Estratto da

Atti degli incontri di logica matematica Volume 6, Siena 28-31 maggio 1989.

Disponibile in rete su http://www.ailalogica.it

Temporal logic of programs:

Temporal semantics, verification and specification

Extended abstract

Fred Kröger
University of Munich

1. Temporal logic

Temporal logic (TL) is a branch of modal logic extending classical (propositional) logic according to the intuitive KRIPKE-like conception that propositions are interpreted over a "time-scale". TL provides logical operators for building new propositions relating the truth of propositions at certain time points.

Example: "A will hold at the next time point that B holds". This will be formalized by A **atnext** B.

The language $\mathcal{L}_{TA}$ of propositional temporal logic

The alphabet of $\mathcal{L}_{TA}$ consists of a denumerable set V of *atomic formulas* and the symbols $\neg,\rightarrow,\circ,\Box$,**atnext**,(,).

Formulas are inductively defined:

- Every $v\in V$ is a formula.
- If A,B are formulas, then $\neg A$, $(A\rightarrow B)$, $\circ A$, $\Box A$, $(A$ **atnext** $B)$ are formulas.

We read: - "nexttime A" for $\circ A$,
- "always A" for $\Box A$.

The following abbreviations can be introduced:

- $\wedge,\vee,\leftrightarrow$,*true*,*false* as usual,
- $\Diamond A$ for $\neg\Box\neg A$ ("sometime A").

In order to save parentheses when writing down formulas we fix the following ordering of the operators w.r.t. their binding:

$$\leftrightarrow \prec \rightarrow \prec \wedge,\vee \prec \textbf{atnext} \prec \neg,\circ,\Box,\Diamond$$

This means that $\leftrightarrow$ binds least, $\neg,\circ,\Box,\Diamond$ bind most. Outermost parentheses are always omitted.

Semantics of $\mathcal{L}_{TA}$

A *temporal (Kripke) structure* $\mathbf{K}$ for $\mathcal{L}_{TA}$ is a denumerable sequence $\{\eta_0,\eta_1,\eta_2,\ldots\}$ of mappings (*states*)

$$\eta_i: V\rightarrow\{\mathbf{f},\mathbf{t}\}$$

where $\mathbf{f}$ and $\mathbf{t}$ are *truth values*. For every formula F, temporal structure $\mathbf{K}$ and $i\in\mathbb{N}_0$, the *truth value of F in state η_i* - denoted by $\mathbf{K}_i(F)$ - is defined inductively:

$\mathbf{K}_i(v) = \eta_i(v)$ for $v \in V$,
$\mathbf{K}_i(\neg A) = \mathbf{t}$ iff $\mathbf{K}_i(A) = \mathbf{f}$,
$\mathbf{K}_i(A \to B) = \mathbf{t}$ iff $\mathbf{K}_i(A) = \mathbf{f}$ or $\mathbf{K}_i(B) = \mathbf{t}$,
$\mathbf{K}_i(\circ A) = \mathbf{t}$ iff $\mathbf{K}_{i+1}(A) = \mathbf{t}$,
$\mathbf{K}_i(\Box A) = \mathbf{t}$ iff $\mathbf{K}_j(A) = \mathbf{t}$ for every $j \geq i$,
$\mathbf{K}_i(A \,\mathbf{atnext}\, B) = \mathbf{t}$ iff $\mathbf{K}_j(B) = \mathbf{f}$ for every $j > i$ or
$\mathbf{K}_k(A) = \mathbf{t}$ for the smallest $k > i$ with $\mathbf{K}_k(B) = \mathbf{t}$.

This definition extends to the operators introduced as abbreviations, e.g.:

$\mathbf{K}_i(A \wedge B) = \mathbf{t}$ iff $\mathbf{K}_i(A) = \mathbf{t}$ and $\mathbf{K}_i(B) = \mathbf{t}$,
$\mathbf{K}_i(\Diamond A) = \mathbf{t}$ iff $\mathbf{K}_j(A) = \mathbf{t}$ for some $j \geq i$.

Remark: $\mathbf{K}_i(\circ A) = \mathbf{K}_i(A \,\mathbf{atnext}\, \mathit{true})$ holds for every $\mathbf{K}$ and i. So, the operator $\circ$ could also be introduced as an abbreviation. The same holds for $\Box$:

$\circ A$ abbreviates $A \,\mathbf{atnext}\, \mathit{true}$,
$\Box A$ abbreviates $A \wedge \mathit{false} \,\mathbf{atnext}\, \neg A$.

Definition: Let A be a formula, F a set of formulas, $\mathbf{K}$ a temporal structure.

$\Vdash_{\mathbf{K}} A$ (A valid in $\mathbf{K}$) iff $\mathbf{K}_i(A) = \mathbf{t}$ for every $i \in \mathbb{N}_0$,
$\Vdash A$ (A valid) iff $\Vdash_{\mathbf{K}} A$ for every $\mathbf{K}$,
$\mathsf{F} \Vdash A$ (A follows from F) iff $\Vdash_{\mathbf{K}} A$ for every $\mathbf{K}$ with: $\Vdash_{\mathbf{K}} B$ for every $B \in \mathsf{F}$.

