# Real-Time Software Specification and Validation with Transnet

Krzysztof M. Sacha
Warsaw University of Technology

**Abstract:** The paper presents the executable specification method for real-time (embedded) systems Transnet. The method is based on an extension to Petri nets. A specification is developed by a problem decomposition into a set of parallel processes. Each process is defined by an extended Petri net with functions assigned to transitions, and conditions and time constants assigned to arcs. It is shown that Transnet matches with the characteristics of the intended class of applications. The available specification languages and the internal representation of data which describe the functional as well as nonfunctional requirements are outlined. The advantages of the Petri net-based representation and the possibilities of automatic net analysis and early validation are discussed.

**Keywords:** Real-time, requirements specification, early validation, Petri net, net analysis.

## 1 Introduction

Transnet is a method for describing real-time computer systems. It has been designed to support the specification and preliminary design steps of the software life-cycle. A Transnet specification is executable and can be examined analytically as well as validated experimentally by simulation.

Basic components of Transnet are the following:

- A method for describing the functional as well as nonfunctional requirements for software.
- A set of specification languages accompanied by software tools for creating the requirements specification.
- A set of analysis techniques accompanied by software tools for inspecting the specification.
- A method and a software tool for simulating the behavior of the specified system.

The goal of this paper is to present basic features of Transnet and to demonstrate that the method is suitable for the intended application area. The most straightforward way to do this is to summarize the characteristics of real-time systems and to draw a correspondence between them and the features offered by Transnet.

A real-time system can be defined as a system "... in which computation is performed during the actual time that an external process occurs, in order that the computation results can be used to control, monitor, or respond in a timely manner to the external process." [1]. An inventory of real-time characteristics which can be derived from the above definition is the following.

*Continuous operation.* The definition does not refer to any "start" or "stop" points for the system activity. On the contrary, the system should — at least in principle — run forever, synchronously with the events which occur within the external process.

*Static environment.* A real-time system must be considered embedded into a larger environment which can be composed of external processes, other systems, people, and interfaces. The structure of the environment is determined by a permanent structure of technical installation and is in most cases complex, but inherently static. This means that the set of external objects, which are to be handled, as well as the interface between the objects and the computer system are constant. This makes the interaction between the system and the environment also static.

*Simultaneity.* The environment of a real-time embedded system is usually composed of a number of partially independent objects, e.g. workcells and machinery, sensors and actuators. The objects operate concurrently, producing data and requesting services from the computer system. To respond appropriately, the computer system should possess the ability for simultaneous processing of concurrent events.

*Timeliness.* The results of a computer operation, i.e. control signals or responses, must be evaluated according to certain rules and within fixed periods of time. Due to physical characteristics of the environment late results can be useless or even dangerous. The conclusion is that the correctness of real-time systems depends not only on the processing results, but also on the time when these results become available. Correct timing is determined by the needs of the environment and not by the computer's processing speed.

*Predictability and analyzability.* Data to be processed may arrive randomly distributed in time. The simultaneous occurrence of several events can lead to a competition for service. Such an indeterminism arises particularly in the case of transient overload and other error situations. Despite this unpredictability, the reactions to be carried out by the computer must be precisely planned and fully predictable. This raises the requirement for the analyzability of a real-time system description.

Several methods for specifying real-time systems have been developed in the last decade. The best known examples are Extended Structured Analysis [2], PAISLey [3] and Statemate [4]. Each of these methods can describe precisely the data and control flow aspects of the system under development. Neither of them, however, can offer the potential for the formal specification and automatic analysis of the non-functional, timing requirements.

Transnet addresses all five characteristics listed above. To show the expressiveness of the Transnet notation and the scope of formal reasoning which can be applied to validate the specification, this paper is organized as follows. Section 2 provides the reader with an overview of the formal foundations and the internal representation of a Transnet specification. The specification languages used in Transnet are briefly presented in Section 3. The discussion of five features of the Transnet method is given in Section 4. Each feature corresponds to one particular characteristic of a real-time system. The last section explains the background, current status, and plans for further research. The paper is written in an informal and readable style. Formal treatment and an algebraic model of the Transnet representation can be found in [5, 6].

## 2 Overview of the Transnet representation

Formal foundation of the Transnet method is a merger of two different models of digital computation, asynchronous processes and Petri nets. Any system is specified as a set of asynchronous processes, each of which is described by an extended Petri net. Particular processes can represent either system objects, such as tasks, data objects, buffers, or environmental objects such as peripheral devices, interfaces, people, etc. The processes can communicate with each other exchanging data during a symmetric and synchronous rendezvous.

A specification can be developed using three different languages. However, it is always compiled and represented in a computer system in the form of an extended Petri net. The extensions to Petri nets enable the definition of functions, data structures and timing. The advantages of such a representation are the following:

- Strict rules for a net structuring enable strong consistency checking.
- Time dependencies can be precisely expressed.
- The properties of a specification can be formally analyzed.
- A net is executable and it offers the potential for early validation.

*A) A process net.*

Each process is defined by its state space and a next-state function. The activity of a process consists of an infinite number of process steps. In each step the next-state function is evaluated thus defining the next state from the state space. The current state is replaced by the next state exactly at the edge between two consecutive process steps. Within a step the current state remains stable.

Classical Petri nets [7] are focused entirely on representing the flow of control and provide no opportunity for representing data. Several extensions to the basic formalism have been already proposed to add data and function modeling capabilities. The results are a number of models, called high level Petri nets, such as Colored Petri Nets [8], Predicate/Transition nets [9], and Environment/Relationship nets [10]. A high level Petri net describes, in fact, a data-flow machine in which multi-sets of tokens move across the network and convey multi-sets of data items.

