Appl. Math. and Comp. Sci., 1992, vol.2, No.2, 217–235

TEMPORAL LOGIC AS A TOOL FOR PROGRAM CORRECTNESS ANALYSIS

RADOSŁAW KLIMEK*

The paper deals with temporal logic and analysis of correctness properties of programs. A classification of program properties is presented. Verification by temporal logic is shown on the examples of synchronization in the producer–consumer problem and a sample parallel system of robots. The analysis of the system concerns three types of safety property.

1. Introduction

Formal analysis of software and proof of its correctness is a very important issue in software engineering. This is a difficult problem and it requires a very precise approach (Dijkstra, 1981). There are many facts which result from each program instruction. Each instruction can influence many other instructions. If program testing can yield some results in the case of sequential program, it fails for a concurrent program which results from unforeseeable time run. The program behavior is not only the input/output relation but also the whole execution sequence, i.e. a program state does not depend only on a program itself but also on other cooperating and competing programs which run in the same time. These time dependencies are very difficult to account for in the accepted model of the program execution.

In ordinary logic valuation of proposition does not depend on time. We can even mention the first–order approach to time (Galton, 1987). A sequence of states connected with a passage of time is considered in a typical model of program execution. In succeeding states variables can have different values. Therefore, formalism used for program analysis should lead to show the dynamic character of a program. Temporal logic, which is classified among model logicals, expands its time domain. The formulae may take on different logic values at connective time points.

The aim of this paper is to present and show temporal logic as a tool for description and verification of software, in particular concurrent software. The following sections of this paper deal with temporal logic and the division of program correctness properties. Temporal logic is discussed in the way it was made in (Galton, 1987). Safety property are described more carefully since it is often taken into account when designing a program. Two examples are presented in order to

* Academy of Mining and Metallurgy, Institut of Automatic Control, 30–059 Kraków, Al. Mickiewicza 30, Poland

show potentialities of temporal logic. The first one is a traditional solution to the question of synchronizing the procedur–consumer problem. It may constitute a certain comparison of temporal logic potentialities with e.g. Habermann's axiomatic theory that can also be used for proving the correctness of concurrent algorithms (Habermann, 1972; Iszkowski and Maniecki, 1982). The second example is a parallel real–time system of robots. First, a primary control program of robots is put forward and then formal verification is made. Verification concerns the three examples of the safety property which are described earlier. In the last section conclusions have been put forward. In appendix, proofs of some temporal logic laws are presented.

2. Temporal Logic

In ordinary logic valuation of a proposition does not depend on time (the passage of time) at all. We always try to reason absolute truth. In this sense we can say that propositions have static character. However, in temporal logic the truth and falsity may depend on a point of time in which a proposition is considered. This is why a proposition has to be considered in a context of other time points (states), i.e. in the whole sequence of them. We assume that set of time points is infinite, discrete and linearly ordered with a smallest element.

Temporal logic introduces new operators in comparison with ordinary logic. There are operators well known in model logic, i.e. $\Box$ *necessity operator* and as well as $\Diamond$ *possibility operator*, so new ones. The list of these operators with a sample use and informal definition is presented below. All definitions deal with a reference point which can be:

$\Box A$ – A holds at all time points *(always)*,

$\Diamond A$ – A holds at least at one time point *(sometime)*,

$\bigcirc A$ – A holds at the time point immediately after the reference point *(next)*,

$A \cup B$ – A holds at all following time points up to a time point at which B holds *(until)*.

As shown above the first three operators are unary and the last one is binary.

A *language* $\mathcal{L}_T$ *of propositional temporal logic* is given by an alphabet of symbols and the definition of a set of strings over this alphabet, called *formulae*. The definition mentioned below is an extension of a similar one for a language of ordinary logic which could also be shown, i.e. language of the temporal logic includes the language of ordinary logic.

Alphabet

- a denumerable set $\mathcal{V}$ of *atomic formulae*,
- the symbols: **true**, **false**, $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, $\Box$, $\Diamond$, $\bigcirc$, $\cup$, (,).

Inductive definitions of formulae:

i) Every atomic formula is a formula.
ii) If A and B are formulae, then $\neg A$, $(A \wedge B)$, $(A \vee B)$, $(A \Rightarrow B)$, $(A \Longleftrightarrow B)$, $\Box A$, $\Diamond A$, $\bigcirc A$ and $A \cup B$ are formulae.

The given set of symbols is greater than necessary. The set: $\neg$, $\Rightarrow$, $\bigcirc$, $\cup$, (,) is sufficient since the remaining operators and constants can be introduced as abbreviations of them.

The priority order of the operators is as follows:

$\neg$, $\Box$, $\Diamond$, $\bigcirc$	have higher priority than all binary operators;
$\cup$	has higher priority than $\wedge$, $\vee$, $\Rightarrow$, $\Longleftrightarrow$;
$\wedge$, $\vee$	have higher priority than $\Rightarrow$, $\Longleftrightarrow$;
$\Rightarrow$	has higher priority than $\Longleftrightarrow$.

The *semantics* of language $\mathcal{L}_T$ is based on the idea of *temporal structure* K which consists of an infinite sequence $\{b_0, b_1, b_2, \ldots\}$ of mappings (*Boolean valuations*):

$$b_i : \mathcal{V} \rightarrow \{\mathbf{f}, \mathbf{t}\}$$

where $\mathbf{f}$ and $\mathbf{t}$ represent, falsity and truth, respectively, b_i is called *state* and b_0 is the *initial state*. It means that a state is a time point and every state is a valuation in the ordinary sense.