A list of some valid formulas and rules

$\neg \circ A \leftrightarrow \circ \neg A$
$\Box A \to A$
$A \to \Diamond A$
$\Box A \to A \,\mathbf{atnext}\, B$
$\Diamond \Box A \to \Box \Diamond A$
$\Box \Box A \leftrightarrow \Box A$
$\Box \circ A \leftrightarrow \circ \Box A$
$\circ(A \to B) \leftrightarrow \circ A \to \circ B$
$\circ(A \,\mathbf{atnext}\, B) \leftrightarrow \circ A \,\mathbf{atnext}\, \circ B$
$\Box(A \wedge B) \leftrightarrow \Box A \wedge \Box B$
$\Box A \vee \Box B \to \Box(A \vee B)$
$\Box(A \to B) \to (\Box A \to \Box B)$
$\Box A \leftrightarrow A \wedge \circ \Box A$
$A \,\mathbf{atnext}\, B \leftrightarrow \circ(B \to A) \wedge \circ(\neg B \to A \,\mathbf{atnext}\, B)$
$A \to B \Vdash \circ A \to \circ B$
$A \to B \Vdash \Box A \to \Box B$
$A \Vdash \Box B \to \Box(A \wedge B)$
$\Box A \to B \Vdash \Box A \to \Box B$

Some further operators

Based on the temporal operators of $\mathcal{L}_{\mathrm{TA}}$, many other operators can be introduced, e.g.:

$A \,\mathbf{unless}\, B$ for $B \,\mathbf{atnext}\, (A \to B)$,
$A \,\mathbf{while}\, B$ for $\neg B \,\mathbf{atnext}\, (A \to \neg B)$,
$A \,\mathbf{before}\, B$ for $\neg B \,\mathbf{atnext}\, (A \vee B)$.

The informal meaning of these operators is:

$A \,\mathbf{unless}\, B$: "If there is a following state in which B holds then A holds up to that point or else A holds permanently",
$A \,\mathbf{while}\, B$: "A holds as long as B holds",
$A \,\mathbf{before}\, B$: "If B holds sometime in the future then B holds before that".

The formal system Σ_{TA}

The following axiomatization of TL can be given:

Axioms:
- all "classically valid" formulas (e.g. $\circ A \to \circ A$)
- $\neg \circ A \leftrightarrow \circ \neg A$,
- $\circ(A \to B) \to (\circ A \to \circ B)$
- $\Box A \to A \wedge \circ \Box A$
- $\circ \Box \neg B \to A \,\mathbf{atnext}\, B$
- $A \,\mathbf{atnext}\, B \leftrightarrow \circ(B \to A) \wedge \circ(\neg B \to A \,\mathbf{atnext}\, B)$

Rules:
(mp) $A, A \to B \vdash B$
(nex) $A \vdash \circ A$
(ind) $A \to B, A \to \circ A \vdash A \to \Box B$

The notion of *derivability* (w.r.t. Σ_{TA}) of a formula A from a set F of formulas - denoted by $\mathsf{F} \vdash A$ - is defined as usual.

Soundness theorem: *If* $\mathsf{F} \vdash A$ *then* $\mathsf{F} \Vdash A$.
Deduction theorem: *If* $\mathsf{F} \cup \{A\} \vdash B$ *then* $\mathsf{F} \vdash \Box A \to B$.

The proofs of these theorems are straightforward.

Completeness

The most general completeness assertion

If $\mathsf{F} \Vdash A$ *then* $\mathsf{F} \vdash A$

does not hold in TL. A simple counterexample is given by the infinite set

$\mathsf{F} = \{A \to B, A \to \circ B, A \to \circ\circ B, A \to \circ\circ\circ B, \ldots\}$.

$\mathsf{F} \Vdash A \to \Box B$ holds, but $A \to \Box B$ cannot be derived from F. (A formal derivation can use only finitely many premises from F.)

However, it is possible to prove

If $\mathsf{F} \Vdash A$ *then* $\mathsf{F} \vdash A$ *for finite sets* F *of formulas.*

For proving this it is sufficient to show:

Completeness theorem: *If* $\Vdash A$ *then* $\vdash A$.

We give a proof outline for this theorem. The basic idea follows the classical HENKIN-method which, however, has now to be modified carefully. (All sets of formulas which are constructed in this method have to be finite.) A finite set $\mathsf{F} = \{A_1, \ldots, A_n\}$, $n \geq 0$, of formulas is called

- *consistent* if it is empty, or $n > 0$ and $\neg(A_1 \wedge \ldots \wedge A_n)$ is not derivable in Σ_{TA},
- *satisfiable* if there is a temporal structure $\mathbf{K}$ such that $\mathbf{K}_0(A_j) = \mathbf{t}$ for all $j = 1, \ldots, n$.

To show the completeness theorem it is sufficient to show the following

Satisfiability theorem: *Every finite and consistent set of formulas is satisfiable.*

(From this the completeness theorem is deduced as in the classical case.) We illustrate the proof idea for this theorem by an example. Let A be the formula $(v_1 \to v_2) \to \Box v_3$, B the

formula $v_2 \to \circ(v_4 \to v_5)$ and $F = \{A,B\}$. ($v_1,\ldots,v_5 \in V$.) In order to "make A and B true" in η_0 we build a (finite, consistent) *completion* F^* of F. F^* consists of sub-formulas of F in such a way that if these formulas become true in η_0 then so will A and B. For technical reasons we include A and B themselves into F^* and obtain e.g.:

$$F^* = \{A, B, v_1 \to v_2, \neg v_1, v_2, v_3, \Box v_3, \circ(v_4 \to v_5)\}.$$

