Validation of formal specifications

Dominique Méry, Yassine Mokhtari

Validation of formal specifications

Dominique Méry and Yassine Mokhtari
LORIA-UMR n^{o}7503
Université Henri Poincaré, France
{mery,mokhtari}@loria.fr

Abstract

TLA (the Temporal Logic of Actions) is a linear temporal logic for specifying and reasoning about reactive systems. The purpose of this paper is to develop an animator and a model checker, both based on a subset of TLA, and illustrates how we can combine these tools to validate TLA specifications.

Introduction

Validation of formal specifications is the process of comparing the formal model of requirements (called formal specification) against the customer needs. It fulfills two rles: the customer must be convinced that all requirements are fully recorded. On the other hand, the designer must be able to use the requirements to produce a structure around which formal reasoning and an implementation can be developed.

In formal methods, validating formal specifications is achieved by a number of techniques including mainly proof and animation (execution). Therefore, the machine assistance in the validation of formal specifications traditionally comes with two forms: animation[1] allows the validation of arbitrarily complex systems but it requires the intervention of the user especially in a particular context like handling non-determinism. Model checking and similar techniques, on the other hand, are largely automatic but they are usually limited to systems whose state space is finite and relatively small. Thus both techniques are possible, and both have their uses. However, model checking is possible on a much smaller set of specifications than animation. In addition, the success of the validation depends on the communication between the customer and the designer. If we consider that the customer has a little knowledge of computing, then using proof at this stage may exclude and restrict the rle of the customer in this process.

In this paper, we will describe a subset of Lamport's logic on which we will develop an animator and a model checker. We will explain how we can combine these tools to do the validation of a small example.

[1] animation is the generation (may in random manner) of a small number of behaviors that the user can study

Language description

We describe here an executable subset of Lamport's Temporal Logic of Actions (TLA) (Lamport 1994) on which an animator and a model checker are based. TLA is a linear temporal logic equipped with a formal semantics and a proof system suitable for specifying and reasoning about reactive systems and their properties. In TLA, a component is described by a temporal formula which has the following canonical form:

$$Init \wedge \Box[N]_x \wedge L$$

where:

- $Init$ is a state predicate describing the initial state.
- N is a transition predicate describing the components steps. x is needed for *stuttering* steps.
- L is a liveness requirement, usually expressed using fairness constraints.

A complete description of TLA can be found in (Abadi & Merz 1996; Lamport 1996).

It is well established that not all TLA specifications could be animated. Mainly, there are two obstacles for the development of executable temporal logic (Fisher & Owens 1993).

- Although, the satisfiability problem of PTL is decidable but it is a complex problem (PSPACE-complete). It is undecidable for the first-order temporal logic and highly complex.
- First-order temporal logic is incomplete. Thus, there is a valid formula for which we cannot exhibit a proof. Consequently, not all formulas could be executed successfully.

Faced with these problems, Merz (Merz 1993) suggests to find a tradeoff between expressiveness and efficient implementability.

In the first time, we have chosen to animate the temporal formula $Init \wedge \Box[N]_x$ i.e. only the safety property. To do so, we restrict the syntax of actions, in order to ensure that successor states of each state is recursively enumerable. The rest of the syntax of actions is as the following:

- Atomic actions are of the form $x' = v$ where v is a (computable) state function.
- Conjunctions and disjunctions of actions are again actions.

- Implications are permitted only if the formula on the left-hand side is a (computable) state predicate; this restriction in particular ensures that the negation of an action cannot in general be expressed.

We have formalized an operational semantics for this subset. A semantics is given by assigning a semantic meaning $[\![F]\!]$ to each syntaxic object F. We recall that an action represents a relation between old states[2] and new states, where the unprimed variables refer to the old state and the primed variables refer to new state. Thus, x'=x+1 is the relation asserting that the value of x in the new state is one greater thant the value of x in the old state. In the logical semantis, the meaning of an action A is a mapping from pairs of states to booleans. We write that $s[\![A]\!]t$ holds for a pair states (s, t). In the operational semantics, when the action hols in a pair of states then, the execution of this action in the first state, say s, can produce a new state t. The operation semantics try to construct partially the state t. This partial information is represented by a *valuation*. Operationnally, we define the meaning of an action as set of valuations. We need two fundamentals operations. The first one ensures that tow valuations τ_1 and τ_2 are compatible, written $\tau \simeq \tau_2$, that is they agree on the values of all variables they both determine. The second one is the operation JOIN that allows the composition of sets of valuations. Finally, we prove the soundness and completeness of our semantics with respect to logical semantics.

