one unit 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 [LAM 94] , we want also to allow steps that leave x unchanged. Letting $[\mathcal{A}]_f$ denote $\mathcal{A} \vee (f' = f)$, this is expressed by the TLA formula

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

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 $\mathrm{WF}_x(x' = x + 1)$ expresses the liveness property that infinitely many $x' = x + 1$ steps (ones that do increment x) occur. The reader is referred to [LAM 94] 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

$$Init \ \wedge \ \Box[\mathcal{N}]_v \ \wedge \ L \tag{3}$$

where *Init* is a predicate characterizing the initial state, $\mathcal{N}$ an action describing how the variables may change, v the tuple of all program variables, and L a liveness condition. The conjunct $\Box[\mathcal{N}]_v$ asserts that every step is either an $\mathcal{N}$ step or else leaves all the variables unchanged.

3. Overview of SPIKE

In this section, we describe the main features of SPIKE[2] [BOU 95b], 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 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* [BOU 95a]. SPIKE has been developed on this principle and incorporates optimizations such as

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

powerful simplification techniques based on conditional rewriting and decision procedures. The system can 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 [STR 00, ARM 02] it was proposed an integration schema of a linear arithmetic decision procedure in SPIKE [STR 00]. 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 [RUS 00].

4. Translating TLA

In this section, we describe a method for representing TLA specifications and their execution in SPIKE. In the sequel, we assume that all the formulas are universally quantified.

4.1. *States*

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.

TLA assumes that there is an 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 our formalization, we assume that every variable has associated a well-defined[3] sort. 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.

As an example, we consider the state space containing the variables x and y which range over natural numbers. To represent this state space in SPIKE, we define the sort $State$ whose constructor is $Tpl : Nat \times Nat \longrightarrow State$, 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.

3. A sort is well-defined if it contains at least one element.

Therefore, the precedent variables x and y are defined as

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

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 5). An action can be considered as a conjunction of equations having the left-hand side either a variable or a primed variable. The equations whose left-hand side is a variable give the conditions to be satisfied in st_1 in order to pass to st_2. 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 by the following two equations: [4] $\mathcal{A}_1 \triangleq (x' = x + 1) \wedge (y' = y)$ is formalized as $Exec(\mathcal{A}_1, st) = Tpl(X(st) + 1, Y(st))$. 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 us consider another TLA action: $\mathcal{A}_2 \triangleq (x = 2) \wedge (x' = x + 1) \wedge (y' = y)$ which is encoded by the 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 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 (3) contains an action that allows "stuttering" steps, *i.e.* it leaves all variables of the system unchanged. We call this action $Stutt$, and we define it as $Exec(Stutt, st) = st$.

4.3. *Transition System Model*

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

4. The symbol $\triangleq$ means equals by definition.

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[5] $Fs : Action^* \times State \longrightarrow State$ is recursively defined as follows:

$$Fs(tr, st) = \begin{cases} st & \text{if} \quad tr = \langle\rangle \\ Fs(tr', Exec(a, st)) & \text{if} \quad tr = \langle a \circ tr' \rangle \end{cases} \tag{4}$$

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'.

4.4. *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 $\mathbf{INV1}$ and is used to prove invariance properties:

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

$\mathbf{INV1}$ expresses the fact that whenever all the actions preserve I then $\Box I$ holds, provided that I holds initially [LAM 94].

We now describe how $\mathbf{INV1}$ 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:

$$Goods(st) \Longrightarrow Goods(Fs(tr, st)) \tag{5}$$

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.

5. X^* is the set of sequences of elements of X.

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:

$$Goods(st) \Longrightarrow Goods(Exec(a, st)) \tag{6}$$

In our formalization, $\mathbf{INV1}$ is implemented by the following proof rule $\mathbf{PR1}$:

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

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 (5) is deduced from (6).