The formulas $\Box v_3$ and $\circ(v_4 \to v_5)$ - considered in η_0 - have some "influence" on the states $\eta_1, \eta_2, \ldots$. We apply a mapping σ on F^* which produces a set $\sigma(F)$ of formulas (which is again finite and consistent) such that if these formulas are true in η_1 then $\mathbf{K}_0(\Box v_3) = \mathbf{t}$ and $\mathbf{K}_0(\circ(v_4 \to v_5)) = \mathbf{t}$. We have

$$\sigma(F^*) = \{\Box v_3, v_4 \to v_5\}.$$

Continuing in the same way we obtain the following sequence $F_0, F_1, F_2, \ldots$ of formula sets:

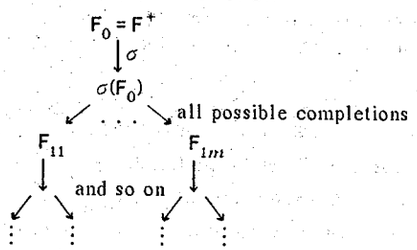
$F = \{A,B\}$
↓ completion
$F_0 = F^* = \{A, B, v_1 \to v_2, \neg v_1, v_2, v_3, \Box v_3, \circ(v_4 \to v_5)\}$
↓ σ
$\sigma(F_0) = \{\Box v_3, v_4 \to v_5\}$
↓ completion
$F_1 = \{\Box v_3, v_4 \to v_5, v_3, \neg v_4, v_5\}$
↓ σ
$\{\Box v_3\}$
↓ completion
$F_2 = \{\Box v_3, v_3\}$
↓ and so on

From this (over-simplified) construction we find $\mathbf{K} = \{\eta_0, \eta_1, \eta_2, \ldots\}$ by

$$\eta_i(v) = \mathbf{t} \text{ iff } v \in F_i. \quad \text{for every } v \in V,\ i \in \mathbb{N}_0.$$

It is clear that with this $\mathbf{K}$ we get $\mathbf{K}_0(A) = \mathbf{K}_0(B) = \mathbf{t}$.

There is, however, a problem in this construction. Suppose, some F_j contains the formulas $\neg\Box v$ and v ($v \in V$). Constructing F_{j+1}, we get $\neg\Box v \in F_{j+1}$ and it may also happen that we obtain again $v \in F_{j+1}$. The same may hold for all the next steps which would end up in $\eta_i(v) = \mathbf{t}$ for all $i \geq j$ and therefore $\mathbf{K}_j(\neg\Box v) = \mathbf{f}$. This, of course, should not be the case. The solution of this problem is found if we observe the fact that in every step, usually more than one completion is possible. The actual construction is to consider all these possibilities resulting not in a sequence F_0, F_1, F_2 of formula sets but in a tree of the form



It can be shown then that there is a path through this tree (yielding $\{\eta_0, \eta_1, \eta_2, \ldots\}$ as before) which avoids the above-mentioned problem. The proof runs by contradiction: Assume there is a node N_1 in the tree such that for some formula A, both $\neg\Box A$ and A are contained in N_1 and in all descendent nodes denoted by $N_2, N_3, \ldots$. If $N_i = \{B_1, \ldots, B_k\}$ let $\hat{N}_i$ denote the formula $B_1 \wedge \ldots \wedge B_k$. It is easy to see that there are only finitely many different N_i, say, $N_1, \ldots, N_n$. Let $C = \hat{N}_1 \vee \ldots \vee \hat{N}_n$. Then one can show:

(1) $\vdash \hat{N}_1 \to C$
(2) $\vdash C \to \circ C$
(3) $\vdash C \to A$

From (2) and (3), $\vdash C \to \Box A$ follows with rule (ind). Together with (1) we obtain $\vdash \hat{N}_1 \to \Box A$ which leads to a contradiction because of $\neg\Box A \in N_1$.

Induction principles

In applications, formulas of the kind

$A \to \Box B$,
$A \to B$ **atnext** C,
$A \to B$ **unless** C,
⋮
$A \to \Diamond B$

are of particular interest. We give some basic proof rules ("induction principles") for proving them:

For $A \to \Box B$, there is already a rule given by Σ_{TA}:

(ind) $A \to B,\ A \to \circ A \vdash A \to \Box B$.

Some variants of (ind) are:

(ind′) $A \to \circ A \vdash A \to \Box A$,
(ind″) $A \to B,\ B \to \circ B \vdash A \to \Box B$.

For $A \to B$ **atnext** C there is the rule

(indatnext) $A \to \circ(C \to B) \wedge \circ(\neg C \to A) \vdash A \to B$ **atnext** C.

For **unless**, **while**, **before** similar rules can be given. For $A \to \Diamond B$ no such principle (in propositional TL) exists.

First-order temporal logic (with equality)

$\mathcal{L}_{TA}$ can be extended to a first-order version $\mathcal{L}_{TP}$ by additional language features:

- variables (partitioned into *global* and *local* variables),
- function and predicate symbols, the equality symbol =,
- quantification $\forall$ (and $\exists$), allowed only over global variables.

