

Formal Verification of Event Driven Systems

Peter Bachmann

Cottbus University of Technology

Department of Computer Science

PO Box 10 13 44, D-03013 Cottbus

e-mail: Peter.Bachmann@informatik.tu-cottbus.de

October, 2006

Abstract

For software systems mainly driven by events, a special specification technique is introduced which allows to define the systems in a very clear way by the separation of event handling from the control of program actions. From this specification, an efficient implementation can be derived automatically. The approach allows to check the correctness of the specification with respect to certain expected properties. Moreover, some interesting temporal properties can be verified using the technique of predicate transformers.

1 Introduction

In a lot of applications (windows-, web applications, embedded systems) the program flow is mainly controlled by raising events. Each event can cause some actions like procedure calls in the program. In this way, the state of the program namely the existence and value of objects is influenced.

The connection between events and caused actions is described in different ways, dependent on the used system and technology. Commonly, events and corresponding actions are directly coupled. In C# (.NET), an event is *delegated* to a sequence of actions by means of *delegates*. This is described in the program parts (classes) where the occurrence of the event is expected. However, not all the events influence the program flow really. For instance, the only effect of moving the mouse over a window is mostly to change the mouse pointer. Either such an event is not delegated to an action or its occurrence is disabled. In the last case, this is expressed by a property of an object like a windows button.

This means, that event handling is distributed over the whole program text. So, there is no clear overview where and under which circumstances events may occur and which effects they release. It depends always on the current general situation of a program which is mainly characterized by the existence and values of program objects, i.e. the current memory assignment. This makes software maintenance complicated.

The basic idea to overcome this problem is to clearly separate event handling from the actions of the program. We introduce a special program state which is responsible for event handling alone. Let us call it the *event state*. Every event may change the event state.

So, there is one cycle in the program flow which is determined by raising events and changing the event state.

The connection between events and actions of the program is also controlled by means of the event state. A control function selects the actions which are executed on the basis of the current event state. Of course, this coupling between events and actions is much more loosely described as before but by a suitable structure of the event state this can be clearly expressed too. The focus of consideration is stronger directed to the actions than to the events.

The advantages of this approach consist in:

- The separation of event-handling from control of actions allows to design the whole system by two distinct persons: one is only responsible for managing events and the other one for the effects depending on the different states.
- The separated description of event-handling cycle as well as control of programm actions can be done in a relative abstract, specification like, manner. For this specification, no knowledge of any programming language is necessary. Nevertheless, an automatic transformation of the specification into the source code of the used programming language is possible. Moreover, the event-handling cycle can now be considered as a special transaction system. As it is shown in sections 3, 4.2 and 5, some optimizations for the transformation into programming language code and verifications of temporal assertions can be carried out.

2 Event Driven Systems

2.1 Structure

An Event Driven System (EDS) is modelled by

- S - a set of *states*,
- $E \subset S^S$ - a finite set of *events*, i.e. each event $e \in E$ is a total function $e : S \rightarrow S$, and
- $\delta : S \rightarrow \mathcal{A}^*$ - a control function, where \mathcal{A} is a certain set of *program actions*, i.e. for each state $s \in S$ it is $\delta(s) \in \mathcal{A}^*$ a finite sequence of program actions which all have to be carried out under the current state s .

2.2 Semantics

The pair (S, E) can be considered as a transition system where to each state s and each event e a transition $s \rightarrow e(s)$ belongs.

A sequence $\epsilon \in E^*$ of events defines the final state after all the events within ϵ are executed in the correspondig order, i.e.:

$$\varepsilon(s) := s, e.\varepsilon(s) := \varepsilon(e(s))$$

where ε denotes the empty sequence.

We extend the effect of a sequence $\epsilon \in E^*$ of events to produce all the intermediate states by considering the function $\epsilon^* : S \rightarrow S^*$ defined as

$$\varepsilon(s) := s, e.\epsilon^*(s) := s.\epsilon^*(e(s))$$

Analogously, we extend the control function δ to $\delta^* : S^* \rightarrow \mathcal{A}^*$, by

$$\delta^*(\varepsilon) := \varepsilon, \delta^*(s.\sigma) := \delta(s).\delta^*(\sigma).$$

Now, for any sequence $\epsilon \in E^*$ of events and any initial state s we get a sequence $\delta^*(\epsilon^*(s))$ of program actions which are executed when ϵ occurs. This defines the meaning of event driven system EDS.