4.5. *Methodology*

In the following, we propose a methodology for reasoning about TLA specifications with deductive methods based on induction and rewriting. In a first step, we build an algebraic specification from the TLA specification. The sorts, the signatures of the functions and the axioms of the algebraic specification are obtained by

1) associating to each variable a well-defined sort;
2) defining for each variable a function playing the rôle of *accessor*;
3) encoding every action in a set of (conditional) rewrite rules;
4) defining auxiliary functions, for example, the addition over the naturals.

The proof rule $\mathbf{PR1}$ is used to prove invariance properties by means of the powerful rewriting and induction techniques provided by SPIKE.

5. A Mutual Exclusion Example

As example, we prove a mutual exclusion property for a simple algorithm allowing that two processes that execute the same program code do not access simultaneously a particular part of the code, considered as critical. The control over the critical region is done by means of semaphores. The example is taken from [LAM 91]. An intuitive picture and the TLA specification of this algorithm are given in Fig.1 and Fig.2, respectively.

The program increments a variable, local to each process, and it consists of three instructions. The status of the concurrent system made of the two processes can be characterized by the variables x, y, pc_1, pc_2 and sem. The variable pc_1 and pc_2 are program counters, taking one of the values "A", "B" and "G", corresponding to each of the instructions. sem stores the value of the semaphore. In a loop, a process can enter in the critical region, represented by the second instruction, if the value of sem

is strictly positive. In this case, sem is decremented by 1, followed by the increment of the local variable. Afterwards, the process leaves the critical region, increments the semaphore and waits again for sem to be positive.

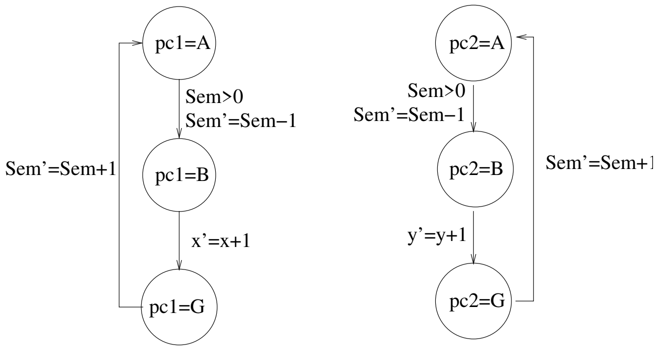

Figure 1. *The mutual exclusion algorithm*

5.1. *TLA Specification*

We are interested in the mutual exclusion property of this algorithm, *i.e.* the property that the algorithm never reaches a situation where $pc_1 = pc_2 =$"B" holds. This is shown by proving in TLA the formula $\Psi \Longrightarrow \Box T$, where:

$$T \triangleq (pc_1 = \text{“}A\text{”} \ \vee \ pc_2 = \text{“}A\text{”}) \tag{7}$$

and Ψ is the formalization of the concurrent system from Fig.2.

5.2. *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.[6]

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:

