An Introduction to Structured Petri Nets

Marshall T. Rose
Department of Information and Computer Science
University of California, Irvine
Sun Oct 30 22:55:41 1983

Computer Mail: MRose.UCI@Rand-Relay

Technical Report Number 218

Introduction and Motivation

The protocol−entities used in computer networks form a large class of concurrent software systems. Traditional methods of specification, intended primarily for the design of sequential software, do not address the issues of timing and failure, which are the central focus of many parallel applications, and in particular network protocols. As such, software of this type presents several challenges for a representation language.

Many techniques have been applied to the problems of specifying the behavior of these entities, ranging from state−encoded techniques at one extreme to history−encoded techniques at the other[SCHW82]. State−encoded techniques tend to simplify the bridge to implementation, while history−encoded methods tend to permit more elegant proofs of correctness. Rather than viewing these approaches as alternatives to each other, we choose to view many of the techniques as complementing each other in going from more abstract specifications to implementation. Our perspective now narrows on the state−encoded approach to specification, and in particular the Petri net model[PETE77], which has been used successfully for describing hardware systems, but has not enjoyed widespread−use in software design.

Recently, another extension to the Petri net model has been proposed, the structured Petri net[MROSE83a]. It is our purpose to explain the notion of structured Petri nets, and to show how they can be used to represent concurrent software systems. We emphasize the fundamental principles of the structured Petri net technique, rather than closely examining how structured Petri nets may be used to specify communications protocols. By using this approach, we hope to gain an understanding of the structured Petri net as a general specification tool.

Philosophy

The philosophy of structured Petri nets finds its basis in two *simple* observations: first, programming languages, while permitting powerful data−manipulation capabilities, do not contain powerful concurrency−constructs with clean semantics; and, second, that Petri net languages, while permitting elegant representation of concurrency, are unable to represent any meaningful handling of data. Structured Petri nets, in response, combine the data−handling features of programming languages with the control−logic features of Petri nets.

Although strongly worded, these observations are quite defensible. Programming languages have evolved from a sequential background, in which flow of control has been viewed strictly within the context of a given program. As concurrent programming has received wider use, programming languages have been augmented with concurrency−constructs. Unfortunately, the designers of programming languages have been unable to introduce powerful methods of indicating concurrency that have simple semantics. The rendezvous−construct found in Ada[1][ADA82], while fully capable of permitting elaborately staged synchronizations, has an amazingly complex set of rules that are difficult, at best, to describe straight−forwardly[2].

Petri nets have evolved from a theoretical background, and have been used primarily as method for hardware design. As such, many of the notions taken completely for−granted in software, such as the reference semantics for variables (i.e., scoping rules), are absent from this approach to representation. Many authors have proposed extensions to the basic Petri net model, including timing[MERL76], memory and processors[BERT82]. When viewed with these extensions in mind, Petri nets continue to suffer from some inherent weaknesses as a method for representing software systems. For example, most extended Petri net models, such as numerical Petri nets[SYMO80], view memory as global, so that any transition can cause any portion of the memory component to be altered. The UCLA Graph Model of Behavior (GMB)[RAZO80], which is a particular extension of Petri nets, solves this problem though the introduction of a data flow graph. The data flow graph clearly defines the particular relations between transitions and memory components. While useful, the data flow graph is cumbersome when compared to the mechanism used by programming languages to define access to variables.

The Basics

We now consider the building−blocks that are used to construct structured Petri nets. This discussion is more conversational than formal: the current definition of structured Petri nets can be found in [MROSE83b].

A software system described by the structured Petri net technique is composed of one or more structured Petri nets. A Petri net is a directed graph with two types of nodes: *places*, which hold *tokens*; and *transitions*, which absorb and produce tokens. A structured Petri net is also a directed graph, but in addition to places and transitions, it is also populated with: *invocations*, which represent the instantiation of another net; and *named subnets*, which represent the substitution of another net. Each net has one or more places designated as *entry* places, and one or more places designated as *exit* places.

A structured Petri net operates in much the same way as a Petri net: the presence of tokens on the *input* places for a transition *enables* that transition, which then *fires* by absorbing the tokens on the input places and producing tokens on the *output* places. There are certainly more formal definitions for enabling and firing, than this terse synopsis, but since our purpose is to gain an intuitive understanding of the structured Petri net technique, no more time is spent on the fundamentals of the Petri net model[3].

