Exception Handling with Petri Net for Digital Systems

Wagner Luiz Alves de Oliveira
DSIF/FEEC/Unicamp, Brazil
wagner@dsif.fee.unicamp.br

Norian Marranghello
DCCE/Ibilce/Unesp, Brazil
norian@dcce.ibilce.unesp.br

Furio Damiani
DSIF/FEEC/Unicamp, Brazil
furio@dsif.fee.unicamp.br

Abstract

Petri net is a language for system modeling. Petri net extensions achieve a wide range of applications, such as: communication protocols, programming languages, computer architectures, distributed systems, industrial process control. These extensions allow the description of parallel systems, which can be simulated at the atemporal, logical, and functional levels with reasonable simplicity. Such extensions have some limitations when describing digital systems. This paper presents an extension to overcome the existing limitations. Emphasis is provided about PNDS mechanisms aimed at exception handling.

1. Introduction

Digital system design has reached a high degree of complexity that prevents its realization without CAD tools. Several specification languages were developed to capture as many hardware characteristics as possible. In this context, Petri nets have shown to be an effective modeling methodology for discrete event systems, due to the existence of a set of Petri net related techniques for dynamic and structural analysis. These techniques allow the formal validation of important properties of the model, such as: liveness (deadlock-freeness), safeness, boundedness. From the study about the use of Petri net extensions in digital system design [1], we could verify that none of them meets all desirable characteristics of a good modeling language. We propose a new Petri net extension to overcome these limitations. This extension is described in section two; section three shows its exception handling mechanisms; section four presents conclusion as well as studies in development.

2. A Petri net for digital system modeling

Further dynamic and structural analysis techniques mentioned in introduction, Petri net extensions present most of the features needed for a good digital system modeling methodology as stated by Gajski et al. [2]: concurrency, state transitions, timing, programming constructs, hierarchy, behavioral completion, process synchronization, communication, and exception handling. Place Chart Nets (PCN [3]) and Petri Nets for Embedded Systems (PNES [4]) are Petri net extensions that present most of these features. PCN is a low-level extension, hierarchical, with preemptive mechanisms, but it lacks timing and programming constructs. PNES has hierarchy, programming constructs and preemptive mechanisms (it lacks interruptive mechanisms), but cannot be classified as high-level Petri net due to its boolean tokens, each of which can either be associated to holding the behavior of a circuit or not. Finally, both time for communication and time for computation have not been simultaneously treated in Petri net extensions. In this context, the authors have developed an extension called Petri Net for Digital Systems (PNDS).

The main features of PNDS can be so stated:

a) High-level description. Three kinds of places allow to model system components (storage and functional) as well as their local states. Tokens can represent the contents of a cell (at a storage unit), the status (data value) of a system component, or a data item related to an ordered list (data sample) to be processed by a behavior associated to a functional component. Transitions allow both to change modes (through removing/ladding tokens from/to places assigned to local states or behaviors) and to perform read/write operations on data values (tokens) belonging to places assigned to storage elements. Special branches are adopted according to the places to which they are connected. Branch related expressions define what or how many tokens must be inserted, removed, read, or written on the assigned places;

b) Hierarchy. Models deal with multiple process in the same structure. There exists one hierarchical instance (copy of behavior) for each complete data sample (ordered list of tokens) present in a functional place;

c) Programming constructs. Place related constructs model system inherent data manipulation, while transition related constructs represent control flow manipulation;

d) Full exception handling. Preemptive mechanisms allow to abort behaviors/local states assigned to a place,
while interruptive mechanisms allow to suspend all instances of a behavior assigned to a functional place; e) Complete time representation. **Timed** and guarded transitions allow the description of system events that may occur in a scheduled or random way, making possible to model both communication time mechanisms and computational time mechanisms.

The following sections survey the operational semantics of places, transitions, and branches.

### 2.1. Places

Three kinds of places are described in the following.