Now, we can define temporal logic operators strictly using temporal structure (a subscript for K means that b_i is a reference state):

$$\begin{aligned}
K_i(\Box A) = \mathbf{t} \quad &\text{iff} \quad K_j(A) = \mathbf{t} \quad \text{for every } j \geq i;\\
K_i(\Diamond A) = \mathbf{t} \quad &\text{iff} \quad K_j(A) = \mathbf{t} \quad \text{for some } j \geq i;\\
K_i(\bigcirc A) = \mathbf{t} \quad &\text{iff} \quad K_{i+1}(A) = \mathbf{t};\\
K_i(A \cup B) = \mathbf{t} \quad &\text{iff} \quad K_j(B) = \mathbf{t} \quad \text{for some } j > i \text{ and}\\
&\qquad\quad K_k(A) = \mathbf{t} \quad \text{for every } k,\ i < k < j.
\end{aligned}$$

Before we present a list of some basic axioms and laws of temporal logic, let us consider the combinations of two model operators. The formula $\Box\Diamond A$ (*infinitely often* A) can be interpreted as there is an infinite number of future time points at which A holds, or otherwise, if A ever becomes false, it is guaranteed to become true again at some later time point. The formula $\Diamond\Box A$ (*eventually henceforth* A) can be interpreted as follows: there is some time point after which A remains true forever.

Duality laws

(T1) $\Box\neg A \Longleftrightarrow \neg\Diamond A$

(T2) $\Diamond\neg A \Longleftrightarrow \neg\Box A$

(T3) $\bigcirc\neg A \Longleftrightarrow \neg\bigcirc A$

T1 and T2 say that operators $\Box$ and $\Diamond$ are *dual* and lead to express the relation between the necessity and the possibility. T3 says that operator $\bigcirc$ is *self-dual.*

Reflexity laws

(T4) $\Box A \Rightarrow A$

(T5) $A \Rightarrow \Diamond A$

These formuae indicate relations between the future and the present.

"Strength" of the operators

(T6) $\Box A \Rightarrow \bigcirc A$

(T6′) $\Box A \Rightarrow \Diamond A$

(T7) $\bigcirc A \Rightarrow \Diamond A$

(T8) $A \cup B \Rightarrow \Diamond A$

(T9) $\Diamond\Box A \Rightarrow \Box\Diamond A$

These formulae indicate implication relations between the operators.

Idempotency laws

(T10) $\Box\Box A \Longleftrightarrow \Box A$

(T11) $\Diamond\Diamond A \Longleftrightarrow \Diamond A$

T10 and T11 mean that, intuitively speaking, the future of the future is equivalent to the future.

Distributivity laws

(T12) $\Box(A \wedge B) \Longleftrightarrow \Box A \wedge \Box B$

(T13) $\Diamond(A \vee B) \Longleftrightarrow \Diamond A \vee \Diamond B$

(T14) $\bigcirc(A \wedge B) \Longleftrightarrow \bigcirc A \wedge \bigcirc B$

(T15) $\bigcirc(A \vee B) \Longleftrightarrow \bigcirc A \vee \bigcirc B$

(T16) $\bigcirc(A \Rightarrow B) \Longleftrightarrow \bigcirc A \Rightarrow \bigcirc B$

These formulae indicate *distributivity* relations for temporal logic operators.

Weak distributivity laws

(T17) $\Box A \vee \Box B \Rightarrow \Box(A \vee B)$

(T18) $\Box(A \Rightarrow B) \Rightarrow (\Box A \Rightarrow \Box B)$

(T19) $\Diamond(A \wedge B) \Rightarrow \Diamond A \wedge \Diamond B$

(T20) $(\Diamond A \Rightarrow \Diamond B) \Rightarrow \Diamond(A \Rightarrow B)$

(T21) $((A \cup C) \vee (B \cup C) \Rightarrow (A \vee B) \cup C$

(T22) $(A \cup (B \wedge C)) \Rightarrow ((A \cup B) \wedge (A \cup C))$

These formulae indicate *one direction* distributivity relations.

Recursion equivalences

(T23) $\Box A \Longleftrightarrow A \wedge \bigcirc \Box A$

(T24) $\Diamond A \Longleftrightarrow A \vee \bigcirc \Diamond A$

(T25) $A \cup B \Longleftrightarrow B \vee (A \wedge \bigcirc(A \cup B))$

These formulae describe the recursion representation for temporal logic operators.

The complete list of axioms and laws of temporal logic can be found in many works (Hailpern, 1982; Kröger, 1987; Manna and Pnueli, 1981).

Variability of the temporal logic propositions on the time axis is not the only admissible interpretation, though this is probably the most natural one. Another interesting and more general interpretation (Manna and Pnueli, 1981) is possible when we consider the variability of some disjoint models. We may also use the comparison that the models are different *worlds* of similar structure but different contents. However, all these *worlds* compose the *universe*.

Many researches on temporal logic and its usage have been conducted in a great number of research institutes. For instance, suggestions have been made to modify the operators so that the time interval of formulae fulfillment may be taken into account (Hailpern, 1982). Temporal logic is used for the theory of fixed points equations (Clarke *et. al.*, 1985; Fariñas–del–Cerro, 1985), it also serves to create systems of automatic reasoning (Dijkstra, 1981; Fariñas–del–Cerro, 1985), or it may be used for logic programming (Fariñas–del–Cerro, 1985).