2.3 Refinements

To define S, E, δ we go into more details.

We introduce a set X of *variables* and set $S = \mathbb{N}^X$ where \mathbb{N} denotes the set of natural numbers. So, each $s \in S$ is a function $s : X \rightarrow \mathbb{N}$ which assigns to every variable x a number $s(x)$, the value of variable x .

An event e is defined by a sequence of statements of form $C \Rightarrow t$. Here, $C = \bigwedge_{i \in [n]} c_i$ is a condition built by a conjunction of the c_i where $[n] = \{1, \dots, n\}$. Each c_i is a comparison where the operands are variables out of X or natural numbers and the operators are from the set $\{<, \leq, =, \geq, >\}$.

The c_i as well as the C define sets of states. A comparison c_i defines all the states for which the comparison is satisfied and condition C defines the states for which all the comparisons in C are satisfied. Therefore, we identify the comparison c_i and condition C with their states and use the same symbols for the formulas as well as for their sets of states.

So, we have $C = \bigcap_{i \in [n]} c_i$. C is satisfied by state s if and only if $s \in C$. If $n = 0$ we have $C = S$.

Every t is a sequence of simple assignments of form $x := e$ where x is a variable and e an expression over variables and numbers as operands and addition and subtraction as operators. The execution of sequence t implements an unconditional transformation of states $t : S \rightarrow S$.

If an event occurs then all the included statements are checked. If for any statement $C \Rightarrow t$ the current state s satisfies condition C , i.e. $s \in C$, then the transformation t is executed. We demand that for any two statements $C \Rightarrow t, C' \Rightarrow t'$ within the same event the set of variables changed in t and t' must be disjoint if $C \cap C' \neq \emptyset$. As a consequence, the order of these statements is not important and, the effect of an event e can be considered as a function $e : S \rightarrow S$.

The control function is also defined by a sequence of statements of form $C \Rightarrow \alpha$. Here, C is a condition as in an event, α is a sequence of program actions. For a current state s , it is $\delta(s)$ the concatenation of all sequences α for which the corresponding condition C is satisfied. This means, that the effect of the control function may depend on the order of the statements $C \Rightarrow \alpha$. However, reordering of these statements is allowed. So, there is no warranty that the control function works in accordance with the given order.

3 Optimizations

3.1 Simplifications

Each condition $C = \bigwedge_{i \in [n]} c_i$ can be simplified in a threefold manner:

- **Elimination of tautologies:**

If for some $j \in [n] : c_j = S$, then $C = \bigwedge_{i \in [n] - \{j\}} c_i$.

If $n = 0$, i.e. $[n] = \emptyset$, then we get $S \Rightarrow t$.

- **Elimination of contradictions:**

If $C = \emptyset$, then a statement $C \Rightarrow t$ can be removed from the event.

- **Elimination of implications:**

If for some $j, k \in [n] : c_k \subseteq c_j$ then $C = \bigwedge_{i \in [n] - \{j\}} c_i$.

For any comparison c it holds that $c = S$ iff c has the form $o \leq o, o = o$ or $o \geq o$ for some operand o . And, it holds that $c = \emptyset$ iff c has the form $o < o$ or $o > o$ for some operand o .

3.2 Check of Contradictions

The method to check whether $C = \emptyset$ is adapted to the tableau-method of propositional calculus:

Stepwise we transform condition C into a set \mathcal{C} of simple conditions which all of them contain only comparisons of form $o < o'$. This transformation is done in such a way that C is satisfiable iff one of the conditions from \mathcal{C} can be satisfied.

Step 1:

In C , replace every comparison of form $o > o'$ by $o' < o$ and every comparison of form $o \geq o'$ by $o' \leq o$.

Step 2:

Start with $\mathcal{C} := \{C\}$.

While there is in \mathcal{C} a C which contains a comparison of form $o \leq o'$: split C into two sets C' and C'' where the comparison $o \leq o'$ in C' is replaced by $o = o'$ but in C'' it is replaced by $o' < o$.

Step 3:

While in \mathcal{C} there is a set C which contains a comparison of form $o = o'$ remove it from \mathcal{C} and do:

- if o is a variable, say x , then replace any occurrence of x in C by o' ,
- otherwise, if o' is a variable, say x , then replace any occurrence of x in C by o ,
- otherwise, if both, o as well as o' , are different natural numbers then remove whole set C from \mathcal{C} .