#### 2.1.1. Type I: local state

Place used for indicating that one or more system entities (tokens) of a specific kind (processes, storage, communication, and functional units) are at a same local state. Two sets define a type-I place: Domain ($D_s$) and Repository ($R_s$). $D_s$ is a non-empty finite subset belonging to a token_type set, restricting the tokens that can occupy the place. $R_s$ ($R_s \subseteq D_s$) indicates what tokens are present in a given moment of the net. Two operations are allowed by involving a place $p$ of type I:

- a) **Token insertion.** It is an operation involving a simple output branch connecting a transition $t$ to $p$. This branch is labeled with an insertion function which denotes a token set $T$ ($T \subseteq D_s$) to be put into $R_s(p)$ when $t$ fires. The intersection between $T$ and the set of tokens enclosed in $p$ should be empty. If insertion function has been omitted, $T$ will be the union set of all token sets to be removed from each place connected to $t$ through input branches;
- b) **Token remotion.** It is an operation involving a simple or conjunctive input branch connecting $p$ to any transition. A simple input branch allows to remove a token set from $p$ according to a remotion expression $\langle \text{min, max, order} \rangle$, where min and max denote the minimal and maximal number of tokens to be removed from the place, and order {Chronological, Random} denotes whether tokens will be removed in a non-deterministic or deterministic way. A conjunctive input branch allows the transition to randomly remove the same token set from each and every input place connected to it. It has a label $\langle \text{min, max} \rangle$.

The PNDS model shown in fig. 1 has one transition (Process) and two type-I places (Waiting and Processing). Such places are divided into two sections: in the upper part, it is described both place identify and domain; in the lower part, the place repository is described by an ordered list enclosed in square brackets. In fig. 1.a, tokens are removed in a chronological way: the firing of Process has caused both the remotion of the two oldest tokens from Waiting and the insertion of such tokens into Processing, what represents the running of the two oldest processes ($p3$ and $p1$). In fig. 1.b, the simple input branch denotes the remotion of tokens in random order, reason for which any two processes have been removed from Waiting.

![Figure 1. Occurrence of Process](image)

#### 2.1.2. Type II: storage

Place used for describing system storage entities (registers, memories) on which read/write operations on data (tokens) can be carried out. Four components define a type-II place: Domain ($D_m$), Address Space (A), Write Latency ($L_w$), and Read Latency ($L_r$). $D_m$ is analogous to $D_s$ (see type-I place). A is a finite set of storage cell addresses (there is only one token for each address). $L_w$ ($L_r$) is a natural number representing how many time units are needed for a token to be updated at (recovered from) the place after a write (read) operation.

Two operations are allowed on a type-II place $p$:

- a) **Write.** It is an operation carried out with help of a writing output branch connecting any transition to $p$. This branch has a writing expression $\langle t_1, a_1 \rangle \cdots \langle t_n, a_n \rangle$, where $t_i \ (t_i \in D_m)$ represent token names that should be stored into place storage cells (addresses) $a_i \ (i=1, \ldots, n)$;
- b) **Read.** It is an operation carried out with the presence of a reading input branch connecting $p$ to any transition. This branch is labeled with an expression that describes what tokens must receive data values to be read from specific addresses of the place. Reading expression has the same format as writing expression.

In fig. 2, it is shown the use of storage places, which are depicted by two concentric ellipses, divided into two parts: in the upper part, it is described the identify, domain {N-bit-hexa}), read and write latencies of the place; in the lower part, a table describes the address space (table header) and the contents of each address (table body). When Transfer fires, the contents of a storage cell in Reg. File I (denoted by $addr$) is read and stored in a token $t$ whose value will be written in the address $addr$ of Reg. File II. Then, the transition related operations (lower section of Transfer) will be carried out. Such operations evaluate the next address to be accessed. The default type of a net variable (as $addr$) is integer and its default value is 0. In spite of addresses being denoted by names, they belong to an enumerated type.
2.1.3. Type III: Functional. Place used for indicating that one or more data samples (token lists) are being processed by one or more functional units of the same kind (described by a behavior). Six components define a type-III place: Input Domain \((D_I)\), Input Repository \((R_I)\), maximal number of samples \((\text{max}_s)\), Output Domain \((D_O)\), Output Repository \((R_O)\), and Behavior \((B)\). \(D_I\) and \(D_O\) are ordered lists. \(D_I\) denotes the types of the tokens required to run any instance (copy) of the place related behavior \(B\), while \(D_O\) denotes the types of those tokens that will store results provided by any instance of \(B\). The amount of data samples allowed to be enclosed in the place at the same time is limited to \(\text{max}_s\) (a non-zero natural number). After an input data sample is formed into \(R_I\), it will be processed by one instance of \(B\), and the resulting data sample will be stored into \(R_O\). \(R_I\) is a structure to store incoming samples. It can be thought of as \(n\) FIFO buffers of length \(\text{max}_s\), where \(n = \#D_I\). Each buffer is assigned to only one entry defined in \(D_I\). So, there can be two or more buffers of the same type. When a token is inserted in the place, it is stored in the less occupied buffer of its type. If there exist two or more buffers with equal number of cells filled then the token will occupy the first one according to \(D_I\) ordering. Analogously, \(R_O\) is a structure to store output samples resulting from the processing of input samples. \(R_O\) can be thought of as \(m\) FIFO buffers of length \(\text{max}_s\), where \(m = \#D_O\). \(B\) denotes the place related functionality: one instance of \(B\) is carried out whenever one sample into \(R_I\) has just completed. \(B\) must be described in one of two ways: Transformational PNDS (T-PNDS) or Routine.

