Animating TLA Specifications [⋆]

Yassine Mokhtari[1] and Stephan Merz[2]

[1] LORIA-UMR n^{o}7503, Université Henri Poincaré, Nancy, France
[2] Institut für Informatik, Universität München, Germany
mokhtari@loria.fr, merz@informatik.uni-muenchen.de

Abstract. TLA (the Temporal Logic of Actions) is a linear temporal logic for specifying and reasoning about reactive systems. We define a subset of TLA whose formulas are amenable to validation by animation, with the intent to facilitate the communication between domain and solution experts in the design of reactive systems.

1 Introduction

The Temporal Logic of Actions (TLA) has been proposed by Lamport [21] for the specification and verification of reactive and concurrent systems. TLA models describe infinite sequences of states, called behaviors, that correspond to the execution of the system being specified. System specifications in TLA are usually written in a canonical form, which consists of specifying the initial states, the possible moves of the system, and supplementary fairness properties. Because such specifications are akin to the descriptions of automata and often have a strongly operational flavor, it is tempting to take such a formula and "let it run". In this paper, we define an interpreter algorithm for a suitable subset of TLA. The interpreter generates (finite) runs of the system described by the specification, which can thus be validated by the user.

For reasons of complexity, it is impossible to animate an arbitrary first-order TLA specification; even the satisfiability problem for that logic is Σ_1^1-complete. Our restrictions concern the syntactic form of specifications, which ensure that finite models can be generated incrementally. They do not constrain the domains of system variables or restrict the non-determinism inherent in a specification, which is important in the realm of reactive systems.

In contrast, model checking techniques allow to exhaustively analyse the (infinite) runs of finite-state systems. It is generally agreed that the development of reactive systems benefits from the use of both animation for the initial modelling phase, complemented by model checking of system abstractions for the verification of crucial system components.

The organization of the paper is as follows: in sections 2 and 3 we discuss the overall role of animation for system development, illustrating its purpose at the hand of a simple example, and discuss executable temporal logics. Section 4 constitutes the main body of this paper; we there define the syntax and semantics of an executable

[⋆] This work was partly supported by a grant from DAAD and APAPE under the PROCOPE program.

subset of TLA, give the interpreter algorithm, and prove it sound and complete with respect to the logical semantics. Section 5 shows how fairness conditions can be taken into account and briefly describes our current prototype implementation, which also contains a model checking component. Finally, section 6 concludes by summarizing our results and comparing them to related work.

2 The role of animation

Requirements capture and analysis is the first step in typical lifecycle models of software engineering. It is the process of identifying and recording the needs of a customer. It fulfills two different roles:

- The customer must be convinced that the requirements are completely understood and recorded.
- The designer must be able to use the requirements to produce a structure around which formal reasoning and an implementation can be developed.

The success of this step depends on the communication between customer and designer. In general, the analysis and requirements document constitutes the interface between *problem domain experts* with little knowledge of computers and *solution domain professionals* with little knowledge of the problem and a large understanding of computer systems and techniques. Thus, the communication should be oriented towards the customer: the formal model of requirements should be understood and communicated to the customer. Validation is the key technique that supports this communication.

In the formal methods community, the validation of formal specifications is mainly based on either (automatic or computer-supported interactive) proof or on animation, that is, execution. The verification of TLA specifications has been amply studied; we focus here on validation by animation. Hoare [16] has studied the feasibility of animating specifications. Emphasizing the tradeoff between expressiveness of the formal notations and their animation, he has initiated a lively debate in the community. Hayes and Jones [17] represent one side of this debate; they explain that specifications need not be executable by considering various problems involving different formal notations. They argue that the expressiveness should not be sacrificed in favor of animation since a formal specification is intended for "human consumption", whereas the additional requirements of executability may easily lead to over-specification. Consequently, the implementor may be tempted to follow the algorithmic structure of the executable specification. Moreover, Hayes and Jones argue that validation is achieved by proof rather than by animation, and that animation techniques should be restricted to particular problems such as the design of user interfaces [15]. As an exponent of the other side, Fuchs [11] argues that the specifications are (preferably) executable. He considers the same problems stated by Hayes and Jones and translates them into a logic specification language (LSL), which is more expressive than Prolog. He emphasizes the value of animation as the primary vehicle of communication between the client and the designer.

The limitations in expressiveness can be overcome by appropriate specification languages. For example, the B-Toolkit [3,4] supports type-checking, animation, and proofs. When there is non-determinism, the user is referred to as an oracle. In DisCo [7], the concept of external functions is used to extend the notations.

Of course, animation should not be thought of as a panacea; it certainly does not exclude formal proofs. Rather, proofs and tests are complementary and should be used in conjunction. As an example, Rushby [26] considers the problem of sorting a sequence. The question whether a sorting function is idempotent can be addressed by proof or by animation. The latter can examine the property for a few representative values. Most importantly, animation can help us to ask the right questions. As with model-checking, the goal is to find a counter-example. When we have gained confidence in both the specification and the property then we can use a theorem-prover to attempt a formal proof.

Related to implementation, Zave and Yeh [31,32] argue that the border between implementations and executable specifications is resource management. Implementations have to meet performance goals by an optimal use of resources while executable specifications should only specify the functional properties of the system. There is also a distinction between prototypes and executable specifications. Prototypes are mainly used to explore only part of the functionality of the system. On the other hand, executable specifications form the basis of the implementation [11].

3 Validating specifications by animation

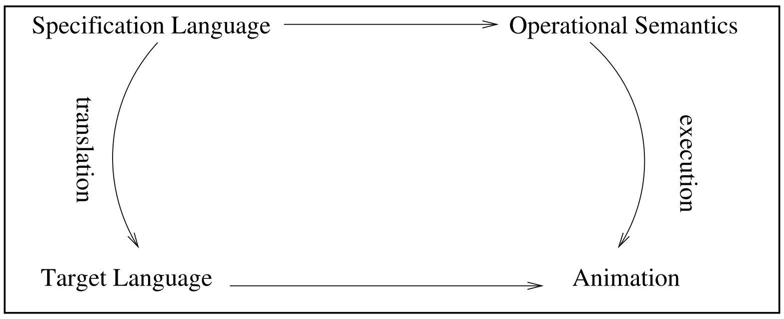

$$\begin{tikzcd}
\text{Specification Language} \arrow[r] \arrow[d, bend right, "\text{translation}"'] & \text{Operational Semantics} \arrow[d, bend left, "\text{execution}"] \\
\text{Target Language} \arrow[r] & \text{Animation}
\end{tikzcd}$$

Fig. 1. Overall approach to animation [27]

Figure 1 illustrates two approaches to animate formal specifications. The first approach consists of (automatically) translating specifications into a target language, which is immediately executable. In the second approach, one defines an operational semantics for the specification language, perhaps using a standard SOS format, which can then be interpreted by a custom-built interpreter or even using a standard tool. Theoretically, the two approaches are equivalent: in both cases we must preserve correctness (and completeness if possible). Technically, the definition of the animation by the translation is a programming activity, and proving properties of the translation is often nontrivial.