In the second time, we are interested in the animation besides the safety part, the fairness part. We have introduced two schedulers one per each kind of fairness requirements. The reader may refer to (Mokhtari & Merz 1999) for a complete description.

An animator

The interpreter algorithm attempts to construct a set of finite behaviors of a certain length. Let $F \triangleq Init \wedge \Box[A]_x$ be a temporal formula and $\mathcal{W} = FV_{temp}(F)$ where $FV_{temp}(F)$ denotes the free variables of F. For a complete definition of free variables, the reader may refer to (Abadi & Merz 1996). Intuitively, $\mathcal{W}$ constitutes the space state. A configuration is a partial mapping with finite domain from *free variables* of F to values, i.e:

$$\mathcal{K} : \mathcal{W} \rightarrow \mathcal{U}$$

In fact, the algorithm constructs a simulation tree where the root is defined by the initial states and every node is represented by a configuration. For each state s_i under construction, we want to generate a set of configurations $\{\mathcal{K}_0, \mathcal{K}_1, \ldots, \mathcal{K}_n\}$. We denote this set by $\mathcal{C}$. Next, we define the operation *configurations* inductively that takes as an argument a (set of) partial mappings and returns a set of configurations. Variables not initialized are provided by the user. Thus, the operation $input(x)$ denotes that the user is given a value for x. If x is a flexible variable then the user is asked to give a value or it can be generated randomly by the animator. If x is a rigid variable then the user is asked once to give a value for x. The figure 1 illustrates the algorithm.

[2] a state is mapping from variables to values

Definition 1 *Let τ be a valuation, The operation* CONFIGURATIONS *is defined as follows:*
CONFIGURATIONS$(\tau) = \{\mathcal{K} : \mathcal{K} \simeq \tau \wedge \mathcal{K}(x) = input(x)$ *for each* $x \in (\mathrm{dom}(\mathcal{K}) \setminus \mathrm{dom}(\tau))\}$

Definition 2 *The generalization of the operation* CONFIGURATION *on sets of valuations is defined as follows:*
CONFIGURATIONS$(T) = \bigcup_{\tau \in T}$CONFIGURATIONS$(\tau)$

Example 1 *Let $F \triangleq x = 0 \wedge \Box[y > 0 \wedge x' = x + 1]_{\langle x,y \rangle}$ where x and y are flexible variables. At time 0, we have by initialization:*

$$\mathcal{C}_0 = \{\mathcal{K}_0 : \mathcal{K}_0(x) = 0 \wedge \mathcal{K}_0(y) = 3\}$$

We assume that the user has chosen to instantiate y with 3. At time 1, we have by the construction of the i^{th} set of configurations:

$$\mathcal{C}_1 = \mathcal{C}_0 \cup \{\mathcal{K}_1 : \mathcal{K}_1(x) = 1 \wedge \mathcal{K}_1(y) = 4\}$$

We assume that the user has chosen to instantiate y with 4. The future set of configurations can be constructed in a similar way

A model checker

The model-checker uses an explicit state enumeration algorithm to check properties of the system, such as invariants. If the system fails to observe these properties, a counter-examples is generated by the verifier.

The verification algorithm in a subset of TLA explores the state graph described by a canonical formula, which encodes all possible executions of the system. An execution of the system is a finite or infinite sequence of states $s_0, s_1, \ldots$ where the initial state s_0 is described by the predicate $Init$. If s_i is any state in the sequence, s_{i+1} is obtained by applying some subaction of the next-state relation whose enabled predicate is true and whose body transforms s_i to s_{i+1}. In general, the state s_i may satisfied a several actions, so there is more than one execution (nondeterminism). In order to define the verification algorithm in TLA more precisely, we need to define some fundamental concepts.

Definition 3 *A state graph is a quadruple $A = \langle Q, Q_0, \Delta, error \rangle$, where Q is a set of states, and $\Delta \subseteq Q \times Q$ is a transition relation with the property that $q = error$ whenever $(error, q) \in \Delta$.*

The special state error state, **error**, is used as the next state whenever an invariant is violated or an error statement is executed. In TLA, a state graph is defined implicitly by a set of actions (transition predicates), where each action maps a state to a successor state. Formally, for all $s_1, s_2 \in Q$, we have $(s_1, s_2) \in \Delta$ if and only if there exists an action a such that $s_2 = s_1[\![a]\!]$.