[1] Ada is a trademark of the Department of Defense (Ada Joint Program Office).

[2] It is not our purpose to single−out Ada, or to unfairly criticize its definition, many other programming languages are also guilty of powerful, yet non−understandable concurrency−constructs.

[3] This should not present any difficulties for readers with a casual familiarity with Petri nets. Readers without any background whatsoever should examine [PETE77] before continuing.

Accessing Data

Tokens traverse the net. The token found in a structured Petri net has a single attribute, its *color*. All tokens of the same color are indistinguishable from each other, although tokens of different colors are easily distinguished. A color may be thought of as representing a dynamically scoped environment. Colors are mapped to *contour−blocks*, which denote a part of a private resource space (i.e., memory component) for a particular process executing in the system.

The color attribute of a token has an ordinal value from the set of all colors, which for our purposes is very, very large. Unique to each process is a *color−generator*, which is a machine that produces new colors for the process whenever the generator is *incremented*. The color−generator has a *current color value*, which is always equal to the color produced by the last increment operation. In short, since each process has its own color−generator, a contour becomes the binding between a particular process and a given color, and as a result, each process has its own private memory−component.

A contour−block contains bindings for variables and a pointer to the immediately scoping contour. This means that by using colorful tokens, structured Petri nets are able to achieve a dynamic scoping mechanism to determine data access. To search for a variable in the context of a particular token, that token's contour−block is first examined. If the variable is not present, then the previous contour is examined. This activity continues until either the variable is found, or there is no previous contour. In the former case, the variable found references the most recent occurrence of the data represented by that name; in the latter case, the variable is undefined in the context of the original token, (no data is represented by that name in that context).

To introduce a new context, an operation similar to entering a new block in a programming language, a token passes through a transition, which increments the color−generator, and produces a token whose color is the current−color−value and whose immediately−scoping contour is the color of the original token. New variables may now be declared and initialized in the new context. To terminate the context, an operation similar to leaving a block in a programming language, a token passes through a transition, which produces a token whose color is the one denoting the immediately−scoping contour of the original token.

Execution

All of the activity of a structured Petri net is controlled by the transitions in the net. A transition may potentially perform a very complicated set of steps when it fires after being enabled. As with the basic Petri net model, we view the firing of a transition to occur as a single atomic action, although internally many actions may occur.

For a transition to be enabled, a boolean expression, known as the enabling conditions of the transition must be satisfied. This expression may reference the number of tokens on the input places for the transition, along with the value of variables in the context of these tokens. It is very important to note that tokens of the same color collaborate to satisfy the enabling conditions, tokens of different colors are *always* considered separately. Hence it is possible for more than one group of colorful tokens to satisfy the enabling conditions, the actual tokens chosen, known as the *eligible* tokens, is chosen non−deterministically. It is also important to emphasize that the testing of the enabling conditions must not cause side−effects.

When a transition is enabled, it fires. At this point several activities happen in sequence:

(1) The eligible tokens are removed from the input places and move to their corresponding input arcs. A *selection rule* is evaluated which determines the number of tokens that are to be *introduced* into the net, and which output arcs are to receive those tokens.

(2) A set of *construction rules* are followed which tell what colors these introduced tokens take on. Normally, the tokens have the same color as the eligible tokens, but this need not be the case. The tokens are then "constructed" and placed on the appropriate output arcs.

(3) The introduced tokens are then manipulated in the context of the firing transition. This means that a sequential code segment is executed that references variables in the context of the eligible and introduced tokens.

(4) Finally, the eligible tokens are removed from the input arcs, and the introduced tokens are placed on their respective output places. At this point, the transition has finished firing.

There are a number of nuances that this firing method permits. The selection rule, permits us to easily build topologies that represent if—then—else or switch decision structures by using a single transition and many output places. This same flexibility can be achieved by using a single input place any many transitions, but it is felt that the former representation might be easier to analyze[4]. Naturally, evaluating the selection rule must not cause side—effects.

The constructor rules allow us to easily push and pop contours to change the scoping of the data. These rules are required to either use the same color as the eligible tokens, increment the color generator and then use the current color value, or pop the most recent contour. If the generator is incremented, then the constructor rules may introduce variables into the newly created context. As we see later, this permits us to instantiate invocations both recursively and in parallel.