A T-PNDS is a PNDS that requires the presence of both a set formed only by input places (without transitions that insert tokens into them) and a set formed only by output places (without transitions that consume tokens from them). To each element of such an only-input (only-output) set, it is associated one token which can get into (leave) the behavior: there should exist one input (output) place of type-I for each token in the input (output) sample in order to store an input argument for the behavior (a result produced by the behavior). One T-PNDS instance will remain active until each only-output place receives a token. At this moment, tokens are removed from these only-output places and corresponding tokens are created in output repository \(R_O\), being the instance deactivated.

A Routine is a sequence of programming statements that describes a computation. It can use data values (tokens) belonging to the data sample acquired from \(R_I\), transforming them into data values of an output sample, which is stored in \(R_O\). One routine instance is created for each completed input sample present into \(R_I\). The instance will be carried out until the last statement is found.

Two operations involving type-III places are allowed:

a) Token Insertion. It is an operation involving a supplier output branch that connects any transition to a type-III place. This branch has an insertion function (like that for simple output branch) which denotes a token set to be put into \(R_I\) when the transition fires, representing input arguments for one or more instances of \(B\). \(R_I\) can be thought of as a matrix \(M_I\) with \(\text{max}_s\) rows and \(#D_I\) columns. \(M_I\) is filled bottom up. It will be activated one instance of \(B\) whenever a row \(l\) of \(M_I\) is completed. Then, such a sample is removed from \(R_I\) but the vacant row \(l\) must not be occupied until a generated output sample is removed from place \((l\text{ is unavailable})\). Any other complete input sample will activate another instance of \(B\). After an instance is finished, an output sample will be stored into \(R_O\). \(R_O\) can be thought of as a matrix \(M_O\) with \(\text{max}_s\) rows and \(#D_O\) columns. All elements of the row \(l\) of \(M_O\) will be filled at once after the related instance is carried out;

b) Token Remotion. It is an operation carried out with help of a collector input branch connecting the place to any transition. Such a branch allows the transition to remove the oldest output sample (the bottom row of \(M_O\)) from \(R_O\) when it fires. This remotion causes both an unavailable \(M_O\) row to become available and every token belonging to either \(M_I\) or \(M_O\) to be shift down one position, making possible the insertion of a new input sample into \(R_I\). A collector input branch has an expression \(<t_1, ..., t_m>\), where \(t_1, ..., t_m\) are tokens belonging to types implicitly assigned by the output domain \(D_O\) of the place.

A functional place describes a behavior with well defined input and output parameters. Arguments to input parameters can be supplied at any time either little by little or at once. Whenever there exists the number of needed arguments to run the behavior, it takes place, and the produced results are stored in the output repository. An embedded concept in a functional place is the ability to parameterize behaviors, which can be concurrently carried out. A complex behavior can be described by a
subnet (T-PNDS) whose behaviors can also be described by subnets and so on. Concurrent behaviors must be described through several functional places, while a sequential behavior can be represented by a routine assigned to a functional place. All instances of a behavior B can be aborted, suspended and resumed by means of firing of transitions connected to the place through branches tuned to exception handling (see next section).

A functional place is represented by three concentric ellipses divided into three parts (fig. 3). The upper part encloses the place name, input domain (table header, listing the type of input parameters), and input repository (table body); the table entries infer the maximal number of samples (3, in the figure). The place related behavior is described in the middle part of the place. In fig. 3, the behavior is given by a routine (Comparator) due to its simplicity. In the lower part of the place, it is represented both the output domain and the output repository.

