Contemporary Engineering Sciences, Vol. 2, 2009, no. 10, 463 - 478

Temporal Logic-based Modeling and Analysis of ASM Designs

Hocine EL-Habib Daho

Department of Informatics
University of Oran, Algeria
dahoh@yahoo.com

Djilali Benhamamouch

Department of Informatics
University of Oran, Algeria

Abstract

Abstract State Machines(ASMs) constitute the basis of an alternative approach to mathematical modelling of discrete dynamic systems. The ASM approach provides an expressive means to specify the operational behavior of a system, but it does not come equipped with a(fixed) logical proof system. Several formal techniques of verifying correctness of ASM designs based on a translation into variants of restricted first-order temporal logic have been investigated. In this paper, we present a method for modeling and analysing ASM designs based on the Temporal Logic of Actions(TLA) which provide an adequate logical reasoning support for ASM models. In TLA, both the ASM design and its required property are modeled by TLA formulas. Analysis of ASM design in TLA is carried out by validating the implication relationship between two formulas. We exemplify our methodology by a case study of a token ring system for mutual exclusion problem formalized in ASMs.

Keywords: Transition Systems, Algebras, Temporal Logic of Actions(TLA), Abstract State Machines(ASMs), Formal Correctness Proof

1 Introduction

Abstract State Machines(ASMs), previously called *Evolving Algebras* and introduced by Y.Gurevich in [9], constitue the mathematical foundation of a practical methodology which has successfully been applied to the design of various kinds of complex dynamic systems.

The ASMs method has been used extensively in different areas, such as software and hardware systems, programming languages, communication protocols and distributed algorithms(see [2, 3] for a comprehensive overview). It provides a flexible formalism to specify the operational semantics of a system at a natural abstraction level in a direct and intuitive way [3]. The ASM approach belong to the family of state-based methods, which model a system as a transition system. An ASM model describes the state space of a system by means of universes(i.e. basic sets) with functions and relations interpreted on them, and the state transitions by means of transition rules by which the system is driven from state to state. In ASMs, states are represented as first-order structures(*Algebras*) over the same signature(*Vocabulary*), and transition rules define the changes over time of the states. In applications, abstract state machines are considered a suitable specification formalism for giving semantics of a system in terms of its set of possible executions(i.e. state sequences).

Besides the standard mathematical techniques underlying the ASM approach that naturally support informal proofs of ASM model properties, there has been work on formal proof systems for ASMs, using various formal verification tools [7, 14, 16]. For instance, [14] use the KIV(Karlsruhe Interactive Verifier) system to mechanically verify the proof of correctness of ASM refinements, both references [4] and [7] show how ASMs can be encoded in the PVS formal system in order to perform mechanical verification of the correctness of ASM specifications or to mechanically check hand proofs using the PVS proof system. While in [16] a model checker approach has been applied for proving correctness of ASM specifications automatically.

Furthermore, other related work in the open literature about verification of ASM models include the work of M.Spielmann[16] and A.Nowack[13] who investigated the verification problem for ASMs using variants of temporal logics to express properties of ASMs. Both work have imposed restrictions on ASM specifications in order to identify classes of ASMs(respectively called Nullary ASMs and Guarded ASMs) that can be verified automatically. Work in [8] introduces the formal language for ASMs (called FLEA), a system for formal reasoning about ASM specifications. They have adopted a modal logic view. Reference [1] presents work on the specification and verification of real-time systems within a logical framework where ASM formalism(e.g. Block ASMs) is used to specify timed algorithms. More precisely, in the context of the verification problem they have used a type of first-order timed logic(FOTL) to formally specify and reason about behavior of real-time ASM specifications by means of FOTL-formulas.

In our current research work[5, 6], we propose to adopt Lamport's Temporal Logic of Actions(TLA)[10] as an appropriate alternative to the logic-based approaches [1, 7, 8, 13, 15], to formally reason about ASM specifications of dynamic systems without imposing restrictions on the specification formalisms.

TLA is a state-based logic which provides the means for describing transition systems(i.e. states, state transitions and thereby the resulting state sequences) and formulating their properties in a single logical formalism, equipped with a relatively complete set of proof rules for reasoning about safety and liveness properties that can be required for systems.