Timing Considerations

The notion of time in structured Petri nets is currently underdeveloped. Nevertheless, structured Petri nets can contain temporal elements, using the concepts presented in [RAZO83]. In addition to enabling conditions and firing rules determining how a transition fires, each transition also has an enabling time and a firing time.

Although the enabling conditions, considered in the context of the eligible tokens, determine if a transition can fire, the enabling time for a transition is the duration of time that the enabling conditions must be continuously satisfied before the transition can actually begin to fire. This permits us to represent activities such as timers. By default, transitions have an enabling time of zero: as soon as the enabling conditions are satisfied, the transition is ready to fire.

To permit performance analysis, the firing time of a transition is the duration of time that occurs between the instant that the eligible tokens are removed from the input arcs, and the instant that the introduced tokens are moved from the output arcs to their respective output places. During this time, other transitions may fire. Since all of the critical actions of the firing have already occurred automatically, no problems result. By default, transitions have a firing time of zero: as such, the transition appears to fire completely in a single atomic

[4] This conjecture is entirely intuitive at this point in time.

action.

Control Hierarchy

A key attribute of the structured Petri net technique is the ability to summarize control information by having several instantiations of a net active at the same time. Structured Petri nets have a simple method of achieving this powerful representational ability.

An invocation in a net is represented with a square instead of a circle. A transition which outputs to an invocation is known as an *entry* transition, while the invocation which receives tokens from an invocation is known as an *exit* transition. There are special semantics associated with these two types of transitions. Briefly stated, when an entry transition fires, it produces exactly one token with a new context. If parameters are to be passed to the invocation, new variables can be created and initialized by the transition's construction rules. When the token is ready to be moved from the output arc leading to an invocation, it is placed on the entry place for the net representing this invocation.

The net must be designed in such a way that it can now begin to execute with this new token. Normally, the first action is to introduce another contour and load it with variables local to the net. At some point in the future, this token reaches the net's exit place. When it moves here, it appears instead in the original net, at the exit transition for the invocation. The exit−transition typically performs the actions thought of as clean−up after a subroutine call, and produces a token with the same color as the eligible token of the corresponding entry transition. The original net can now continue to along that particular path of execution.

Although this method sounds tricky, it is really quite simple: the entry and exit transitions act like push and pop operators on a stack, saving and restoring execution context. It is interesting to consider that this really does handle recursion and concurrent execution properly. Even though one path of execution is suspended when an invocation is instantiated, other paths in that same net and in other nets are still running. It is now easy to see how a single structured Petri net may be used to represent many instantiations of an object, by having tokens of different contexts traversing it.

In addition to the invocation, there is a second control mechanism requiring mention: A named subnet is represented as a dashed square, and denotes the substitution of another net. Named subnets are often used as an alternative to tail−recursion in a net.

Suppose each top−level net in your system represented a particular state. To denote moving from one state to another, you could use reference these nets as invocations. Unfortunately, it is conceptually unclear as to what state your system was really in, since each invocation eventually returns to its exit place. In contrast, named subnets do not save or restore context. Rather, when a token is ready to be moved from an output arc leading to a named subnet, it appears on the entry place for that subnet and continues in its present context. This allows you to easily represent the state of your system.

Synchronization

Since tokens of different colors compete but do not co−operate to enable transitions, and since memory is changed only in the context of the eligible and introduced tokens for a firing transition, we now consider how tokens of different colors can communicate and exchange information. Two special transition disciplines, the *boundary* and *split* transitions help to permit this capability.

A boundary transition is used to permit fixed entities to communicate with each other. The firing rules for a boundary transition clearly specify which variables are to be communicated. These are copied into the context of a *blank* token. Each blank token is unique, and easily distinguished from other types of colorful tokens. Conceptually, the information communicated is fully copied to the context of the blank token, so that the information can indeed be viewed as "traveling" from one process to another. When the blank token is removed from the output arc, it is placed on the corresponding input arc of another boundary transition. At this point, the other process can retrieve the information from the context of the blank token, copying it into its own current context. The boundary transition, while more rigid in its application than split transitions, has a very clear conceptual basis.