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

6. The full SPIKE specification can be accessed at `http://www.loria.fr/~stratula/papers/mutex.spike`.

$$
\begin{array}{lcl@{\qquad}lcl}
\multicolumn{6}{l}{Init_{\Psi} \triangleq (pc_1 = \text{“}A\text{”}) \wedge (pc_2 = \text{“}A\text{”}) \wedge (x = 0) \wedge (y = 0) \wedge (sem = 1)} \\
\alpha_1 & \triangleq & \wedge\, (pc_1 = \text{“}A\text{”}) \wedge (sem > 0) & \alpha_2 & \triangleq & \wedge\, (pc_2 = \text{“}A\text{”}) \wedge (sem > 0) \\
 & & \wedge\, pc_1' = \text{“}B\text{”} & & & \wedge\, pc_2' = \text{“}B\text{”} \\
 & & \wedge\, sem' = sem - 1 & & & \wedge\, sem' = sem - 1 \\
 & & \wedge\, x' = x & & & \wedge\, x' = x \\
 & & \wedge\, y' = y & & & \wedge\, y' = y \\
 & & \wedge\, pc_2' = pc_2 & & & \wedge\, pc_1' = pc_2 \\
\beta_1 & \triangleq & \wedge\, pc_1 = \text{“}B\text{”} & \beta_2 & \triangleq & \wedge\, pc_2 = \text{“}B\text{”} \\
 & & \wedge\, pc_1' = \text{“}G\text{”} & & & \wedge\, pc_2' = \text{“}G\text{”} \\
 & & \wedge\, x' = x + 1 & & & \wedge\, y' = y + 1 \\
 & & \wedge\, sem' = sem & & & \wedge\, sem' = sem \\
 & & \wedge\, y' = y & & & \wedge\, x' = x \\
 & & \wedge\, pc_2' = pc_2 & & & \wedge\, pc_1' = pc_1 \\
\gamma_1 & \triangleq & \wedge\, pc_1 = \text{“}G\text{”} & \gamma_2 & \triangleq & \wedge\, pc_2 = \text{“}G\text{”} \\
 & & \wedge\, pc_1' = \text{“}A\text{”} & & & \wedge\, pc_2' = \text{“}A\text{”} \\
 & & \wedge\, sem' = sem + 1 & & & \wedge\, sem' = sem + 1 \\
 & & \wedge\, x' = x & & & \wedge\, x' = x \\
 & & \wedge\, y' = y & & & \wedge\, y' = y \\
 & & \wedge\, pc_2' = pc_2 & & & \wedge\, pc_1' = pc_1 \\
\mathcal{N}_1 & \triangleq & \alpha_1 \vee \beta_1 \vee \gamma_1 & \mathcal{N}_2 & \triangleq & \alpha_2 \vee \beta_2 \vee \gamma_2 \\
 & & & & & \\
\multicolumn{6}{l}{\mathcal{N} \triangleq \mathcal{N}_1 \vee \mathcal{N}_2} \\
\multicolumn{6}{l}{\omega \triangleq \langle x, y, sem, pc_1, pc_2 \rangle} \\
\multicolumn{6}{l}{Init_{\Psi} \ \wedge\ \Box[\mathcal{N}]_{\omega} \ \wedge\ L}
\end{array}
$$

Figure 2. *The TLA specification of the mutual exclusion algorithm*

where $rule\_name$ is the rule identifier.

5.2.1. *Sort Constructors*

The most important sorts in our algebraic specification correspond to: i) **Space states**: $State$, with the constructor set $\{Tpl\}$; ii) **Counter values**: $Char$; with the constructor set $\{$“A”,“B”,“G”$\}$; iii) **Action identifiers**: $Action$, with the constructor set $\{\alpha_1,\beta_1,\gamma_1,\alpha_2,\beta_2,\gamma_2,Stutt\}$, where $Stutt$ is the identifier of the action allowing “stuttering steps”; iv) **Booleans**: $Bool$, defined by $\{T,F\}$; v) **Natural numbers**: Nat, with the constructor set $\{0,s\}$; vi) **Traces**: $Action^*$, with the constructor set $\{\langle\rangle,\circ\}$. We denote the equality (inequality) predicates for the sorts $Char$ and Nat by $=_c$ ($\neq_c$) and $=_n$ ($\neq_n$), respectively.

Since the mutual exclusion algorithm uses five variables (x, y and sem are natural numbers, and pc_1 and pc_2 take counter values), the state constructor is declared as: $Tpl : Nat \times Nat \times Nat \times Char \times Char \longrightarrow State$.

5.2.2. *Rewrite Rules*

The accessor functions to the state components are $X,\ Y,\ Sem : State \longrightarrow Nat$ and $Pc_1,\ Pc_2 : State \longrightarrow Char$. The corresponding rewrite rules of these functions are: i) $\frac{X(Tpl(c_1,c_2,c_3,c_4,c_5))}{c_1}\ \langle R_1, (True)\rangle$, ii) $\frac{Y(Tpl(c_1,c_2,c_3,c_4,c_5))}{c_2}\ \langle R_2, (True)\rangle$, iii) $\frac{Sem(Tpl(c_1,c_2,c_3,c_4,c_5))}{c_3}\ \langle R_3, (True)\rangle$, iv) $\frac{Pc_1(Tpl(c_1,c_2,c_3,c_4,c_5))}{c_4}\ \langle R_4, (True)\rangle$, v) $\frac{Pc_2(Tpl(c_1,c_2,c_3,c_4,c_5))}{c_5}\ \langle R_5, (True)\rangle$.