The model adopted by the Transnet method is much simpler. The basic definition of Petri net is extended by associating:

- places with variables,
- transitions with data processing functions, and
- arcs which lead from places to transitions with boolean conditions (predicates).

The extension enables us to model basic processing structures, i.e. composition, conditional selection, and parallel computation. An example net of a simple device monitor process is shown in Fig. 1.

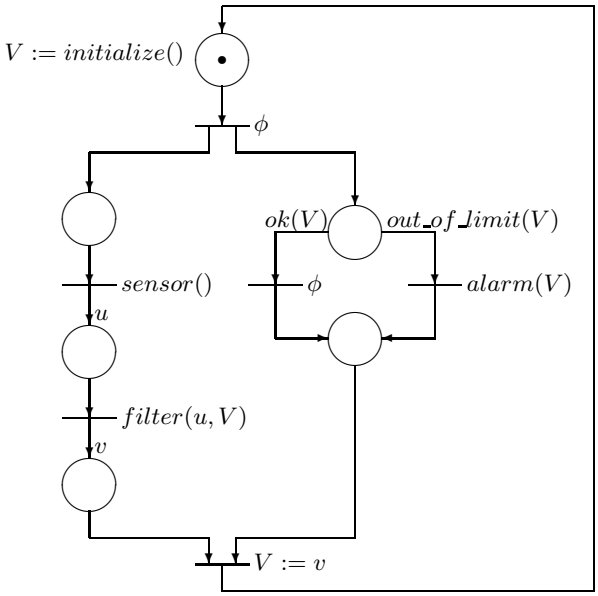


Figure 1: A device monitor process

The semantics of the extensions to Petri nets is easily discernible in Fig. 1. When a transition fires, the associated function is evaluated and the results are assigned to variables associated with the output places of this transition. Two successive transitions fire sequentially, e.g. $u := sensor()$ followed by $v := filter(u, V)$. Places with more than one output arc correspond to conditional selection. When a token resides in such a place, the conditions associated with its output arcs are examined and only the arcs associated with conditions which have been evaluated to 'true' are considered.

The device monitor process shown in Fig. 1 is cyclic and runs forever. Each cycle corresponds to a single process step. The process net is composed of two parallel branches. The left-hand branch is responsible for reading sensor data from a sensor device and executing a filtration algorithm. The filtered value $v$ is subsequently stored as the current value of the state variable $V$. In the right-hand branch the current value $V$ is examined. If the value falls within the limits, i.e. the condition $ok(V)$ evaluates to 'true', no action is performed. Otherwise, an alarm message is produced by the $alarm(V)$ function.

A Petri net which defines a process contains a distinguished place, referred to as a *terminal place*, which is marked with a token at the beginning of each process cycle. This place is indicated in Fig. 1 by a dot. The variables associated with the terminal place are called *terminal variables*, while the other ones are referred to as *nonterminal variables*. Terminal variables constitute the process state and retain the process history (e.g. $V$ in Fig. 1). Nonterminal variables are restricted in scope to a single process cycle and serve only as value-holders for conveying the data between the successive evaluations of functions (e.g. $u, v$ in Fig. 1).

The initial values are assigned to terminal variables by an *initialization function* before starting the net execution. The initialization function is defined by a constant expression, i.e. the one which is not allowed to contain any variable. During the net execution, new values are assigned to terminal variables exactly when the token is being deposited into the terminal place, and remain stable until the end of the current cycle. When the terminal place is marked, the values of all nonterminal variables are invalidated.

The semantics of functions associated with transitions can be defined twofold.

- An atomic function can be either intrinsic or defined by an appropriate subprogram in a programming language (actually a C function).
- A complex function is defined by a separate subnet composed of places, arcs, and transitions associated with simpler functions.

The decomposition of complex functions into subnets will be treated in more detail in Section 2-D.

The execution of a Petri net with parallel branches is indeterministic due to indeterministic scheduling. To prevent side-effects and indeterministic results of a process step, the process net must comply with the single assignment rule. This means that any variable used within a process net can be assigned a value only once, in one place in the process net. This implies, in fact, that no local iteration inside the process net is allowed. The consequences of this assumption will be discussed in detail in Section 4-C.

The current position of a token during a process net execution can be interpreted as the program counter which points to the currently evaluated function. According to this interpretation a Petri net which describes a process should be safe and conservative (but not strictly conservative as parallel branches inside the process net are still allowed).

*B) Inter-process communication.*

The inter-process communication is based on a two-way, symmetric, and synchronous rendezvous. Two processes can interact with each other by exchanging the arguments supplied at the invocation of a rendezvous. Neither common variables nor buffered message transfer are provided.

The synchronous communication principle simplifies the conceptual background of the model, as a specification consists of processes only, without any implicit buffers, counters, etc. Moreover, this complies with the tendency of using synchronous communication primitives in real-time programming environments, e.g. in the QNX operating system or Ada. Synchronous primitives, augmented with timeouts, enable a process which sends the data to check whether the operation has or has not been completed successfully.

A rendezvous between two processes is modeled in a Transnet specification by a transition with two input arcs and two output arcs. The transition is called an *exchange transition*, and it belongs to both cooperating processes with a pair of one input arc and one output arc belonging to one process, and the other pair belonging to the other process. The *exchange function* assigned to such a transition is a quasi-function which returns in each process the value supplied as the function argument by the other process.

An example of the inter-process interaction is shown in Fig. 2. The device simulator process offers the device data $X$ to the monitor process through the exchange transition *sensor*. No useful data is transferred in the opposite direction.