The second approach favors compositionality of the translation, which simplifies formal reasoning.

3.1 Criteria for animation systems

Breuer and Bowen [6] identify three qualitative measures that can be used to compare animation systems: coverage, efficiency and sophistication. Utting [28] suggests three additional evaluation criteria: interactivity, transparency and operational equivalence. Among these criteria, we have chosen the following requirements which allow us to classify the different techniques of animation:

- the expressiveness of the animation language,
- efficiency, measured as the time and space requirements for execution,
- correctness: every outcome of the execution conforms to the original specification, and
- completeness: every model of the original specification is a possible outcome of the animation.

3.2 Using animation to validate systems

We illustrate what kinds of specification errors the animation may help to detect, and why it is useful to combine animation and proof. Figure 2 shows an example due to Gravell [15], but reformulated in TLA$^+$, a specification language introduced by Lamport [19,20] and based on top of TLA. The specification should be easy to understand even without detailed knowledge of TLA$^+$.

module *Counter*

VARIABLE
x

$$\begin{aligned}
\mathit{Init} &\triangleq x = 0 \\
\mathit{Next} &\triangleq \land\, x \leq 995 \\
&\ \land\, x' = x + 5 \\
\mathit{Fair} &\triangleq \mathrm{WF}_x(\mathit{Next})
\end{aligned}$$

$$\begin{aligned}
\mathit{Spec} &\triangleq \land\, \mathit{Init} \\
&\ \land\, \Box[\mathit{Next}]_x \\
&\ \land\, \mathit{Fair}
\end{aligned}$$

THEOREM
$\mathit{Invariant} \triangleq \Box(x \in \mathit{Nat} \land x < 1000)$

Fig. 2. Specification of a counter.

As explained by Gravell, there are two problems with this specification. The first one concerns the test $x \leq 995$. When $x = 995$, the action Next is enabled and can thus be executed. But, the invariant is not preserved. This kind of errors can be found easily by animation. 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 initial state, and again after the execution of any action.

On the other hand, the animation points out that the invariant, when corrected, could be strengthened by asserting that $x \bmod 5 = 0$. For example, this stronger invariant implies that an implementation of the counter could internally use an 8 bit representation of integers. On the other hand, it could be the case that this invariant holds just "by accident" and is not actually desired by the client.

Although this example is trivial, it illustrates what we expect from using animation. As a realistic example, Gravell [15] has discovered an error in a specification which had been published in a book about Modula-3.

3.3 Executable Temporal Logic

Temporal logic [18,23] is a standard framework for specifying and reasoning about reactive systems. It combines classical logic for assertions concerning single states with temporal operators expressing assertions that relate several states. The behaviors of reactive systems are modelled as infinite sequences of states. Temporal logic can be used to model reactive systems at a high level of expressiveness. If we can directly execute (a reasonable subset of) temporal logic specifications, they can be validated early on, without constructing and verifying intermediate refinements of the original specification. For this reason, there has been some interest in executing temporal logic specifications [9], and we review the central points.

The "execution" of a temporal formula F aims to build a model for F. The outcome is therefore a model $\mathcal{M}$ such that $\mathcal{M} \models F$ holds. In general, an animator will only produce finite prefixes of infinite behaviors. It should be the case that any finite sequence of states produced by the animator can be extended to a model of the input specification. Moreover, the animator should be able in principle to produce prefixes of every model, possibly with some guidance by the user in the case of non-determinism.

The construction of models for propositional temporal logic specifications is usually based on automata-theoretic techniques [30,8]. The satisfiability problem is decidable in the propositional case, but it is rather complex (PSPACE-complete). For first-order temporal logics, the satisfiability problem is highly undecidable (Σ_1^1-complete), and it is not in general possible to incrementally construct models. Faced with these problems, Merz [24] suggests to find a tradeoff between expressiveness and efficient implementability.

We have chosen a subset of first-order TLA as our specification language. As opposed to model checking, we want to be able to animate specifications that involve complex, unbounded data structures, while we are prepared to give up on full coverage of the state space. Compared to other temporal logics, TLA differs in that it emphasizes automata-like descriptions of reactive systems, which have proven to be scalable to specifications of realistic size. It provides a simple logical language to describe both systems and their properties, an important point when one is interested in refinement, and thereby simplifies the interface between the domain and solution experts.

4 Animating TLA specifications

4.1 Overview

TLA specifications are usually written in the canonical form [21]

$$\mathit{Init} \wedge \Box[N]_x \wedge L$$

where *Init* describes the initial states of the machine, N is the next-state relation, written as a disjunction of possible moves, x is a tuple of all variables of interest and L is a formula that describes the fairness requirements. A TLA specification thus defines an *abstract machine* whose state space is defined by the variables of the specification and whose transition relation is described by actions (transition predicates). An *execution* is modelled as an infinite sequence of states, called *behavior*, where a terminating execution is modelled as repeating the final state.

As a first step, we ignore fairness conditions, which will be considered in section 5.1, and concentrate on the safety part of specifications, written as

$$\mathit{Init} \wedge \Box[N]_x$$

Such a formula is satisfied by a behavior whose initial state satisfies the predicate *Init* and where every pair of successive states satisfies N, or else doesn't change x. Thus, the key requirement for animation is the ability to generate a *finitely representable* set of successors of each state. We can now state this idea more formally:

Definition 1. A specification in TLA can be characterized by a triple (St, I, A) where St is a set of all possible states, $I \subseteq \mathit{St}$ is the set of initial states, and A is a set of actions.

The interpretation of a state predicate p, written $[\![p]\!]$, is a mapping from states to the booleans. The interpretation of an action a, written $[\![a]\!]$, is a boolean valued function on steps, where a step is a pair of states. For each action $a \in A$, we can define its *enabling condition*, written $\text{ENABLED}\,(a)$ as the state predicate that is true precisely in those states where a may be executed. Semantically, $\text{ENABLED}\,(a)$ is defined by

$$s[\![\text{ENABLED}\,(a)]\!] \;\triangleq\; \exists\, t \in \mathit{St} : s[\![a]\!]t$$

A necessary condition for a specification to be executable is that for every (reachable) state s and every action $a \in A$, the set

$$\{t \in \mathit{St} : s[\![a]\!]t\}$$

of legal successor states of s w.r.t. a be recursively enumerable. We will restrict the action syntax in order to ensure this condition:

- 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 allowed 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.

The following sections formally define the syntax and semantics of our executable subset of TLA and describe an algorithm that generates (finite) models of such specifications. We show that our interpreter algorithm is both correct and complete as explained in section 3.1.