Definition 4 *If $(s, t) \in \Delta$, then t is a successor of s, and s is a predecessor of t.*

Definition 5 *A finite sequence of states $\langle s_0, \ldots, s_n \rangle$ is called a path if $s_0 \in Q_0$ and s_i is a successor of s_{i-1} for all $1 \leq i \leq n$.*

Input Let $F = Init \wedge \Box[A]_x$ where $Init = \bigvee_{i=1}^{m} \bigwedge_{j=1}^{n} x_{ij} = c_{ij}$.

Output simulation tree

Initialization $\mathcal{C}_0 = \text{CONFIGURATIONS}(\bigcup_{i=1}^{m} \text{JOIN}_{j=1}^{n}\{[x_{ij} \leftarrow c_{ij}]\})$.

Construction build a set of nodes:

1. choose any subaction of A, saying B.
2. $\mathcal{C}_{i+1} = \bigcup_{s \in \mathcal{C}_i} \text{CONFIGURATIONS}(s[\![B \vee x' = x]\!])$.

Figure 1: The interpreter algorithm

Input Let $F = Init \wedge \Box[A]_x$ where $Init = \bigvee_{i=1}^{m} \bigwedge_{j=1}^{n} x_{ij} = c_{ij}$.
Hashtable Reached
Queue Unexpanded
Boolean deadlock

Output if an error has occurred then report error

Initialization build an initial state from init:

1. $\mathcal{C}_0 = \text{CONFIGURATIONS}(\bigcup_{i=1}^{m} \text{JOIN}_{j=1}^{n}\{[x_{ij} \leftarrow c_{ij}]\})$.
2. check that each state s in the set of successors satisfy the invariant. If it is not so then report error
3. Reached = Unexpanded = $\{s | s \in \mathcal{C}_0\}$

Successors Calculate the set of successors states of the state s:

1. $\mathcal{C}_i = \text{CONFIGURATIONS}(s[\![A]\!])$.
2. for each state s' in $\mathcal{C}_i$, check that s' satisfy the invariant. If it is not so then report error.
3. If s' satisfy the invariant and not in Reached add s' in Reached and Unexpanded.

Verifier build the state graph with bread-first search algorithm:

1. while Unexpanded $\neq \emptyset$ do
2. Remove a state s from Unexpanded
3. Calculate the set of successors states of the state s

Figure 2: A simple on-the-fly algorithm

Definition 6 *A state s is reachable if there exists a path $\langle s_0, \ldots, s \rangle$*

The algorithm in figure 2 checks whether an error is reachable. It is an on-the-fly algorithm, which generates and explores new states only when all previous states are known to be error-free. The states are stored in a hash-table, so that is that it can be decided efficiently whether or not a newly-reached state is old (has been examined already) or new (has not been examined already). New states are stored in queue of active states (whose successors still need to be generated). The verifier does a breadth-first search since it produces a shortest error trace if a problem is occurred.

Validation

Let us illustrate our framework for validation of TLA specifications by using a simple example, the algorithm of Hyman. This algorithm consists of two processes that tried to access to their critical section. The purpose of the system is to ensure mutually exclusive access to the critical sections for both the processes without starvation. In the below, the algorithm is given in a pseudo-algorithm:

```
var turn : bit;
process P1
 loop
  noncritical: ...;
  while (turn != 0)
   do if P2.noncritical then turn := 0 fi
  enddo;
  critical: ...
  endloop
||
process P2
 loop
  noncritical: ...;
  while (turn != 1)
   do if P1.noncritical then turn := 1 fi
  enddo;
  critical: ...
endloop
```

Writing and validating formal specifications

Like programs, writing a specification is a hard task and it is not errors free. In addition, Pneuli[3] points out that the process of verification is not enough since the initial specification may be inadequate, i.e. it fails to capture the user requirements. In the first time, writing the Hyman's algorithm seems a trivial task but it is not the case. As an exercise, we left the reader to write his own specification. We claim the following, that the animator help us to write a good specification that captures the user requirements. Indeed, testing increases our understanding of the specification and simply means validating that desirable scenarios are allowed by our TLA model, and that undesirable ones are forbidden. In respect to this algorithm, we can check that each process must waits until its turn i.e. that we have specified the loop correctly. This is the main difficulty in this algorithm.