It is worth noting, that the exchange transition is executed in the device simulator process alternatively with another transition, associated with an empty function. This emulates real-time message transfer from one process to the other: The device simulator can never be stopped by the monitor process. To preserve the semantics of this control structure, the Transnet interpreter always gives a priority to the exchange transition over the other one (if both of them are ready for execution).

A Transnet specification is composed of process nets which have been 'glued' by exchange transitions. The concepts introduced so far can be formally defined in the following way. A transnet specification is a seven-tuple $\Sigma =< C, D, \pi, \sigma, \gamma, PT, IN >$ satisfying the requirements below:

1. $C =< P, T, I, O >$ is a Petri net composed of [7] a set of places $P$, a 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 $P, T$ are disjoint.

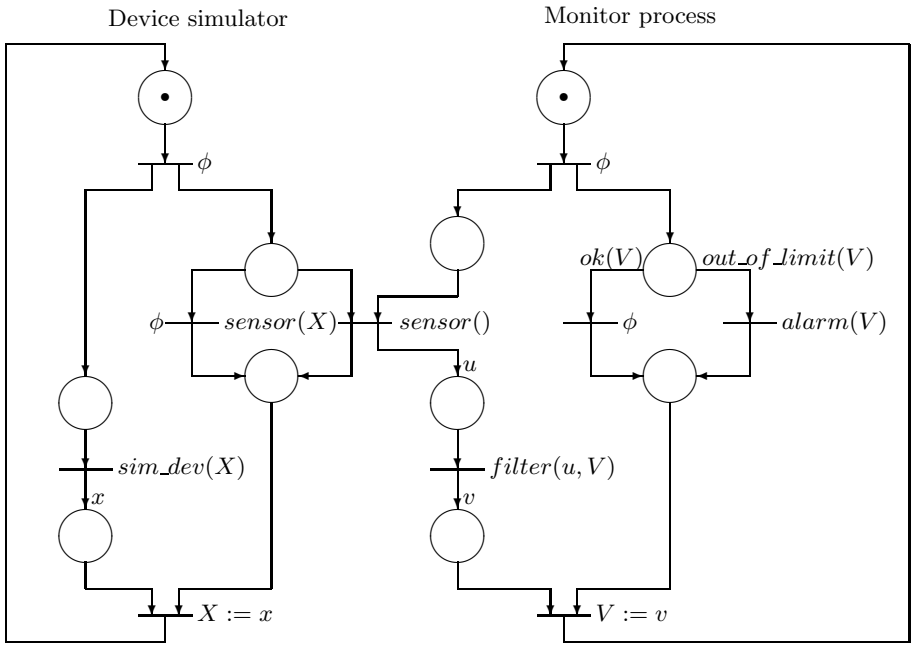


Figure 2: The interaction between the device simulator and the monitor processes

2. $D =< A, B, V, F >$ is a data path composed of [5] a set of variables $A$, a set of values $B$, a set of valuations $V : (\forall v \in V)(v : A \longrightarrow B)$, and a set of functions $F : (\forall f \in F)(f : B^* \longrightarrow B^*)$, where $B^*$ denotes the set of all lists over $B$.
3. $\pi : P \longrightarrow A^*$ is a function which associates places with lists of variables,
4. $\sigma : T \longrightarrow F \times A^* \times A^*$ is a function which associates transitions with signatures of functions; each signature consists of a function, a list of input variables, and a list of output variables.
5. $\gamma : P \times T \longrightarrow F \times A^*$ is a function which associates arcs with signatures of boolean functions.
6. $PT \subseteq P$ is a set of terminal places.
7. $IN : \pi(PT) \longrightarrow B$ is an initialization function. It is defined from the set of terminal variables into the set of values.

There is a token in each terminal place of the Petri net $C$ in the initial marking $M_0$.

Thus, the specification is formal and it offers the potential for automatic analysis of important properties such as reachability, liveness (i.e. deadlock-freedom), and transition liveness. The limitations imposed on the inter-process communication contribute greatly to the specification net analyzability. The extended Petri net which describes a specification preserves the properties of safeness and conservation. The lack of common variables makes the specification free from side-effects which could be caused by varying speed of the processes' execution.

*C) Timing.*

Extended Petri nets described so far enable the user to specify control and data processing aspects of the system under development. However, realistic modeling of real-time systems requires tools for describing time-dependent aspects of the system's behavior.

Three major extensions to Petri nets for handling time have been suggested in literature. Each of the models is derived from classical Petri nets by assigning time constants to transitions or places of the net.

Timed Petri nets [11] are defined as Petri nets with a finite firing duration associated with each transition. The firing rule of Petri nets is modified in two points. First, a transition must fire as soon as it is enabled. Second, when a transition with a firing duration $\Delta t$ fires, tokens are immediately removed from each of its input places and are deposited into each of its output places after the time $\Delta t$. The firing duration can model the evaluation time of a function associated with the transition.

Time Petri nets [12] are defined as Petri nets with two values of time, $t_1$ and $t_2$, associated with each

transition. The firing rule of Petri nets is modified as follows: A transition must be enabled at least through the time $t_1$ until it can fire. Assuming that the transition has continuously been enabled, it must fire before the expiration of the time $t_2$. Time Petri nets can most naturally be used to model the timeouts defined in a communication protocol.

The third model described in [13] is different in that time constants are assigned to places rather than transitions. Consider a place associated with a time value $\Delta t$. A token must reside in that place at least through the time $\Delta t$ until it can enable any transition. An enabled transition fires immediately. The model is, in fact, equivalent to Timed Petri nets. Each timed place can be modeled by a sequence of two places separated by a timed transition. Similarly, each timed transition can be modeled by a sequence of two transitions separated by a timed place.