The semantics of $\mathcal{L}_{TP}$ is given as in classical logic; in each state η_i, a value $\eta_i(x)$ is assigned to variables x. For global variables x we request:

$$\eta_i(x) = \eta_j(x) \quad \text{for all } i,j.$$

In the axiomatization we can add some further axioms and rules, e.g.:

- $\forall x A \to A_x(t)$ if $A_x(t)$ has no other occurrences of local variables in the scope of a temporal operator than A
- $\forall x \circ A \to \circ \forall x A$

- $A \to \circ A$ if A does not contain local variables
- $x = x$
- $x = y \to (A \to A_x(y))$ if A does not contain temporal operators
- $A \to B \vdash A \to \forall x B$ if there is no free occurrence of x in A

($A_x(t)$ denotes the effect of substituting t for x in A after appropriate renaming of bound variables of A which occur in t.) Soundness is again easily proved. The formal system is, however, not complete. In fact, the logic can not be axiomatized completely. The strongest incompleteness result (to our knowledge) is [3]:

- *If the first-order language contains at least two binary function symbols then the logic is incomplete.*

The idea to prove this is to interprete Peano arithmetic in the logic: 0 and the successor function are definable, the two function symbols can be axiomatized to be addition and multiplication.

The well-founded ordering principle

Let TL^{wf} be a first-order temporal logic containing

- variables ranging over a well-founded set Z,
- a binary predicate symbol $\leq$ interpreted by the well-founded ordering on Z.

In TL^{wf} we can formulate a basic proof rule for deriving formulas of the form $A \to \Diamond B$:

(wfo) $A(x) \to \Diamond(B \vee \exists x'(x' < x \wedge A(x'))) \vdash \exists x A(x) \to \Diamond B$ if B does not contain x.

Extensions

Temporal logic is investigated (and applied) in the literature in many other forms. We only give a few keywords:

- branching time logic: The underlying Kripke structure is not totally but only partially ordered;
- other modifications of the "time scale": not discrete, finite;
- more powerful temporal operators, in particular "past" operators;
- interval logic: Interpretation of formulas not over "time points" but over "time intervals";
- multi-dimensionality: Further dimensions besides "time";
- probabilistic additions: Truth values varying over the real interval [0,1] interpreted as probabilities.

2. Programs and their temporal semantics

Temporal logic is applicable to many kinds of (mainly imperative) programs, in particular parallel programs. Here we illustrate this applicability by a very simple class of parallel programs containing, e.g., the following (skeleton of a) program Π:


```
initial ex=true ∧ bf=0 ∧ be=n ∧ n>0;
cobegin loop α_00: ... ;
                ⋮
                α_0k0: ... ;
                α_1: await be>0 then be := be-1;
                α_2: await ex=true then ex := false;
                α_30: ... ;
                ⋮
                α_3k3: ... ;
                α_4: ex := true;
                α_5: bf := bf+1
         end,
         loop β_0: await bf>0 then bf := bf-1;
                β_1: await ex=true then ex := false;
                β_20: ... ;
                ⋮
                β_2l2: ... ;
                β_3: ex := true;
                β_4: bf := bf+1;
                β_50: ... ;
                ⋮
                β_5l5: ...
         end
coend
```

The first line defines (under the keyword **initial**) the *initial condition* of Π. The two *cyclic loops* between **cobegin** and **coend**, each of them enclosed by **loop** and **end**, are thought to be computed in parallel. The loops consist of sequences of labeled *atomic statements* (seperated by ";"), the α's and β's being unique *labels* of the statements. The statements (labeled by) $\alpha_4,\alpha_5,\beta_3,\beta_4$ are *assignments* like in usual imperative programming languages; the statements $\alpha_1,\alpha_2,\beta_0,\beta_1$ manage some *synchronization* of the two parallel components of Π.

The general form of a program (of our sample class)

The programs we consider are of the form

initial R;
cobegin $\Pi_1,\ldots,\Pi_p$ **coend** $(p \geq 1)$

where

- every Π_i is of the form

  loop $\alpha_0^{(i)}$: $\mathfrak{a}_0^{(i)}$; $\alpha_1^{(i)}$: $\mathfrak{a}_1^{(i)}$; ... ; $\alpha_{m_i}^{(i)}$: $\mathfrak{a}_{m_i}^{(i)}$ **end**

- every $\mathfrak{a}_j^{(i)}$ is of the form

  $a := t$ or
  await B **then** $a := t$

  (a is a *program variable*, t is a term, and B is a formula),

- all $\alpha_j^{(i)}$ in Π are pairwise different.

For further use we let $\mathfrak{M}_{\Pi_i} = \{\alpha_0^{(i)},\ldots,\alpha_{m_i}^{(i)}\}$ and $\mathfrak{M}_\Pi = \bigcup_{i=1}^{p} \mathfrak{M}_{\Pi_i}$.

Modelling the parallel computation of $\Pi_1,\ldots,\Pi_p$

The (*interleaving*) model of computation of a parallel program of the described form is informally given as follows:

- The computation is a sequence of interleaved executions of the atomic statements (beginning with some $\alpha_0^{(i)}$):
  $\ldots\ ;\ \alpha_{j_1}^{(i_1)};\ \alpha_{j_2}^{(i_2)};\ \alpha_{j_3}^{(i_3)};\ \ldots$
- The relative ordering of the $\alpha_j^{(i)}$ for every fixed i is maintained;
- Any **await** B **then** ... may occur as the next executed statement only if B holds.