One may argue that our restrictions on executable formulas are too severe and result in a specification language of insufficient expressivity. On the other hand, it is clear that for fundamental reasons of computability theory, it is impossible to syntactically characterize the full set of specifications such that the set of legal successor states is recursively enumerable. We expect to extend the class of allowed specifications given more experience with our present prototype animator.

4.2 The logic of transitions

Like TLA, the subset of TLA has two tiers, with the temporal formulas defined on top of nontemporal transition formulas. We now define the syntax and semantics of transition formulas.

Syntax We assume given a denumerable set of variables $\mathcal{V}$. These are partitioned into denumerable sets $\mathcal{V}_R$ of rigid variables and $\mathcal{V}_F$ of flexible variables.

We also assume given a sequence $\mathcal{L}$ of symbols, partitioned into a sequence $\mathcal{L}_P$ of predicate symbols and a sequence $\mathcal{L}_F$ of function symbols. To each of the symbols in $\mathcal{L}$ is assigned a natural number, its arity.

The syntax of nontemporal transition formulas has three tiers:

1. The first tier concerns the *constant formulas* whose meaning is state-independent. Rigid variables may occur in these formulas.
2. The second tier concerns the *state formulas* whose meaning is state-dependent. State formulas comprise *state functions* and *state predicates*. Both *rigid variables* and *flexible variables*, whose value is state-dependent, may occur in state formulas.
3. The third tier concerns the *transition formulas*; they comprise only *transition predicates* called *actions*. Transition formulas may contain primed occurrences of flexible variables.

We will explicitly define state and transition formulas. Constant formulas are state formulas that do not contain flexible variables.

State Functions. The set of state functions is the smallest set such that:

- If $x \in \mathcal{V}_F \cup \mathcal{V}_R$ then x is a state function.
- If $f \in \mathcal{L}_F$ is a function symbol of arity n and $v_1, \ldots, v_n$ are state functions then $f(v_1, \ldots, v_n)$ is a state function.

State Predicates. The set of state predicates is the smallest set such that:

- If $p \in \mathcal{L}_P$ is a predicate symbol of arity n and $v_1, \ldots, v_n$ are state functions then $p(v_1, \ldots, v_n)$ is a state predicate.
- If v_1 and v_2 are state functions then $v_1 = v_2$ is a state predicate.
- If P is a state predicate then $\neg P$ is a state predicate.
- If P and Q are state predicates then $P \wedge Q$ is a state predicate.

Further connectives like $\vee$, $\Rightarrow$ and $\equiv$ are defined as standard abbreviations.

Actions. The set of actions (transition predicates) is the smallest set such that:

- If P is a state predicate then P is an action.
- If $x \in \mathcal{V}_F$ and v is a state function then $x' = v$ is an action.
- If A and B are actions then $A \wedge B$ and $A \vee B$ are actions.
- If P is a state predicate and A is an action then $P \Rightarrow A$ is an action.

We sometimes write $x' = x$ where x is a finite list $\langle x_1, \ldots, x_n \rangle$ of flexible variables to denote the conjunction of all actions $x_i' = x_i$.

Logical semantics The basic semantical concept of $\mathcal{L}$ is a structure $\mathcal{M}$ that consists of:

- a non-empty domain $\mathcal{U}$ called a *universe*.
- an n-ary function $\mathcal{M}(f) : \mathcal{U}^n \to \mathcal{U}$ for every n-ary function symbol f.
- an n-ary predicate $\mathcal{M}(p) \subseteq \mathcal{U}^n$ for every n-ary predicate symbol p.

A *rigid variable valuation* (with respect to $\mathcal{U}$) assigns some $\xi(x) \in \mathcal{U}$ to every $x \in \mathcal{V}_R$. In this paper, we assume given a fixed structure $\mathcal{M}$ and valuation ξ of the rigid variables.

The semantics is defined in terms of *states*. A *state* is a mapping from flexible *variables* $\mathcal{V}_F$ to *values* from $\mathcal{U}$. Thus, a state $s \in St$ assigns a value $s(x)$ to every flexible variable $x \in \mathcal{V}_F$.

State functions. The meaning $[\![v]\!]$ of a state function v is a mapping from states to values in $\mathcal{U}$. For every state s, we define $s[\![v]\!]$ by induction as follows:

- If $x \in \mathcal{V}_F$ then $s[\![x]\!]$ is $s(x)$.
- If $x \in \mathcal{V}_R$ then $s[\![x]\!]$ is $\xi(x)$
- $s[\![f(v_1, \ldots, v_n)]\!]$ is $\mathcal{M}(f)(s[\![v_1]\!], \ldots, s[\![v_n]\!])$

State Predicates. The meaning $[\![P]\!]$ of a state predicate P is a mapping from states to booleans, so $s[\![P]\!]$ equals true or false for every state s. We say that a state s *satisfies* a predicate P iff $s[\![P]\!]$ equals true. The semantics of state predicates is inductively defined as follows:

- $s[\![v_1 = v_2]\!]$ is true iff $s[\![v_1]\!]$ and $s[\![v_2]\!]$ are equal.
- $s[\![p(v_1, \ldots, v_n)]\!]$ is true iff $(s[\![v_1]\!], \ldots, s[\![v_n]\!]) \in \mathcal{M}(p)$.
- $s[\![\neg P]\!]$ is true iff $s[\![P]\!]$ is false.
- $s[\![P \wedge Q]\!]$ is true iff both $s[\![P]\!]$ and $s[\![Q]\!]$ are true.

Actions. An action (transition predicate) represents a relation between states, where the unprimed variables refer to the state before the transition and the primed variables refer to the state thereafter.

Formally, we say that a pair (s, t) of states *satisfies* an action $\mathcal{A}$, and we write s$[\![\mathcal{A}]\!]$t iff:

- If $\mathcal{A}$ is a state predicate P then $s[\![\mathcal{A}]\!]t$ is true iff $s[\![P]\!]$ is true.
- If $\mathcal{A}$ is $x' = v$ then $s[\![\mathcal{A}]\!]t$ is true iff $t[\![x]\!]$ and $s[\![v]\!]$ are equal.
- If $\mathcal{A}$ is $A \wedge B$ then $s[\![\mathcal{A}]\!]t$ is true iff $s[\![A]\!]t$ and $s[\![B]\!]t$ are true.
- If $\mathcal{A}$ is $A \vee B$ then $s[\![\mathcal{A}]\!]t$ is true iff $s[\![A]\!]t$ or $s[\![B]\!]t$ is true.
- If $\mathcal{A}$ is $P \Rightarrow A$ then $s[\![\mathcal{A}]\!]t$ is true iff $s[\![P]\!]$ is false or $s[\![A]\!]t$ is true.