3. Correctness Properties of Program

Let us consider a typical concurrent program

$$P = P_1 || ... || P_m$$

with input variables $\overline{x} = (x_1, ..., x_k)$ and local (shared) variables $\overline{y} = (y_1, ..., y_n)$. Let $b(\overline{x})$ be the precondition of the program that restricts the set of the input

data. The formula at l enables us to define the label of the actually executed instruction. The formula holds when any process executes instruction of label l. Finally, the formula $\models W$ means that W holds for every suffix of states of any computing (a sequence of states from the initial to the terminal one).

Correctness properties of a program are traditionally divided into safety, liveness and precedence properties (Iszkowski and Maniecki, 1982; Lamport, 1977; Owicki and Lamport, 1982). This classification is connected with some temporal logic operators, correspondingly $\Box$, $\Diamond$, $\cup$.

Safety property can be informally defined as *nothing bad will ever happen* (Kröger, 1987). The bad event could be the buffer overflow, deadlock, etc. The property is expressible by the formula:

$$\models \Box\neg W$$

where W states an undesirable property of a program. The formula holds continuously through the execution of the program and this is why it is sometimes called invariance property. Using the proper rule of temporal logic the property is also expressible by the equivalent formula:

$$\models \neg\Diamond W$$

The program precondition can be included in the following way:

$$\models b(\bar{x}) \Rightarrow \Box\neg W$$

Liveness property can be informally said that when some condition holds, something good happens as a result (Manna and Pnueli, 1981). The good event could be the access to a critical section, responsiveness (in a real–time system), etc. The property is expressible by the formula:

$$\models W_1 \Rightarrow \Diamond W_2$$

The formula states that when W_1 holds then W_2 will have to hold in one of the following states. The fact that the formula holds for all computations can be written down:

$$\models \Box(W_1 \Rightarrow \Diamond W_2)$$

Precedence property links together succeeding satisfaction of two conditions, for example the absence of unsolicited response. The property is expressed by the formula:

$$\models W_1 \cup W_2$$

The formula means that in the future W_2 will hold and until this time W_1 holds in all states.

Let us return to the safety property to give some examples of it.

i) *Partial correctness*
Partial correctness of a program means that if the first condition holds and the program terminates, then the second condition will hold too. The first condition is called precondition of the program and restricts the input data for which the program is supposed to be *correct*. The second condition is called postcondition of the program and is connected with the *correctness* of the output data. Formally we can write:

$$\models b(\overline{x}) \Rightarrow \Box(\text{at } \overline{l}_e \Rightarrow R(\overline{x}, \overline{y}))$$

where $\overline{l}_e$ is the set of terminal labels of the program and $R(\overline{x}, \overline{y})$ is an output relation connected with postcondition and holds between the input and output data.

ii) *Clean behavior*
Clean behavior is such a requirement that there cannot be any error such as, for example, an attempt to put data into the full buffer. We can write:

$$\models b(\overline{x}) \Rightarrow \Box \bigwedge_{l} (\text{at } l \Rightarrow W_l)$$

where l is a label for which W_1 states the condition of the clean behavior. The right side of the implication is the conjunction taken over *potentially dangerous* labels in the program.

iii) *Mutual exclusion*
Mutual exclusion is such a requirement that it is never the case that more than one process can execute a critical section. The critical section is treated as a kind of indivisible macro–operation. We can write:

$$\models b(\overline{x}) \Rightarrow \Box\neg(\text{at } C_1 \wedge \text{at } C_2)$$

where $C_i \subseteq L_i$ for $i = 1, 2$ states a critical section in component processes (L_i is a set of program labels).

iv) *Deadlock freedom*
Deadlock freedom is such a requirement that it is never the case that any process waits for an event that will never happen. We can write:

$$\models b(\overline{x}) \Rightarrow \Box(\bigwedge_{i=1}^{m} \text{at } l^i \Rightarrow \bigvee_{i=1}^{m} E_i)$$

where E_i is a formula that states possibility of the skip to the next state for the label l^i different from the terminal label and called a waiting label to note the state in which holds:

$$\text{at } l^i \wedge \neg E_i$$

4. Deadlock in Producer–Consumer Problem

To begin with, let us consider the well known producer–consumer problem version with a limited buffer. We want to show that the deadlock cannot appear in the standard solution of synchronization question of this problem. Let us present the matter precisely.

Synchronization of the problem has to include mutual exclusion of producer processes and consumer processes which operate in the critical section (buffer of communicates). Next, it is necessary to guarantee that the processes cannot put elements into the full buffer and take elements from the empty buffer. All these postulates can be realized in the way shown in Figure 1 by two semaphore operations P and V and by a proper structural instruction (Iszkowski and Maniecki, 1982).

```
var
   Buffer: shared array [1 .. max] of ElemBuf;
   empty, full: semaphore: = max, 0;

procedure Producer;
begin
   repeat
      P(empty);
      region Buffer do
      begin
         .
         .
         .
      end;
      V(full)
   until false;
end {Producer};

procedure Consumer;
begin
   repeat
      P(full);
      region Buffer do
      begin
         .
         .
         .
      end;
      V(empty)
   until false;
end {Consumer};

Process (1..M) Producer;
Process (1..N) Consumer;
```

Fig. 1. Typical solution of the synchronization question in the producer–consumer problem.