The operational behavior(semantics) of an ASM specification is directly defined by TLA-logical formulas and the TLA-proof techniques can be applied to formally prove the correctness of ASM specifications. Using this framework, both ASM specifications and required properties are represented by formulas in the same logic. In particular, we provide some basic rules to translate ASM models into TLA$^+$ specifications. TLA$^+$ is a formal specification language based on Zermelo-Fränkel set theory, first-order logic and the linear-time temporal logic TLA[11]. In addition to the operators of TLA, it contains operators for defining and manipulating data structures and syntactic structures for handling large specifications. The TLA$^+$-logical framework offers a potential mathematical framework into which ASM model elements are directly translated to their most natural equivalents in TLA$^+$.

The structure of the rest of this paper is as follows: Section 2 briefly presents the basic notions of the ASM approach. In section 3, we give an overview of the main features of the TLA logic and the language TLA$^+$. Section 4 is mainly about the formal representation of an ASM design using the specification language TLA$^+$. this section provides a description of how each element of an ASM model has to be represented in TLA$^+$. In section 5, we present a case study to illustrate the process of formal modelling and analysis of ASM model using the TLA$^+$-logical framework. Finally, in section 6, we conclude our contribution and outline future research directions.

2 Basic Concepts of Abstract State Machines

Abstract State Machines(ASMs)[3] are used for modelling systems as transition systems. They define a state-based computational model, where computations(runs) are finite or infinite sequences of states $\{S_i\}$, obtained from a given initial state S_0 by repeatedly executing transition rules. In ASMs, states are defined as many-sorted first-order structures over a given signature Σ (a vocabulary), and the transition relation is specified by transition rules for describing changes to states. States are implicitly given in an ASM model, and are usually described in terms of functions in the underlying signature. Abstract state machines are considered appropriate for giving semantics of a system in terms of its set of possible executions.

2.1 The Basic Model

An abstract state machine model, M, can be defined as a tuple of the form $M = \langle\ \Sigma,\ Prog\ ,\ Init\ \rangle$, where Σ is a signature, $Init$ is a closed formula over Σ describing the initial state and $Prog$ is a finite set of transition rules.

2.1.1 States :

States of M are variants of first-order structures over a given signature Σ. They are also called Σ-*Algebras*. A signature Σ consists of a collection of domain names(also called universe or set) and a collection of function names, each function name f coming with a fixed arity n and profile $T_1 \times ... \times T_n \rightarrow T_k$ where $T_i (1 \leq i \leq n)$ and T_k are universe names (written $f : T_1 \times ... \times T_n \rightarrow T_k$), or simply $f : T_k$ if $n = 0$.

A Σ-*Algebra*(or state) S consists of a nonempty set T^S for each universe T(the carrier set of T), and a function $f^S : T_1^S \times ... \times T_n^S \rightarrow T_k^S$ for each function name $f : T_1 \times ... \times T_n \rightarrow T_k$ in Σ.

The universe names may be marked as *dynamic* or *static* according to whether or not the set of objects they contain may vary. Function names in Σ can be declared as :

- *Static* : static function names have the same fixed interpretation in each computation state; that is, static functions never change during a run.

- *Dynamic* : the interpretation of dynamic function names can be changed by the transitions occurring in a given computation step; that is, dynamic functions change during a run as a result of the specified system's behavior. Dynamic functions represent the internal state of the system.

Every ASM-signature Σ is assumed to contain the following logic symbols: static nullary functions $True$, $False$, $Undef$, the equality sign $=$, and the sort(universe) $Boolean$ with its usual boolean operators($\neg$, $\wedge$, $\vee$, etc.)

2.1.2 Transition rules :

Transition rules describe how transitions between states(*algebras* of signature Σ) can occur. They define the changes over time of the states. The basic transition rules are syntactic expressions generated as follows :

- *Update rule* : An update rule is the simplest transition rule, also called a local function update or simply update and has the form $c := t$ where c is a variable(a dynamic nullary function in ASM terminology) and t is a closed term over Σ. The update $c := t$ transforms the current state into a new state, in which the denotation of c has been changed into the current denotation of t.