Operational semantics We now complement the traditional, logical semantics of TLA defined above with an operational semantics that allows us to effectively evaluate actions. This operational semantics is based on the first-order structure $\mathcal{M}$, which we assume to be effectively presented. We are not concerned with how exactly $\mathcal{M}$ is defined; in practice, it will be provided by a host language, possibly extended with algebraic data types [14].

Informally, if $s[\![\mathcal{A}]\!]t$ holds for a pair (s, t) of states then the execution of $\mathcal{A}$ in state s can produce the new state t. We effectively construct t with the help of an operational semantics of actions that allow us to build the state t incrementally. At each step of the construction, we have partial information about the state t. This partial information is represented by a *valuation*. Operationally, we define the meaning of an action in a state s as a set of valuations with finite domains. For example, $s[\![x' = x + 1]\!]$ is the set that contains just the valuation $[x \leftarrow \mathcal{M}(+)(s[\![x]\!], \mathcal{M}(1))]$. We need two fundamental notions. The first one ensures that two valuations τ_1 and τ_2 are compatible, that is, they agree on the value of all variables they both determine. The second notion is the operation JOIN that allows us to compose sets of valuations.

Definition 2. A valuation τ is a (possibly partial) mapping from $\mathcal{V}_F$ to $\mathcal{U}$ .i.e.

$$\tau : \mathcal{V}_F \to \mathcal{U}$$

Note that a state s can be regarded as a valuation with dom$(s) = \mathcal{V}_F$. However, in the following the domain of τ will often be finite. We write $[x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n]$ for the valuation τ with dom$(\tau) = \{x_1, \ldots, x_n\}$ and $\tau(x_i) = v_i$. In particular [ ] denotes the trivial valuation that is nowhere defined.

Definition 3. Let τ_1 and τ_2 be two valuations. We say that τ_1 and τ_2 are compatible, written $\tau_1 \simeq \tau_2$, iff $\tau_1(x) = \tau_2(x)$ for every $x \in$ dom$(\tau_1) \cap$ dom(τ_2).

Definition 4. The composition of two compatible valuations τ_1 and τ_2, written $\tau_1 \bullet \tau_2$, is the valuation such that:

- dom$(\tau_1 \bullet \tau_2) =$ dom$(\tau_1) \cup$ dom(τ_2) and

– for every variable $x \in \text{dom}(\tau_1 \bullet \tau_2)$:

$$(\tau_1 \bullet \tau_2)(x) \;=\; \begin{cases} \tau_1(x) \text{ if } x \in \text{dom}(\tau_1) \\ \tau_2(x) \text{ if } x \in \text{dom}(\tau_2) \end{cases}$$

Definition 5. The operation JOIN is defined by

$$S \text{ JOIN } T \;=\; \{\rho \bullet \tau : \rho \in S, \tau \in T, \rho \simeq \tau\}$$

for any two sets S and T of valuations.

We note the following properties of these definitions, which are later used to establish the soundness and completeness of our semantics.

Lemma 6. *If* $\tau \in \text{JOIN}_{i=1}^{n}\{[x_i \leftarrow c_i]\}$ *then dom*$(\tau) = \{x_1, \ldots, x_n\}$ *and* $\tau(x_i) = c_i$ *for every* $x_i \in$ *dom*(τ).

Lemma 7. *Let* S, T *be two sets of valuations and t be a state.*
There exists some $\tau \in (S \text{ JOIN } T)$ *such that* $t \simeq \tau$ *iff there exist* $\tau_1 \in S$ *and* $\tau_2 \in T$ *such that* $t \simeq \tau_1$ *and* $t \simeq \tau_2$.

Now, we can define the meaning of an action as a mapping from states to sets of valuations. The semantics of actions is defined by structural induction as follows:

– $s[\![x' = v]\!] = \{[x \leftarrow s[\![v]\!]]\}$.
– If P is a state predicate and $s[\![P]\!]$ is true then $s[\![P]\!] = \{[]\}$.
– If P is a state predicate and $s[\![P]\!]$ is false then $s[\![P]\!] = \{\}$.
– $s[\![A \wedge B]\!]$ is $s[\![A]\!]$ JOIN $s[\![B]\!]$.
– $s[\![A \vee B]\!]$ is $s[\![A]\!] \cup s[\![B]\!]$ where $\cup$ is set union.
– If $s[\![P]\!]$ is true then $s[\![P \Rightarrow A]\!]$ is $s[\![A]\!]$.
– If $s[\![P]\!]$ is false then $s[\![P \Rightarrow A]\!]$ is $\{[]\}$.

4.3 Temporal formulas

Temporal formulas are built on the basis of transition formulas as defined above. We now define their syntax and semantics.

Syntax The only form of temporal formulas is

$$\mathit{Init} \wedge \Box[N]_x$$

where:

– *Init* is a state predicate describing the initial values of x. We restrict *Init* to be of the form $\bigvee_{i=1}^{m} \bigwedge_{j=1}^{n} x_{ij} = c_{ij}$ where c_{ij} is a constant and x_{ij} is a variable.
– N is an action describing the next-state relation.
– x is a tuple of flexible variables that appear in N. We require each step to satisfy N or else leave all variables in x unchanged. We may consider such steps as being performed by the environment.

Semantics The semantics of temporal formulas is defined on behaviors, that is, infinite sequences of states. If $\sigma = \langle s_0, s_1, \ldots \rangle$ is a behavior then $\sigma_{|n}$ is its prefix $\langle s_0, \ldots, s_n \rangle$. Let St^∞ denote the set of all behaviors.

Definition 8. Let $\sigma = \langle s_0, s_1, \ldots \rangle \in St^\infty$ be a behavior. We say that σ *satisfies* the temporal formula $\mathit{Init} \wedge \Box[N]_x$, written $\sigma[\![\mathit{Init} \wedge \Box[N]_x]\!]$ iff $s_0[\![\mathit{Init}]\!]$ is true and for all $n \in \mathit{Nat}$, $s_n[\![N \vee x' = x]\!]s_{n+1}$ is true.

A fundamental result about TLA asserts that all formulas are invariant under stuttering, that is, finite repetitions of identical states. Formally, *stuttering equivalence* is the finest equivalence relation on behaviors such that any two behaviors $\pi \circ \langle s, s \rangle \circ \sigma$ and $\pi \circ \langle s \rangle \circ \sigma$ are stuttering equivalent. Invariance under stuttering allows refinement to be represented by logical implication. All formulas written in our restricted logic are therefore also stuttering invariant. The proof of the following proposition appears in [2].

Proposition 9. *Let $F \equiv \mathit{Init} \wedge \Box[A]_x$.*
If σ and τ are two stuttering equivalent behaviors, then $\sigma[\![F]\!]$ holds iff $\tau[\![F]\!]$ holds.