Figure 3 shows a 2-integer comparator. Two integers are acquired from a storage component (Arguments) to be processed by a functional component (Comparator). The greater number is chosen and then it is inserted into another storage component (Results). Insert 1 argument and Insert 2 arguments provide tokens (arguments) for Comparator. Their enabling depends on the net variable control: when control is equal to either 0 or 1, Insert 1 argument will be enabled; otherwise, Insert 2 arguments will be enabled. Transfer Result is responsible for storing the result of processing into Results. The first produced result will be stored into the address R1, the second one into R2, and the third one into R3. This process will be repeated while cont is TRUE.

2.2. Transitions

Transitions represent possible events in the modeled system. A transition is enabled when the associated event may occur at the considered moment due to certain rules and conditions being true. Firing of an enabled transition corresponds to the occurrence of the related event. To each transition there may be associated operations (to be effected on the net variables), a guard, and a firing time interval. The enabling rule of a PNDS transition depends on its guard and related branches. To each kind of branch is defined an enabling rule, as described in the following:

a) Reading input branch. Rule composed by two conditions. The first one is the branch reading expression must be in compliance with the related place p (addresses belonging to A(p)). The second condition refers to latency. It must have elapsed a minimal timing interval T since the prior operation O involving p, so defined:

if O=reading operation ⇒ T = L_R(p) × n, otherwise
if O=writing operation ⇒ T = L_W(p) × n,

where n is the number of tokens involved in operation O.

b) Writing output branch. Rule composed by two conditions. The first one is that its writing expression must be compatible with the place definitions (D_M and A). The second condition is related to read/write latency such as that defined to reading branch.

c) Simple and conjunctive input branches. Each related input place must have at least one set of tokens whose cardinality matches the remotion expression.

d) Simple output branch. The connected place must have no token belonging to token set T, resulting from the evaluation of the branch insertion function, besides being T in compliance with the place domain.

e) Supplier output branch. The cardinality of token set T, resulting from the evaluation of branch insertion function, must be less or equal to the amount of free cells in the input repository of the connected place, besides being T in compliance with the place input domain.

f) Collector input branch. Rule composed by two conditions. The first one is that there must exist at least one token sample in the corresponding place output repository. The second condition is that the amount of tokens described by the branch remotion expression must be equal to the cardinality of place output domain.

Suspensive, preemptive, and restore branches are tuned to exception handling. A suspensive branch allows a transition t to temporarily suspend all active instances of the behavior related to a type-III place. When a T-PNDS describes the place behavior, the suspension of instances
will occur due to impossibility of enabling the T-PNDS transitions. When a routine describes the behavior, each of its instances will finish the current statement and then halt. A preemptive branch allows to remove all tokens from the connected place (type I or III), independently of any conditions. Such a fact has distinct means: for type-I places it means that local states maintained by distinct system components will be extinguished at once; for type-III places it means that both computation instances related to the place will be aborted and any existing input or output samples in the place will be extinguished at once. A preemptive branch also allows to reset storage cells of type-II places. Finally, a restore branch allows a transition \( t \) to resume all instances of the corresponding functional place related behavior, previously suspended by the firing of another transition connected to the same place through a suspensive input branch.

A preemptive input branch has distinct enabling rules. When it is connected to a type-I/type-III place, the rule is that the place has at least one token. For type-II places, it does not exert any constraint in the enabling of the related transition. The enabling rule of a suspensive input branch is that the connected type-III place has at least one active instance. A restore output branch does not exert any constraint in the enabling of the related transition.

The enabling rule of a PNDS transition is composed by two conditions: its guard must evaluate to TRUE; and the enabling rule of each related branch must be satisfied.

To fire, a transition must be enabled and satisfy its firing rule. PNDS adopt a time mechanism similar to the one found in Merlin’s Time Petri Nets [5]: a firing time interval \([\text{min}, \text{max}]\) assigned to each transition, indicating the minimum and maximum delays for the actual firing of the transition after its enabling. Extended in PNDS, this mechanism is useful to model communication time. Transitions without such an interval denote occasional events in the modeled system which may occur or not. On the contrary, a transition with firing time interval must fire, unless it becomes unavailable due to the firing of any concurrent transition at the period comprehended by the interval. The interval \([0,0]\) denotes immediate firing. The firing rule for an enabled transition is that its lower time limit must be lower than or equal to the lowest among the upper time limits of all enabled transitions.