Step 4:

While \mathcal{C} is not empty do:

Take any $C \in \mathcal{C}$. Let O be the set of all operands of comparisons in C and \mathbb{N} the set of natural numbers. Now, try to build a function $s : O \rightarrow \mathbb{N}$ in the following way:

- if o is a number then set
 $s(o) := o$
- otherwise, if o is a variable then set

$$s(o) := \begin{cases} 0 & \text{if there is no comparison } o' < o \text{ in } C \\ \max\{s(o') \mid o' < o \in C\} + 1 & \text{otherwise} \end{cases}$$

If such an function s does not exist or for the existing s there is a comparison $o' < o$ in C such that $s(o) \leq s(o')$ then remove C from \mathcal{C} .

Theorem 3.1 *The method above always terminates and, condition C is a contradiction if and only if set \mathcal{C} becomes empty.*

Proof:

The original condition C is a contradiction if and only if after step 2 every set in \mathcal{C} is a contradiction.

In step 3, all equations are resolved. But, if there is an equation which cannot be resolved because the left-hand side and right-hand side are different, the corresponding set is a contradiction and is removed from \mathcal{C} .

In step 4, by constructing the function s , we try to find values $s(o)$ for all operands o such, that all the comparisons in the corresponding conditions are fulfilled. Function s exists if and only if the transitive closure of relation $<$ is asymmetrically since in this case it defines a partial ordering which always can be completed to a total

one. But, for numbers as operands on the right-hand side function s may not meet the needed order. This must be checked separately.

A remaining condition of \mathcal{C} can be satisfied by the founded function s . Therefore, the original condition C can be satisfied and is not a contradiction. \square

3.3 Reordering

A specification as defined in section 2.3 for the events as well as for the control function, consists of a sequence

$$C_1 \Rightarrow x_1; \dots C_n \Rightarrow x_n;$$

where the C_i are conditions (conjunctions of comparisons) and the x_i are either sequences of simple assignments (in the case of events) or sequences of actions (in the case of control function).

From such a specification we can get a text of a programming language, like Java, C, C++ or C#, just by transforming it into a sequence of **if**-statements

$$\mathbf{if}(C_1) x_1; \dots \mathbf{if}(C_n) x_n;$$

But, in some cases it would be advantageous to reorder this sequence and to build nested **if**-statements. This may prolong the length of the text but improve the efficiency of execution. We do this by the following steps:

Step 1:

We reorder the sequence $C_1 \Rightarrow x_1; \dots C_n \Rightarrow x_n$; such that if $C_i \subseteq C_j$ then $j \leq i$.

Step 2:

We build the following sequence of nested **if**-statements:

$$\begin{aligned} & \mathbf{if}(C_1) \{x_1; \mathbf{if}(C_2) x_2; \dots \mathbf{if}(C_n) x_n; \mathbf{return}; \} \\ & \mathbf{if}(C_2) \{x_2; \mathbf{if}(C_3) x_3; \dots \mathbf{if}(C_n) x_n; \mathbf{return}; \} \\ & \dots \\ & \dots \\ & \dots \\ & \mathbf{if}(C_{n-1}) \{x_{n-1}; \mathbf{if}(C_n) x_n; \mathbf{return}; \} \\ & \mathbf{if}(C_n) x_n; \end{aligned}$$

Step 3:

We remove each **if**-statement $\mathbf{if}(C_i)\{\dots\}$ if there exists an j where $C_i \subseteq C_j$.

Step 4:

In each remaining **if**-statement

$$\mathbf{if}(C_i) \{x_i; \mathbf{if}(C_{i+1}) x_{i+1}; \dots \mathbf{if}(C_n) x_n; \mathbf{return}; \}$$

we replace C_{i+k} by C'_{i+k} if C_{i+k} has the form $C_i \wedge C'_{i+k}$.

Step 5:

In each remaining **if**-statement

$$\mathbf{if}(C_i) \{x_i; \mathbf{if}(C_{i+1}) x_{i+1}; \dots \mathbf{if}(C_n) x_n; \mathbf{return}; \}$$

we remove the contained **if**-statement $\mathbf{if}(C_{i+k}) x_k$; if $C_i \cap C_{i+k} = \emptyset$.