4.4 Soundness and completeness of our semantics

We now define an operational semantics for temporal formulas, based on the operational semantics for actions, and relate it to the logical semantics by proving soundness and completeness theorems.

Let $F \equiv \mathit{Init}/\ \Box[A]_x$ be a temporal formula. We write $\textsc{traces}(F)$ to denote the set of finite behaviors that satisfy F. Formally, $\textsc{traces}(F)$ is defined by induction as follows:

Definition 10. $\textsc{traces}(F)$ is the smallest set such that:

- If $\mathit{Init} \equiv \bigvee_{i=1}^m \bigwedge_{j=1}^n x_{ij} = c_{ij}$ then $\langle s_0 \rangle \in \textsc{traces}(F)$ iff $s_0 \simeq \tau$ for some $\tau \in \bigcup_{i=1}^m \textsc{join}_{j=1}^n \{[x_{ij} \leftarrow c_{ij}]\}$.
- If $\langle s_0, \ldots, s_n \rangle \in \textsc{traces}(F)$ then $\langle s_0, \ldots, s_{n+1} \rangle \in \textsc{traces}(F)$ iff $s_{n+1} \simeq \tau$ for some $\tau \in s_n[\![Ax' = x]\!]$.

$\textsc{traces}(F)$ is therefore the set of finite behaviors that can be constructed by repeatedly applying the operational semantics for actions from section 4.2. It defines the operational semantics of temporal formulas.

Lemmas 11 and 12 establish the soundness and completeness of our semantics for the initial predicate and for actions.

Lemma 11. *$s_0[\![\bigwedge_{i=1}^n x_i = c_i]\!]$ is true iff $s_0 \simeq \tau$ for some $\tau \in \textsc{join}_{i=1}^n \{[x_i \leftarrow c_i]\}$.*

Lemma 12. *Let $\mathcal{A}$ be an action and s, t be states. $s[\![\mathcal{A}]\!]t$ is true iff $t \simeq \tau$ holds for some $\tau \in s[\![\mathcal{A}]\!]$.*

These lemmas are the essential steps in proving the following theorems 13 and 14, which assert the soundness and completeness of the operational semantics with respect to the logical semantics.

Theorem 13 (Soundness). *Let $F \equiv Init \wedge \Box[N]_x$ and $\sigma \in St^\infty$ be a behavior. If $\sigma_{|n} \in \text{TRACES}(F)$ holds for all $n \in \mathbb{N}$ then $\sigma[\![F]\!]$ is true.*

Theorem 14 (Completeness). *Let $F \equiv Init \wedge \Box[N]_x$ and $\sigma \in St^\infty$ be a behavior. If $\sigma[\![F]\!]$ is true then $\sigma_{|n} \in \text{TRACES}(F)$ holds for all $n \in \mathbb{N}$.*

4.5 The interpreter algorithm

The interpreter algorithm iteratively constructs a set of finite behaviors that are prefixes of models of an executable specification F. Let $F \equiv Init / \Box[A]_x$ be a temporal formula and $\mathcal{W} = FV_{temp}(F)$ be the free flexible variables of F. Intuitively, $\mathcal{W}$ constitutes the space state. A *configuration* $\mathcal{K}$ is a mapping from $\mathcal{W}$ to values, i.e.

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

In fact, the algorithm constructs a forest of simulation trees whose roots correspond to the initial states and whose nodes are configurations. This tree represents the set of finite behaviors allowed by the formula F.

For each state under construction, we want to generate a set $\mathcal{C}$ of configurations, based on valuations τ produced by the operational semantics defined above. It only remains to assign values to any variables in $\mathcal{W}$ that are not in the domain of τ. Uninitialized variables typically correspond to environment inputs; their values have to be provided by the user or be randomly generated by the animator. If τ is a valuation with $\text{dom}(\tau) \subseteq \mathcal{W}$, we let $\text{CONFIGURATIONS}(\tau)$ denote the valuation $\tau \bullet input$ where $input$ is some valuation with domain $\mathcal{W} \setminus \text{dom}(\tau)$. We extend CONFIGURATIONS to sets of valuations in the obvious way. Figure 3 illustrates the algorithm.

Input Let $F \equiv Init \wedge \Box[A]_x$ where $Init \equiv \bigvee_{i=1}^{m} \bigwedge_{j=1}^{n} x_{ij} = c_{ij}$.
Output simulation tree
Initialization $\mathcal{C}_0 = \text{CONFIGURATIONS}\left(\bigcup_{i=1}^{m} \text{JOIN}_{j=1}^{n}\{[x_{ij} \leftarrow c_{ij}]\}\right)$.
Construction

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

Fig. 3. The interpreter algorithm.

Example. To illustrate the algorithm, consider the following example. Let

$$F \equiv x = 0 \wedge \Box[y > 0 \wedge x' = x + 1 \wedge y' = y + 1]_{\langle x,y \rangle}$$

where x and y are flexible variables. At time 0, we may have by initialization:

$$\mathcal{C}_0 = \{[x \leftarrow 0, y \leftarrow 3]\}$$

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

$$\mathcal{C}_1 = \mathcal{C}_0 \cup \{[x \leftarrow 1, y \leftarrow 4]\}$$

The construction continues in a similar way. Figure 4 illustrates the simulation tree.

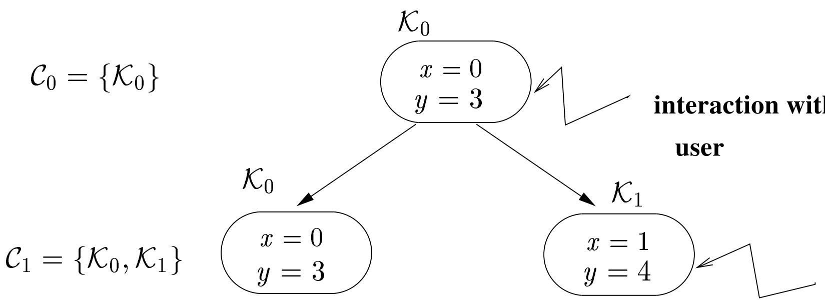


Fig. 4. Simulation tree

5 Extensions

We extend the basic set of executable TLA formulas considered so far by fairness conditions and give a brief indication of the current implementation, which complements the animator with a simple model checker for the analysis of finite-state specifications.

5.1 Fairness requirements

Fairness conditions are used to constrain the nondeterministic choices present in the specifications of concurrent systems at an abstract level of description. For example, consider the specification of the manager of a shared resource. We may use a fairness condition to require that every request must be eventually served. A typical implementation may queue the waiting requests. But, including a queue in the specification of the resource manager has at least two drawbacks. On one hand, this solution over-specifies the requirements and on the other hand it enforces implementation decisions at the requirements level. Observe moreover that without fairness requirements, a TLA component specified by a canonical formula $\mathit{Init} \wedge \Box[N]_v$ may at some point simply stop operating, since every stuttering action satisfies the safety requirement.