Likewise, if f is a dynamic function of arity n, and t_1, ..., t_n, t are terms of Σ, then $f(t_1, ...,t_n) := t$ is also a function update. The effect of this update is that the denotation of f in the next state is equal to its denotation in the old state, except that the function value on the current values of t_1, ..., t_n is changed into the current value of t. This update also transforms the current state into a new state.

- *Conditional rule* : The conditional rule is a conditional update rule which specify a precondition for updating. It has the form $If\ g\ Then\ R_T\ Else\ R_F$ where g, the guard, is a boolean expression and R_T , R_F are arbitrary update rules. The meaning of this rule is that whenever the guard g evaluates to *true* then apply R_T at the current state otherwise apply R_F. Note that the rules R_T and R_F can take the form of a block rule(see below).

- *Block rule* : the block rule has the form

block
R_1
R_2
....
R_n
endblock

Where R_i for $1 \leq i \leq n$ are transition rules. The block rule groups a set of transition rules and fires them simultaneously. In ASMs, with the block rule we construct the overall ASM-program, and for brevity we always omit the keywords "block" and "endblock", and use indentation to eliminate ambiguity.

2.1.3 Computations :

A computation(run) of an ASM M is a finite (or infinite) sequence of states $\langle S_0, S_1, ..., S_k, ...\rangle$ such that S_0 refers to some given initial state and each state $S_{i+1}(i \geq 0)$ is obtained as the result of firing the program $Prog$ at S_i.

$$S_0 \overset{Prog}{\rightarrow} S_1 \overset{Prog}{\rightarrow} S_2 \overset{Prog}{\rightarrow} ... \overset{Prog}{\rightarrow} S_k \overset{Prog}{\rightarrow} ...$$

In this way, an ASM specification M which can be considered as given by a program $Prog$ together with an initial state S_0, models computations of dynamic systems through finite or infinite ASM runs.

3 Overview of TLA and TLA$^+$

The Temporal Logic of Actions(TLA) was proposed by Lamport [10] as a logic for specifying and reasoning about reactive, distributed, and particular asynchronous systems. TLA uses a single logical formalism for describing transition systems and formulating their properties. It is an extension of classical first-order logic by some linear-time temporal logic operators with a relatively complete set of proof rules. The semantics of TLA is defined in terms of states and behaviors. A state is an assignment of values to variables, and a behavior is an infinite sequence of states. A TLA formula is interpreted as a boolean function on behaviors.

In TLA we distinguish two classes of variables called *rigid variables* and *flexible variables*. The rigid variables represent quantities that do not change with time, they are also called constants. The flexible variables represent quantities that may change with time, and are just called variables.

TLA formulas are built up from *state functions* using the usual boolean operators($\wedge, \vee, \neg, \Rightarrow$) and the operators $'$ (prime) and $\Box$ (read as always). A *state function* is defined as a non-boolean expression built upon variables and constant symbols. *State functions* are interpreted over single states. For example, the value of the *state function* $x + 1$ is 3 at a state s in which the value of x is 2. A *state predicate* is a boolean expression built from variables and constant symbols. For example, $x > 1$ is true in the state s. An *action* is a boolean-valued expression which can be made from variables, primed variables and constant symbols. *Actions* are interpreted over pairs of states. The unprimed variables are interpreted in the first state of a state pair and the primed variables in the second. For example, the *action* $y' = x + 1$ is true over the pair of states $\langle s\ ,\ t\rangle$ iff the value of y in t is equal to the value of x in s plus 1. A pair of states satisfying *action* A is called an A *step*. We write f' for the expression obtained by priming all the variables of the tuple(i.e. a *state function*) f , and $[A]_f$ for $(A \vee Unchanged\ f)$ where $Unchanged\ f \triangleq (f' = f)$, so an $[A]_f$ *step* is either an A *step* or a *step* that leaves f unchanged.

As usual in temporal logic, if F is a formula then $\Box F$ is true of a behavior iff it is true in every state of it. So, $\Box\,[A]_f$ holds over a behavior iff every pair of consecutive states in it is either an A *step* or a *step* that leaves f unchanged.

The standard way of specifying a system in TLA is with a formula in the "canonical form": $Init \wedge \Box\,[Next]_f$, where

- $Init$ is the initial-state predicate, a formula describing all legal initial states of the system

- $Next$ is the next-state relation, which specifies all possible steps(pairs of successive states) in a behavior of the system. It is a description of actions that describe the different system operations.

