Safety verification of software using structured Petri nets

Krzysztof Sacha

Warsaw University of Technology, Institute of Control and Computation Engineering, ul. Nowowiejska 15/19, 00-665 Warszawa, Poland

Abstract. A method is described for the analysis and the verification of safety in software systems. The method offers a formal notation for describing the software structure, the means for defining safe and unsafe states of the system and a technique for the software simulation and analysis. The modeling process is based on an extension to Petri nets, which enables the modeler to represent control as well as data processing aspects of the software. The Petri net-based model can be analyzed using the concept of a modified reachability tree or can be used as a framework for a simulated execution. The model can be build in an early phase of the software development process, thus creating the potential for early verification and validation of safety.

1 Introduction

The problem of safety has always played a pivotal role in the development of control systems in such application areas as railway transport, process industry, power plants, etc. The technological shift from traditional hardwired systems towards software-based systems issued a new challenge — it has been recognized that safety analysis should also be incorporated into the software development process and embedded somehow into the software lifecycle. However, the problem of selecting the appropriate methods for safety analysis in software domain has not been solved as yet.

IEC 1508 (draft) standard [1] defines safety in terms of probabilistic measures (the notion of risk). Such a definition encourages two general methods of increasing system safety:

– employing redundant architectures,
– increasing the reliability of components.

The drawback of such an approach within the software domain is, that there are neither the means to measure the software reliability nor even a satisfactory definition. Hence, there are not many methods which can be practically applied to ensure the desired level of software reliability. A short survey of practices and methods which are applied within the software lifecycle in order to enable and to conduct safety analysis for software can be found in [2].

Nevertheless a number of factors have been identified which influence the software reliability. Among them are the following:

- Exact and unambiguous requirement specification.
- Early validation of safety of the specification.
- Credible verification of the design and the implementation against the requirements specification by means of:
  - proof of correctness,
  - testing.

At the current level of the software technology testing cannot guarantee program correctness, while full proofs of correctness are not feasible, as yet. Therefore it seems practical to develop an approach of partial proofs of correctness, i.e. proofs related to selected features which are particularly resistive to testing. Those features correspond to all aspects of interprocess synchronization and communication. The reasoning is such, that a sequential program can effectively be tested. Testing a set of parallel processes is much more difficult.

The goal of this paper is to describe a formal method for the software specification and prototyping which addresses all of the above mentioned problems. The method — called Transnet — offers the potential of formal specification combined with an efficient technique of rapid prototyping and safety analysis. The paper is organized as follows. Section 2 provides the reader with an overview of the method which is based on an extension to Petri nets. Section 3 presents an illustrative example. A description of techniques used for analysis and prototyping is given in Section 4. Timing aspects are considered and formally defined in Section 5. Tools supporting the use of the method and plans for future work are discussed in Conclusions.

2 Overview of the method

Many methods and techniques have been developed for specifying real-time control systems in a formal way. Formal methods are based on mathematical theories, such as: algebra, temporal logic, finite-state machines, functional programming or Petri nets. Transnet is a method which adopts a modified model of Petri nets.

A classical Petri net [3] can be viewed as a bipartite directed graph consisting of nodes which are places and transitions, and oriented arcs. Places are represented graphically by circles, and transitions by bars (or rectangles).The arcs join places to transitions and transitions to places in such a way that neither two places nor two transitions are linked directly. Places of a net can be marked with tokens, drawn as dots. Tokens can move between places as result of transition firings. A transition is enabled, i.e. ready to fire, if all places that input the transition have a token. Firing a transition removes a token from each of its input places and deposits a token in each of its output places. The current distribution of tokens among places which is called a marking, defines the current state of the net.

Petri net is usually interpreted as a control flow graph of a modeled system. Places correspond to conditions within the system, while transitions correspond to actions. A condition can be fulfilled, i.e. marked with a token in the current system state, or not. An action, i.e. a transition firing, can move tokens between places, thus reflecting a change to the system state.

Classical Petri nets are focused entirely on representing the flow of control and provide no opportunity for representing data and time-dependent aspects of the system operation. Hence, various extensions to the basic model have been invented and described in the literature [4–8].

The net model adopted by Transnet [9] restricts the net structure to a composition of three basic building blocks:

– sequential composition,
– alternative selection,
– parallel branching.

This restriction complies with the basic control structures used in program design and implementation. Moreover, it helps in maintaining readability of the specification.