In general, fairness conditions assert that an action that is enabled often enough will eventually be executed. Standard interpretations of "often enough" are either "forever from some point onwards" (*weak fairness*) or "infinitely often" (*strong fairness*). Clearly, strong fairness implies weak fairness. In TLA, these fairness conditions can be defined as

$$\mathrm{WF}_f(A) \triangleq (\Box\Diamond\langle A\rangle_f) \vee (\Box\Diamond\neg\textsc{Enabled}\ \langle A\rangle_f)$$
$$\mathrm{SF}_f(A) \triangleq (\Box\Diamond\langle A\rangle_f) \vee (\Diamond\Box\neg\textsc{Enabled}\ \langle A\rangle_f)$$

We allow conjunctions of weak and strong fairness conditions to appear in executable formulas. They are implemented with the help of *schedulers*, ensuring that every infinite behavior that is a limit of the finite behaviors produced by the animator satisfies the fairness conditions as well as the safety part of the given specification. Of course, the choice of a fixed scheduler destroys the completeness property, since the nondeterminism of the original specification is constrained. In our implementation, the user has the freedom to override the scheduler's choice of which action to execute next.

For the sake of clarity, we separately define schedulers for specifications that contain either weak or strong fairness conditions. It is easy to combine the algorithms to obtain a scheduler for specifications that contain both types of fairness requirements.

Weak Fairness. The scheduling algorithm for weak fairness is a simple round robin scheduler that cycles through the list of actions that have associated fairness conditions. In particular, an action that is continuously enabled from some point on will eventually be chosen for execution, thus satisfying the weak fairness condition.

Formally, let $F \equiv \mathit{Init} \wedge \Box[\bigvee_{i=1}^{l} A_i]_x \wedge \bigwedge_{i=1}^{m} \mathrm{WF}_x(A_i)$. We adapt the definition of $\textsc{traces}(F)$ as follows:

Initialization If $\mathit{Init} \equiv \bigvee_{i=1}^{p} \bigwedge_{j=1}^{q} x_{ij} = c_{ij}$ then $\langle s_0\rangle \in \textsc{traces}(F)$ iff $s_0 \simeq \tau$ for some $\tau \in \bigcup_{i=1}^{p} \textsc{join}_{j=1}^{q}\{[x_{ij} \leftarrow c_{ij}]\}$.

Body Assume that $\langle s_0, \ldots, s_n\rangle \in \textsc{traces}(F)$.

1. If $s_n[\![\langle A_{(n \bmod l)}\rangle_x]\!] \neq \emptyset$ then $\langle s_0, \ldots, s_{n+1}\rangle \in \textsc{traces}(F)$ iff $s_{n+1} \simeq \tau$ for some $\tau \in s_n[\![\langle A_{(n \bmod l)}\rangle_x]\!]$.
2. If $s_n[\![\langle A_{(n \bmod l)}\rangle_x]\!] = \emptyset$ then $\langle s_0, \ldots, s_{n+1}\rangle \in \textsc{traces}(F)$ iff $s_{n+1} \simeq \tau$ for some $\tau \in s_n[\![(\bigvee_{i=1}^{l} A_i) \vee x' = x]\!]$.

Theorem 15 (Soundness). *Let $F \equiv \mathit{Init} \wedge \Box[\bigwedge_{i=1}^{l} A_i]_x \wedge \bigwedge_{i=1}^{m} \mathrm{WF}_x(A_i)$. For any behavior $\sigma \in St^{\infty}$, if $\sigma_{|n} \in \textsc{traces}(F)$ holds for all $n \in \mathbb{N}$ then $\sigma[\![F]\!]$ is true.*

Strong Fairness. The scheduling algorithm for strong fairness maintains a list of actions with associated fairness conditions and at every step tries to execute the first enabled action from that list, which is then moved to the end of the list. Intuitively, this corresponds to a priority scheduler where actions that cannot be executed move towards the beginning of the list. If the action is infinitely often enabled without being executed, it will eventually be the first enabled action in the list, and thus be executed by the scheduler.

Formally, let $F \equiv \mathit{Init} \wedge \Box[\bigwedge_{i=1}^{l} A_i]_x \wedge \bigwedge_{i=1}^{m} \mathrm{SF}_x(A_i)$. We adapt the definition of $\textsc{traces}(F)$ as follows, simultaneously defining a sequence $P_0, P_1, \ldots$ of lists of actions with fairness conditions (we use superscripts to refer to the element of a list at a given position):

Initialization

1. If $Init \equiv \bigvee_{i=1}^{p} \bigwedge_{j=1}^{q} x_{ij} = c_{ij}$ then $\langle s_0 \rangle \in \text{TRACES}(F)$ iff $s_0 \simeq \tau$ for some $\tau \in \bigcup_{i=1}^{p} \text{JOIN}_{j=1}^{q}\{[x_{ij} \leftarrow c_{ij}]\}$.
2. $P_0 = \langle 1, \ldots, m \rangle$

Body Assume that $\langle s_0, \ldots, s_n \rangle \in \text{TRACES}(F)$.

1. If $\text{SCHED}(P_n, s_n) = i \neq 0$ then $\langle s_0, \ldots, s_{n+1} \rangle \in \text{TRACES}(F)$ iff $s_{n+1} \simeq \tau$ for some $\tau \in s_n[\![\langle A_i \rangle_x]\!]$, and $P_{n+1} = \langle P_n^1, \ldots, P_n^{i-1}, P_n^{i+1}, \ldots, P_n^m, P_n^i \rangle$.
2. If $\text{SCHED}(P_n, s_n) = 0$ then $\langle s_0, \ldots, s_{n+1} \rangle \in \text{TRACES}(F)$ iff $s_{n+1} \simeq \tau$ for some $\tau \in s_n[\![(\bigvee_{i=1}^{l} A_i) \vee x' = x]\!]$ and $P_{n+1} = P_n$.

Here, $\text{SCHED}(P, s) = \begin{cases} 0 & \text{if } s[\![\langle A_{P^j} \rangle_x]\!] = \emptyset \text{ for all } j \\ \min\{j : s[\![\langle A_{P^j} \rangle_x]\!] \neq \emptyset\} & \text{otherwise} \end{cases}$

Theorem 16 (Soundness). *Let* $F \equiv Init \wedge \Box[\bigwedge_{i=1}^{l} A_i]_x \wedge \bigwedge_{i=1}^{m} \text{SF}_x(A_i)$. *For any behavior* $\sigma \in St^{\infty}$, *if* $\sigma_{|n} \in \text{TRACES}(F)$ *holds for all* $n \in \mathbb{N}$ *then* $\sigma[\![F]\!]$ *is true.*

5.2 Model checker

The animator is complemented by a model checker for finite-state TLA$^+$ specifications. The model checker is based on our subset of TLA and uses an explicit state enumeration algorithm to check the properties of the system and produce counterexamples to those properties that do not hold of the system. It uses an on-the-fly algorithm, which interleaves the generation of the state space and the search for errors, avoiding the construction of the complete state space. 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.