. f consists of the variables the system operations can change.

TLA$^+$ is a formal specification language based on (untyped) Zermelo-Fraenkel set theory in which every value is a set, first-order logic, and TLA [11]. TLA$^+$ supplements TLA with operators for defining and manipulating data structures and mechanisms(syntactic structures) for writing specifications modularly. A TLA$^+$ specification is organized as a collection of modules. Logically, a TLA$^+$ module consists of a list of statements, where a statement can be a *declaration*, a *definition*, an *assumption* or a *theorem*. It has the following form:

MODULE	$\langle$*Name*$\rangle$
CONSTANTS	$\langle$*List of constant parameters*$\rangle$
VARIABLES	$\langle$*List of variable parameters*$\rangle$
ASSUME	$\langle$*Properties of constants*$\rangle$
TYPE INVARIANT	$\langle$*Properties of variables*$\rangle$
INIT	$\langle$*Initial values of variables*$\rangle$
SPEC	$\langle$*A TLA formula describing possible behaviors*$\rangle$
THEOREM	$\langle$*A TLA formula stating properties of spec*$\rangle$
END.	

Below are some of the TLA$^+$ notations used to represent functions :

. The TLA$^+$ expression $[x \in S \mapsto e(x)]$ equals the function f whose domain is the set S such that $f[d] = e(d)$ for every $d \in S$.

. The TLA$^+$ notation $[A \to B]$ is the set of all functions from the set A into the set B.

. If f is a function, the TLA$^+$expression [f EXCEPT![i] = j] is the function which is equal to f except in i where the returned value is j.

. The function application is expressed using square brakets, so $f[i]$ is the value obtained from function f with argument i.

4 Modeling ASM Designs using TLA$^+$

TLA$^+$ provides a key tool for formalizing the operational behavior(execution) of ASM designs. In this section we provide some basic recipes of how to represent the different aspects of an ASM model using the TLA$^+$ specification language. In this translation process we re-use the same names(symbols) that occurred in the ASM-signature Σ to indicate the close correspondence with respect to their values. An ASM model translates into a TLA$^+$module, a specification unit in TLA$^+$. In the following we describe for each aspect of an ASM model how to represent it in The TLA$^+$:

Sorts(Universes) representation : sorts have different TLA$^+$encoding depending on their being static or dynamic :

. A static sort U, i.e. a sort which does not change during computation, is represented as a TLA$^+$ constant parameter(*rigid variable*) U, declared in the TLA$^+$module with a *CONSTANTS* statement, as follows : **Constants** U.

. A dynamic sort D, i.e. a sort which may change during computation, is encoded as a TLA$^+$variable parameter(*flexible variable*) D, declared with a *VARIABLES* statement, as follows: **Variables** D.

Functions representation : ASM basic functions are classified in static functions which remain constant, and dynamic function which may change interpretation during computation.

. A static function f together with its signature(e.g. $f : S \to T$) that provides the type information of the function symbol f, is encoded as a TLA$^+$constant parameter f together with a set membership assumption(such as $f \in [S \to T]$) declared in the *ASSUME* statement of the TLA$^+$module, asserting that its value is a function in the set of all functions(with domain S and range a subset of T), described by the TLA$^+$construct $[S \to T]$.

. A dynamic function h together with its typing information, is encoded as a TLA$^+$variable parameter h. The type property of the variable h will be specified by a TLA$^+$typing predicate namely a state predicate which has not to be assumed but to be proved as an invariant of the resulting TLA$^+$temporal formula identifying behaviors of the ASM model.

TLA$^+$modelling of ASM rules : We now describe how ASM transition rules are represented as TLA$^+$-*actions* which provide their logical semantics. In the following we will restrict our investigation to basic update rules, conditional and block rules. To simplify notation, let consider R and R_i ($i \geq 1$) stand for ASM rules, *exp* for an expression, g for a boolean expression(condition), x for a variable(dynamic nullary function), f for a dynamic function with arity $n > 0$, $V(R)$ for the set of variables potentially changed by R, V for the set of all declared ASM-program variables, and $[[R]]$ for the resulting TLA$^+$formula.