On the other hand, Transnet extends the definition of Petri net by associating (Fig. 1):

– places with variables,
– transitions with data processing functions,
– arcs which lead from places to transitions with Boolean expressions.

The semantics of the extensions to Petri nets is as follows. A transition is enabled if all places that input the transition have a token (classics) and all Boolean expressions associated with the input arcs to the transition evaluate to *true* (extension). Firing a transition removes and deposits tokens as usually, but additionally the function associated with the transition is evaluated and the results are substituted to variables associated with output places. This way, an execution of a net can model the flow of control within the modeled system as well as data values evaluated during the system operation.

A specification of a control system is build as a set of concurrent processes, each of which is modeled by an extended Petri net. A process can be a representation of a system object, such as task or data buffer, or an environmental object, such as discrete model of a physical process. Processes in a specification can cooperate with each other, exchanging messages during a symmetric and synchronous rendezvous, modeled by an **exchange transition** i.e. a transition with input and output arcs belonging to both cooperating processes. A complete specification is formal and executable and therefore can be used as a system prototype.

The introduction of variables and arc expressions changes dramatically the semantics of the model, as the net marking can no longer be used to characterize the state of the model. The real results of the modeled computation are stored as values of variables, while the current marking describes only an internal state of the computation. A more elaborate treatment of this problem is given below.

All processes of a specification are cyclic and run forever. Each process has a distinguished place, referred to as the **terminal place**, which is marked in the inactive state of the process, just between two consecutive cycles of execution. The terminal place holds a token in the initial net marking. The variables associated with the terminal place which are called **terminal variables**, store the results of the computation and retain the process history between the consecutive process cycles. The values of terminal variables are changed at the end of the current process cycle and are stable throughout the next cycle. Other variables do not retain the process history. They comply with the single assignment rule — the idea borrowed from functional languages — and are used only as value-holders which are invalidated when the terminal place of the process receives a token.

The **specification state** is defined as a vector of values of all terminal variables. A sequence of states produced during the net execution, called a **trace**, determines the behavior of the specification. This can be observed from the outside and interpreted by people or devices. The execution of a specification net is indeterministic due to indeterministic scheduling. This implies that a specification net has many traces and the meaning of a specification is characterized by a **trace set**, including all traces that may be produced during the specification net execution. The trace set of a specification can be subject to analysis and validation against the user requirements. This includes also the requirement for system safety.

The method for the specification analysis is based on the construction of the reachability graph — a standard analysis technique developed for various kinds of Petri nets. The initial marking of the net contains a single token in the terminal place of each process. Due to the restrictions imposed on the net structure, the entire net is conservative, hence bounded. Therefore, the reachability graph of the net itself, i.e. without regarding variables, is finite.

Reachability graph of the specification can be analyzed in order to verify the structural correctness of the interprocess communication. This involves examination of the reachability of selected net markings and verification against deadlock. The Boolean expressions associated with arcs can influence the reachability graph in such a way, that the arcs related to conditions which evaluate to *false* become ineffective and can be removed. This enables the modeler to simulate various scenarios of the specification net execution by assuming different values of particular expressions.

3 Example: Railroad Crossing

Consider a railroad crossing equipped with a semaphore (*green* — *red*) which controls the movement of trains, and a gate (*up* — *down*) which controls the road traffic. Both devices are controlled by a computer system which receives and processes the information related to the train position. The semaphore is *red* and the gate is *up* in the initial state of the crossing.

A specification of the railroad crossing control system can be split into four parallel activities which can be modeled by a set of four parallel processes (Fig. 1):

– keeping track of the current train position (places $p_1 \dots p_3$),
– deciding: *red* or *green*, and *up* or *down* (places $c_1 \dots c_6$),
– operating the semaphore (places s_1, s_2),
– operating the gate (places g_1, g_2).

All processes are cyclic and run forever. The processes communicate with each other to maintain the required state of the railway devices. By convention, arc expressions not shown explicitly in the figure are all equal to *true*.