4 Predicate Transformer

4.1 Basics

Use of predicate transformer is a well investigated method for the verification of programs. Without reviewing the whole theory, we consider two special cases of predicate transformers. This idea goes back to Sifakis ([Si82]) who applied this theory to general transition systems.

A transition system can be considered as a triplet $T = (Q, T, R)$ where

- Q is a countable set of *states*,
- $T = \{t_1, \dots, t_n\}$ is a set of transitions and
- $R = \{r_1, \dots, r_n\}$ is a set of binary relations on Q .

The meaning of the whole transition system can be considered as the union $r := \bigcup R$, also a binary relation on Q .

In our case, an event system can be considered as a transition system too. Here, S is the set of possible states and any event, say e , is a possible transition which transforms the current state. But, we have the special case that every event is a *total* function $e : S \rightarrow S$. As a consequence, $r := \bigcup E$ is a relation defined for all states $s \in S$, i.e. $sr := \{s' \mid (s, s') \in r\} \neq \emptyset$ always holds.

Analogously to Sifakis, for any relation $r \subseteq S \times S$ we define two predicate transformer $\langle r \rangle : \wp(S) \times \wp(S)$ and $[r] : \wp(S) \times \wp(S)$ by

Definition 4.1

$$\langle r \rangle(A) := \{s \mid sr \subseteq A\} \text{ and } [r](A) := \{s \mid sr \cap A \neq \emptyset\}.$$

In the sense of Dijkstra ([Di76]), $\langle r \rangle(A)$ is the *weakest liberal precondition* of proposition A which means that for any state $s \in \langle r \rangle(A)$ all *results* sr of s by using relation r must meet the postcondition A . Note that for any state s which $sr = \emptyset$ we have $s \in \langle r \rangle(A)$. And, $\langle r \rangle(\emptyset) = \{s \mid sr = \emptyset\}$ is the set of all states for which r is not defined. Therefore, $\overline{\langle r \rangle(\emptyset)} = \{s \mid sr \neq \emptyset\}$ is the set of all states for which r is defined, for which at least one result exists.

The *weakest precondition* is defined by $\langle r \rangle(A) \cap \overline{\langle r \rangle(\emptyset)}$. It contains exactly all states s with $rs \neq \emptyset$ and $sr \subseteq A$. Note that if r is always defined, as in our case, then $\overline{\langle r \rangle(\emptyset)} = S$ and therefore, weakest liberal precondition and weakest precondition coincide.

In terms of event systems, $\langle r \rangle(A)$ is the set of all states for which all by *any* event transformed states, fulfil the postcondition A .

The meaning of predicate transformer $[r]$ is slightly different. $[r](A)$ is the set of all states which can be transformed by *some* events into states which fulfil the postcondition A .

It is obvious that:

Lemma 4.2 $\forall s : (s \in S \rightarrow sr \neq \emptyset) \rightarrow (\forall A : A \subseteq S \rightarrow \langle r \rangle(A) \subseteq [r](A))$

Proof: $rs \neq \emptyset \wedge sr \subseteq A \rightarrow sr \cap A \neq \emptyset$ □

As we will see in section 5, both predicate transformers are useful to verify interesting properties of event systems. However, only one is necessary, because of:

Lemma 4.3

$$[r](A) = \overline{\langle r \rangle(\overline{A})} \text{ and } \langle r \rangle(A) = \overline{[r](\overline{A})}.$$

Proof:

$$[r](A) = \{s \mid sr \cap A \neq \emptyset\} = \overline{\{s \mid sr \cap A = \emptyset\}} = \overline{\{s \mid sr \subseteq \overline{A}\}} = \overline{\langle r \rangle(\overline{A})}$$

and analogously

$$\langle r \rangle(A) = \{s \mid sr \subseteq A\} = \{s \mid sr \cap \overline{A} = \emptyset\} = \overline{\{s \mid sr \cap \overline{A} \neq \emptyset\}} = \overline{[r](\overline{A})}$$

□

And, for the special case of event $e : S \rightarrow S$ we get:

Lemma 4.4 $\langle e \rangle = [e]$

Proof: $\langle e \rangle(A) = \{s \mid \{e(s)\} \subseteq A\} = \{s \mid e(s) \in A\} = \{s \mid \{e(s)\} \cap A \neq \emptyset\}$ \square