Basic update rules : The most basic forms of updates that can appear in an ASM model are called local function updates. They are similar to assignments in imperative programming languages :

- Updates of the form, $R :: x := exp$. The effect of this update upon a state is to change the value of x into the value of exp, and leaves all variables other x unchanged. In TLA$^+$, the corresponding change to x is expressed by the TLA$^+$-*action* $[[R]]$ using the prime $'$ operator and the $Unchanged$ construct as follows : $[[R]] \triangleq x' = exp \wedge Unchanged\ (V - \{x\})$.

- Updates of the form, $R :: f(t_1, ..., t_n) := exp$, is represented by the TLA$^+$-*action* $[[R]]$ using the EXCEPT construct, the prime $'$ operator and the $Unchanged$ construct as follows:
$[[R]] \triangleq f' = [f \text{ EXCEPT!}[t_1, ..., t_n] = exp] \wedge Unchanged\ (V - \{f\})$, where the TLA$^+$-expression $[f \text{ EXCEPT!}[t_1, ..., t_n] = exp]$ represents the new function $\hat{f}$ that is the same as f except that $\hat{f}\,[t_1, ..., t_n] = exp$.

- Conditional rule of the form, $R :: If\ g\ then\ R_1\ else\ R_2\ endif$, is the most common means of specifying a precondition for updating. Conditional rules are formally represented using the TLA$^+$logical conditional choice operator IF *condition* THEN $formula1$ ELSE $formula2$ which is equivalent to
$((condition \wedge formula1) \vee (\neg condition \wedge formula2))$. The rule R is translated as follows :

$$\begin{aligned}[[R]] \triangleq \quad & \wedge \text{ IF } g \text{ THEN } [[R_1]] \\ & \qquad \text{ELSE } [[R_2]] \\ & \wedge Unchanged\ (V - V(R))\end{aligned}$$

where $[[R_1]]$ and $[[R_2]]$ are the TLA$^+$modelling of the rules R_1 and R_2, and $V(R) = V(R_1) \cup V(R_2)$.

- Block rule $R ::$ *block* $R_1\ R_2\ ...R_n$ *endblock*, groups a set of transition rules. With a block rule we construct the overall ASM-program. All transition rules that are grouped into a block are performed in parallel. TLA$^+$ supports different specification styles(*interleaving* and *noninterleaving* styles) for representing the concurrent execution of transition rules composing the block rule. With the block rule, we adopt the interleaving representation which is more convenient for reasoning about ASM-program. The TLA$^+$description in the interleaving style for the block rule R is constructed from the TLA$^+$modelling of the rules $R_1, ..., R_n$ as follows :

$$
\begin{aligned}
[[R]] \triangleq \quad & \vee\ [[R_1]] \wedge Unchanged\ (V(R) - V(R_1)) \\
& \vee\ [[R_2]] \wedge Unchanged\ (V(R) - V(R_2)) \\
& \vee\ ... \\
& \vee\ [[R_n]] \wedge Unchanged\ (V(R) - V(R_n))
\end{aligned}
$$

Where $V(R) = V(R_1) \cup V(R_2) \cup ... \cup V(R_n)$.

The TLA formula representing the semantics of an ASM specification(model)M is a safety formula Φ of the following form :

$$\Phi = [[M]] \triangleq INIT \wedge \Box\,[Next]_V$$

where $INIT$ is the state predicate representing the initial condition of the ASM model M, $Next$ is the TLA$^+$next-state action representing the block rule(ASM-program), and V is the collection of ASM-program variables. The TLA$^+$ formula Φ describing an ASM model can be used to infer any safety property of an ASM system through logical reasoning. Formula $\Phi = [[M]]$ is satisfied only by all admissible behaviors. The formula Φ produced in this way for an ASM model M describes the behaviors(executions or runs) of this ASM.

Lemma 4.1 *Every run of an ASM model M is a model of the formula Φ([[M]]), and conversely, every model of Φ is a run of M.*

5 Formal Analysis of ASM Designs

The purpose of giving a logical characterization of the behavior of an ASM design using the TLA logic has been to be able to formally state and verify behavioral properties using the proof rules of TLA. In TLA both ASM designs and their required properties are represented in the same logic. The assertion " An ASM design M has the property P " is expressed in TLA by the validity of the formula $[[M]] \Rightarrow [[P]]$, where $[[M]]$ represents the TLA semantics of system ASM specification and $[[P]]$ is the logical expression of the informal property P. So analysis of ASM designs based on TLA is converted to prove some formula *true* or *false*.