The simulation of a PNDS is performed by changing classes, each one describing a possible global state of the modeled system. A class \( C_i \) (\( i \in \mathbb{N} \)) is a triple \((M_i, T_i, V_i)\), where \( M_i \) is a net marking, \( T_i \) is a time domain, and \( V_i \) is a set of values for the net variables. A reachability graph describes the relationship among the classes.

The firing of a transition \( t \) effects the changing of the present class \( C_i \) to a new class \( C_{i+1} \), involving five steps: deactivation of behavior instances and exclusion of all tokens from repositories related to type-III places connected to \( t \) through preemptive branches; suspension (resuming) of behavior instances relating to type-III places connected to \( t \) through suspensive branches (restore branches); reading (writing) of tokens involving type-II input (output) places, and exclusion (addition) of tokens involving type-I and III input (output) places (relating to \( t \)); setting of the transition operations results; and updating of the net times.

The properties of a PNDS model verify the validation or not of the properties of actual system. We have already defined the following properties for PNDS: boundedness (that enables the actual implementation of the modeled system), reinitiability (the system will return to initial state after finishing one or more tasks), and liveness (that asserts a deadlock-free system). After achieving the proper results, the net model of the system should be converted to a low-level Petri net in order to allow a direct mapping of the net onto a circuit description.

### 3. PNDS and exception handling

In a modeled system, a particular event (named exception) can require that a computation mode (local state/behavior) is interrupted or aborted immediately, transferring the net control flow for other computation mode. There are two kinds of exceptions: preemption (the current mode is aborted) and interrupt (the current mode is temporarily suspended). As it was mentioned in section 2, the only Petri net extensions dealing exceptions are PCN and PNES. In PNES, a supermode is defined as a place corresponding to a subnet representative of operations that can be aborted by the firing of a special transition called exception. PCN has similar concept. Thus, both extensions lack interruptive mechanisms. For the sake of preemption, PNDS makes use of preemptive input branches. Such branches are used in the net of fig. 7 (thick dashed arcs), which models an instruction pipeline with four stages (Fetch, Decode, Exec, Store). Part of the net models a timed variation on a clock signal; another part represents a five-module clock divider (place 5-Module). If a processing error happens at any pipeline stage (functional places), the corresponding transition (StartTask1, ..., StartTask4) will be unenabled as soon as the transition Clock 5 fires. This fact will cause one or more places related to stage synchronization (Sinc1, ..., Sinc4) to acquire one token, which enables the transition Pipeline Error. If Pipeline Error remains enabled for 1 time unit, it will fire, removing all tokens enclosed at its input places. Another part of the model should give an adequate handling for such a situation. So, pipeline errors will not interfere in the clock mechanism, avoiding to propagate errors to other parts of the net.
PNDS uses suspensive input branches and restore output branches to represent interrupt (thin dashed arcs in fig. 8). In fig. 8, the transition **Suspend** allows to temporally suspend all active instances of Routine Fiat Lux. On the other hand, **Resume** has a contrary effect: it allows to resume running all previously suspended instances of Routine Fiat Lux. It can be observed in fig. 8 the existence of two running (active) instances, since the two last entries of the table representing the place input repository are unavailable. Suppose the net variable **interrupt** be 3; if such a value changes to 4, the transition **Suspend** will be enabled, occurring immediately (due to its interval [0,0]), which will cause such instances to run their current statement and then to be suspended. After instances being suspended, only the firing of **Resume** can reactivate them. For that firing, it is necessary both to assign the value 0 to net variable **interrupt** (which will enable Resume) and to elapse at least 100 time units without any update on **interrupt** (which will unenable **Resume**).

4. Conclusions

Based on studies about Petri net extensions used for digital system modeling, we proposed an extension called Petri Net for Digital Systems (PNDS). Its current version is focused at system specification. The treatment of high-level synthesis aspects has been left for future versions. This paper shown the main features and components of PNDS, emphasizing exception handling description. PNDS components, representational language, and analysis methodologies are undergoing a formalization process. Future work will include the implementation of a software framework capable of capturing digital system PNDS descriptions.

5. Acknowledgement

This work has been partially supported by FAPESP under Grant number 99/04321-7.

6. References