Theorem. *Synchronization of the producer–consumer problem as shown in Figure 1 is free of a deadlock.*

Proof. Suppose that the deadlock is possible. It would appear if producer processes waited for semaphore signal from consumers and consumer processes waited for semaphore signals from producers but in both these cases signals would never come. Let us consider the situation when the producers wait for signals from the consumers. The proof for the reverse situation is analogical (symmetrical). First, let us note that if the critical section is not occupied, then we can write:

$$\max = \text{empty} + \text{full} \tag{1}$$

Let us express the fact that producers are suspended and they will remain suspended:

$$\begin{aligned} &K_i(\neg\Diamond \text{empty} > 0) = \mathbf{t} \iff \\ &K_i(\Box\neg \text{empty} > 0) = \mathbf{t} \iff \\ &K_i(\Box \text{empty} = 0) = \mathbf{t} \end{aligned} \tag{2}$$

If the semaphore signals from consumers are not coming, it means that it is satisfied:

$$\begin{aligned} &K_i(\Box \text{full} = 0) = \mathbf{t} \iff \\ &K_i(\Box \max - \text{empty} = 0) = \mathbf{t} \iff \\ &K_i(\Box \text{empty} = \max) = \mathbf{t}. \end{aligned} \tag{3}$$

Expressions (2) and (3) cannot be satisfied simultaneously thus the processes of producers and processes of consumers cannot be suspended simultaneously, either. It means that the deadlock is not possible. ■

5. Sample of Robots System

Now, let us consider a sample of a real–time system presented in Figure 2. This is a system of cooperating robots and assembly–transport belts. The connections among all the elements are also shown in Figure 2. This sample is an extension of a similar one from (Szmuc, 1989).

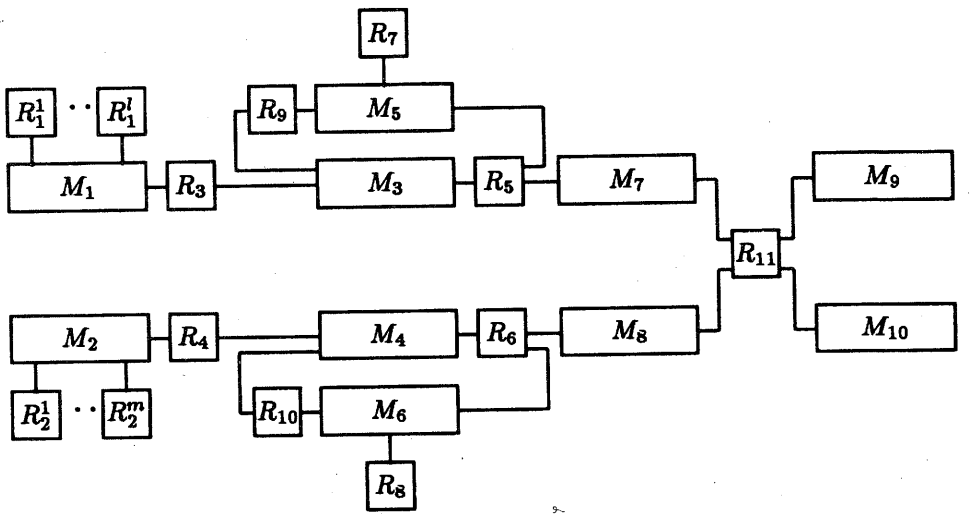

Fig. 2. The system of cooperating robots.

Robots $R_1^1, ..., R_1^l$ $(R_2^1, ..., R_2^m)$ realize assembly operation of articles which are in containers of belt M_1 (M_2). Robot R_3 (R_4) carries an element over belt M_3 (M_4). Robot R_5 (R_6) takes one element from belt M_3 (M_4) and checks its quality. When it is up to a standard it is carried over belt M_7 (M_8), otherwise it is carried over belt M_5 (M_6) to improve its quality. The improvement is realized by robot R_7 (R_8). When the operation is terminated, the element is placed on belt M_3 (M_4) once again but now it is done by robot R_9 (R_{10}). When the element from belt M_3 (M_4) is not up to a standard for the second time, it is carried over belt M_7 (M_8). Afterwards, robot R_{11} takes elements from belts M_7 and M_8 one by one and checks their quality. If any element is not up to a standard, then it is carried over belt M_{10} and the operation is repeated. If elements taken are good quality then the robot assembles them and places them on belt M_9. Belt M_1 (M_2) moves one step when one cycle of robots $R_1^1, ..., R_1^l$ $(R_2^1, ..., R_2^m)$ is completed. Belts M_3, M_5, M_7 (M_4, M_6, M_8) and M_9, M_{10} move one step when one element is placed on them.

5.1. Primary Control Program of Robots

When designing control procedures of robots we use the set of some fixed and elementary operations. Operations from outside this set are not important for the correctness of the program since they deal with nothing but technology. This is the reason why the technology operations can be omitted. The set of elementary operations is:

init (r)	–	initiate the work of robot r;
finish (r)	–	finish the work of robot r;
move (m)	–	move belt m;
get (m)	–	get an element from belt m;
put (m)	–	put an element on belt m;
improve (m)	–	improve quality of an element on belt m;
attempt (m)	=	true if an element from belt m was attempted to improve its quality, otherwise attempt(m) = false;
correct (m)	=	true if an element from belt m is up to a standard, otherwise correct(m) = false;
assemble (m_1, m_2)	–	assemble elements from belts m_1 and m_2.