Both predicate transformer are monotonic:

Lemma 4.5 *If $A \subseteq A'$ then $\langle r \rangle(A) \subseteq \langle r \rangle(A')$ and $[r](A) \subseteq [r](A')$.*

Proof: It holds

$$\langle r \rangle(A) = \{s \mid sr \subseteq A\} \subseteq \{s \mid sr \subseteq A'\} = \langle r \rangle(A').$$

And from $A \subseteq A'$ follows $\bar{A}' \subseteq \bar{A}$, consequently $\langle r \rangle(\bar{A}') \subseteq \langle r \rangle(\bar{A})$ what means $[r](A) = \langle r \rangle(\bar{A}) \subseteq \langle r \rangle(\bar{A}') = [r](A')$. \square

For any relation $r \subseteq S \times S$ we define relations $r^n \subseteq S \times S$ for $n \in \mathbb{N}$ and $r^* \subseteq S \times S$ by

$$r^0 := \{(s, s) \mid s \in S\}, \quad r^{n+1} := r \circ r^n \quad \text{and} \quad r^* := \bigcup \{r^n \mid n \in \mathbb{N}\}.$$

Now, for $r = \bigcup E$, the predicate transformers $\langle r^* \rangle$ and $[r^*]$ says:

If for the current state s we have $s \in \langle r^* \rangle(A)$ and a finite number of any events was raised then the resulting state s' **must** fulfil A , i.e. for any sequence $\epsilon \in E^*$ of events we get $\epsilon(s) \in A$.

And, if $s \in [r^*](A)$ and a finite number of any events was raised then it can happen that a resulting state s' fulfils A . In other words, there exists a sequence $\epsilon \in E^*$ of events such that $\epsilon(s) \in A$.

Lemma 4.6 $\langle r \rangle(A \cap A') = \langle r \rangle(A) \cap \langle r \rangle(A')$

Proof:

$$\begin{aligned} \langle r \rangle(A \cap A') &= \{s \mid sr \subseteq (A \cap A')\} = \{s \mid sr \subseteq A \wedge sr \subseteq A'\} \\ &= \{s \mid sr \subseteq A\} \cap \{s \mid sr \subseteq A'\} = \langle r \rangle(A) \cap \langle r \rangle(A') \end{aligned}$$

\square

Lemma 4.7 $[r](A \cup A') = [r](A) \cup [r](A')$

Proof:

$$\begin{aligned} [r](A \cup A') &= \{s \mid sr \cap (A \cup A') \neq \emptyset\} = \{s \mid sr \cap A \neq \emptyset \vee sr \cap A' \neq \emptyset\} \\ &= \{s \mid sr \cap A \neq \emptyset\} \cup \{s \mid sr \cap A' \neq \emptyset\} = [r](A) \cup [r](A') \end{aligned}$$

\square

4.2 Calculation of predicate transformers

It is well known (e.g. [Ho78]) that for a simple assignment like $x := e$ it holds that

Lemma 4.8 $\langle x := e \rangle(A) := A_e^x$

where A_e^x is the assertion got from a logical formula of A by substituting there all free occurrences of variable x by expression e .

And by lemma 4.4 we get

Lemma 4.9 $[x := e](A) := A_e^x$

Furthermore, it holds

Lemma 4.10 $\langle r \circ r' \rangle = \langle r' \rangle \circ \langle r \rangle$.

Proof:

$$\begin{aligned} \langle r \circ r' \rangle(A) &= \{s \mid s(r \circ r') \subseteq A\} = \{s \mid (sr)r' \subseteq A\} = \{s \mid sr \subseteq \{s' \mid s'r' \subseteq A\}\} \\ &= \{s \mid sr \subseteq \langle r' \rangle(A)\} = \langle r \rangle(\langle r' \rangle(A)) = (\langle r' \rangle \circ \langle r \rangle)(A). \end{aligned}$$

□

Similarly:

Lemma 4.11 $[r \circ r'] = [r'] \circ [r]$.

Proof:

$$[r \circ r'](A) = \overline{\langle r \circ r' \rangle(\bar{A})} = \overline{\langle r \rangle(\langle r' \rangle(\bar{A}))} = [r](\overline{\langle r' \rangle(\bar{A})}) = [r](\overline{[r'](A)}) = ([r'] \circ [r])(A).$$

□

However, we get

Lemma 4.12 $\langle r \cup r' \rangle(A) = \langle r \rangle(A) \cap \langle r' \rangle(A)$.