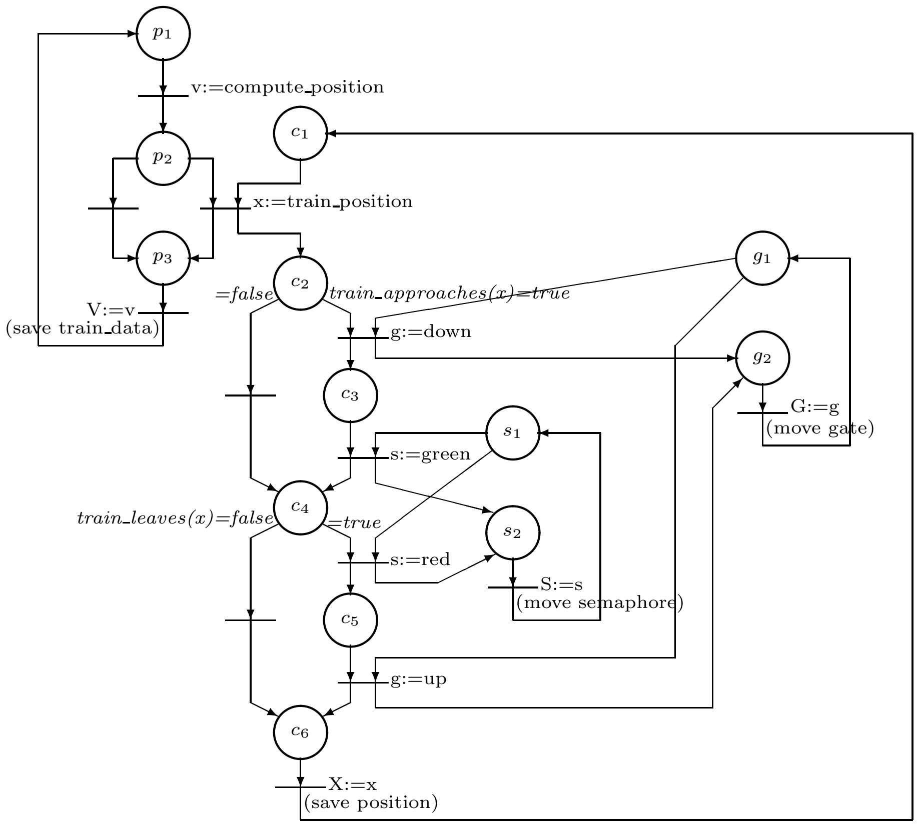

Fig. 1. Transnet model of the railroad crossing control system (terminal variables capitalized, arc expressions in italics, comments in parenthesis)

The first process (p places) models the computation of the current train position. The function *compute_position* can be viewed as a simulator of the train movement or as a procedure which receives the information from the railway track system and maintains the real train position. When the computation is finished, i.e. the place p_2 is marked, the train position is offered to the next process which decides on the necessary action. The communication between both processes is organized in such a way that it cannot block the process which keeps track of the train position — a token can always be moved from the place p_2 to p_3, even if the place c_1 is not marked at the moment.

The decision process (c places) starts with taking the new train position and storing it as a value of the local variable x. Next, when the place c_2 is marked, the process evaluates Boolean expression *train_approaches*, to check whether the train is just approaching the crossing. If this is the case (*train_approaches=true*), two consecutive commands for closing the gate and displaying the green signal on the semaphore are issued and passed to the device controllers. Otherwise (*train_approaches=false*), the token is moved to the place c_4 and nothing other is done. Thus, in both cases the token appears in the place c_4. Afterwards, the process evaluates Boolean expression *train_leaves*, to check whether the train is just leaving the crossing. If this is the case, two commands for displaying the red signal on the semaphore and opening the gate are issued. Otherwise, the token is moved to the place c_6. In the last step the current train position is stored as a value of the terminal variable X.

