IEEE Real-time Embedded Systems Workshop, Dec. 3, 2001

Fault Analysis Using Petri Nets

Krzysztof Sacha, *Member IEEE*

Abstract-- **The paper describes the specification method for dependable systems, called Transnet. The method offers a formal notation for describing the software structure, the means for defining safe and un-safe states and techniques 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. Petri net-based model can be analyzed using the concept of a modified reachability graph and a state graph or can be a framework for simulated execution. The model can be built in the specification phase, thus creating the potential for early validation of the software behavior.**

Index Terms—software specification, rapid prototyping, fault analysis, fault prevention, Petri nets

1 Introduction

Computer systems are used in many application areas in which a malfunction of the system can cause significant losses or even endanger the environment or human life. Examples of such areas are air and railway transport, process control and medical devices. The systems which are used in such or similar application areas are expected to exhibit always an acceptable behavior. This property of a system is often referred to as **dependability**. Any departure from the acceptable behavior is considered a system **failure**. Failures are caused by **faults**, which can arise in different phases of the system lifecycle.

Most of the techniques which have been devised for fault analysis are targeted towards hardwired systems and do not match the characteristics of software. A crucial difference between hardware and software system is that a program can neither break nor wear-out. Software faults can always be traced back to mistakes, which have been made during software specification, design or implementation.

To detect and remove faults, the software can be verified and validated against the requirements specification. The weakest point of this procedure is the requirements specification. Any fault or ambiguity in the specification can result in a fault in the software implementation.

Fault avoidance is another approach to increase software dependability. Most of the measures applied throughout the development process attempt to make the development more strict and formal. It is important that the process of adding rigor and formality could start from the very beginning, i.e. from developing a formal requirements specification which defines the space of all behaviors, which can be exhibited by the software. In the next step the unacceptable, e.g. dangerous, behaviors can be identified and defined in terms of the same formalism. Finally, one can check whether an unacceptable behavior can be deduced from the specification. If this is the case, the specification can be modified and the analysis repeated.

A special case arises, when the formal model which underlies a software specification is discrete. In such a case the space of all behaviors is discrete, and a definition of an unacceptable behavior can be reduced to a definition of unacceptable states. The evaluation of the software behavior can be conducted as a verification whether or not such states belong to the state space of the specification.

This paper describes a method for software specification, which addresses the problem of fault analysis. The method consists of a mathematical formalism based on extended Petri nets and techniques for rapid prototyping and fault analysis. Section 2 provides the reader with an overview of the formal model. Section 3 describes the techniques used for fault analysis. An example is presented in Section 4. The results are summarized in Conclusions.

2 Formal Model

Petri net [1] can be represented as a bipartite directed graph composed of nodes, which are places and transitions, and oriented arcs. Places are represented graphically by circles, and transitions by bars (Figure 3). Neither two places nor two transitions can be 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, called a marking, defines the current state of the net.

Petri net is usually interpreted as a control flow graph of the modeled system. Places correspond to conditions, while transitions correspond to actions. In this paper Petri nets are used for modeling the flow of control within the software under design. The flow of a token through the net models the execution of a software process. Transitions correspond to actions, e.g. functions which have been executed. A transition firing moves a token between places, thus reflecting a change to the process state.

Krzysztof Sacha is with the Department of Control and Computation Engineering, Warsaw University of Technology, Warsaw, Poland. Email: k.sacha@ia.pw.edu.pl.

Marked Petri net can be viewed as an abstract machine which starts with an initial marking μ_0, selects an enabled transition t_0, computes the next marking μ_1, and continues: $\mu_0, \mu_1 \ldots \mu_k, \mu_{k+1} \ldots$ as long, as there exists at least one transition t_k enabled in marking μ_k. If more than one transition is enabled in a given marking μ_k, then the choice of the firing transition is random. This way a marked Petri can produce many different computations.

2.1 Representation of data

Petri net provides very little opportunity for representing data. This compromises also the capability for modeling the flow of control, which depends on the results of some earlier computation. To overcome this restriction various extensions to the model have been suggested. The models of Coloured Petri Nets [2] and Environment/Relationship Nets [3] both assign data values to tokens, and provide a mechanism to compute the values, and to control the choice of the firing transition. These models are complex, and describe in fact a data-flow machine in which multi-sets of tokens convey multi-sets of data items.