Example: A possible execution of the program

initial $a=0 \wedge b=0$;
cobegin loop α_0: $a := a+1$; α_1: **await** $b\neq 0$ **then** $a := a\times b$; α_2: $a := a+3$ **end**,
loop β_0: $a := 2\times a$; β_1: **await** $a\neq 1$ **then** $b := 2\times b$; β_2: $b := b+1$ **end**
coend

is as follows:

action	values of a, b
	initially 0 0
α_0: $a := a+1$	1 0
β_0: $a := 2\times a$	2 0
β_1: **await** $a\neq 1$ **then** $b := 2\times b$	2 0
β_2: $b := b+1$	2 1
α_1: **await** $b\neq 0$ **then** $a := a\times b$	2 1
⋮	

Another possible beginning of an execution is:

action	values of a, b
	initially 0 0
β_0: $a := 2\times a$	0 0
α_0: $a := a+1$	1 0

This latter situation illustrates a typical problem of parallel programming: After the execution of β_0 and α_0 no one of the two parallel components can proceed (the program is in a *deadlock*) since the "**await**-conditions" of both α_1 and β_1 are false then.

We now define the formal notions covering these intuitive ideas.

Definition: A *program state* is a triple $\eta=(\mu,R,\lambda)$ with:

- μ assigns a value $\mu(a)$ to every program variable,
- $R=\{\alpha_{j_1}^{(1)},\ \ldots\ ,\alpha_{j_p}^{(p)}\}$ ($\alpha_{j_i}^{(i)}$ is the "next statement in Π_i"),
- $\lambda\in R\cup\{\mathrm{NIL}\}$ (next statement to be executed).

An *execution sequence* is an infinite sequence $\mathfrak{W}_\Pi=\{\eta_0,\eta_1,\eta_2,\ldots\}$ of program states with:

- R "is true in $\eta_0=(\mu_0,R_0,\lambda_0)$",
- $R_0=\{\alpha_0^{(1)},\ \ldots\ ,\alpha_0^{(p)}\}$,
- if $\eta_k=(\mu,R,\alpha_j^{(i)})$ then $\eta_{k+1}=(\mu',R',\alpha')$ with
  -- $R'=(R\setminus\{\alpha_j^{(i)}\})\cup\{\alpha_{j\oplus 1}^{(i)}\}$ (where $\oplus$ denotes addition modulo m_i+1),
  -- μ' = "μ changed according to $\alpha_j^{(i)}$",
- if $\eta_k=(\mu,R,\mathrm{NIL})$ then $\alpha_{j_i}^{(i)}$ is **await** B_i **then** ... for $i=1,\ldots,p$, B_i "is false in η_k", and $\eta_{k+1}=\eta_k$.

Example: In the above deadlock example we have e.g.:

	$\mu(a)$	$\mu(b)$	R	λ
η_0:	0	0	α_0,β_0	β_0
η_1:	0	0	α_0,β_1	α_0
η_2:	1	0	α_1,β_1	NIL
η_3:	1	0	α_1,β_1	NIL

and so on

Temporal semantics of a program Π

The basic idea of describing the "behaviour" (the *semantics*) of a program Π within temporal logic is to specialize the general temporal structures of TL to execution sequences of programs. This can be performed by further axioms (and one rule) which are called *program axioms* and build the *temporal semantics* of Π: They describe (axiomatically) all possible execution sequences of Π.

The language $\mathcal{L}_{T\Pi}$ for this purpose is a first-order temporal language $\mathcal{L}_{TP}$ with:

- program variables of Π as local variables,
- other variables as global variables,
- predicate and function symbols of Π
- (additional) atomic formulas:
  -- α for every $\alpha\in\mathfrak{M}_\Pi$ (α true in $\eta=(\mu,R,\lambda)$ iff $\alpha=\lambda$),
  -- $at\alpha$ for every $\alpha\in\mathfrak{M}_\Pi$ ($at\alpha$ true in $\eta=(\mu,R,\lambda)$ iff $\alpha\in R$),
  -- *nil* (*nil* true in $\eta=(\mu,R,\lambda)$ iff $\lambda=\mathrm{NIL}$),
- abbreviation:
  -- $start_\Pi$ for $at\alpha_0^{(1)}\wedge\ldots\wedge at\alpha_0^{(p)}\wedge R$

The program axioms for our kind of programs are:

(B1)	$start_\Pi \rightarrow \Box A \vdash A$	("$start_\Pi$ holds in the initial state")
(B2)	$nil\wedge A \rightarrow \bigcirc(nil\wedge A)$	("If *nil* then nothing is changed")
(Π1)	$\alpha\rightarrow\neg\alpha'$ for $\alpha\neq\alpha'$	("No two actions execute at the same time")
(Π2)	$\alpha\rightarrow at\alpha$	("An action may only execute if it is ready to")
(Π3)	$at\alpha_j^{(i)}\rightarrow\neg at\alpha_k^{(i)}$ for $j\neq k$, $j,k=0,\ldots,m_i$, $i=1,\ldots,p$	("In every Π_i, no two actions are ready to execute at the same time")
(Π4)	$at\alpha\rightarrow\neg nil$ if α labels a statement $a := t$ $at\alpha\wedge B\rightarrow\neg nil$ if α labels a statement **await** B **then** ...	("If some action is able to execute then some action has to execute")
(Π5)	$at\alpha\wedge\neg\alpha\rightarrow\bigcirc at\alpha$	("An action ready to execute but not executed remains ready")
(Π6)	$\alpha\wedge A_a(t)\rightarrow\bigcirc A$ if α labels a statement $a := t$ or **await** ... **then** $a := t$, A is a formula without temporal operators and without atomic formulas as defined above	(describes the effect on μ of executing α)
(Π7)	$\alpha_j^{(i)}\rightarrow\bigcirc at\alpha_{j\oplus 1}^{(i)}$ for $j=0,\ldots,m_i$, $i=1,\ldots,p$	(describes the effect on R of executing $\alpha_j^{(i)}$)

Additional to this rule and axioms there should also be some *data axioms* describing the involved data.