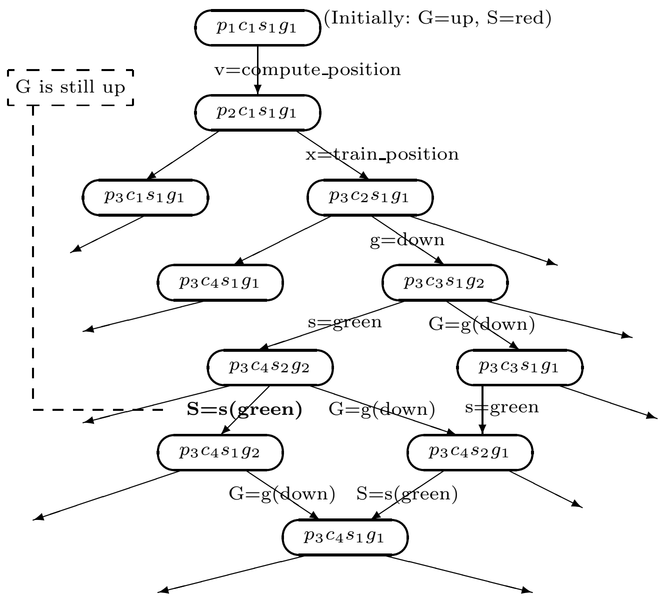

Fig. 2. Part of the reachability graph of the railroad control system

The remaining two processes are device controllers: of the gate and of the semaphore. The gate controller (g places) waits for a command (place g_1 marked), and after receiving the command and storing it as a value of the local variable g (place g_2 marked) it operates the gate accordingly (G=g). Thus, the value of the terminal variable G reflects the current state of the gate. The semaphore controller (s places) is nearly identical.

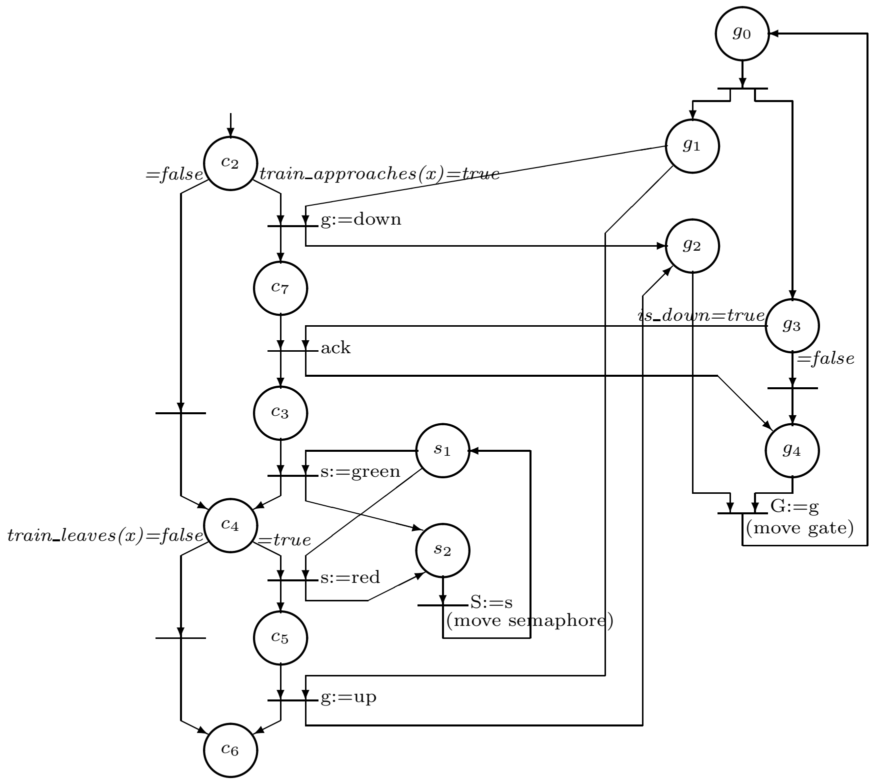

Fig. 3. Part of the modified Transnet model of the railroad crossing control system

It can be noted that the model preserves the parallelism inherent to the problem: the train, the semaphore and the gate are separate devices which operate concurrently, and concurrently with the decision process. A similar problem of modeling a railroad crossing control system by means of Petri nets was considered in [10]. Comparing both models one can note, that Transnet model closer relates to a real design of the control software than to an abstract description of the requirements.

The requirement for safe operation of the railroad crossing devices can be formulated in such a way, that no valuations of terminal variables in which *G=up* and *S=green* should appear in any trace of the specification. At the first glance it could seem that this requirement is fulfilled by the specification in Fig. 1, as the commands for moving the gate *down* and for displaying the signal *green* are issued by the decision process consecutively and in the right order. However, a look at the reachability graph (Fig. 2) shows, that a scenario of the net execution is possible which leads directly to the dangerous valuation. The un-safe behavior of the specification results from the possibility of internal delay in both device controller processes.

Two solutions can be suggested to eliminate the possibility of reaching the dangerous state, both of which result in a change to the process structure. The first one requires that the gate and the semaphore processes are combined to form a single and sequential control process which can guarantee proper sequencing of the device operation. The other solution is based on a feedback from the gate — the device should be able to signal that the state *down* has been reached. This can be modeled (Fig. 3) by extending the specification net and adding new transitions which synchronize the decision process with the gate controller (synchronization by transition *ack* which can fire only if *is_down=true*).