Time Petri nets are more general than the other two models. Timeout values associated with transitions can be used to model timed places as well as timed transitions, but the converse is not true. Moreover, an enumerative method for the Time Petri net analysis has been developed recently [14].

The aforementioned time models have been generalized in Time ER nets introduced in [10]. The new model is formally defined and it offers a tremendous modeling power. On the other hand, however, time ER nets cannot offer any potential for the net analysis.

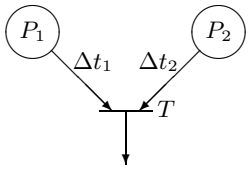


Figure 3: Time constants associated to arcs

Therefore, the Transnet representation of time is based on still another extension to classical Petri nets. The proposed model is more general than Timed Petri nets, and easier to analyze than Time Petri nets. Time constants can be assigned to arcs which lead from places to transitions (Fig. 3). The meaning of a time value $\Delta t_i$ is such, that a token must reside in the input place $P_i$ at least through the time $\Delta t_i$ 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.

The Transnet model is unambiguous and removes one limitation of the previous models, which have been indicated in [10], i.e. the inability to specify the cases where the firing time of a transition depends on the timing characteristics of only a subset of its input places. As shown in Fig. 3, the firing time of the transition $T$ can be related separately to the time instants of depositing tokens in places $P_1$ and $P_2$.

The applications of this time model in Transnet are twofold. A direct one is to define timeouts for potentially infinite operations. E.g. the process $P$, part of which is shown in Fig. 4, will wait for a rendezvous with another process $Q$ not longer than through the time $\Delta t$. If the exchange transition *get_value* does not fire within the time $\Delta t$, then the transition *set_default* becomes enabled and the execution of the process $P$ will be continued.

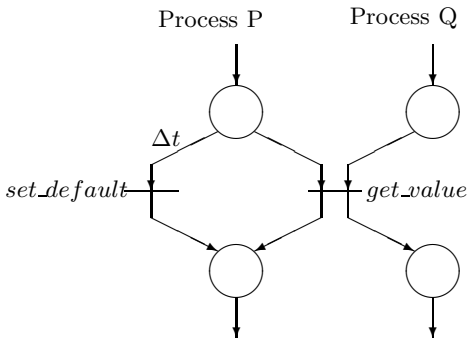


Figure 4: Timeouted operation

The other application is to model the evaluation time of a function associated with a transition within a process net. It can easily be seen that such an evaluation time $\Delta t$ can be modeled by a construction shown in Fig. 5.

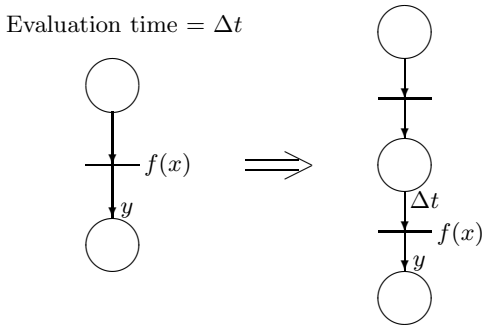


Figure 5: Evaluation time of a function $f(x)$

*D) Hierarchical net structure.*

The graphical representation of Petri nets is readable only for relatively small nets. Therefore a Transnet description of a process net is organized as a sequence of diagrams, which gradually expose more and more details. A high level diagram represents the whole process with complex functions associated with transitions. Each lower level diagram represents a more detailed view of a particular function. The decomposition can be continued down to the level of simple functions, i.e. the intrinsic functions or functions which are directly implemented in a programming language.

To preserve consistency between the levels of abstraction, the semantics of the leveling process must be defined more precisely. A transition from a given diagram which is represented by a subnet in the lower level diagram will be referred to as a *substitution transition*. The substitution transition is connected to the surrounding places by means of input and output arcs. Moreover, the transition is associated with a function, a list of input variables, and a list of output variables.

A subnet which represents a detailed description of the substitution transition is a directed graph which begins with a single transition and ends with a single transition. Input variables of the subnet are the ones that are referenced but not assigned with values within this subnet. Output variables of the subnet are the ones that are associated with the last transition of this subnet.

The translation of a substitution transition and its subnet into an equivalent non-hierarchical net is done in three steps:

1. Delete the substitution transition.
2. Insert a copy of the subnet, connect the input arcs of the substitution transition to the first transition of the subnet, and connect the output arcs of the substitution transition to the last transition of the subnet.
3. Assign the input variables of the substitution transition to the input variables of the subnet and the output variables of the substitution transition to the output variables of the subnet.

Each subnet is a template. From that individual copies can be made to replace various substitution transitions. Each copy of a given subnet has its own set of variables.

Following this functional decomposition, time values are assigned to transitions at each level of the diagrams. The correctness and the consistency between the levels of diagrams can be checked automatically. A more detailed discussion on the verification of the timing requirements will be given in Section 4.

# 3 Specification languages

Transnet offers three languages for specifying the requirements for real-time computer systems

- the functional-like language PAISLey,

- the graphical language of annotated Petri nets, and
- the direct textual language Net.

*A) PAISLey.*

Basic ideas of a process and the inter-process communication which are used in Transnet, have been borrowed from the executable specification language PAISLey [3, 15]. In fact, each PAISLey specification can be translated into the form of an extended Petri net (but not vice versa). The detailed discussion of a translation algorithm can be found in [6].

PAISLey follows the style of functional programming. However, it has been extended to incorporate the notion of a process state. The state space of a process is defined as a set of values composed of simpler sets using set operations: union, cartesian product, and enumeration. The next-state function of a process is defined by a functional expression, which can be composed of functions and combining forms: composition, conditional selection and tuple-formation. It can easily be seen that the composition of functions corresponds to sequential evaluation of these functions, while tuple-formation corresponds to the parallel evaluation of the component functions.