Proof:

$$\begin{aligned} \langle r \cup r' \rangle(A) &= \{s \mid s(r \cup r') \subseteq A\} = \{s \mid (sr \cup sr') \subseteq A\} = \\ &= \{s \mid sr \subseteq A \wedge sr' \subseteq A\} = \{s \mid sr \subseteq A\} \cap \{s \mid sr' \subseteq A\} = \langle r \rangle(A) \cap \langle r' \rangle(A). \end{aligned}$$

□

But:

Lemma 4.13 $[r \cup r'](A) = [r](A) \cup [r'](A)$.

Proof:

$$[r \cup r'](A) = \overline{\langle r \cup r' \rangle(\bar{A})} = \overline{\langle r \rangle(\bar{A}) \cap \langle r' \rangle(\bar{A})} = \overline{\langle r \rangle(\bar{A})} \cup \overline{\langle r' \rangle(\bar{A})} = [r](A) \cup [r'](A).$$

□

By lemma 4.12 we get for $r = \bigcup E$:

Lemma 4.14 $\langle r \rangle(A) = \bigcap \{\langle e \rangle(A) \mid e \in E\}$.

And, by lemma 4.10

Lemma 4.15 $\langle r^n \rangle = \langle r \rangle^n$

Together we have

Theorem 4.16 $\langle r^* \rangle(A) = \bigcap \{\langle r \rangle^n(A) \mid n \in \mathbb{N}\}$.

But:

Theorem 4.17 $[r^*](A) = \bigcup \{[r]^n(A) \mid n \in \mathbb{N}\}$.

Proof:

$$\begin{aligned} [r^*](A) &= \overline{\langle r^* \rangle(\overline{A})} = \overline{\bigcap \{ \langle r \rangle^n(\overline{A}) \mid n \in \mathbb{N} \}} = \\ &= \bigcup \{ \overline{\langle r \rangle^n(\overline{A})} \mid n \in \mathbb{N} \} = \bigcup \{ [r]^n(A) \mid n \in \mathbb{N} \}. \end{aligned}$$

□

It remains to describe the calculation of $\langle e \rangle$. In order to formalize that we introduce for any finite subset $I \subset \mathbb{N}$ of natural numbers and any set $\{f_i \mid f_i : X \rightarrow X \wedge i \in I\}$ of functions the operation $\circ_{i \in I} f_i$, defined by

$$(\circ_{i \in I} f_i)(x) := \begin{cases} x & \text{if } I = \emptyset, \\ (\circ_{i \in I - \{k\}} f_i)(f_k(x)) & \text{if } k = \max(I). \end{cases}$$

Using this operation, we get for $t = (x_1 := e_1, \dots, x_n := e_n)$ and $[n] = \{1, \dots, n\}$:

$$\langle t \rangle := \circ_{i \in [n]} \langle x_i := e_i \rangle.$$

Finally, for $e = (C_1 \Rightarrow t_1, \dots, C_n \Rightarrow t_n)$ we claim that

Theorem 4.18

$$\langle e \rangle(A) := \bigcup_{I \subseteq [n]} \left(\bigcap_{i \in I} C_i \cap \bigcap_{i \in \bar{I}} \overline{C}_i \cap (\circ_{i \in I} \langle t_i \rangle)(A) \right).$$

Proof:

By lemma 4.10 and the fact that the order of t_i is not important we have for any $I \subset \mathbb{N}$: $\langle \circ_{i \in I} t_i \rangle = \circ_{i \in I} \langle t_i \rangle$.