There are two kinds of properties that designers check with model checking tools: *safety properties* and *liveness properties*. Checking safety properties can be reduced to reachability analysis, whereas checking liveness properties amounts to searching for cycles in the state graph. More precisely, the procedure of verification is described in the following.

Given an executable TLA specification G and a property f expressed in propositional TLA, the verification procedure [12] is defined as follows:

1. Build a Büchi automaton $B_{\neg f}$ for the negation of the formula f.
2. Compute the product $B_G \otimes B_{\neg f}$ of the transition system that corresponds to the specification G and the automaton $B_{\neg f}$; the accepting runs of this automaton correspond to infinite computations of G accepted by $B_{\neg f}$
3. Check whether the language accepted by $B_G \otimes B_{\neg f}$ is empty or not.

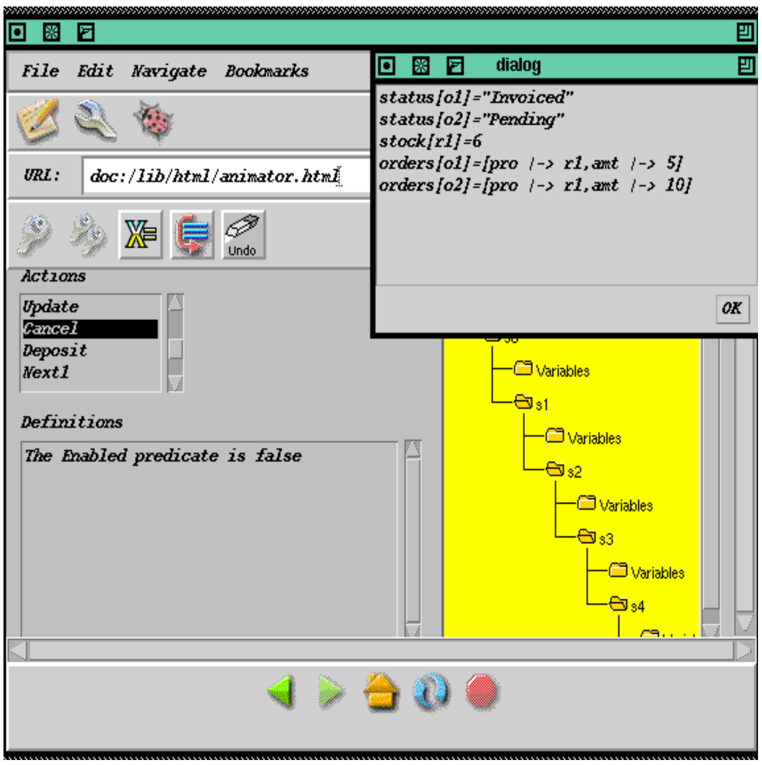
The first step is based on the relation between Büchi automata and LTL. Vardi and Wolper [29] showed that any LTL formula can be translated into a Büchi automaton which *accepts* precisely those (infinite) system executions that satisfy the LTL formula. The algorithm [12] is based on a tableau procedure and computes the states of the Büchi automaton by computing the set of subformulas that must hold in each reachable state and in each of its successor states. Liveness conditions give rise to the set of accepting states of $B_{\neg f}$.

The language of $B_G \otimes B_{\neg f}$ is nonempty iff the automaton contains an acceptance cycle. The search for acceptance cycles can be interleaved with the construction of the product automaton [13].

5.3 Implementation

We have implemented the tools outlined in our paper in Java. The figure 5 illustrates a screen-shot of our environment for TLA$^+$. The tools include the following components:

Source Editor Edit source files using a point-and-click editor. The Source Editor also serves integrated display mechanism for the other TLA$^+$ tools component.
Compiler Manager Build all TLA$^+$ modules.
Animator Animate a TLA$^+$ specification.
Model checker Model checker for TLA$^+$ specifications.



```
status[o1]="Invoiced"
status[o2]="Pending"
stock[r1]=6
orders[o1]=[pro |-> r1,amt |-> 5]
orders[o2]=[pro |-> r1,amt |-> 10]
```

```
The Enabled predicate is false
```

Fig. 5. An example of a scenario

Figure 5 illustrates the animation of the invoice system specification. It consists of a system which is composed of the following components: *set of orders*, *stock*, and *entry flows*. An order contains one reference to a product of a certain quantity. It may change its state from the state *pending* to *invoiced* if and only if the ordered quantity of the product is less than or equal to the quantity of this product in stock. The same reference of the product may occur in different orders. The entry flows allow us to change the state of the set of orders and the stock: (i) entries in the set of orders : new orders (**receive operation**) or cancelled orders (**cancel operation**). (ii) entry in stock : new entry of quantities of products in stock at warehouse (**deposit operation**).

Further, the figure shows an example of a user's interaction with the animator. We assume that the user initializes the system with two orders, namely o_1 and o_2 and one reference, namely r_1. Then the user chooses the scenario which consists in carrying out the sequence of actions: $\mathit{Receive}(o_1, r_1, 5)$,$\mathit{Receive}(o_2, r_1, 10)$,$\mathit{Deposit}(r_1, 11)$, $\mathit{Update}(o_1)$. Next, we reach a state depicted by the figure in which the first order o_1 is invoiced and the stock is updated. Finally, we can try to cancel the order o_1. The animator displays the following message "The Enabled predicate is false" which means that we cannot change the status of the order which is already invoiced. Thus, the execution of the action $\mathit{Cancel}(o_1)$ is not authorized. For futher explanations, the reader may refer to [33].

6 Conclusion

We have presented an executable subset of TLA and have established the soundness and completeness of our operational semantics with respect to the logical semantics. The interpreter algorithm needs to store only the current state in order to compute its successors; there is no need to refer to the entire history. Nevertheless, we store the choices present at each step so that the user can backtrack and explore different branches of the tree. Note in particular that because of the possibility of taking stuttering steps, and because all specifications in the subset defined in this paper are machine closed [1], a temporal formula in our subset is satisfiable if and only if the initial condition is. The user is alerted when only the stuttering action is enabled at some point; he can then decide whether the system state corresponds to a deadlock or to a quiescent state where all activity has terminated successfully.

We intend this work as a first step towards the development of an animator for TLA and TLA$^+$ [19]. We have developed a prototype in Java which faithfully incorporates the ideas outlined in this paper.