6 Case Study : A token ring algorithm

In this section we use a case study to illustrate the process of formal modelling and analysis of an ASM model using the TLA$^+$-logical framework. The case study is a variant of distributed computing system, namely a token ring algorithm which is a primitive solution of exclusive sharing of resources, formalised in terms of ASM notation as presented in [12].

The setting is as follows : there are n processes arranged in a logical ring, which from time to time want to use some shared resource(e.g. disk) which must not be accessed by any two processes simultaneously. This is done by *token* passing among the processes. When a process receives token, it means that the resource is free and reserved for it. The process can use it and after its work is done, it passes the token to the next process in the ring. Sometimes can arise a situation, when a process receives token but it does not want to use the resource, then it simply passes the token to the next process without doing anything. Note, that there is only one token in the system and only one process can be its owner and use the resource.The goal of the algorithm is to guarantee at any time, that there is at most one process using the resource.

The ASM model of the above informal description of the algorithm together with its temporal translation and correctness condition are given below:

6.1 ASM Specification :

The ASM model $M = \langle\ \Sigma,\ Prog\ ,\ Init\ \rangle$ of the above system of token passing processes is defined as follows :

The ASM signature The signature Σ consists of the following universes and functions together with their associated sorts :

. Static universes :

$$Processes = \{0, 1, ..., n-1\}$$
$$Status \quad = \{Wait, Owner, User, Pass\}$$

. Static functions :

$next : Processes \longrightarrow Processes$ such that $next(i) = (i+1) \bmod n$
$0 \quad :\longrightarrow Processes$

. Dynamic functions :

$$state : Processes \longrightarrow Status$$
$$hastoken : Processes \longrightarrow Boolean$$
$$request : Processes \longrightarrow Boolean$$

Initial state : $Init$ is a first-order formula specifying the initial state of any run.
$\forall\, i \in Processes : (state(i) = Wait \wedge request(i) = true) \wedge hastoken(0) = true \wedge \forall\ j \neq 0 : hastoken(j) = false$

Integrity constraints : these are first-order descriptions which express implicit assumptions about the application domain.

$OneOwnerOfToken \triangleq \exists i : hastoken(i) \wedge \forall j : j \neq i \Rightarrow \neg hastoken(j)$
$OwnerProcess \triangleq hastoken(i) \Leftrightarrow (state(i) = Owner \vee state(i) = User)$.

ASM program : the set of transition rules describing behavior of the token ring system.The rule $RECEIVE$ represents situation, when process i receives token. After receiving it, process i makes decision whether to use the resource or not(see rule $DECIDE$). Token passing is described by the rule $PASS$, process i passes the token to its successor and moves to the state Wait.

$RECEIVE :: If\ state(i) = Wait \wedge hastoken(i)\ Then\ state(i) := Owner$

$$DECIDE :: \quad If\ \ state(i) = Owner\ Then \\ If\ request(i)\ Then \\ state(i) := User \\ Else \\ state(i) := Pass$$

$USE :: If\ state(i) = User\ Then\ state(i) := Pass$

$$PASS :: \quad If\ \ state(i) = Pass\ Then \\ hastoken(i) := false \\ hastoken(next(i)) := true \\ state(i) := Wait$$

6.2 TLA$^+$ Specification :

In the following we present the temporal translation of the ASM specification M of the token ring system. The TLA$^+$module that encodes the ASM model M by following the translation rules is as follows:

Module *TokenRingASM*
Extends $Naturals$
Constants $Processes,\ Status, next$

Assume
$\land\ Processes = 0..n-1$
$\land\ Status = \{Wait, Pass, User, Owner\}$
$\land\ next \in [Processes \to Processes]$
$\quad \land\ next = [i \in Processes \mapsto i + 1 \bmod n]$
$\land\ n \in Nat \land n > 0$