A split transition is used to permit arbitrary entities to communicate with each other and to synchronize. Split transitions have the same number of input arcs as output arcs[5], each pair of arcs correspond to a token of a potentially different color. The enabling conditions require that each input place have at least one token, and the selector rules produce exactly one token for each output arc. Each introduced token has the same color as its corresponding eligible token. During the manipulative part of firing, variables may be considered in the context of any of the eligible tokens. This permits a simple rendezvous and information exchange mechanism. Split transitions are allowed to violate a key tenet of the data access rules for tokens, but do so in a controlled fashion.

An Example

Finally, let us view an example of a concurrent system which is described by structured Petri nets. The solution to this simple problem demonstrates a few of the fundamental concepts of the structured Petri net technique.

Conceptually, the system generates prime numbers by sifting a sequence of ascending natural numbers through a filter of processes. Each process in the filter has associated with it a prime number. When it receives a number to consider, the process sees if that number is evenly divisible by the prime number associated with the process. If so, the number can not be a prime number, and is filtered out. If not, then the process passes the number to the next process in the filter. If there isn't another process, then the number has successfully passed through all prime numbers known to the system and must therefore be a prime[6]. Stated more precisely, the composition of the system is:

(1) There are two types processes in the system, a driver process, and many generator processes. The driver process is the top–level process that is instantiated to guide the system. Each generator process represents a particular prime number.

(2) A driver process begins by starting the first of the generator processes, with the prime number 2. Then, the driver perpetually iterates, by incrementing the number and communicating it to its child, so as to enumerate 3, 4, ..., and so on.

(3) Each generator process begins by printing out the number that it was invoked with. It then enters an perpetual loop. The loop begins by receiving a number from the parent of the process. If this number is evenly divisible by the process' invocation number, then this new number is not a prime and may be discarded. Otherwise, if the generator

[5] At least two of each.

[6] This conclusion is true since ascending numbers, starting at 3, are given to the filter which initially contains only one process, which is associated with the prime number 2.

has not started a successor, the number is a prime, so the generator starts another generator process with the new number. Otherwise, the process simply communicates this number to its child.

Three structured Petri nets are used to specify the system. The MAIN net (figure 1) is the driver process. When instantiated, it begins by creating a contour for local variables: **x**, which is the number currently being enumerated, and **n**, which will be used as a synchronization identifier. Control then forks. One fork instantiates the PNF net, which is made with parameters **p** and **n**. Note that since instantiating an invocation requires the creation of a new contour, the **n** parameter is not the **n** variable in MAIN's local contour. These parameters are initialized, and the PNF net is invoked. Since PNF is believed to never terminate, there is no exit transition associated with the invocation. Considered from a rigorous perspective, this is incorrect, but in order to emphasize the nature of PNF (and keep the figures used in this example simple), MAIN was purposely constructed in this fashion.

The other fork begins the enumeration loop, **x** is incremented, and then the S net is instantiated at the C entry point, with parameters **n** and **x**.

The PNF net (figure 2) is the specification for the generator process. When instantiated, it begins by creating a contour for local variables: **l**, which indicates if a child has been created for this process, and **x**, which is used to hold communicated numbers. As a part of the firing rules, the statement **print(p)** is called. This procedure prints out the value of the variable **p** in the current context (recall that when invoked, PNF was given two parameters, **p** and **n**).

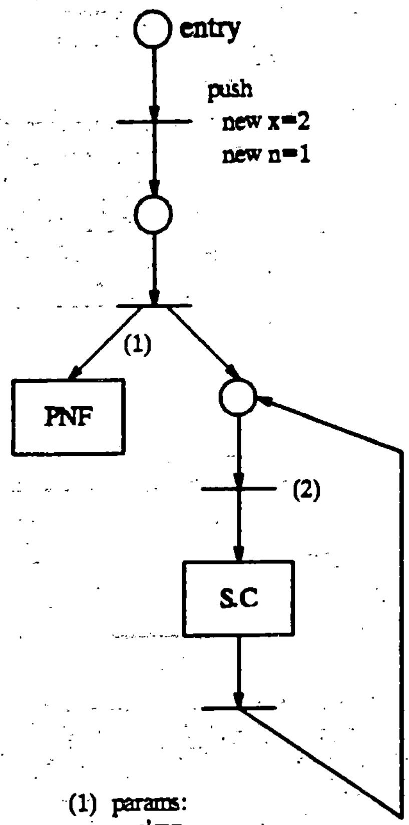
The S net is now instantiated at the P entry point. When the invocation returns, the variable **x** is copied from the contour used to instantiate the invocation into the current execution context[7]. This roughly corresponds to a value−result parameter passing mechanism. The variable **x** is compared against **p**. If **x** is evenly divisible by **p**, then control loops back. Otherwise, if **l** is **FALSE**, then control forks. One fork instantiates the PNF net, which copies **x** into **p** and increments **n**. The other fork sets **l** to **TRUE** and loops back. If **l** is **TRUE**, then the S net is instantiated at the C entry point.