Processes can interact through *exchange functions*, which perform synchronized two-way data transmission. An exchange function is identified by a unique name and a prefix which specifies the details of the inter-process synchronization. Two exchange functions that have been initiated in two different processes can both terminate only if they match with each other, i.e. if they have both the same name and matching prefixes. Before termination the functions exchange arguments, i.e. each of them returns the other's argument as its value.

A PAISLey program which describes the specification in Fig. 2 is given in Fig 6. The specification is composed of two processes: *device* and *monitor*. The state space of the *device* process is a set $DEVICE\_STATE$, and the state space of the *monitor* proces is a set $MONITOR\_STATE$.

$(device[initial\_state], monitor[initial\_value]);$

$initial\_state :\rightarrow DEVICE\_STATE;$
$device : DEVICE\_STATE \rightarrow DEVICE\_STATE;$
$device[x] = proj - 1[(sim\_dev[x], dsensor[x])];$
$sim\_dev : DEVICE\_STATE \rightarrow DEVICE\_STATE;$
$dsensor : DEVICE\_STATE \rightarrow \{`null'\};$
$dsensor[x] = xr\_sensor[x];$

$initial\_value :\rightarrow MONITOR\_STATE;$
$monitor : MONITOR\_STATE \rightarrow MONITOR\_STATE;$
$monitor[v] = proj - 1[(filter[(v, msensor[null])], test[v])];$
$msensor : \{`null'\} \rightarrow DEVICE\_STATE;$
$msensor[null] = x\_sensor[null];$
$filter : MONITOR\_STATE \times DEVICE\_STATE \rightarrow MONITOR\_STATE;$
$test : MONITOR\_STATE \rightarrow \{`null'\};$
$test[v] = /out\_of\_limits[v] : alarm[v], True : null/;$

$DEVICE\_STATE = TEMPERATURE \times LEVEL \times VALVES;$

Figure 6: PAISLey specification of the device and the monitor processes

The next-state function of the process *device* is a function *device* from the set $DEVICE\_STATE$ into the set $DEVICE\_STATE$. The function *device* is composed of a few functions and combining forms. The internal part is a tuple formation of functions *sim_dev* and *dsensor*. The evaluation of the tuple formation is equivalent to parallel computation of both component functions. The result returned is the pair of values computed by the component functions. The external function *proj-1* selects the first element of the pair and returns this element as the value of the function *device*.

The function *dsensor* is defined as an *exchange function* which transfers data between both processes of the specification. The exchange functions are distinguished from other functions by a prefix $x$, $xr$, or $xm$. An $x$-type function can match with any other exchange function with the same name (e.g. *x-sensor* with *xr-sensor*). When initiated this function waits for a match. An $xr$-type function can also match

with any other exchange function with the same name, but it does not wait for a match. If no matching function has been initiated, this function terminates immediately and returns its own argument. An *xm*-type function is similar to *x*, except that it cannot match with another *xm*-type function.

In the example program the exchange functions *xr-sensor* and *x-sensor* match with each other. The *xr-sensor* function does not wait for the partner. It can be noted, that useful data are transferred unidirectionally, from the *device* to the *monitor* process.

The advantage of PAISLey is that the language is a powerful tool for expressing functional requirements for the processing of data. The specification is formal, executable, and free from unnecessary sequencing constraints.

The disadvantages are twofold. The first one is that the language has no potential to express the requirements for time-dependent aspects of the system's behavior. The only mechanism which is offered for this purpose are meaningful comments which can be used by a simulator. The other disadvantage is that larger programs are hardly readable.

The main application area intended for PAISLey within the Transnet approach is for specifying particular functions within a process net and for defining data types. The general structure of a specification net can be more readably defined using a graphical language of annotated Petri nets.

*B) Graphical language of annotated Petri nets.*

The graphical language used in Transnet is composed of elements which are typical for Petri net graphs. These are circles for places, bars for transitions and lines with arrowheads for arcs. The additional information related to functions, conditions, variables and time values is introduced through textual windows associated with particular elements.

The use of the graphical language is based on an interactive editor which supports drawing a net. Each time a new element is created, the editor opens a window on the screen and prompts the user to enter the list of data items associated with this element. The drawings produced by the net editor are similar in shape to those shown in Fig. 1 and 2.

Petri nets tend to get messy when there are processes which cooperate with several different partners. Therefore the graphical language can most efficiently be used for drawing a preliminary draft of the specification under development, and for drawing separate diagrams of particular processes or sub-nets.

To improve the readability of the diagrams, particular nets can be structured in a hierarchy of diagrams. A single transition of an upper layer diagram can be refined to the form of a module, i.e. a sub-net in a lower layer diagram. The rules for hierarchical net construction, described in Section 2-D, are well defined and the net analyzer tool can check for consistency between the diagrams.

The non-functional timing requirements can be defined by means of time values associated with arcs and transitions. A constant associated with an arc represents a timeout value as described in Section 2-C. A time value associated with a transition represents the evaluation time of a function associated with the transition. To model some amount of uncertainty, the value can be defined as a random variable. The range of the variable is defined by a pair of constants. During the net simulation the actual evaluation time is defined by drawing a value from within the range of the random variable.

*C) Textual language Net.*

The textual language is the lowest level language which can be considered as a "net assembler". A program consists of segments, each of which describes a process type, a process (usually being an instance of a process type), a sub-net in a hierarchical net description (this is called a module), or a data type. Particular instructions of the net-type segments (all but data segments) describe the elements of an extended Petri net, i.e. places, transitions, arcs, functions associated with transitions and conditions associated with arcs, variables, and time constants. Data types are defined in a set notation, similarly as in PAISLey.