The model adopted in this paper is much simpler [4]. The definition of Petri net is extended by associating:

- places with variables,
- transitions with functions, and
- arcs from places to transitions with Boolean functions.

A transition is enabled if all places that input the transition have a token and all Boolean functions associated with arcs that input the transition evaluate to *true*. Firing a transition removes a token from each of the input places of the transition, and deposits a token in each of the output places. Additionally, the function associated with the transition is evaluated, and the results are substituted to variables associated with output places.

2.2 Overview of the Transnet model

The model adopted by Transnet is based on two concepts of digital computation: Concurrent processes and Petri nets. A **specification** is built as a set of concurrent processes, each of which is modeled by an extended Petri net. A **process** can represent a system object, such as a task, or an environmental object, such as a discrete model of a physical process. A complete specification is formal and executable and can be used as a prototype of software.

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 with a token 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. The values of terminal variables are changed at the end of the current process cycle and are stable throughout the next cycle. Other variables of the process do not retain the process history. They are used only as value-holders, which invalidate when the terminal place receives a token.

Variables comply with the single assignment rule: Any variable used within a process net can be assigned a value only once during a run through a process net. This implies that no local iteration inside the process net is allowed.

A process net can be composed of three building blocks:

- sequential composition, modeled by a sequence of transitions,
- alternative selection, modeled by a set of transitions with a common input place,
- parallel branching, modeled by a transition with a set of output places,

which correspond to the basic control structures used in program design and implementation. The result is that Petri nets used in this paper are safe Petri nets, which means that at most one token can reside in a place.

Two processes of a specification can interact with each other according to the concept of a rendezvous. Neither common variables nor buffered message transfer are provided. A rendezvous between two processes is modeled in a Transnet specification by a transition with two input arcs and two output arcs. Such an exchange transition belongs to both cooperating processes with a pair of one input arc and one output arc belonging to the one process, and the other pair of arcs belonging to the other process. The exchange function associated with such a transition returns in each process the value supplied as the function argument in the invocation within the other process.

The introduction of variables changes the semantics of the model, as the net marking can no longer be considered the *state* of the model. The real results of the computation are stored as values of variables, while the current marking describes only an internal state of the computation.

An **external state** of a specification, i.e. a state which can be observed from the outside and interpreted by people or devices, is defined as a vector of values of all terminal variables. A sequence of external states produced during the net execution, is called a **trace**. The execution is non-deterministic, and a specification can have many traces. The meaning of the specification is characterized by a **trace set**, including all traces that may be produced during the execution. The trace set can be subject to analysis and validation against the user requirements.

3 FAULT ANALYSIS

The flow of control within a Transnet specification is dictated by Petri net. The analysis techniques based on the construction of a reachability tree [1] can be used in order to verify the structural correctness of the interprocess communication. A disadvantage is, that the reachability tree does not take into account any information related to data values and decision points.

3.1 Modified reachability graph

Reachability graph of a marked Petri net is composed of nodes, which correspond to subsequent markings of the net, and oriented arcs, each of which describes the firing of an enabled transition. An arc is labeled by the transition fired.

The graph takes the form of a tree, with the initial marking in the root node. The construction of the graph proceeds from the root by firing all transitions enabled in the current marking. A branch of the tree is finished when a marking is reached with no transition enabled, or a marking which has previously appeared in the tree. The reachability graph of a safe Petri net is finite.

Reachability graph deals with net marking only. Boolean functions associated with arcs can influence the shape of the graph because they can control the firing of transitions. A firing of a transition is reflected in the graph by an arc between the two consecutive markings. In a modified reachability graph an arc label consists of two elements: The fired transition and the Boolean function which must have evaluated to *true* in order to enable the transition.

The values of Boolean functions are determined by the current valuation of variables. The specification models a software under design, and each computation reflects a particular scenario of the software execution. This enables the modeler to prototype the software and simulate various scenarios of execution by selecting and firing enabled transitions, and computing the values of variables.

A simplified analysis can be performed, by having abandoned the computation of variables and just assuming the values of particular Boolean functions. The arcs related to functions which evaluate to *false* become ineffective and can be removed from the modified reachability graph. The reduced reachability graph can be verified against deadlock and reachability of dangerous states.