The dangerous valuation can be reached in the specification in Fig. 1 also in case when the train leaves the crossing. The nature of the problem is similar to the previous one so the solution is also similar — the semaphore should be able to signal the *red* state, and the decision process should postpone opening the gate until the moment in which displaying the red light has been acknowledged.

When the modifications to the model are finished, the correctness of the final specification can formally be proved. Part of the reachability graph shown in Fig. 2 which has been modified in order to reflect changes made to the specification, is depicted in Fig. 4. The un-safe valuation of terminal variables G and S can never be reached.

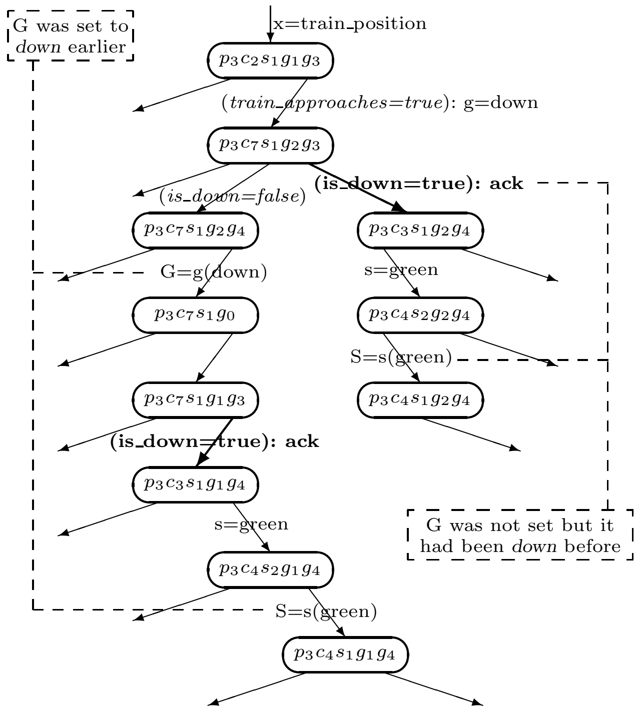

Fig. 4. Part of the modified reachability graph of the railroad control system of Fig. 3

4 Analysis Techniques

The analysis of Transnet specification is based on the analysis of a modified reachability graph of the net. The graph consists of nodes which correspond to the markings of places, and arcs which correspond to transition firings. Boolean arc expressions can influence the shape of the reachability graph in that they can control the firing of transitions which have a common input place. To reflect this influence, each arc of the modified graph is labeled with a name of the firing transition and a name of the Boolean expression which must have evaluated to true in order to enable the firing transition.

It can be noted that the modified graph with all Boolean expressions evaluated to *true* is identical to the classical reachability graph, i.e. the graph with no arc expressions at all. Such a graph represents an indeterministic net in which all conditions in all alternative branches can eventually be fulfilled. The analysis of this graph can help in detecting deadlock conditions due to improper process communication and synchronization. Deadlock can involve the entire specification net or a group of processes (livelock).

The existence of arc expressions extends the scope of analysis and enables the modeler to simulate various scenarios of the specification net execution. This can be done by assuming different values of particular arc expressions, and removing from the reachability graph those arcs which correspond to expressions which have evaluated to *false*. After the removal of selected arcs, the process of graph analysis can be repeated.

The application of the modified reachability graph analysis can be demonstrated using the example railroad crossing control system from the previous section. Consider, e.g., that the gate broke down and cannot be closed (i.e. it remains always in *up* position). This means, that the value of the arc expression *is_down* always evaluates to *false*, and the respective arcs (bold in Fig. 4) should be removed from the modified reachability graph. The remaining graph is safe in that green signal can never be displayed, but at the same time it has a livelock: the decision process is blocked with the token held in the place c_7.

The analysis of the modified reachability graph can be used to verify correctness of the control flow within the specification. In those cases when the processing of data is based on enumerative substitution of values, the reachability graph can also help in the analysis of the results of data processing. This was the case illustrated in the previous part of this section. Otherwise, the processing of data can be validated according to the concept of rapid prototyping.

The specification described in the form of an extended Petri can be executed by simulated execution of the net. Starting from the initial net marking and an initial valuation of terminal variables, a complete trace of the specification can be computed. Due to the inherent parallelism of Petri net the computation is indeterministic. Hence, the experiment can be repeated many times, thus allowing the modeler to investigate different scenarios of the execution. This way the behavior of the specification can be studied and validated. The validation process can include safety requirements which can be described by a set of prohibited values of terminal variables.