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

There are seven actions in the mutual exclusion algorithm. We now express each action as rewrite rules. We start with the first action, α_1. The rewrite rule

$$\frac{Exec(\alpha_1, st)}{Tpl(Sem(st)-1, X(st), Y(st), \text{“B”}, Pc_2(st))}\ \langle R_1\alpha_1, cond\rangle$$

is applied when the condition $cond = (Pc_1(st) =_c \text{“A”} \wedge Sem(st) > 0)$ is satisfied (the action α_1 is executed when it is enabled). The cases in which α_1 is disabled are given by the following rewrite rules:

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

$$\frac{Exec(\alpha_1, st)}{st}\ \langle R_3\alpha_1, (Sem(st) \quad \not> \quad 0)\rangle$$

The table from Fig.3 displays the rewrite rules describing all the actions, both enabled and disabled.

5.2.3. *Proving the Invariance Property*

The property T (see formula (7)) can be expressed by the functions Pc_1 and Pc_2 as $(Pc_1(st) =_c \text{“A”})\ \vee\ (Pc_2(st) =_c \text{“A”})$.

Unfortunately, we can't use our proof rule $\mathbf{PR1}$ for proving the property T because it is not an inductive invariant. As in TLA [LAM 91], we must find another property (stronger invariant), $Goods$, satisfying the three conditions:

$$Goods(st_0) \tag{8}$$

$$Goods(st) \Longrightarrow (Pc_1(st) =_c \text{“A”})\ \vee\ (Pc_2(st) =_c \text{“A”}) \tag{9}$$

$$Goods(st) \Longrightarrow Goods(Exec(a, st)) \tag{10}$$

where the initial state st_0 is $Tpl(0, 0, 1, \text{“A”}, \text{“A”})$.

The invariance property $Goods : State \longrightarrow Bool$ is:

$$\begin{array}{llll}
\bigvee & (Sem(st) =_n 1) & \wedge & (Pc_1(st) =_c \text{“A”}) \wedge (Pc_2(st) =_c \text{“A”}) \\
\bigvee & (Sem(st) =_n 0) & \wedge & \vee\, (Pc_1(st) =_c \text{“A”}) \wedge ((Pc_2(st) =_c \text{“B”}) \vee (Pc_2(st) =_c \text{“G”})) \\
 & & & \vee\, (Pc_2(st) =_c \text{“A”}) \wedge ((Pc_1(st) =_c \text{“B”}) \vee (Pc_1(st) =_c \text{“G”}))
\end{array}$$