Let now s be any state where $e(s) \in A$ and let $I_s := \{i \mid s \in C_i\}$. This means that $e(s) = (\circ_{i \in I_s} t_i)(s)$. It follows $(\circ_{i \in I_s} t_i)(s) \in A$ and $s \in \langle \circ_{i \in I_s} t_i \rangle(A) = (\circ_{i \in I_s} \langle t_i \rangle)(A)$. Furthermore, it is $s \in (\bigcap_{i \in I_s} C_i \cap \bigcap_{i \in \bar{I}_s} \overline{C}_i)$ and therefore,

$$s \in (\bigcap_{i \in I_s} C_i \cap \bigcap_{i \in \bar{I}_s} \overline{C}_i \cap (\circ_{i \in I_s} \langle t_i \rangle)(A)).$$

But, for any $I \neq I_s$ we have $s \notin (\bigcap_{i \in I} C_i \cap \bigcap_{i \in \bar{I}} \overline{C}_i \cap (\circ_{i \in I} \langle t_i \rangle)(A))$, what means, that $s \in \langle e \rangle(A)$. □

5 Applications

5.1 Invariant and Trajectory

Definition 5.1 An assertion A is called an invariant if $A \subseteq \langle r \rangle(A)$.

If some state s meets the invariant A , i.e. $s \in A$, then $sr \subseteq A$, consequently: $sr^* \subseteq A$ and if $S_0 \subseteq A$ then $S_0 r^* \subseteq A$ too.

Lemma 5.2 It is A an invariant if and only if A is fixed point of $F(X) = X \cap \langle r \rangle(X)$.

Proof: $A \subseteq \langle r \rangle(A) \Leftrightarrow A = A \cap \langle r \rangle(A)$ □

Lemma 5.3 A is an invariant if and only if $A = \langle r^* \rangle(A)$.

Proof: Let A be an invariant. By theorem 4.16 we have

$$\langle r^* \rangle(A) = A \cap \bigcap \{ \langle r \rangle^{n+1}(A) \mid n \in \mathbb{N} \}$$

and therefore: $\langle r^* \rangle(A) \subseteq A$. By definition 5.1 and lemma 4.5 we get $\forall n \in \mathbb{N} : A \subseteq \langle r \rangle^n(A)$, i.e. $A \subseteq \bigcap \{ \langle r \rangle^n(A) \mid n \in \mathbb{N} \} = \langle r^* \rangle(A)$.

If, vice versa, $A = \langle r^* \rangle(A)$ then by theorem 4.16 we have $\langle r^* \rangle(A) \subseteq \langle r \rangle(A)$ and therefore $A \subseteq \langle r \rangle(A)$. □

Theorem 5.4 *It is $\langle r^* \rangle(A)$ the greatest invariant which is less or equal to A .*

Proof: By theorem 4.16 and lemma 4.6 we have

$$\langle r^* \rangle(A) = A \cap \bigcap \{ \langle r \rangle^{n+1}(A) \mid n \in \mathbb{N} \} \subseteq \bigcap \{ \langle r \rangle^{n+1}(A) \mid n \in \mathbb{N} \} = \langle r \rangle(\langle r^* \rangle(A)).$$

That means, $\langle r^* \rangle(A)$ is an invariant and $\langle r^* \rangle(A) \subseteq A$.

Let now A' be any invariant less or equal to A . Then, by lemma 5.3 and lemma 4.5 we get $A' = \langle r^* \rangle(A') \subseteq \langle r^* \rangle(A)$. \square

Definition 5.5 *An assertion A is called a trajectory if $A \subseteq [r](A)$.*

Analogously to lemma 5.2 we get

Lemma 5.6 *It is A a trajectory if and only if A is fixed point of $G(X) = X \cap [r](X)$.*

Proof: $A \subseteq [r](A) \Leftrightarrow A = A \cap [r](A)$ \square

However, the calculation of the fixed point is slightly different.

The lemma 5.3 cannot be adapted to $[r^*]$. A counterexample is given by:

$$S = \{0, 1, 2, 3\}, r = \{(0, 1), (0, 3), (1, 3), (2, 2), (3, 2)\}.$$

Here, set $\{2\}$ is a trajectory since $\{2\} \subseteq [r](\{2\}) = \{2, 3\}$ but $[r^*](\{2\}) = \{0, 1, 2, 3\}$. On the other hand, we have $[r^*](\{0, 1\}) = \{0, 1\}$ but $[r](\{0, 1\}) = \{0\}$, and $\{0, 1\}$ is not a trajectory.

As a consequence, also theorem 5.4 cannot be adapted, as the counterexample shows. We have indeed to calculate a fixed point of $G(X)$ in accordance with the general theory.