3.2 State graph

S**tate graph** of a Transnet specification represents the set of computations of a specification. It consists of nodes and oriented arcs. A node consists of the current marking and the current valuation of variables, while an arc corresponds to a transition firing. Each arc is labeled by the transition fired and the Boolean function which must have evaluated to *true* in order to enable the transition.

The set of markings is finite. The set of valuations can, in general, be infinite. A special case arises, when the set of values is finite, i.e. all variables are of enumerative type. The set of valuations and the state space are finite. The state graph of such a specification is finite, and equivalent to a finite state machine of the specification. The advantages of Transnet notation over the notation of finite state machines are the following:

- direct description of parallelism, inherent to the problem at hand,
- well defined structure of processes of the software,
- compact size of Transnet specification.

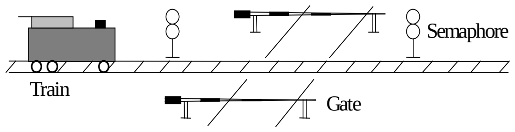
State graph of a Transnet specification can be subject to fault analysis. A fault in a specification can be defined as a valuation of terminal variables, which is unacceptable from the application viewpoint. The presence of faults can be verified by exhaustive analysis of the state graph.

Unfortunately, the state space of a specification can be large and a means of limiting the number of states is needed. This can be helped by layered construction of Transnet specification in the form of a hierarchy of models [4], which gradually expose more and more details.

4 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 road traffic (Figure 1). 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.

Figure 1: Railroad crossing.



A specification of the railroad crossing control system (and of the developed software) can be split into four parallel activities which can be modeled by a set of four parallel processes (Figure 3):
– keeping track of the current train position (places $p_1...p_3$),
– deciding: *red* or *green*, and *up* or *down* (places $c_1...c_6$),
– operating the semaphore (places s_1 , s_2),
– operating the gate (places g_1 , g_2).

The first process (*p* places) computes the current train position. The function *compute_position* can be a simulator or a procedure which maintains the real train position. When the computation is finished, i.e. place p_2 is marked with a token, the train position is offered to the 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. By convention, Boolean functions not shown in the figure are all equal to *true*. In the last step the current train data is stored as a value of terminal variable *V*.

The next process (*c* places) takes the new train position and stores it as a value of variable *x*. When place c_2 is marked, it evaluates Boolean function *train_approaches*, thus checking whether or not the train approaches the crossing. If this is the case (*train_approaches=true*), two consecutive commands for closing the gate and displaying green on the semaphore are transferred to appropriate device controllers by exchange transitions. Otherwise, the token is moved directly to place c_4. Afterwards, the process evaluates Boolean function *train_left*, thus checking whether or not the train has just left the crossing. If this is the case, two commands for displaying red on the semaphore and opening the gate are issued. The current train position is stored as a value of terminal variable *X*.

The remaining two processes are device controllers. The gate controller (*g* places) receives the command, stores it as a value of variable *g* (place g_2 marked) and operates the gate accordingly (*G=g*). Thus, the value of terminal variable *G* reflects the current state of the gate. The semaphore controller (*s* places) is almost identical.

The requirement for safety of the railroad crossing is such that no valuation of terminal variables in which *G=up* and *S=green* can appear in any trace of the specification. It could seem that this requirement is fulfilled by the specification in Figure 3. This hypothesis can be verified by analyzing of the state graph of the specification.

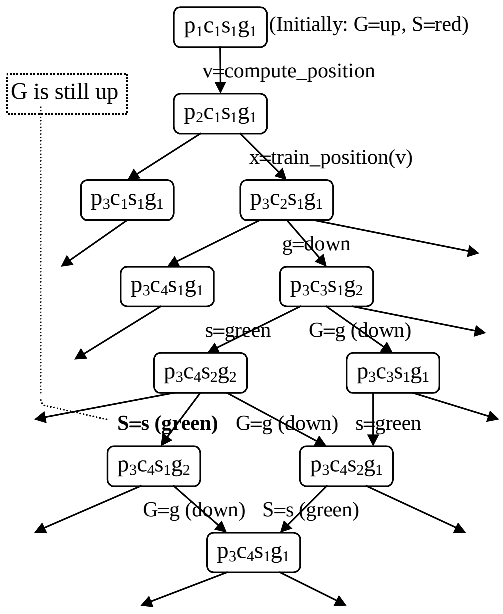


Figure 2. Part of the state graph

A relevant part of the state graph is shown in Figure 2. The values of variables have been shown nearby arcs, as commands to substitute a new value to a variable, which has been changed by the transition firing. One can note that an un-safe scenario of execution exists, which leads to the dangerous valuation. This results from the possibility of internal delay in both device controller processes.