Actions	**Status**	**Rewrite rule**
α_1	Enabled	$\frac{Exec(\alpha_1,st)}{Tpl(X(st),Y(st),Sem(st)-1,\text{“}B\text{”},Pc_2(st))}\ \langle R_1\alpha_1, cond\rangle$ where $cond = (Pc_1(st) =_c \text{“}A\text{”} \wedge Sem(st) > 0)$.
	Disabled	$\frac{Exec(\alpha_1,st)}{st}\ \langle R_2\alpha_1, (Pc_1(st) \neq_c \text{“}A\text{”})\rangle$
	Disabled	$\frac{Exec(\alpha_1,st)}{st}\ \langle R_3\alpha_1, (Sem(st) \not> 0)\rangle$
β_1	Enabled	$\frac{Exec(\beta_1,st)}{Tpl(X(st)+1,Y(st),Sem(st),\text{“}G\text{”},Pc_2(st))}\ \langle R_1\beta_1, Pc_1(st) =_c \text{“}B\text{”}\rangle$
	Disabled	$\frac{Exec(\beta_1,st)}{st}\ \langle R_2\beta_1, (Pc_1(st) \neq_c \text{“}B\text{”})\rangle$
γ_1	Enabled	$\frac{Exec(\gamma_1,st)}{Tpl(X(st),Y(st),Sem(st)+1,\text{“}A\text{”},Pc_2(st))}\ \langle R_1\gamma_1, Pc_1(st) =_c \text{“}G\text{”}\rangle$
	Disabled	$\frac{Exec(\gamma_1,st)}{st}\ \langle R_2\gamma_1, (Pc_1(st) \neq_c \text{“}G\text{”})\rangle$
α_2	Enabled	$\frac{Exec(\alpha_2,st)}{Tpl(X(st),Y(st),Sem(st)-1,Pc_1(st),\text{“}B\text{”})}\ \langle R_1\alpha_2, cond\rangle$ where $cond = (Pc_2(st) =_c \text{“}A\text{”} \wedge Sem(st) > 0)$.
	Disabled	$\frac{Exec(\alpha_2,st)}{st}\ \langle R_2\alpha_2, (Pc_2(st) \neq_c \text{“}A\text{”})\rangle$
	Disabled	$\frac{Exec(\alpha_2,st)}{st}\ \langle R_3\alpha_2, (Sem(st) \not> 0)\rangle$
β_2	Enabled	$\frac{Exec(\beta_2,st)}{Tpl(X(st),Y(st)+1,Sem(st),Pc_1(st),\text{“}G\text{”})}\ \langle R_1\beta_2, Pc_2(st) =_c \text{“}B\text{”}\rangle$
	Disabled	$\frac{Exec(\beta_2,st)}{st}\ \langle R_2\beta_2, (Pc_2(st) \neq_c \text{“}B\text{”})\rangle$
γ_2	Enabled	$\frac{Exec(\gamma_2,st)}{Tpl(X(st),Y(st),Sem(st)+1,Pc_1(st),\text{“}A\text{”})}\ \langle R_1\gamma_2, Pc_2(st) =_c \text{“}G\text{”}\rangle$
	Disabled	$\frac{Exec(\gamma_2,st)}{st}\ \langle R_2\gamma_2, (Pc_2(st) \neq_c \text{“}G\text{”})\rangle$
$Stutt$	Enabled	$\frac{Exec(Stutt,st)}{st}\ \langle RStutt, (True)\rangle$

Figure 3. *Rewrite rules of actions*

We can write this invariant in another way as follows:

$$\begin{aligned} Goods(Tpl(x_1,x_2,0,x_4,x_5)) = \ & \bigvee\ (x_4 =_c \text{“}A\text{”}) \wedge ((x_5 =_c \text{“}B\text{”}) \vee (x_5 =_c \text{“}G\text{”})) \\ & \bigvee\ (x_5 =_c \text{“}A\text{”}) \wedge ((x_4 =_c \text{“}B\text{”}) \vee (x_4 =_c \text{“}G\text{”})) \end{aligned}$$

$$Goods(Tpl(x_1,x_2,1,x_4,x_5)) = (x_4 =_c \text{“}A\text{”}) \wedge (x_5 =_c \text{“}A\text{”})$$

$$x_3 \neq_n 0 \wedge x_3 \neq_n 1 \Longrightarrow \neg Goods(Tpl(x_1,x_2,x_3,x_4,x_5))$$

In SPIKE, the proofs of (8) and (9) are straightforward. The major effort was put while proving (10). In the following, we describe how SPIKE deals with this proof.