We are not alone in studying animation techniques for temporal logics. The classical approach for finite-state systems is based on the correspondence between propositional temporal logic (PTL) and finite-state automata on ω-words such as Büchi automata. Wolper [30] shows that for a given propositional temporal logic formula ϕ, one can construct an ω-automaton which accepts precisely those models that satisfy ϕ. This automaton can be taken as the basis of a PTL animator. While fully general for propositional logics, this approach is inherently restricted to finite-state systems. Moreover, the resulting automaton can be prohibitively large.

Other approaches are based on symbolic manipulation to ensure that each step in the execution obeys the specification. For example, METATEM [10] is based on full linear-time temporal logic, where formulas are restricted to the form

$$\text{past time antecedent} \Rightarrow \text{future time consequent}$$

which means "on the basis of the past do the future". Given a program consisting of a set of rules R_i of the above form, the interpreter attempts to construct a model of the formula $\Box \bigwedge_i R_i$. It proceeds informally in the following manner:

1. find those rules whose past-time antecedents evaluate to true in the current history;
2. "jointly execute" the consequents of applicable rules with any commitments carried forward. This will result in the current state being completed and the construction of a set of commitments to be carried out in the future;
3. repeat the execution process from the new commitments and the new history resulting from 2 above.

METATEM is more general in that it has an input language equivalent in expressiveness to full first-order temporal logic. The interpreter can therefore not ensure that finite behaviors constructed up to a certain point are in fact prefixes of models. Besides, it has to store the complete history of each run in order to evaluate past-time formulas in the antecedents of rules.

Finally, there have been extensions of PROLOG to incorporate temporal modalities [5,25]. However, these approaches are based on sets of rules that may involve temporal logic, and answer queries for such historical databases; they are not intended for the construction of behaviors of reactive systems.

References

1. M. Abadi and L. Lamport. The Existence of Refinement Mappings. Theoretical Computer Science 81(2):253–284, 1991.
2. M. Abadi and S. Merz. On TLA as as logic. in M. Broy (ed.): Deductive Program Design. Springer-Verlag, NATO ASI series F, 1996.
3. J.-R Abrial. The B-Book. Cambridge University Press, 1996.
4. B-core. B-Toolkit. User's Manual, Release 3.2. Technical Report, B-core, 1996
5. M. Baudinet. Temporal Logic Programming is Complete and Expressive. In Proc. ACM Conf. on Princinples of Programming Languages, pp. 267–280 (1989)
6. P. Breuer and J. Bowen. Towards Correct Executable Semantics for Z. In J. Bowen and J. Hall, editors, Proc. 8th Z Users Workshop (ZUM94), Workshops in Computing, pages 185-212. Cambridge, Springer-Verlag, Berlin, 1994.
7. http://www.cs.tut.fi/laitos/DisCo/DisCo-english.fm.html
8. E.A. Emerson and E.M. Clarke. Using branching-time temporal logic to synthesize synchronization skeletons. Science of Computer Programming **2** (1982), pages 241–266.
9. M. Fisher and R. Owens. An introduction to Executable Modal and Temporal Logics. In Michael Fisher and Richards Owens editors, Executable Modal and Temporal Logics, volume 897 of Lecture Notes in Computer Science , pages 1–20, Springer-Verlag, 1993.
10. M. Fisher. Concurrent METATEM—The Language and its Applications. In Michael Fisher and Richards Owens editors, Executable Modal and Temporal Logics, volume 897 of Lecture Notes in Computer Science , Springer-Verlag, 1993.

11. N. E. Fuchs. Specifications are (preferably) executable. Software Engineering Journal,7(5),pages 323-334,1992
12. R. Gerth, D. Peled, M. Vardi and P. Wolper Simple on-the-fly automatic verification of linear temporal logic PSTV, XIII, pages 3-18, 1995
13. P. Godefroid and G. J. Holzmann On the verification of temporal properties PSTV, XIII, pages 109-124, 1993
14. J. Goguen and G. Malcolm. Algebraic Semantics of Imperative Programs. MIT Press, 1997.
15. A. Gravell. Executing Formal Specifications Need Not Be Harmful. Available on the WWW at URL http://dsse.ecs.soton.ac.uk/ amg/papers.html.
16. C.A. Hoare. An Overview of Some Formal Methods for Program Design. IEEE Computer, pages 85-91, 1987.
17. I. J. Hayes and C. B. Jones Specifications are not (necessarily) executable. Software Engineering Journal,4(6),pages 320-338,1989
18. F. Kröger Temporal Logic of Programs. EATCS Monographs on Theoretical Computer Science 8. Berlin, Springer-Verlag, 1987
19. L. Lamport. The module structure of TLA$^+$ Research Report 1996–002, Digital Equipment Corporation, Systems Research Center.
20. L. Lamport. The operators of TLA$^+$ Research Report 1997–006a, Digital Equipment Corporation, Systems Research Center.
21. L. Lamport. The temporal logic of actions. ACM Transactions on Programming Languages and Systems, 16(3):872–923, May 1994.
22. A. C. Leisenring. Mathematical Logic and Hilbert's ε-Symbol. Gordon and Breach, New York, 1969.
23. Z. Manna and A. Pnueli The Temporal Logic of Reactive and Concurrent Systems - Specifications New-York etc.: Springer-Verlag, 1992
24. S. Merz. Efficiently Executable Temporal Logic Programs. In Michael Fisher and Richards Owens editors, Executable Modal and Temporal Logics, IJCAI'93 Workshop, volume 897 of Lecture Notes in Computer Science , pages 69–85, France, 1993, Springer.
25. M.A. Orgun and W.W. Wadge. Towards a unified theory of intensional logic programming. Journal of Logic Programming **13**(4), pp. 413ff (1992)
26. J. Rushby. Formal Methods and their Role in the Certification of Critical Systems. SRI Technical Report CSL-95-1, March 1995.
27. L. Sterling, P. Ciancarini and T. Turnidge. On the animation of "not executable" specifications by Prolog. International Journal of Software Engineering and Knowledge Engineering, 6(1):63-87.
28. M. Utting. Animating Z: Interactivity, transparency and equivalence. Technical Report 94-40. Software Verification Research Center.
29. M. Vardi and P. Wolper An automata-theoretic approach to automatic program verification Proceedings on the First Symposium on Logic in Computer Science, pages 322-331,1986
30. P. Wolper. Temporal Logic Can Be More Expressive. Information and Control 56. pages 71–99, 1983
31. P. Zave and R. T. Yeh. Executable requirement specification for embedded system. Proc. 5th Int. Conf. on Software Engineering, San Diego, California, pages 295-304, 1981.
32. P. Zave An operational approach to requirements specifications. IEEE Trans. SE-8(3), pages 250-269, 1982.
33. Y. Mokhtari The invoice system problem in TLA+ Proc. International Workshop on Comparing Systems Specification Techniques, University of Nantes, France March 1998.