3. Verification of programs

A *program property* is a property valid for every execution of the program. The temporal semantics leads to the possibility of formal verification of program properties A of a program Π by:

- describing A in $\mathcal{L}_{T\Pi}$,
- deriving A in the formal system of first-order temporal logic augmented by rule (B1) and the other program axioms.

Examples of (typical) program properties

The intention of the sample program at the beginning of section 2 is that of a *producer/consumer* program: The first parallel component (the *producer*) cyclically "produces" in its statement list $\alpha_{00};\ldots;\alpha_{0k_0}$ some object and stores it (in $\alpha_{30};\ldots;\alpha_{3k_3}$) into a *buffer* shared with the other (*consumer*) component. The latter cyclically gets an object from the buffer (in $\beta_{20};\ldots;\beta_{2l_2}$) and "consumes" it in $\beta_{50};\ldots;\beta_{5l}$. The synchronization is to achieve that

- the producer can only store an object when the buffer is not full,
- the consumer can only get an object when the buffer is not empty,
- producer and consumer must not have access to the buffer at the same time (this could be required by implementation details).

For formally describing some typical program properties by means of this example we introduce the following abbreviations:

$in\text{PROD}$ for $at\alpha_{00} \vee \ldots \vee at\alpha_{0k_0}$,
$in\text{PUT}$ for $at\alpha_{30} \vee \ldots \vee at\alpha_{3k_3}$,
$in\text{CONS}$ for $at\beta_{50} \vee \ldots \vee at\beta_{5l_5}$,
$in\text{GET}$ for $at\beta_{20} \vee \ldots \vee at\beta_{2l_2}$.

E.g., inPROD informally means that the producer is "in its produce section".

Suppose now, *producedobj* and *consumedobj* are program variables containing as values the objects which are handled in the cycles of the producer and the consumer, respectively. Then the following program properties could be of interest:

$start_\Pi \to producedobj=s$ **before** $consumedobj=s$ (partial correctness)
$producedobj=s \to \Diamond consumedobj=s$ (no loss property)
$start_\Pi \to \Box\neg(in\text{PUT} \wedge in\text{GET})$ (mutual exclusion of buffer access)
$start_\Pi \to \Box(at\alpha_1 \wedge at\beta_0 \to be>0 \vee bf>0)$
$start_\Pi \to \Box(at\alpha_1 \wedge at\beta_1 \to be>0 \vee ex=true)$
$start_\Pi \to \Box(at\alpha_2 \wedge at\beta_0 \to ex=true \vee bf>0)$
$start_\Pi \to \Box(at\alpha_2 \wedge at\beta_1 \to ex=true)$ } (deadlock freedom)
$at\alpha_1 \to \Diamond at\alpha_{30}$
$at\beta_0 \to \Diamond at\beta_{20}$ } (starvation freedom)
$(producedobj=s \wedge \Diamond producedobj=s' \wedge s \neq s') \to$
$consumedobj=s$ **before** $consumedobj=s'$ (FIFO-behaviour of the buffer)

Verification rules

Program properties can be divided into three classes according to their syntactical form:

$A \to \Box B$: invariance properties
$A \to B$ **atnext** C:
$A \to B$ **before** C: } precedence properties } safety properties
$\vdots$
$A \to \Diamond B$: liveness properties

Basic proof principles for these kinds of properties can be derived from the corresponding purely logical proof rules together with some of the program axioms. In order to give an example, let us introduce the following abbreviations:

B **invof** α for $\alpha \wedge B \to \circ B$ ("B is an invariant of executing α"),
B **invof** $\mathfrak{M}$ for B **invof** $\alpha_1 \wedge \ldots \wedge B$ **invof** α_k where $\mathfrak{M}=\{\alpha_1,\ldots,\alpha_k\}$.

A basic proof rule for showing invariance properties is:

(inv) $A \to B,\ B$ **invof** $\mathfrak{M}_\Pi \vdash A \to \Box B$

It can be derived with the logical rule (ind″) as follows (suppose $\mathfrak{M}_\Pi=\{\alpha_1,\ldots,\alpha_k\}$):

(1) $A \to B$ assumption
(2) $\alpha_i \wedge B \to \circ B$ for $1 \le i \le k$ assumption
(3) $nil \wedge B \to \circ B$ program axiom (B2)
(4) $\alpha_1 \vee \ldots \vee \alpha_k \vee nil$ from definition of *nil*
(5) $B \to \circ B$ from (2), (3) and (4)
(6) $A \to \Box B$ from (1) and (5) by (ind″)

The only program axiom used in this justification of (inv) is (B2). Since (ind″) is a simple consequence of (ind), we may note this fact as:

(ind), (B2) ⇒ (inv).

In the same way we obtain rules (atnext), (before), ... for precedence properties:

(indatnext), (B2) ⇒ (atnext),
(indbefore), (B2) ⇒ (before),
$\vdots$

As an example, we note the rule

(before) $\alpha \wedge A \to \circ\neg C \wedge \circ(A \wedge B)$ for every $\alpha \in \mathfrak{M}_\Pi$, $nil \wedge A \to \neg C \ \vdash\ A \to B$ **before** C.

For liveness properties the situation is similar in the result (that one obtains a verification rule from the logical rule (wfo)) but must be handled with some more care.

An example of a verification

As an example, we indicate the formal verification of deadlock freedom of the producer/consumer program Π.