The textual language Net is extremely primitive. It was designed not as a tool for constructing specifications, but for making local corrections to previously developed nets. Moreover, is was found a convenient tool for defining and refining data types.

*D) Compilation.*

Regardless of the language used originally for developing a specification, the description is always converted to an internal representation which is an extended Petri net. The description of a specification is kept in a few data files. Particular files contain the library of process types, the library of module types, the library of data types, and the description of the set of processes which comprise the specification under development.

It is worth noting that only the description of the set of processes is directly tied up to a particular system. The items of library files can be re-used in the specifications of various system.

The compilers of particular languages have the potential for incremental compilation. This means, that even an incomplete program can be converted into a piece of internal representation and added to the appropriate files. In this way particular parts of a specification can be developed using different languages. The consistency between the files can be controlled by an appropriate net analyzer.

# 4 Features of the representation

*A) Continuous operation.*

Unlike many algorithmic specifications which have well defined *start* and *stop* points, Transnet processes are cyclic and can run continually. The exchange transitions provide for synchronous as well as real-time inter-process interaction. These concepts have been inherited from PAISLey. The new features in Transnet are the opportunities for formal specification of time-dependent behavior, and for formal analysis of the safety-related properties of deadlock-freedom, starvation-freedom, and state reachability. Both features will be discussed in detail later in this section.

*B) Simultaneity.*

Transnet processes are executed in parallel and can interact with concurrently running objects from the environment. Following the object-oriented approach, each process encapsulates local data structures and the operations on data. To prevent indeterministic side-effects during simultaneous execution, no common variables are allowed.

The other form of simultaneity allowed in Transnet is the concurrent execution of parallel branches within a process net. This feature has been inherited from PAISLey, but it seems a less important factor. In most practical cases parallel branches can be serialized automatically [6].

*C) Static structures and boundedness.*

It can be noticed that the structure of a specification net which can be described in Transnet is static. This means that the processes as well as the links between them can neither be created nor deleted dynamically. However, this cannot be considered a serious drawback, as the method is aimed at the specification of real-time embedded systems, which are inherently static.

An important consequence of the static perspective of Transnet is boundedness. All the components of a Transnet specification are static and bounded. This applies to the structure of the specification net as well as to the structure of data. This guarantees that a system specified in Transnet can consume only bounded amount of resources.

As it was mentioned earlier, no local iteration within a process net is allowed. This seems unusual and should be discussed in detail. There are, in general, two different uses of an iterative structure. The first one is for traversing an (unbounded) data structure. The other one is for implementing a repetitive algorithm which modifies a value of a variable in a loop.

Relating to the first case, all data structures in Transnet are bounded. The processing of consecutive items of a bounded structure composed of $n$ items can be defined by means of a sub-net with $n$ parallel branches, each branch dealing with one item. Suitable notation for compact description of such a sub-net is provided by the PAISLey *index notation* [16]. Similar structure is available in the textual language Net.

The second case, i.e. an iterative computational algorithm, cannot be described in Transnet. This is a direct consequence of the single assignment rule. According to this rule, no repetitive computation

which modifies a value of a variable in a loop is allowed. If such an algorithm is needed, it should be encapsulated within a function. This is why Transnet cannot be applied in the detailed design phase.

Unfortunately, boundedness of data structures and the lack of unbounded iteration is not sufficient for guaranteeing that a Transnet process can consume only bounded amount of time. As shown in Fig. 7 an equivalent of an unbounded *while* loop can easily be constructed (the same is true for PAISLey). In this example, a process P sends an initial value of $x$ to another process Q and starts waiting for a response. The process Q repeats the instructions:

$X := x$

$x :=$ *function(X)*

as long as the condition $B$ evaluates to 'true'. When the condition $B(x)$ turns to 'false', the process Q responds by sending the latest value of $x$. The initial value of $X$ in the process Q is such that $\neg B(X)$ holds.

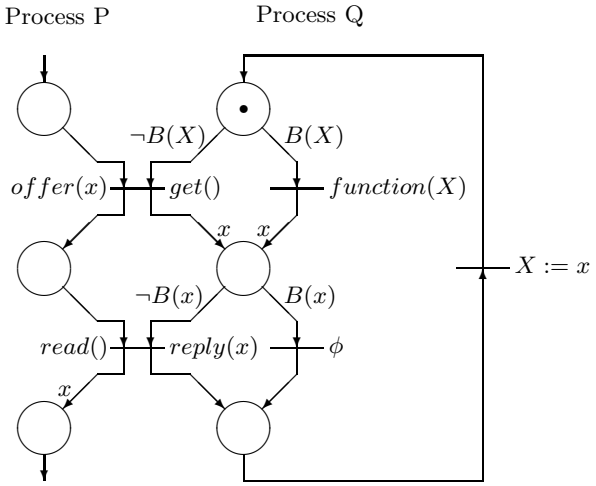


Figure 7: An equivalent of the loop: *while (B) do x=function(x)*

*D) Timeliness.*

Transnet offers two language mechanisms for specifying timing constraints on the duration of elementary operations:

- time limits for the evaluation of functions associated with transitions,
- timeout values assigned to particular branches in a conditional selection control structure.

Consider the first option. The construction of a specification net is layered. This means that a transition associated with a compound function in a higher layer, may be refined in a lower layer into a sub-net composed of transitions associated with simpler functions. In each layer a lower and an upper bound of the evaluation time can be defined for each transition. These parameters can be interpreted in two different ways in the layered net definition:

- as an estimation of the evaluation time of the function associated with a transition, and
- as constraints imposed on the total execution time of a sub-net representing this transition in a lower layer.