Variables $state, request, hastoken$

$TypeInvariant \triangleq$
$\land\ state \in [Processes \to Status]$
$\land\ request \in [Processes \to Boolean]$
$\land\ hastoken \in [Processes \to Boolean]$

$Vars \triangleq \langle state, request, hastoken \rangle$

$INIT \triangleq$
$\land\ state = [i \in Processes \mapsto Wait]$
$\land\ hastoken = [i \in Processes \mapsto IF\ i = 0\ THEN\ True\ ELSE\ False]$
$\land\ request = [i \in Processes \mapsto True]$

$RECEIVE(i) \triangleq$
IF $state\,[i] = Wait \land hastoken\,[i]$
THEN $\land\ state' = [state\ EXCEPT!\,[i] = Owner]$
$\quad \land\ Unchanged\ \langle hastoken, request \rangle$
ELSE $Unchanged\ \langle state, hastoken, request \rangle$

$DECIDE(i) \triangleq$
IF $state\,[i] = Owner \land request\,[i]$
THEN $\land\ state' = [state\ EXCEPT![i] = User]$
$\quad \land\ Unchanged\ \langle hastoken, request \rangle$
ELSE $Unchanged\ \langle hastoken, request, state \rangle$

$USE(i) \triangleq$
IF $state\,[i] = User$ THEN
$\quad \land\ state' = [state\ EXCEPT![i] = Pass]$
$\quad \land\ Unchanged\ \langle hastoken, request \rangle$
ELSE $Unchanged\ \langle state, hastoken, request \rangle$

$PASS(i) \triangleq$
IF $state\,[i] = Pass$ THEN
$\land\ hastoken' =$
$[hastoken\ EXCEPT![i] = False, EXCEPT![i + 1 \bmod n] = True]$
$\land\ state' = [state\ EXCEPT![i] = Wait]$
$\land\ Unchanged\ \langle request \rangle$
ELSE $Unchanged\ \langle state, hastoken, request \rangle$

$NEXT(i) \triangleq RECEIVE(i) \lor DECIDE(i) \lor USE(i) \lor PASS(i)$

$\Phi = [[M]] \triangleq INIT \land \Box\,[\exists i \in Processes : NEXT(i)]_{Vars}$

END.

6.3 Correctness Proof of the ASM Model :

As an example of correctness requirement that the token ring system should satisfy is exclusive sharing of resource(mutual exclusion property), meaning that at any moment there can be at most one process using the resource(i.e. No two processes are trying to use a resource at the same time). This property of mutual exclusion for such a system is an example of invariance(safety) properties, those which are true at every state of the system execution. In TLA$^+$, this mutual exclusion property can be formally expressed by the temporal formula $\Box Mutex$, where $Mutex$ is a state predicate of the following form:

$$Mutex \triangleq \forall i, j \in Processes : i \neq j \Rightarrow \neg(state\,[i] = User \wedge state\,[j] = User)$$

The formal correctness proof of the ASM specification M which simulates the token ring system behavior, is reduced to proving that the corresponding TLA$^+$formula $[[M]]$ satisfies the mutual exclusion property $\Box Mutex$. In TLA$^+$, the assertion that the TLA$^+$formula $[[M]]$ satisfies the property $\Box Mutex$ takes the form of the following theorem(a TLA formula):

$$\text{THEOREM: } [[M]] \Rightarrow \Box Mutex$$

Which asserts that every behavior satisfying the TLA$^+$specification $[[M]]$ also satisfy the property $\Box Mutex$, i.e. the predicate $Mutex$ is true through every behavior satisfying this specification $[[M]]$.

The proof of $[[M]] \Rightarrow \Box Mutex$ is a relatively straightforward application of classical first-order reasoning and simple temporal facts which are embodied in the TLA proof rules [10]. In order to prove this property, we will take advantage of one of the rules of TLA, namely the rule $INV1$:

$$INV1 : \frac{I \wedge [N]_f \Rightarrow I'}{I \wedge \Box\,[N]_f \Rightarrow \Box I}$$