5 Conclusion

Transnet method is aimed at the development of real-time software systems. An executable specification can be used as a prototype or can be analyzed using a state graph or a modified reachability graph. The method enables the modeler to define unacceptable states of the software and to verify the reachability of such states. The problems for future research. are: An environment for automatic transformation of the specification net structure, and means for expressing timing aspects of the software execution [4].

6 References

[1] Peterson, J. L: Petri net theory and modeling of systems, Prentice-Hall, Inc., 1981.

[2] Jensen, K: Coloured Petri Nets: A High Level Language for System Design and Analysis, LNCS 483, Springer-Verlag, pp. 342-416, 1991.

[3] Ghezi, C. et al.: A unified high-level Petri net formalism for time-critical systems, IEEE Trans. Software Eng., vol. 17, pp. 160-172, Feb. 1991.

[4] Sacha, K: Real-Time Software Specification and Validation with Transnet, Real-Time Systems Journal, vol. 6, pp. 153-172, 1994.

[5] Sacha, K.: Safety Verification of Software Using Petri Nets, LNCS 1516, pp. 329-342, 1998.

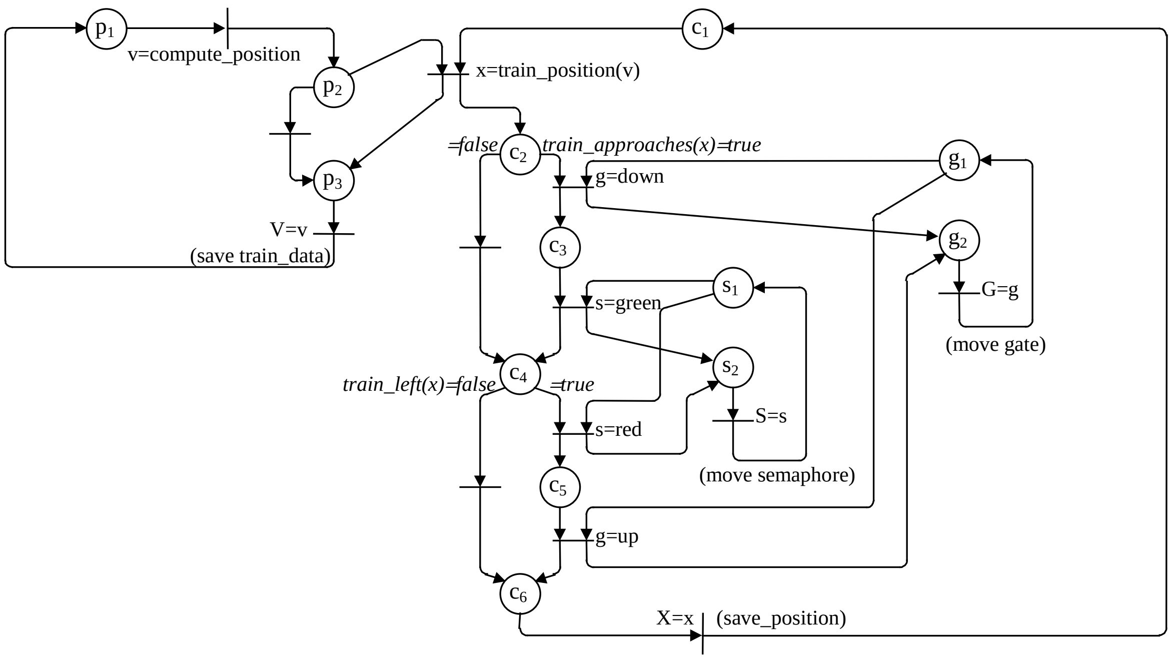


Figure 3. Transnet specification of the railroad crossing control system (Terminal places: p_1, c_1, s_1 and g_1)