5 Timing aspects

Still another extension introduced by Transnet to Petri nets is a mechanism for modeling the flow of time and specifying timing constraints. The extension is very simple and consists of time constants assigned to arcs which lead from places to transitions. The meaning of a time constant Δt is such, that a token must reside in the input place of the arc at least through the time Δt until it can enable the transition pointed by the arc. An arc without an associated time constant is assumed to have the default time value equal to zero.

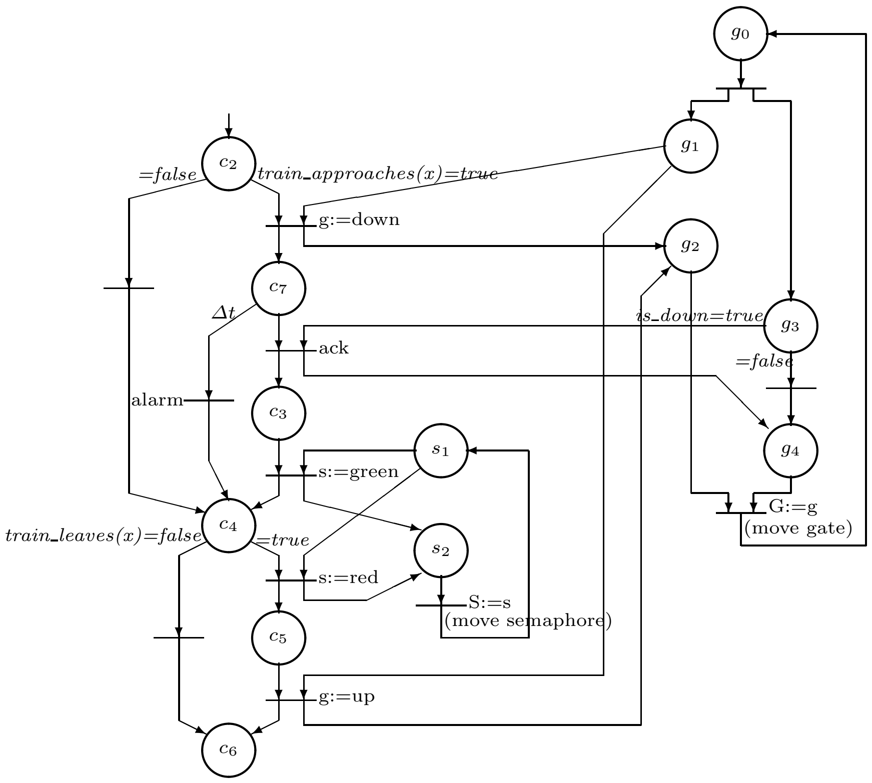

Fig. 5. Part of the model with a timeout

The applications of this time model in Transnet are twofold. First, it is used to define timeouts for potentially infinite operations. Consider, e.g., the model of a railroad crossing system shown in Fig. 3. It has been demonstrated in the previous section that a breakdown of the gate in its *up* position can block the decision process forever, with a token held in the place c_7. To resolve this livelock of the net, an additional transition is needed to react to the damage (the transition labeled *alarm* in Fig. 5). However, the additional transition should not fire during a regular functioning of the system — this can be forced by assigning a timeout value to the arc which joins the place c_7 with the added *alarm* transition.

The second application of time constants is to model the evaluation time of a function associated with a transition within a process net. One can note that such an evaluation time Δt can be modeled by a construction shown in Fig. 6.

Comparing Transnet model of time to other simple models, which assign time constants to transitions [4] or to places [5], one can note that Transnet model is equally simple, but more expressive. Time constants assigned to arcs can easily model timed transitions (e.g. as in Fig. 6) and timed places. The converse, however, is not true, as the semantics of a timeout can be described by means of time constants assigned neither to places nor to transitions. Moreover, Transnet model of time allows the firing time of a transition to be related separately to the time instants in which tokens are being deposited in each of the input places of this transition.

On the other hand, Transnet model of time is slightly less expressive, but much more simple, than the model proposed in [6] in which time intervals are assigned to transitions. This results from the fact, that Transnet model can describe only lower (but not upper) bound imposed on the firing time of a transition. Such a model matches the characteristics of programming languages and operating system which provide the tools for timeout operations, but do not provide any means to guarantee the upper bound of the execution time.