In the case of our specification, I is the state predicate $Mutex$ which is an invariant, N is the next-state action $\exists i : NEXT(i)$ and f is the tuple $Vars$ of variables. The rule $INV1$ tells us that we must prove the following statements:

$$INIT \Rightarrow Mutex : (1)$$
$$Mutex \wedge [\exists i \in processes : NEXT(i)]_{Vars} \Rightarrow Mutex' : (2)$$

from which we can deduce $[[M]] \Rightarrow \Box Mutex$ as follows:

$$\begin{aligned}
[[M]] \Rightarrow \quad & \{ \text{ By definition of } [[M]] \ \} \\
& INIT \wedge \Box\ [\exists i \in Processes : NEXT(i)]_{Vars} \\
\Rightarrow \quad & \{ \text{ By the implication (1) } \} \\
& Mutex \wedge \Box\,[\exists i \in Processes : NEXT(i)]_{Vars} \\
\Rightarrow \quad & \{ \text{ By the implication (2) and the rule INV1 } \} \\
& \Box Mutex.
\end{aligned}$$

7 Conclusion and Future Work

The aim of our work is to provide formal reasoning techniques for ASMs by using the TLA$^+$-logical framework. In this paper, we have shown how the TLA$^+$-framework can be used to formally modelling and reasoning about the correctness of ASM designs. The power of this modeling method is analysis of ASM models by formally reasoning in TLA. Future work will concentrate on the development of a model translator, namely ASM2TLA$^+$ translator, to perform the translation of an ASM model into a TLA$^+$ model which can be verified automatically using the TLA$^+$ model checker called TLC[11].

References

[1] D.Beauquier, A.Slissenko, A first-order logic for specification of timed algorithms: Basics properties and a decidable class. Annals of Pure and Applied Logic, 113(1-3):13-52,2002.

[2] E.Börger , High-level system Design and Analysis using Abstract State Machines. In Current Trends in Applied Formal Methods(FM-Trends98), number 1641 in LNCS, pages 1-43. Springer-Verlag, 1999.

[3] E.Börger, R.Stärk : Abstract State Machines : A Method for High-Level System Design and Analysis. Springer 2003.

[4] A.Dol, T.Gaul, U.Vialard,and W.Zimmerman, ASM-based mechanized verification of compiler back-ends. In Proceedings of the 28th Annual Conference of the German Society of Computer Science. Technical Report, Magdeburg University, 1998.

[5] H. El-Habib Daho, D.Benhamamouch, Verifying the Correctness of ASM Programs using TLA$^+$. Technical report, Department of Informatics,University of Oran, January 2008.

[6] H. El-Habib Daho, D. Benhamamouch, Formal Verification of ASM Models Using TLA$^+$. In E.Börger et al.(Eds.), ABZ 2008, LNCS 5238, Springer-Verlag 2008.

[7] A. Gargantini, E. Riccoben, Encoding Abstract State Machines in PVS. In Abstract State Machines : Theory and Applications, Vol.1912 of LNCS.Springer-Verlag, 2000.

[8] R. Groenboom, G.R. Lavalette, A Formalisation of Evolving Algebras. In Proceedings Accolade 95, pages 17-28, 1995.

[9] Y. Gurevich, Evolving Algebras : An attempt to discover semantics. Bulletin of the EATCS, (43):264-284, February 1991.

[10] L. Lamport, The Temporal Logic of Actions. ACM Transactions on Programming Languages and systems, 16(3):872-923, May 1994.

[11] L. Lamport, Specifying Systems : The TLA$^+$Language and Tools for Hardware and Software Engineers. Addison-Wesley, Boston 2003.

[12] P. Matousek, Evolving Algebras and Formalisation of Dynamic Systems. Technical report, Department of Informatics, VSB Technical University of Ostrav, Czech Republic, 1999.

[13] A. Nowack, Deciding the Verification Problem for Abstract State Machines. In Proceedings of ASM 2003, LNCS, 2589:341-371, 2003.

[14] G. Schellhorn, W. Ahrendt, Reasoning about Abstract State Machines : The WAM case study. Journal of Universal Computer science,3(4):377-413, 1997.

[15] M. Spielmann, Automatic verification of Abstract State Machines. In Proceedings of 11th international conference on computer-Aided Verification(CAV'99), Vol.1633 of LNCS, pages 431-442. Springer-Verlag, 1999.

[16] K. Winter, Model Checking Abstract State Machines. Journal of Universal Computer Science, 3(5):689-701, 1997.

Received: July, 2009