There are control procedures of robots in Figure 3. In the case of robots $R_1^1, ..., R_1^l$ and R_3 there is only one procedure as it is assumed that there is a controller of these robots. The procedures for robots $R_2^1, ..., R_2^m$ with R_4 and R_6, R_8, R_{10} are not shown since they are analogous to those listed. The only difference for them are individual indexes.

Control procedures from Figure 3 compose the concurrent program, which we can be writen:

Program = Robots1–3 || Robot5 || Robot7 || Robot9 || Robot11

```
        procedure Robots1–3
        begin
α0 :      move (M1);
α1 :      init (R1^1);
α2 :      finish (R1^1);
            { ..... }
α3 :      init (R1^l);
α4 :      finish (R1^l);
α5 :      init (R3);
α6 :      put (M3);
αe :      finish (R3);
        end {Robots1–3};

        procedure Robot7
        begin
δ0 :      init (R7);
δ1 :      improve (M5);
δe :      finish (R7);
        end {Robot7};
```

```
        procedure Robot5
        begin
β0 :      init (R5);
β1 :      get (M3);
β2 :      if correct (M3)
β3 :         then put (M7)
β4 :         else if attempt (M3)
β5 :            then put (M7)
β6 :            else put (M5);
βe :      finish (R5);
        end {Robot5};

        procedure Robot9
        begin
η0 :      init (R9);
η1 :      get (M5);
η2 :      {any instruction};
η3 :      put (M3)
ηe :      finish (R9);
        end {Robot9};
```

```
        procedure Robot11
        begin
μ0 :      init (R11);
μ1 :      get (M7);
          while not correct (M7) do
          begin
μ2 :         put (M10);
μ3 :         get (M7)
          end;
μ4 :      get (M8);
          while not correct (M8) do
          begin
μ5 :         put (M10);
μ6 :      get (M8)
          end;
μ7 :      assemble (M7, M8);
μ8 :      put (M9);
μe :      finish (R11);
        end {Robot11};
```

Fig. 3. Control procedures of robots.

5.2. Verification of Correctness – Safety Property

When verifying the sample system we concentrate on the safety property and examine its three types. First, let us define the following functions:

len (m) – current number of elements on belt m;

max (m) – maximum admissible number of elements on belt m;

Mutual exclusion. Mutual exclusion is necessary in the case of all these resources (belts) to which more than one process try to get the access (operations: get, put, improve). For these cases (critical sections) we can write:

I. M_3 and Robots1–3, Robot5, Robot9

$$\Box\neg(\text{at } \alpha_3 \wedge \text{at } \beta_1)$$

$$\Box\neg(\text{at } \alpha_3 \wedge \text{at } \eta_3)$$

$$\Box\neg(\text{at } \beta_1 \wedge \text{at } \eta_3)$$

all these expressions have to be satisfied together and thus conjunction can be taken over them:

$$\begin{aligned} \models \quad & \Box\neg(\text{at } \alpha_3 \wedge \text{at } \beta_1) \wedge \Box\neg(\text{at } \alpha_3 \wedge \text{at } \eta_3) \wedge \Box\neg(\text{at } \beta_1 \wedge \text{at } \eta_3) \iff \\ \iff & \Box(\neg(\text{at } \alpha_3 \wedge \text{at } \beta_1) \wedge \neg(\text{at } \alpha_3 \wedge \text{at } \eta_3) \wedge \neg(\text{at } \beta_1 \wedge \text{at } \eta_3)) \iff \\ \iff & \Box\neg((\text{at } \alpha_3 \wedge \text{at } \beta_1) \vee (\text{at } \alpha_3 \wedge \text{at } \eta_3) \vee (\text{at } \beta_1 \wedge \text{at } \eta_3)) \end{aligned} \tag{4}$$

II. M_5 and Robot5, Robot7, Robot9

$$\Box\neg(\text{at } \beta_6 \wedge \text{at } \delta_1)$$

$$\Box\neg(\text{at } \beta_6 \wedge \text{at } \eta_1)$$

$$\Box\neg(\text{at } \delta_1 \wedge \text{at } \eta_1)$$

and together:

$$\models \Box\neg((\text{at } \beta_6 \wedge \text{at } \delta_1) \vee (\text{at } \beta_6 \wedge \text{at } \eta_1) \vee (\text{at } \delta_1 \wedge \text{at } \eta_1)) \tag{5}$$

III. M_7 and Robot5, Robot11. Let us consider two other cases:

a)

$$\Box\neg(\text{at } \beta_3 \wedge \text{at } \mu_1)$$

$$\Box\neg(\text{at } \beta_3 \wedge \text{at } \mu_3)$$

and together:

$$\models \Box\neg((\text{at } \beta_3 \wedge \text{at } \mu_1) \vee (\text{at } \beta_3 \wedge \text{at } \mu_3)) \tag{6a}$$

b)

$$\Box\neg(\text{at } \beta_5 \wedge \text{at } \mu_1)$$
$$\Box\neg(\text{at } \beta_5 \wedge \text{at } \mu_3)$$

and together:

$$\models \Box\neg((\text{at } \beta_5 \wedge \text{at } \mu_1) \vee (\text{at } \beta_5 \wedge \text{at } \mu_3)) \tag{6b}$$

Expressions (4), (5), (6a) and (6b) describe the requirement of mutual exclusion that is necessary for the correct execution of the program. Exclusion can be realized by the semaphore operations or simply by a proper structural instruction. The mutual exclusion for belt M_3 and instructions which operate on this belt are shown in Figure 4.

var
M_3: **shared** BufTyp;

α_6 : **region** M_3 **do** put (M_3);

β_1 : **region** M_3 **do** get (M_3);

η_3 : **region** M_3 **do** put (M_3);

Fig. 4. Mutual exclusion for one of the belts.