The consistency of the time limits defined in different layers guarantees feasibility of the constraints imposed for particular operations. This feature can be checked automatically.

The second mechanism involves timeouts for potentially infinite operations, such as the execution of exchange transitions. Consider once more the unbounded *while* loop. The net shown in Fig. 8 differs from the one in Fig. 7 in that the last transition of the process P has been substituted by a conditional selection with a timeout. The result is that if the exchange transition did not fire within the time period $\Delta t$, than the timed transition becomes enabled thus allowing the process P to continue.

Both mechanisms enable us to specify bounds on the duration of all actions defined in a specification net. Moreover, no unbounded control structures, i.e. iteration and recursion, are provided. This

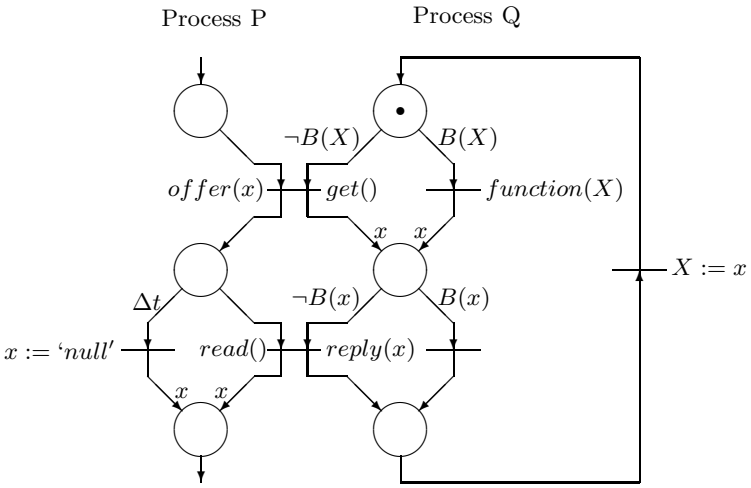


Figure 8: A time-outed *while* loop

guarantees bounded consumption of time by all process steps, and implies full predictability of the time-dependent behavior of the specified system.

The mechanisms for describing time-dependent behavior are not completely independent. As indicated in Section 2-C the method of assigning timeout values to arcs is more general as it can model the other one. Transnet languages allow both mechanisms to be used, as the mechanisms are conceptually different and describe different real-life requirements.

*E) Analyzability.*

The extensions to Petri nets, particularly those which constrain the indeterminism of alternative selection, limit the possibilities of the net analysis. Two such extensions have been introduced in Transnet:

- Predicates assigned to arcs, which violate the distinction between the control and the data processing structures, and make the enumerative analysis methods inapplicable.
- Timeout values, which make Petri nets computationally equivalent to Turing machines.

To overcome these problems, three different levels of the specification net analysis are offered in Transnet. At the first level the basic Petri net is defined as a net without predicates and time which is derived from each Transnet specification. The following structural properties of the basic Petri net are examined using the standard construction of the reachability tree [7]:

- correctness of the net definition,
- safeness and conservation, and
- reachability and liveness.

At the second level the problem of predicates which correspond to conditional selection is handled. Due to the use of variables, the state space of the specification net can, in general, be infinite (or very large). However, the values of variables can influence the reachability tree only through the boolean conditions which control the firing of transitions connected to the same input place. To analyse the behavior of a net with boolean conditions, the reachability tree which has been constructed in the previous step is modified. The arcs of the reachability tree are associated with labels, each of which consists of a transition identifier and a condition which must be fulfilled in the previous marking to fire the transition.

The full reachability tree represents the indeterministic Petri net in which all boolean conditions can eventually be fulfilled. To simulate various scenarios of the specification net execution, some conditions can be assumed 'false' by removing the respective arcs from the reachability tree. A reduced tree represents an execution with a particular combination of 'true' and 'false' conditions. Because the number of boolean conditions is finite (and not very large), the analysis can be exhaustive. The existence of a deadlock situation in a reduced reachability tree indicates that this particular combination of 'true' and 'false' conditions can be dangerous. The detailed analysis of each of these conditions must be performed

manually. An example construction of a reduced reachability tree can be found in [17].

At the third level the Petri net with timeouts is analyzed. The technique used for this purpose is also based on a modified reachability tree. The nodes of a classical reachability tree correspond to 'states' of a Petri net, i.e. to subsequent markings. Adding timeouts changes the notion of the net 'state', which can no longer be described as a marking.

Consider an arc which leads from a place $p$ to a transition $t$, and is associated with a timeout value $\Delta t$. The arc will be called:

- *closed* if no token resides in the place $p$,
- *active* if the place $p$ is marked with a token,
- *open* if the place $p$ has continued to be marked with a token at least through the time $\Delta t$.

A state of a Petri net with timeouts can be defined as a pair $S =< M, E >$ consisting of a marking $M$ and a timeout vector $E$. A marking $M$ describes the distribution of tokens among places, as usually in Petri nets. A vector $E$ describes the current (residual) values of timeouts associated with particular active arcs. The size of the vector $E$ can vary in time. Each entry describes one active arc and contains the length of the time interval remaining for opening the arc.

The execution of a Petri net with timeouts is controlled by the net marking $M$ and the values of time intervals in $E$. The net executes by shifting time intervals and firing transitions. A transition $t$ is enabled in a state $S =< M, E >$ if

- it is enabled in the sense of classical Petri nets, *and*
- has all input arcs open (i.e. the residual values of timeouts associated with these arcs are equal to zero).

Let $S =< M, E >$ be a state of a Petri net with timeouts. To define the rules for computing the next state $S' =< M', E' >$ two cases have to be considered.