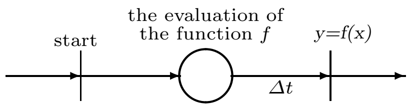

Fig. 6. Evaluation time of a function *f(x)*

Transnet model of time can be defined formally as a finite extension to Petri nets. Having such a definition one can use time-extended Petri net throughout the analysis process, instead of a classical Petri net.

Formally, Petri net can be defined [3] as a four-tuple $C = (P, T, I, O)$ composed of a finite set of places P, a finite set of transitions T, an input function $I : T \longrightarrow 2^P$, and an output function $O : T \longrightarrow 2^P$. It is assumed that the sets P and T are disjoint. A graphical interpretation of both functions is such that the function I defines the arcs which lead from places to transitions, while the function O defines the arcs which lead from transitions to places.

A marked Petri net is a pair (C, μ_0) composed of a Petri net C and an initial marking $\mu_0 : P \longrightarrow N$ which assigns natural numbers (with zero) to places. A number $\mu_0(p)$ is interpreted as a number of tokens in the place p. The marking of Petri net can be changed as result of a transition firing. A transition t is enabled in a marking μ if:

$$\mu(p) \geq \#(p, I(t)) \qquad \text{for all } p \in P$$

where $\#(p, I(t))$ equals 1, if $p \in I(t)$, or 0, if $p \notin I(t)$. A transition t enabled in a marking μ can fire. Firing the transition t leads to a new marking μ' defined as follows:

$$\mu'(p) = \mu(p) - \#(p, I(t)) + \#(p, O(t)) \qquad \text{for all } p \in P$$

A marked Petri net is **bounded** if the set $R(C, \mu_0)$ composed of all markings which can be reached from μ_0 by firing transitions is finite. The property of boundedness is of practical importance, as it guarantees, that the net can be analyzed completely by an exhaustive search through the set $R(C, \mu_0)$.

Petri net with timeouts is a three-tuple $K = (C, \mu_0, \tau_0)$ composed of a marked Petri net (C, μ_0) and a function $\tau_0 : P \times T \longrightarrow R$ which assigns rational numbers to arcs leading from places to transitions. A number $\tau_0(p, t)$ is interpreted as a time value which has been assigned to the arc from place p to transition t.

A state of a Petri net with timeouts can be defined as a pair $s = (\mu, \tau)$ composed of a marking μ and a vector of time values τ. The marking μ describes the current distribution of tokens among places. The vector τ describes the current (residual) values of time intervals associated with particular arcs. The size of the vector τ is equal to the number of arcs leading from places to transitions. An entry $\tau(p, t)$ is a rational number which is:

- not defined, if the arc (p, t) is closed, i.e. no token resides in the place p,
- zero, if the arc (p, t) is open, i.e. the place p has continued to be marked with a token at least through the time interval $\tau_0(p, t)$,
- equal to the length of the time interval remaining for opening the arc, if the arc (p, t) is active, i.e. the place p is marked with a token, but not open, as yet.

The execution of a Petri net with timeouts is controlled by the net marking μ and the values of the time intervals in τ. The net executes by shifting time intervals and firing transitions. A transition t is enabled in a state $s = (\mu, \tau)$ if it has all input arcs open, i.e.:

$$\tau(p, t) = 0 \qquad \text{for all } p \in I(t)$$

Let $s = (\mu, \tau)$ be a state of a Petri net with timeouts. The next state $s' = (\mu', \tau')$ can be computed in two steps:

1. If a transition t is enabled in s, then the computation moves to the step 2. If no transitions are enabled in s and Δt is the shortest non-zero time interval in τ, than the execution of the step 1 remains the marking μ unchanged, and shifts all time intervals in τ by subtracting the value Δt. Formally, step 1 computes for all p, t:

$$\text{new } \tau(p, t) = \begin{cases} \tau(p, t) - \Delta t & \text{if } \tau(p, t) \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

   The elements of t which were not defined before executing step 1, remain undefined. Step 1 repeats until a transition becomes enabled or all arcs in τ become open or closed.