Corollary 6.1. Synchronization realized as shown in Figure 4 ensures mutual exclusion.

Proof. The proof follows directly from the definition of semaphore operations P and V. ∎

Clean behavior. Some belts are served by some processes and in fact they create the producer–consumer problem version with a limited buffer. Therefore, two requirements of clean behavior should be analyzed. (Another requirement of clean behavior so that two processes cannot operate on the same element of a belt in the same time is ensured and follows from the requirement of mutual exclusion).

I. Requirement not to take an element from an empty belt.

The requirement is expressible by:

$$\begin{aligned}\models \Box\ & ((\text{at } \beta_1 \Rightarrow \text{len}(M_3) > 0) \quad \wedge \\ & \wedge(\text{at } \eta_1 \Rightarrow \text{len}(M_5) > 0) \quad \wedge \\ & \wedge(\text{at } \mu_1 \Rightarrow \text{len}(M_7) > 0) \quad \wedge \\ & \wedge(\text{at } \mu_3 \Rightarrow \text{len}(M_7) > 0) \quad \wedge \\ & \wedge(\text{at } \mu_4 \Rightarrow \text{len}(M_8) > 0) \quad \wedge \\ & \wedge(\text{at } \mu_6 \Rightarrow \text{len}(M_8) > 0))\end{aligned} \tag{7}$$

II. Requirement not to put an element down on a full belt.
The requirement is expressible by:

$$\begin{aligned} \models \Box \quad & ((\text{at } \alpha_6 \Rightarrow \text{len}(M_3) < \max(M_3)) \quad \wedge \\ & \wedge(\text{at } \beta_3 \Rightarrow \text{len}(M_7) < \max(M_7)) \quad \wedge \\ & \wedge(\text{at } \beta_5 \Rightarrow \text{len}(M_7) < \max(M_7)) \quad \wedge \\ & \wedge(\text{at } \beta_6 \Rightarrow \text{len}(M_5) < \max(M_5)) \quad \wedge \\ & \wedge(\text{at } \eta_2 \Rightarrow \text{len}(M_3) < \max(M_3)) \quad \wedge \\ & \wedge(\text{at } \mu_2 \Rightarrow \text{len}(M_{10}) < \max(M_{10})) \quad \wedge \\ & \wedge(\text{at } \mu_8 \Rightarrow \text{len}(M_9) < \max(M_9)) \quad \wedge \\ & \wedge(\text{at } \mu_{10} \Rightarrow \text{len}(M_{10}) < \max(M_{10}))) \end{aligned} \tag{8}$$

These requirements for clean behavior expressed by formulae (7) and (8) can also be implemented by semaphore operations. However, these operations should precede the operations which are the result of mutual exclusion. The process should be suspended, if need be, because of an empty or full belt and then because of the critical section which is not free. The clean behavior for belt M_3 was shown in Figure 5.

```
var
   M3: shared BufTyp;
   empty, full: semaphore := max, 0;

      P(empty);                          P(full);
α6 :  region M3 do put (M3);        β1 :  region M3 do get (M3);
      V(full);                           V(empty);

                 P(empty);
              η3 : region M3 do put (M3);
                 V(full);
```

Fig. 5. Clean behavior for one of the belts.

Corollary 6.2. Synchronization realized as shown in Figure 5 ensures clean behavior.

Proof. Let us note once again that if the critical section is not occupied, then we can write:

$$\max = \text{empty} + \text{full} \tag{9}$$

Case 1. Suppose that it is possible to take an element from an empty belt. In this case, when operation V is executed and the process has left the critical section, the proper semaphore variable will be incremented and it will be satisfied:

$$K_i(\Diamond \text{empty} > \max) = \mathbf{t} \iff$$
$$K_i(\Diamond(\max - \text{full}) > \max) = \mathbf{t} \iff$$
$$K_i(\Diamond \text{full} < 0) = \mathbf{t}$$

The last inequality is not satisfied since the semaphore variable cannot have a negative value.

Case 2. Suppose now that it is possible to put an element down on a full belt. We write:

$$K_i(\Diamond \text{full} > \max) = \mathbf{t} \iff$$
$$K_i(\Diamond(\max - \text{empty}) > \max) = \mathbf{t} \iff$$
$$K_i(\Diamond \text{empty} < 0) = \mathbf{t}$$

This inequality is also not satisfied because of the reason mentioned in Case 1.

Thus, there is no possibility of both underflow and overflow of the belt. ■

Deadlock freedom. As it was already said, in case of some belts we have the producer–consumer problem. Usually, this problem cannot result in a deadlock (see section 4). However, there is a possibility of a deadlock in the system under consideration. It is connected with a service of belts M_3 and M_5 by robots R_5 and R_7. Suppose that robot R_9 has taken an element and belts M_3 and M_5 are full. Robot R_5 takes another element from belt M_3 and ascertains that it is not up to a standard. In the same time process Robots1–3 puts another element down on belt M_3 and the belt becomes full once again. Robot R_5 also ascertains that the element taken has not been improved on belt M_5. The deadlock arises just now. Robot R_5 is trying to put the element down on belt M_5 but it is impossible as the belt is full. Robot R_9 does not take any element from belt M_5 because it is trying to put an element down on belt M_3 but it is impossible as robot R_5 is not taking any elements.