Firstly, a **Generate** rule is applied and 1701 instances are produced after the variables a and st are replaced respectively with the elements of the test set describing the sorts $Action$, i.e. $\{\alpha_1,\beta_1,\gamma_1,\alpha_2,\beta_2,\gamma_2,Stutt\}$, and $State$, i.e. $\{Tpl(x_1,x_2,x_3,x_4,x_5)\}$, where $x_1,x_2,x_3 \in \{0,1,s(s(y))\}$ and $x_4,x_5 \in \{\text{“}A\text{”},\text{“}B\text{”},\text{“}G\text{”}\}$. 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 α_1 and $Tpl(0, 0, 1, \text{“}A\text{”}, \text{“}A\text{”})$. We obtain the following conjecture:

$$\begin{aligned} Goods(Tpl(0,0,1,\text{“}A\text{”}, \quad \text{“}A\text{”})) \quad &= T \\ \Longrightarrow \quad & Goods(Exec(\alpha_1, Tpl(0,0,1,\text{“}A\text{”}, \text{“}A\text{”}))) = T \end{aligned}$$

whose condition, equivalent to (8), has been already shown to be true.

To simplify $Goods(Exec(\alpha_1, Tpl(0, 0, 1, \text{“}A\text{”}, \text{“}A\text{”}))) = T$, a case rewriting is applied using the rewrite rules corresponding to α_1 (see Fig.3):

$$(\text{“}A\text{”} =_c \text{“}A\text{”}) = T \wedge (1 > 0) = T \Longrightarrow Goods(Tpl(0,0,0,\text{“}B\text{”},\text{“}A\text{”})) = T$$

$$(1 > 0) = F \Longrightarrow Goods(Tpl(0,0,1,\text{“}A\text{”},\text{“}A\text{”})) = T$$

$$(\text{“}A\text{”} =_c \text{“}A\text{”}) = F \Longrightarrow Goods(Tpl(0,0,1,\text{“}A\text{”},\text{“}A\text{”})) = T$$

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 $\mathbf{PR1}$, the formula (5) is valid. On the other hand, SPIKE is able to prove it completely automatically. The proof is more complex than that of (10), requiring 46 **Generate** rules. By (9), we conclude that $(Pc_1(st) =_c \text{“}A\text{”}) \vee (Pc_2(st) =_c \text{“}A\text{”})$ for any state st accessible from st_0 (for which $Goods$ has been shown previously to be valid).

6. Related Works

We are certainly not the first ones to encode TLA in a theorem prover. Indeed, there are many formalizations of TLA, most of them being more complete in the sense that they are not restricted only to invariant properties.

TLP [ENG 94] 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 the Larch Prover, based on the first-order logic, as its main verification back-end. The input to TLP are the soundness properties, expressed in TLA, together with their proofs, written in a human readable hierarchically structured form. TLP uses LP to check each step of the proof, translating any assertion into a theorem in the verifier's logic and its proof into instructions for the verifier. Although this design is very flexible in principle, it is not clear how hard it would be to add a new back-end. Moreover, LP is considered as a proof assistant and not as an automatic prover because it requires some interactivity. This feature precludes some desirable automation especially for the invariance proof.

In [WRI 91], Joakim von Wright and Thomas Långbacka defined the semantics of the basic concepts of TLA and proved parts of the basic proof rules of TLA as

HOL theorems [GOR 93]. HOL is a theorem prover for the higher-order logic. This logic is more expressive than the SPIKE logic. However, reasoning using HOL can be slow and tedious for many reasons. It has fewer built-in heuristics for automatically attempting to prove a theorem. Furthermore, the invariance proof – based on the action level – is often arithmetic reasoning, at which HOL is notoriously inefficient.

To verify invariant properties of specifications written in the DisCo language, Kellomäki [KEL 96] expresses the semantics of these specifications in TLA which is translated afterwards to higher-order logic. PVS [CRO 93] is used as the underlying theorem prover. This is the only detailed work about automating invariant proofs.

There are two encodings of TLA in Isabelle [PAU 94]. In the first, Kalvala [KAL 95] uses a definitional approach that explicitly represents states and behaviors. For actual verification purposes, this work is somewhat limited because it is restricted to state variables over natural numbers. In the second, Merz [MER 99] improves the first encoding substantially and provides quantification over flexible variables. However, there is no important degree of automation because of the lack of predefined theories such as the linear arithmetics.