Proposition: (1) $start_\Pi \to \Box(at\alpha_1 \wedge at\beta_0 \to be>0 \vee bf>0)$
(2) $start_\Pi \to \Box(at\alpha_1 \wedge at\beta_1 \to be>0 \vee ex=true)$
(3) $start_\Pi \to \Box(at\alpha_2 \wedge at\beta_0 \to ex=true \vee bf>0)$
(4) $start_\Pi \to \Box(at\alpha_2 \wedge at\beta_1 \to ex=true)$

We show: (5) $start_\Pi \to \Box$**exor**$(in\text{PUT} \vee at\alpha_4, in\text{GET} \vee at\beta_3, ex=true)$

exor denotes the "exclusive or" of propositional logic, here applied to three formulas

P,G,$ex{=}true$ where

P is $in\mathrm{PUT} \vee at\alpha_4$,
G is $in\mathrm{GET} \vee at\beta_3$.

Denote **exor**$(P,G,ex{=}true)$ by B. (So (5) reads $start_\Pi \to \Box B$.) We have:

(a) $start_\Pi \to at\alpha_0 \wedge at\beta_0 \wedge ex{=}true \to B$,

(b) $\gamma \wedge B \to \circ B$ for every $\gamma \in \mathfrak{M}_\Pi$.

(a) is trivial; (b) has to be checked for all $\gamma \in \mathfrak{M}_\Pi$. For example, let $\gamma \in \{\alpha_{00},\ldots,\alpha_{0k_0},\alpha_1,\alpha_5\}$. Then $\gamma \wedge B$ implies $\neg P$ and also $\circ\neg P$. The validity of G and $ex{=}true$ remains unchanged under execution of γ, so we obtain $\gamma \wedge B \to \circ B$. The other cases run similarly.

From (a) and (b) we get (5) by (inv), and from (5) we also can derive very easily the propositions (2), (3), (4). In order to prove (1) it can be shown in just the same way:

(6) $start_\Pi \to \Box(bf \geq 0 \wedge be \geq 0)$

and furthermore (using (6)):

(7) $start_\Pi \to \Box((in\mathrm{PROD} \vee at\alpha_1) \wedge (in\mathrm{CONS} \vee at\beta_0) \to be > 0 \vee bf > 0)$.

(1) is then very easily derived from (7) by temporal logic means.

4. Axiomatic specification of programs

Program verification (with TL) means:

- Some program Π is given; the execution behaviour of Π is described in TL;
- program properties of Π are described in TL;
- the program properties are derived from the execution behaviour by the logical means of TL.

Another aspect is given by the keyword of (formal) program *specification*: This means (formally) describing the desired properties of a program which is to be constructed. We give a short introduction into a particular approach [2] to this field.

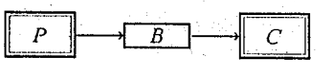
Abstract modules

Properties of parallel programs which are to be specified can be divided into two classes:

- Functionality properties (what is the program to do ?),
- synchronization properties (what must be observed w.r.t. parallel execution ?).

For illustration we consider the producer/consumer system of section 2. Before it was solved in the way given there, the problem could have been given, e.g., as follows.

- A parallel program is to be constructed containing two components P (producer) and C (consumer) using a shared buffer B:



- Functionality:
  P consists of cyclic loops producing objects and storing them into B;
  C consists of cyclic loops of getting objects from B and consuming them;
  B is a complex object offering two operations:

  -- put an object into B,
  -- get an object from B;
  The internal structure of B guarantees a ("queue"-like) *first-in-first-out* (FIFO) behaviour of putting and getting objects.
- Synchronization:
  The access to B is mutually exclusive;
  "put" is only possible if B is not full;
  "get" is only possible if B is not empty.

This also shows that the synchronization given in the final program is actually forced by synchronization properties of the buffer operations "put" and "get". A formal framework for specifying the buffer B and its desired functionality and synchronization properties could be as follows:

module BUFFER(element):put,get
  variable b:queue
  procedures put(**in**:element),
    get(**out**:element)
  functions empty: →queue,
    top: queue → element,
    ⋮
    isempty: queue → boolean,
    isfull: queue → boolean
    } auxiliary functions (see below)
  laws
    ⋮
    axioms in a formal language $\mathcal{L}$(BUFFER)
    describing functionality and synchronization properties of put and get
    ⋮
    } (see below)
endofmodule

We call such a notation an *abstract module*. This module has the name BUFFER and depends on a "sort" element denoting the range of produced objects. For putting and getting objects, two procedures put (with an "input"-parameter of sort element) and get (with an "output"-parameter of this sort) are provided which are to work on the buffer realized by the variable b. The sort of b is denoted by queue. For specifying put and get, some auxiliary functions will be usefull.

The computation model of such a module M is generally defined in the following way:

- At any time, procedures of M may be called from (parallel) program parts;
- a call of procedure f is "accepted" by M by creating a new instantiation of f;
- before executing its "body", f may be forced to wait (for synchronization);
- there is a particular last action of f called termination;
- from instantiation to termination, the effect of f is given by a sequence of atomic actions which may be interleaved with other procedure calls.

An outline of the language $\mathcal{L}$(M) for an abstract module M

$\mathcal{L}$(M) is the language in which the specification of M (under the keyword **laws**) is to be carried out. We define $\mathcal{L}$(M) to be a first-order temporal language with

- the functions of M which are not boolean-valued as function symbols,
- the boolean-valued functions of M as predicate symbols.