The requirement for deadlock freedom in this case is expressible by:

$$\models \Box((\text{at } \beta_6 \wedge \text{at } \eta_2) \Rightarrow (\text{len}(M_5) < \max(M_5) \vee \text{len}(M_3) < \max(M_3))) \qquad (10)$$

As we can see, the program in the form as shown in Figure 3 is not correct since it is open to the deadlock.

In order to remove the possibility of the deadlock one should modify procedure Robot5. When the element which has been taken from belt M_3 is not up to a standard it is checked not only whether the element has already been attempted to be improved but also whether it is possible to put it down on belt M_5. The new version of the procedure Robot5 which is now called Robot5′ is shown in Figure 6.

```
        procedure Robot5'
        begin
β0 :      init (R5);
β1 :      get (M3);
β2 :      if correct (M3)
β3 :         then put (M7)
β4 :         else if attempt (M3) or (len(M5) = max(M5))
β5 :            then put (M7)
β6 :            else put (M5);
βe :      finish (R5);
        end {Robot5'};
```

Fig. 6. Deadlock freedom in the sample system.

Corollary 6.3. The program from Figure 3 including the modification from Figure 6 is free of deadlock.

Proof. If the deadlock is possible it will be shown by the expression which is the inverse of (10), namely:

$$\begin{aligned}
&K_i(\neg\Box((\text{at } \beta_6 \wedge \text{at } \eta_2) \Rightarrow (\text{len}(M_5) < \max(M_5) \vee \text{len}(M_3) < \max(M_3)))) \iff \\
&K_i(\Diamond\neg((\text{at } \beta_6 \wedge \text{at } \eta_2) \Rightarrow (\text{len}(M_5) < \max(M_5) \vee \text{len}(M_3) < \max(M_3)))) \iff \\
&K_i(\Diamond((\text{at } \beta_6 \wedge \text{at } \eta_2) \wedge (\text{len}(M_5) = \max(M_5) \wedge \text{len}(M_3) = \max(M_3)))) \iff \\
&K_i(\Diamond(\text{at } \beta_6 \wedge \text{at } \eta_2) \wedge \text{len}(M_5) = \max(M_5) \wedge \text{len}(M_3) = \max(M_3)))
\end{aligned} \tag{11}$$

From procedure Robot5' it appears that if $\text{len}(M_5) = \max(M_5)$ is satisfied the next executed instruction is the instruction labeled β_5. Instruction β_6 is not executed and expression (11) is not satisfied. Thus the deadlock is not possible. ■

Modification of procedure Robot5 excludes the possibility of the deadlock but it creates the situation in where elements which are not up to a standard may be immediately carried over belt M_7 without an attempt to improve its quality. Afterwards, having been checked by robot R_{11}, elements are carried over belt M_{10} and thereby discarded. (From the point of view of technology in order to reduce the probability of that situation the relative speed of work of robot R_7 should be selected in such a way that belt M_5 will be full as rarely as possible).

6. Conclusions

Program correctness properties and its classification have been discussed in the paper. First, a traditional solution to the synchronization question of the producer–consumer problem has been verified. Next, a sample parallel system of robots has been presented. Verification of the system has concerned three types of the safety property. Temporal logic has been used for the description of properties and for verifications.

Temporal logic makes program verification easier. It is a convenient notation for expressing the dynamic character of a program in a natural way. Even though it is not suitable for expressing the iteration in program (Wolper, 1983) it still seems that temporal logic is a powerful tool for analysis and proving correctness of concurrent programs. That has been shown in this paper on the example of verifications. Future studies shall discuss a problem of reduction of processes together with formulae which describe these processes. The formulae are expressed in temporal logic and the reduction occurs during a program verification. Selected tools and methods should be easily applied in a system for automatic reasoning for temporal logic, which will be the next step in the research.

Acknowledgement

I wish to thank prof. Tomasz Szmuc for his comments and reading a draft of the paper.

References

Clarke E.M., Browne M.C., Emerson E.A. and Sistla A.P. (1985): *Using Temporal Logic for Automatic Verification of Finite State Systems.*– In: Apt K. R. (ed.): Logics and Models of Concurrent Systems. – *Berlin: Springer–Verlag*, pp.3–25.

Dijkstra E.W. (1981): Introduction: *Why correctness must be a mathematical concern.*– In: Boyer R.S., Moore J.S. (eds.): The Correctness Problem in Computer Science; International Lecture Series in Computer Scienc. – *London, New York: Academic Press*, pp.1–8.

Fariñas–del–Cerro L. (1985): *Resolution Model Logicals.*– In: Apt K. R. (ed.): Logics and Models of Concurrent Systems. – *Berlin: Springer–Verlag*, pp.27–55.

Galton A. (1987): *Temporal Logics and their Applications.*– *London, San Diego: Academic Press.*

Habermann A.N. (1972): *Synchronization of Communicating Processes.*– Communications of the ACM, v.15, No.3, pp.171–176.