2. If a transition t_j is enabled in s, then the next step s' is computed for all p, t as follows:
   a) $\mu'(p) = \mu(p) - \#(p, I(t_j)) + \#(p, O(t_j))$
   b)

$$\tau'(p,t) = \begin{cases} \tau_0(p,t) & \text{if } \; t = tj \;\text{ and }\; \mu'(p) \geq \#(p, I(t_j)) \\ \tau_0(p,t) & \text{if } \; \mu(p) - \#(p, I(t_j)) < \#(p, I(t)) \\ & \text{and } \;\; \mu'(p) \geq \#(p, I(t)) \\ \tau(p,t) & \text{otherwise} \end{cases}$$

It can be noted that states (μ, τ) of a Petri net with timeouts which have the same marking μ can differ from each other only with respect to the values held in vector τ. But from step 1 above results, that those values can never exceed the values stored in the initial vector τ_0. Because all values are rational numbers, hence the number of different values of vector τ is also finite. This leads to the following:

Theorem 1. *A Petri net with timeouts (C, μ_0, τ_0) has bounded number of states if and only if marked Petri net (C, μ_0) is bounded.*

The theorem is mathematically interested and shows that the analyzability of a Petri net with timeouts relays on the analyzability of the underlying marked Petri net. However, the practical implications are less important, as the state space of Petri net with timeouts is significantly larger than the one of classical Petri net.

It is interesting to observe, that the same proof and the same remark apply to a similar theorem in [6].

Conclusions

Transnet is a method for describing real-time computer systems. It can be used during the specification and preliminary design steps of the software life cycle as well as during the software verification phase [9]. Transnet model of the software structure can also be used for the analysis and the verification of safety. The method offers a formal notation for defining safe and un-safe states of the system and a technique for the software simulation and analysis.

To be practical, the method has to be supported by a set of software tools for creating, analyzing and executing the specification. Central element of the Transnet CASE system is a data base which stores a description of the specification under development. The description can be introduced into computer files by means of a graphical Petri net editor or textual editor. Management tools include decompilers and a browser which can look through the data base and produce a report including the list of processes, the list of exchange transitions, data types, etc.

The net analyzer enables the modeler to build a reachability tree of a Petri net and to answer questions related to deadlock and reachability of selected markings. The net analyzer does not evaluate functions. However, it can be instructed to assume particular values of arc expressions and to modify the tree by removing the arcs associated with expressions assumed *false*.

The net simulator executes the full model of a specification, taking into account data processing functions as well as time values. The problem of coping with huge volume of data produced during the simulation has been solved by exploiting the concept of validation points. A validation point can be identified by a particular net marking or submarking which determine strict points in the net execution. For each validation point a list of variables can be defined. When a validation point is reached during the net execution, the current time and the values of variables are recorded for further analysis.

The drawback of the current CASE system is that it consists of a set of separate programs, part of which have been implemented under DOS and part under Windows. Our current work is directed towards the unification of the system and shaping them in a coherent development environment, with a unified window-based user interface.

Acknowledgments

I wish to thank the anonymous reviewers, who provided comments and suggestions that improved this paper.

The work has been supported by a Warsaw University of Technology statutory grant 504/036, 1998.

References

1. IEC 1508 (draft). Functional Safety: Safety-Related Systems, IEC (1995)
2. Cegiela R., Sacha K., Zalewski A.: Task A3: Safety Analysis for the Software Domain. Copernicus Joint Research Project CP 94 1594 on Integraton of Safety Analysis Techniques for Process Control Systems. IASE. Wroclaw (1997)
3. Peterson, J., L.: Petri net theory and modeling of systems. Prentice-Hall Inc. (1981)
4. Ramchandani C.: Analysis of asynchronous concurrent systems by timed Petri nets. Massachusets Inst. Technol. Tech. Rep. **120** (1974)
5. Coolahan J. E., Roussopoulos N.: Timing requirements for time-driven systems using augmented Petri nets. IEEE Trans. Software Eng. **SE-9** (1983) 603–616
6. Berthomieu B., Diaz M.: Modeling and Verification of Time Dependent Systems Using Time Petri Nets. IEEE Trans. Software Eng. **17** (1991) 259–273
7. Jensen K.: Coloured Petri Nets. Advances in Petri Nets 1986. Brauer W., Riesig W., Rozenberg G. (eds) Springer-Verlag. (1987)
8. Ghezi C., Mandrioli D., Morasca S., Pezze M.: A unified high-level Petri net formalism for time-critical systems. IEEE Trans. Software Eng. **17** (1991) 160–172
9. Sacha K.: Real-Time Software Specification and Validation with Transnet. Real-Time Systems Journal. **6** (1994) 153–172
10. Leveson N. G., Stolzy J. L.: Safety Analysis Using Petri Nets. IEEE Transactions on Software Engineering. (1987)