Furthermore, $\mathcal{L}(M)$ contains special atomic formulas (besides those of the form $p(t_1,...,t_n)$ with predicate symbol p and terms $t_1,...,t_n$). Let f be a module procedure of the general form

$$f(\textbf{in}:sort_1,...,sort_n,\textbf{out}:sort_{n+1},...,sort_m)$$

Let $INS(f)=\{f_0,f_1,...\}$ be a denumerable set of f-*instantiation-symbols* and $(t_1,...,t_m)$ be the *actual parameter list* of f_i, i.e. the "argument list" of the call of f denoted by f_i. For $f_i \in \bigcup_{f \in M} INS(f)$, the following formulas (given with their informal meaning) are atomic formulas of $\mathcal{L}(M)$:

init:	"M is in its initial state"
$inst f_i(t_1,...,t_m)$:	"The i-th call of the procedure f (i.e. f_i) with parameters $t_1,...,t_m$ is instantiated"
$wait f_i(t_1,...,t_m)$:	"f_i has been instantiated and waits for executing (its body)"
$start f_i(t_1,...,t_m)$:	"f_i starts to execute its body"
$exec f_i(t_1,...,t_m)$:	"f_i executes some action of its body"
$in f_i(t_1,...,t_m)$:	"f_i is in its body"
$term f_i(t_1,...,t_m)$:	"f_i terminates its execution"

Example: In the language $\mathcal{L}$(BUFFER) we could write the formula

$$start put_i(e) \textbf{ before } start put_j(e')$$

with the informal meaning: "The execution of the body of the instantiation $put_i(e)$ starts before the execution of the body of the instantiation $put_j(e')$".

We still introduce an abbreviation:

$$start f_i(t_1,...,t_m) \rightarrow \textbf{new } y=h(y)$$
$$\text{for} \quad start f_i(t_1,...,t_m) \wedge y=y_0 \rightarrow \Box(term f_i(t_1,...,t_m) \rightarrow \bigcirc(y=h(y_0)))$$

describing the effect of the call $f_i(t_1,...,t_m)$: "If $f_i(t_1,...,t_m)$ starts executing (its body) with y having the value y_0 then immediately after termination of this call, y will have the value $h(y_0)$."

Specification of a buffer

Finally we are able now to give the full specification of a buffer b as investigated above. The informal description is:

(a) put(e) stores e in b;
(b) get(r) fetches an element from b on its out-parameter r;
(c) put and get are to terminate in finite time;
(d) order preserving: Objects are fetched from b in the same order as they are stored;
(e) priorities: Different instantiations of put are to be executed in a FIFO-manner (and the same for get);
(f) put has the waiting condition that b is not full; get has the waiting condition that b is not empty;
(g) the bodies of different calls of put and/or get are to be mutually excluded;
(h) initially the buffer is empty.

The formal specification is given by the following abstract module:

module BUFFER(element):put,get
  variable b:queue
  procedures put(**in**:element),
    get(**out**:element)
  functions empty: →queue, (empty buffer)
    top: queue→element, (top element of the buffer)
    rest: queue→queue, (buffer without top element)
    append: queue×element→queue, (storing an element into the buffer)
    length: queue→nat, (number of elements in the buffer)
    cap: →nat, (capacity of the buffer)
    isempty: queue→boolean, (test of being empty)
    isfull: queue→boolean, (test of being full)
  laws top(append(empty,e)) = e,
    top(append(q,e)) = top(q) if q≠empty, } for realizing (d)
    rest(append(empty,e)) = empty,
    rest(append(q,e)) = append(rest(q),e) if q≠empty,
    length(empty) = 0,
    length(append(q,e)) = length(q)+1,
    cap ≠ 0,
    isempty(q) ↔ length(q) = 0,
    isfull(q) ↔ length(q) = cap,
    $start put_i(e)$ → **new** b = append(b,e), } (a),(b),(d)
    $start get_i(r)$ → **new** r = top(r), } (a),(b),(d)
    $start get_i(r)$ → **new** b = rest(b), } (a),(b),(d)
    $start put_i(e)$ → ◇$term put_i(e)$, } (c)
    $start get_i(r)$ → ◇$term get_i(r)$, } (c)
    $start put_i(e)$ → ¬isfull(b), } (f)
    $start get_i(r)$ → ¬isempty(b), } (f)
    $\neg(in put_i(e) \wedge in put_j(e'))$ for $i \neq j$ } (g)
    $\neg(in get_i(r) \wedge in get_j(r'))$ for $i \neq j$ } (g)
    $\neg(in put_i(e) \wedge in get_j(r))$ } (g)
    $inst put_i(e)$ **before** $inst put_j(e')$ →
      $start put_i(e)$ **before** $start put_j(e')$ for $i \neq j$ } (e)
    $inst get_i(r)$ **before** $inst get_j(r')$ →
      $start get_i(r)$ **before** $start get_j(r')$ for $i \neq j$ } (e)
    init → b=*empty* } (h)
endofmodule

References

The material of sections 1-3 is taken from [1]. In that textbook there is also contained a detailed list of relevant literature.

[1] F. Kröger: *Temporal logic of programs*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1987

[2] F. Kröger: *Abstract modules: Combining algebraic and temporal logic specification means*. Techn. Sci. Inform. **6**, 559-572 (1987)

[3] F. Kröger: *On the interpretation of arithmetic in temporal logic*. To appear in: Theor. Comp. Sci.