Hailpern B.H. (1982): *Verifying Concurrent Processes Using Temporal Logic.*– Lecture Notes in Computer Science. – *Berlin: Springer–Verlag*, v.129.

Henzinger T.A., Manna Z. and Pnueli A. (1991): *Temporal proof methodologies for real–time systems..*– Eigtheenth Annual ACM Symp. on *Principles of Programming Languages.* – Orlando, Florida, Jan. 21–23: pp.353–366.

Iszkowski W. and Maniecki M. (1982): *Cuncurrent Programming.*– *Warsaw:* Scientific –*Engineering Press.*

Krõger F. (1987): *Temporal Logics of Programs.*– EATCS Monographs on Theoretical Computer Science, – *Berlin, Heidelberg: Springer–Verlag.*

Lamport L. (1977): *Proving corecteness of multiprocess programs.*– IEEE Trans. Software Engeneering, v.Se–3, pp.125–143.

Manna Z. and Pnueli A. (1981): *Verification of concurrent programs: temporal framework.*– In: Boyer R.S., Moore J.S. (eds.): The Correctness Problem in Computer Science; International Lecture Series in Computer Science. – *London, New York: Academic Press*, pp.215–272.

Owicki S. and Lamport L. (1982): *Proving liveness properties of concurrent programs.*– *ACM Trans. Programming Languages and Systems*, v.4, No.3, pp.455–495.

Szmuc T. (1989): *Poprawność współbieżnych systemów oprogramowania.*– Cracow: Sci. Bull. of Staszic Academy of Mining and Metallurgy, No.1231, Automatics, Bulletin 46.

Wolper P. (1983): *Temporal logic can be more expressive.*– Information and Control, v.56, No.1–2, pp.72–99.

Received April 9, 1992

Appendix

Proofs of some temporal logic laws (Kröger, 1987).

Proof of T3

$$\begin{aligned} K_i(\neg \bigcirc A) = \mathbf{t} &\iff K_i(\bigcirc A) = \mathbf{f} \\ &\iff K_{i+1}(A) = \mathbf{f} \\ &\iff K_{i+1}(\neg A) = \mathbf{t} \\ &\iff K_i(\bigcirc \neg A) = \mathbf{t}. \end{aligned}$$

■

Proof of T9

$$\begin{aligned} K_i(\Diamond\Box A) = \mathbf{t} &\Rightarrow K_j(\Box A) = \mathbf{t} \quad \text{for some } j \geq i \\ &\Rightarrow K_k(A) = \mathbf{t} \quad \text{for every } k \geq j \text{ and some } j \geq i \\ &\Rightarrow (K_k(A) = \mathbf{t} \quad \text{for some } k \geq j) \text{ for every } j \geq i \\ &\Rightarrow K_j(\Diamond A) = \mathbf{t} \quad \text{for every } j \geq i \\ &\Rightarrow K_i(\Box\Diamond A) = \mathbf{t}. \end{aligned}$$

■

Proof of T10

$$\begin{aligned} K_i(\Box\Box A) = \mathbf{t} &\iff K_j(\Box A) = \mathbf{t} \text{ for every } j \geq i \\ &\iff K_k(A) = \mathbf{t} \quad \text{for every } k \geq j \text{ and every } j \geq i \\ &\iff (K_k(A) = \mathbf{t} \text{ for every } k \geq i \\ &\iff K_i(\Box A) = \mathbf{t}. \end{aligned}$$

■

Proof of T16

$$
\begin{aligned}
K_i(\bigcirc(A \Rightarrow B)) = \mathbf{t} &\iff K_{i+1}(A \Rightarrow B)) = \mathbf{t} \\
&\iff K_{i+1}(A) = \mathbf{f} \quad \text{or} \quad K_{i+1}(B) = \mathbf{t} \\
&\iff K_i(\bigcirc A) = \mathbf{f} \quad \text{or} \quad K_i(\bigcirc B) = \mathbf{t} \\
&\iff K_i(\bigcirc A \Rightarrow \bigcirc B) = \mathbf{t}.
\end{aligned}
$$

■

Proof of T17

$$
\begin{aligned}
K_i(\Box A \vee \Box B) = \mathbf{t} &\Rightarrow K_i(\Box A) = \mathbf{t} \quad \text{or} \quad K_i(\Box B) = \mathbf{t} \\
&\Rightarrow K_j(A) = \mathbf{t} \quad \text{for every } j \geq i \text{ or } K_j(B) = \mathbf{t} \text{ for every } j \geq i \\
&\Rightarrow K_j(A) = \mathbf{t} \quad \text{or } K_j(B) = \mathbf{t} \text{ for every } j \geq i \\
&\Rightarrow K_j(A \vee B) = \mathbf{t} \quad \text{for every } j \geq i \\
&\Rightarrow K_i(\Box(A \vee B)) = \mathbf{t}.
\end{aligned}
$$

■

Proof of T23

$$
\begin{aligned}
K_i(A \wedge \bigcirc \Box A) = \mathbf{t} &\iff K_i(A) = \mathbf{t} \text{ and } K_i(\bigcirc \Box A) = \mathbf{t} \\
&\iff K_i(A) = \mathbf{t} \text{ and } K_j(A) = \mathbf{t} \quad \text{for every } j \geq i+1 \\
&\iff K_j(A) = \mathbf{t} \quad \text{for every } j \geq i \\
&\iff K_i(\Box A) = \mathbf{t}.
\end{aligned}
$$

■