The S net (figure 3) is the net that synchronizes the processes. The S net has two entry points P (synchronize with parent) and C (synchronize with child), and consists of a single transition, a **split** transition. The enabling conditions for this transition specify that the value of the variable **n** in the context of a token from arc t_1 must be equal to 1 more than the value of the variable **n** in the context of a token from arc t_2. When the transition fires, the value of the variable **x** in the context of the eligible token from arc t_1 is set to the value of the variable **x** in the context of the eligible token from arc t_2. After this manipulation, each token reaches an exit place.

Described simply, the S net synchronizes a parent and its child, and copies **x** from the context of the parent to the context of the child.

This example demonstrates an interesting property of structured Petri nets, the ability to perform horizontal− and vertical−multiplexing. Although several PNF processes will be executing, only one structured Petri net is required to represent them, since the colorful tokens contain all of the state information. This is an example of horizontal−multiplexing. There are several advantages to this type of capability. For example, when specifying a system, you needn't know before−hand the number of processes that will be running in the system. Instead you must specify the structure of each type of system exactly once.

[7] The x-prime (x') notation is used to denote this.



(1) params:
p'=x
n'=n+1
(2) x=x+1
params:
x'=x
n'=n

Figure 1. MAIN

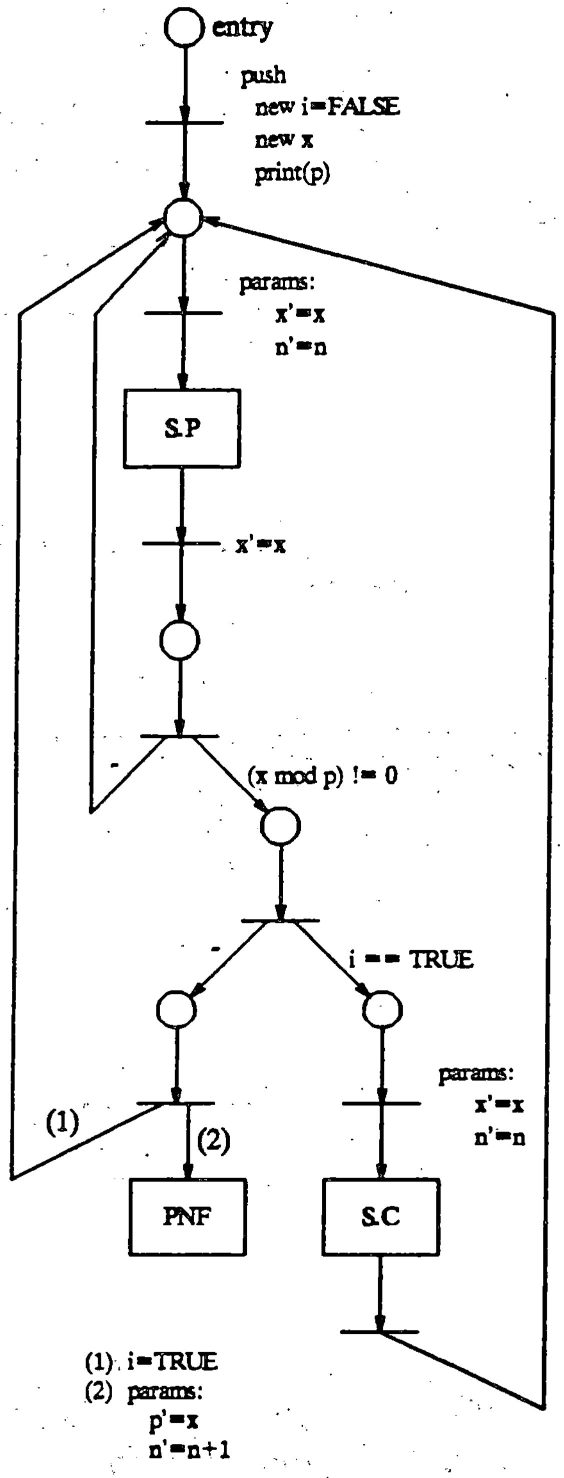

entry
push
new i=FALSE
new x
print(p)
params:
x'=x
n'=n
S.P
x'=x
(x mod p) != 0
i == TRUE
params:
x'=x
n'=n
(1)
(2)
PNF
S.C
(1) i=TRUE
(2) params:
p'=x
n'=n+1

Figure 2. PNF

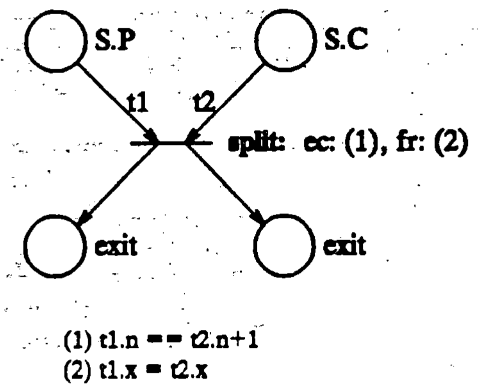


Figure 3. S

References

[ADA82] AdaTEC, "Reference Manual for the ADA Programming Language", ACM Special Publication, (July, 1982).

[BERT82] G. Berthelot, R. Terrat, "Petri Net Theory for the Correctness of protocols", originally appearing in: Proceedings, Second International Workshop on Protocol Specification, Testing, and Verification, C.A. Sunshine, editor, North–Holland Publishing Company, pages 325–342, (May, 1982); also appearing in: IEEE Transactions on Communications, volume 30, number 12, pages 2476–2505, (December, 1982).