1. Assume that no transitions are enabled in $S$. Let $\Delta t$ be the shortest non-zero time interval in $E$. The next state $S'$ is computed from $S$ as follows.
   a) $M' = M$,
   b) $E'$ has the same entries as $E$ and for each entry $e$

$$E'(e) = \begin{cases} E(e) - \Delta t & \text{if } E(e) \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

2. Assume that a transition $t$ is enabled in $S$. The next state $S'$ reached from $S$ by firing $t$ can be computed as follows.
   a) $M'$ is obtained by removing tokens from the input places of the transition $t$ and depositing tokens in the output places of the transition $t$.
   b) $E'$ is obtained by removing from $E$ the entries that are related to the arcs which originate in the input places of the transition $t$, and introducing the time values associated with the new arcs activated.

The construction of the modified reachability tree of a Petri net with timeouts is analogous to the construction of the classical reachability tree. The nodes of the modified reachability tree correspond to the above defined 'states', i.e. pairs composed of a marking and a timeout vector. An examination of the modified reachability tree can answer the questions of liveness and transition liveness. Moreover, the *response time*, i.e. the time interval between particular net states, can be evaluated.

*F) Prototyping.*

Any Transnet specification is executable and constitutes a prototype of the system under development. The prototype can be executed by an interpreter tool. The net execution involves firing transitions and evaluating functions according to the rules defined in the previous Section 4-E. The result of the net execution is a sequence of the consecutive net markings and a sequence of the valuations of variables.

The problem is that the volume of data makes the analysis very difficult. To cope with this problem, the net interpreter exploits the concept of *validation points*, similar to the one introduced in SREM [18]. A validation point is identified by a particular net marking or submarking, which defines a strict point in the net execution. For each validation point a list of variables is defined. When a validation point is reached during the net execution, time and the values of indicated variables are recorded for further analysis.

The net interpreter provides a possibility for early validation of the specified system. There is no demand for completeness of a specification prior to the net execution, as the interpreter tool can effectively deal with incompleteness. Once an undefined function has been encountered, two options are available: an operator can be asked to enter the result or a default value can be substituted. Undefined data types are equivalent to integer.

# 5 Conclusion

The features of the Transnet method match with the most important characteristics of real-time systems. The method offers the potential for the software specification, formal analysis and early validation. Neither of the existing methods can offer such a wide range of capabilities.

An additional feature of the Transnet system, not described in this paper is the possibility of the automatic transformation of the specification net structure. This can help the designer to compare different design decisions and to compromise functionality and performance. A set of transformations which have been proved is briefly described in [6].

Formal foundations which underlie the Transnet method have been developed and described in [5, 6, 17]. Basic tools which support the method have been implemented and tested in a limited, experimental version. More elaborate version is currently under development.

# Acknowledgement

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

# References

[1] *IEEE Standard Computer Dictionary*, IEEE, New York, 1990.

[2] P. T. Ward, "The transformation schema: An extension of the data flow diagram to represent control and timing", *IEEE Trans. Software Eng.*, vol. SE-12, pp. 198–210, Feb. 1986.

[3] P. Zave and W. Shell, "Salient features of an executable specification language and its environment", *IEEE Trans. Software Eng.*, vol. SE-12, pp. 312–325, Feb. 1986.

[4] D. Harel et al., "Statemate: A working environment for the development of complex reactive systems", *IEEE Trans. Software Eng.*, vol. 16, pp. 403–414, April 1990.

[5] K. Sacha, "Transnet: A method for transformational development of embedded software", *Microprocessing and Microprogramming*, 32, pp. 617–624, 1991.

[6] K. Sacha, "Transformational implementation of PAISLey specifications using Petri nets", *Software Eng. Journal*, vol. 7, pp. 191–204, 1992.

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

[8] K. Jensen, "Coloured Petri nets: A high level language for system design and analysis", in G. Rozenberg (ed): *Advances in Petri nets 1990*, LNCS 483, Springer-Verlag, pp. 342–416, 1991.

[9] H. J. Genrich, "Predicate/Transition nets", in W. Brauer, W. Reisig and G. Rozenberg (eds): *Advances in Petri nets 1986*, LNCS 254, Springer-Verlag, pp. 207–247, 1987.

[10] C. Ghezzi et al., "A unified high-level Petri net formalism for time-critical systems", *IEEE Trans. Software Eng.*, vol. 17, pp. 160–172, Feb. 1991.

[11] C. Ramchandani, "Analysis of asynchronous concurrent systems by timed Petri nets", *Massachusets Inst. Technol. Tech. Rep. 120*, Feb. 1974.

[12] P. Merlin and D. J. Faber, "Recoverability of communication protocols", *IEEE Trans. Commun.*, vol. COM-24, Sept. 1976.

[13] J. E. Coolahan, N. Roussopoulos, "Timing requirements for time-driven systems using augmented Petri nets", *IEEE Trans. Software Eng.*, vol. SE-9, pp. 603-616, Sept. 1983.

[14] B. Berthomieu and M. Diaz, "Modeling and verification of time dependent systems using time Petri nets", *IEEE Trans. Software Eng.*, vol. 17, pp. 259–273, March 1991.

[15] P. Zave, "An Insider's Evaluation of PAISLey", *IEEE Trans. Software Eng.*, vol. 17, pp. 212–225, March 1991.

[16] P.Zave, "An Operational Approach to Requirements Specification for Embedded Systems", *IEEE Trans. Software Eng.*, vol. 8, pp. 250–269, May 1982.

[17] K. Sacha, "Real-Time Specification Using Petri Nets", submitted for EUROMICRO'93.

[18] M. W. Alford, "A requirements engineering methodology for real-time processing requirements", *IEEE Trans. Software Eng.*, vol. SE-3, pp. 60-69, Jan. 1977.