Theorem 5.7 *It is $G^\times(A) := \bigcap \{G^n(A) \mid n \in \mathbb{N}\}$ the greatest trajectory less or equal to A .*

Proof:

It is $G^{n+1}(A) = G^n(A) \cap [r]G^n(A)$ and therefore, $G^n(A) \subseteq G^i(A)$ for all $i \leq n$. Let now s be any state with $s \in \bigcap \{G^{n+1}(A) \mid n \in \mathbb{N}\}$. Then, for all $n \in \mathbb{N}$ we have $s \in G^{n+1}(A)$ and therefore, $s \in [r](G^n(A))$.

That means: for all $n \in \mathbb{N}$ a state s_n exists with $s_n \in sr$ and $s_n \in G^n(A)$. But, relation $r = \bigcup E$ is the union of finite many functions and therefore, set sr is always finite. As a consequence, at least one of the elements of set sr , say s' , must occur infinitely often in the sequence s_n . We have $\forall n \in \mathbb{N} : s' \in G^n(A)$, therefore $s' \in \bigcap \{G^n(A) \mid n \in \mathbb{N}\}$ and, finally, $s \in [r](\bigcap \{G^n(A) \mid n \in \mathbb{N}\})$. This shows that $\bigcap \{G^n(A) \mid n \in \mathbb{N}\} \subseteq [r](\bigcap \{G^n(A) \mid n \in \mathbb{N}\})$ and proves that $G^\times(A)$ is a trajectory.

If A' is any trajectory with $A' \subseteq A$ then we get by monotocy of G and, obviously $G^\times(A) \subseteq A$: $A' = G^\times(A') \subseteq G^\times(A) \subseteq A$. That proves the rest. \square

5.2 Some Temporal Assertions

Usually, verification of reactive systems is done by means of model-checking: from any formula of temporal logic an automaton is constructed and it is checked whether this automaton is compatible with the original system. Using this method needs a sound knowledge of temporal logic.

In our case, we restrict ourself on certain assertions which can be proved using the predicate transformers described above. This may be already very helpful to find

out some mistakes in the specification. In the following we show it by some examples. Remember, that S_0 means the set of initial states.

- A is invariant: *if A is fulfilled then it is always fulfilled.*
- A is invariant and $S_0 \subseteq A$: *A is always fulfilled.*
- A is trajectory: *if A is fulfilled then it can always be fulfilled.*
- A is trajectory and $S_0 \subseteq A$: *A can always be fulfilled.*
- $S_0 \subseteq \langle r^* \rangle(\bar{A})$: *A can never be fulfilled.*
- $S_0 \subseteq [r^*](A)$: *After a certain number of steps, A can be fulfilled.*
- $A \cap [r^*](A) = \emptyset$: *If A is sometimes fulfilled then afterwards never.*
- $S_0 \cap [r^*](A) \neq \emptyset \wedge A \cap [r^*](A) = \emptyset$: *A is almost once fulfilled.*

Maybe, somebody is especially interested in the effect of an event, say e . Then, for instance, we get:

- $\langle e \rangle(A)$: *the condition under which after e always A holds.*
- $S_0 \subseteq \langle (r - e)^* \rangle(\bar{A})$: *If e does not occur then A is never fulfilled.*
- $S_0 \subseteq \langle (r - e)^* \rangle(\bar{A}) \wedge S_0 \cap [r^*](A) \neq \emptyset$: *A can be fulfilled if e occurred but never else.*

Other cases can be constructed if we extend our constructions to

- $\langle r \rangle^* := \bigcup \{ \langle r \rangle^n(A) \mid n \in \mathbb{N} \}$ and
- $[r]^\times(A) := \bigcap \{ [r]^n(A) \mid n \in \mathbb{N} \}$.

Now, for instance, we get

- $S_0 \subseteq \langle r \rangle^*(A)$. *After a certain number of steps, A must be fulfilled.*
- $S_0 \subseteq [r]^\times(A)$: *In every step, A can be fulfilled.*

References

- [Ba92] P. Bachmann: *Mathematische Grundlagen der Informatik*, Akademie-Verlag Berlin, 1992
- [Di76] E.W.Dijkstra: *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, NJ, 1976.
- [Ho78] C.A.R. Hoare: *some properties of predicate transformers*, JACM 25 (3) (1978) 461-480.
- [Si82] J. Sifakis: *A UNIFIED APPROACH FOR STUDYING THE PROPERTIES OF TRANSITION SYSTEMS*, Theoretical Computer Science 18 (1982) 227-258, North-Holland Publishing Company