[MERL76] P.M. Merlin, D.J. Farber, "Recoverability of Communication Protocols -- Implications of a Theoretical Study", originally appearing in: IEEE Transactions on Communications, Volume COM–24, Pages 1036–1043, (September, 1976); also appearing in: Communication Protocol Modeling, C.A. Sunshine, editor, Artech House, (1981).

[MROSE83a] M.T. Rose, "Observations on the Relations between Numerical Petri Nets and Algorithmic Representations", Department of Information and Computer Science, University of California, Irvine, (May, 1983).

[MROSE83b] M.T. Rose, "Structured Petri Nets", Working Paper (constantly updated), Department of Information and Computer Science, University of California, Irvine, (October, 1983).

[PETE77] J.L. Peterson, "Petri Nets", Computing Surveys, volume 9, number 3, pages 224-252, (September, 1977).

[RAZO80] R.R. Razouk and G. Estrin, "Modeling and Verification of Communication Protocols in SARA: The X.21 Interface", IEEE Transactions on Computers, Volume C–29, Number 12, Pages 1038–1052, (December, 1980).

[RAZO83] R.R. Razouk, C.V. Phelps, "Performance Analysis Using Timed Petri Nets", Technical Report Number 206, Department of Information and Computer

Science, University of California, Irvine, (August, 1983).

[SCHW82] R.L. Schwartz, P.M. Melliar–Smith, "From State Machines to Temporal Logic: Specification Methods for Protocol Standards", originally appearing in: Proceedings, Second International Workshop on Protocol Specification, Testing, and Verification, C.A. Sunshine, editor, North–Holland Publishing Company, pages 3–20, (May, 1982); also appearing in: IEEE Transactions on Communications, volume 30, number 12, pages 2486–2496, (December, 1982).

[SYMO80] F.J.W. Symons, "Introduction to Numerical Petri Nets, a General Graphical Model of Concurrent Processing Systems", originally appearing in: Australian Telecommunication Research, Volume 14, Number 1, Pages 28–32, (1980); also appearing in: Communication Protocol Modeling, C.A. Sunshine, editor, Artech House, (1981).