Validation and verification of a property

When we have gain confidence in our TLA model and be convinced that this model captures the algorithm. We turn back to the rest of the requirements. Indeed, we wish to verify the invariance property that assert the two processes never are togther in the critical section. We start by formulating our invariant. The validation of the invariant can be done by first checking that the predicate $Init$ satisfies the invariant and secondly by checking that the execution of actions are closed with respect to the invariant. This is done by checking the validity of the invariant at the current state before the execution of the action and checking the validity of the invariant after the execution of this action. With the animator, we can validate the invariant as explained above. But, we have only validated it. In addition, the animator can be used to find the right invariant. When we have gain confidence that the invariant is true then we can use model checking to prove it provided the state space is finite. In respect to this algorithm, we found that the invariant is breaked and the algorithm is not correct.

Back to our requirements, it remains to ensure that each process can be in its critical section without starvation i.e. each process that has formulated a request for the critical section eventually can get it. This is a liveness requirement. In order to satisfy this requirement, we must define a fairness requirements. And again, we are faced with another problem in writing our TLA specification i.e. which kind of fairness requirements we must define on which actions to guarantee this requirement. To do so, we formulate the negation of our requirements, we build its automata and then we can use our model checking constrained by our schedulers to test which kind of fairness is required. We start our checking without fairness requirements. Our tool generates a counter-example. By examining, the counter-example we have noticed that the first process remains in the loop while the second process can have accessed several times to its critical section. Thus, we must define a fairness requirement to allow to the first process to break this loop and goes eventually to its critical section. Thus, we add a weak fairness. But, again it doesn't work. Thus we try with a strong fairness on the same action, now the tool generates another counter-example. If we continue the same process, we can find the all fairness requirements needed. Note that this method can give good results only if our specification is deterministic. Thus, the tool does not generate any counter-example. This fact does not mean that our specification is free of errors. This is due to the incompleteness of our tool since we don't generate all the traces. What we have gained is the confidence that may be the fairness requirements that we have found are convenient for our specification. Thus, we have finished the validation of our specification and we are ready to use a model checker to verify the liveness property definitively.

[3]http://www.wisdom.weizmann.ac.il/ amir/invited-talks.html

Conclusion

We have shown how two tools, an animator and a model checker, can be combined to form a single framework for validation. Together, tools[4] can be applied to problems that would be difficult to handle with either tool separately. The combination of these tools is facilitated by using the same language. The experience gained by the current prototype in Java indicates that it is possible to create a reasonably efficient tool for the validation of TLA specifications. Therefore much work is still needed of course like applying our methodology on more realistic examples. We have chosen to develop another model checker and do not consider the use of other model checkers like SMV (McMillan 1993) and Murϕ (Dill 1996) for the same reasons invoked in (Yu, Manolios, & Lamport).

[4]you can drop an email to the one of the author to get a copy of these tools

References

Abadi, M., and Merz, S. 1996. On TLA as a logic. In Broy, M., ed., *Deductive Program Design*, NATO ASI series, 235–272. Springer-Verlag.

Dill, D. L. 1996. The murϕ verification system. In *Conference on Computer-Aided Verification*, LNCS, 390–393. Springer-Verlag.

Fisher, M., and Owens, R. 1993. Introduction to executable modal and temporal logics. In Fisher, M., and Owens, R., eds., *Executable Modal and Temporal Logics*, volume 897 of *LNCS*, 1–20. Springer.

Lamport, L. 1994. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems* 16(3):872–923.

Lamport, L. 1996. Refinement in state-based formalisms. Technical Report 1996-001, Digital Equipment Corporation, Systems Research Center, Palo Alto, California.

McMillan, K. L. 1993. *Symbolic model checking*. Kluwer Academic Publishers.

Merz, S. 1993. Efficiently executable temporal logic programs. In Fisher, M., and Owens, R., eds., *Executable Modal and Temporal Logics*, volume 897 of *LNCS*, 69–85. Springer.

Mokhtari, Y., and Merz, S. 1999. Animating tla specifications. In Ganzinger, H.; McAllester, D.; and Voronkov, A., eds., *6th International Conference on Logic for Programming and Automated*, volume 1705 of *Lecture Notes in Artificial Intelligence*, 92–110. Springer-Verlag.

Yu, Y.; Manolios, P.; and Lamport, L. Model checking tla+ specifications. Disponible www.research.digital.com/SRC/tla/.