Finally, Mokkedem and Ferguson have verified TLA specifications with the Eves prover [MOK 96] which is based on Zermelo-Fraenkel theory and an untyped first-order logic. Eves is not a fully automatic verification system, and the user must guide it towards proofs by supplying heuristics and performing proof steps. So, an interactivity is needed in some invariance proofs for resembling hand proofs.

7. Conclusion

We have described an attempt of 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.

SPIKE is a 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. We also plan to add quantification and refinement mappings. These will allow the mechanization of more complex examples.

Moreover, we found the hand encoding to be distracting. So, another issue of our work will be the design of a translator which translates TLA specifications and theorems into the language of SPIKE.

8. References

[ARM 02] ARMANDO A., RUSINOWITCH M., STRATULAT S., "Incorporating Decision Procedures in Implicit Induction", *Journal of Symbolic Computation*, , 2002, to appear.

[BOU 95a] BOUHOULA A., RUSINOWITCH M., "Implicit Induction in Conditional Theories", *Journal of Automated Reasoning*, vol. 14, num. 2, 1995, p. 189–235.

[BOU 95b] BOUHOULA A., RUSINOWITCH M., "SPIKE-User Manual", LORIA INRIA, December 1995.

[CRO 93] CROW J., OWRE S., RUSHBY J., SHANKAR N., SRIVAS M., "A Tutorial Introduction to PVS", *Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, April 1993.

[ENG 92] ENGBERG U., GRONNING P., LAMPORT L., "Mechanical Verification of Concurrent Systems with TLA", *Computer-Aided Verification*, LNCS, 1992.

[ENG 94] ENGBERG U., "Reasoning in the Temporal Logic of Actions", PhD thesis, Aarhus University, Danmark, 1994.

[GAR 95] GARLAND S. J., GUTTAG J. V., "An Overview of LP, the Larch Prover", *Proceedings of the 3rd Conference on Rewriting Techniques and Applications*, vol. LNCS 355, Springer-Verlag, 1995, p. 137-151.

[GOR 93] GORDON M., MELHAM T., Eds., *Introduction to HOL*, Cambridge U. Press, 1993.

[KAL 95] KALVALA S., "A formulation of TLA in Isabelle", *http://www.dcs.warwick.ac.uk/ sk/papers/tla.dvi*, March 1995.

[KEL 96] KELLOMÄKI P., "Mechanical Verification of DisCo Specifications", *Binational Symposium on Specification, Development, and Verification of Concurrent Systems*, The Technion, Haifa, January 1996.

[LAM 91] LAMPORT L., "The Temporal Logic of Actions", Technical Report num. 79, 1991, DEC-SRC, Palo Alto, California, USA.

[LAM 94] LAMPORT L., "The Temporal Logic of Actions", *ACM Trans. on Prog. Lang. and Systems*, vol. 16, 1994, p. 872–923.

[MER 99] MERZ S., "An Encoding of TLA in Isabelle", Technical report, December 1999, Institut für Informatik. Universität München, Germany.

[MOK 96] MOKKEDEM A., FERGUSON M., "Eves – A prover for TLA^{+}", Technical report, 1996, INRS-Télécomunications.

[PAU 94] PAULSON L., "Isabelle: A Generic Theorem Prover", *Lecture Notes in Computer Science*, vol. 828, 1994, Springer-Verlag.

[RUS 00] RUSINOWITCH M., STRATULAT S., KLAY F., "Mechanical verification of an ideal incremental ABR conformance algorithm", EMERSON E. A., SISTLA A. P., Eds., *Proceedings of 12th International Conference on Computer-Aided Verification (CAV'2000)*, vol. LNCS 1855, Springer-Verlag, July 2000, p. 344–357.

[STR 00] STRATULAT S., "Preuves par récurrence avec ensembles couvrants contextuels. Applications à la vérification de logiciels de télécommunications", PhD thesis, Université Nancy I, November 2000.

[WRI 91] VON WRIGHT J., "Mechanising the temporal logic of actions in HOL", *Proceedings of the 1991 International Worshop on the HOL Theorem Proving System and its applications*, Davis, California, USA, August 1991, p. 155-159.