On Petri Nets and Predicate-Transition Nets

Andrea Röck

INRIA - project CODES
Roquencourt - BP 105
Le Chesnay Cedex 78153, FRANCE

Ray Kresman

Department of Computer Science
Bowling Green State University
Bowling Green, OH 43403 USA

ABSTRACT

Petri nets are used to study many types of networked systems. [1] have designed a software tool for analysis and simulation of Petri nets. We extend this tool to handle a form of Petri net known as Predicate Transition (PrT) nets. We implement mechanisms that automate the process of folding a Petri net to a PrT net and finding invariants. We can compute invariants for systems that use either Petri net or the extended PrT net. Invariants are required to prove certain properties of the system being modeled, such as liveness and safety. Finally, for Petri net, we use the invariants to prove these two properties.

Categories and Subject Descriptors D.2 [Software Engineering]: Design Tools and Techniques – *Petri nets.* Software/Program Verification – *Formal methods.*

1 Introduction

Several models have been used to study process cooperation in computer systems. Communication protocols have been fertile grounds for a number of these approaches. Many protocols may be described by a Finite State Automaton [2]. Finite state representations use state transition diagrams, state transition matrices [3] or even decision tables.

Another representation is Petri nets [4]. They are found in application domains such as networks, databases, programming languages, and others. Like variants of FSA, it is a graphical representation of the underlying physical system. Tokens are used in these nets to simulate the dynamic and concurrent activities. As a mathematical tool, it is possible to set up state equations, algebraic equations, and other mathematical models governing the behavior of these systems. As a graphical tool, Petri nets can be used as a visual-communication aid similar to flow charts, block diagrams, and network diagrams. There is even a major standards group that is responsible for the standardization of Petri nets [5]. A number of extensions to basic Petri nets have been proposed.

One extension is to associate a firing delay with each transition. This delay specifies the time that the transition has to be enabled before it can actually fire. If the delay is a random distribution function the resulting net class is called a stochastic Petri net. Another variation provides for a delay bound in which both the lower and upper bounds are specified [6].

For a communication system modeled as a Petri net, one can prove certain properties or assertions about the underlying system. Figure 1 shows a Petri net model of the *alternating bit protocol*. The transmission medium is modeled in the middle and the two processes, send message/receive ack and send ack/receive message, are on either side. The different states of the system may be easily inferred from Figure 1. Because Petri nets are fairly simple structures, the representation of larger systems can get clumsy and the proof of such nets gets very involved. Communication protocols have a large number of states and it is difficult to capture all of the interactions using simple Petri nets. Predicate/Transitions nets (PrT) are a more generalized version of Petri nets[7, 11].

This paper concerns PrT nets and their use in constructing certain properties of communication protocols. [1] is a software tool to study Petri nets. We provide certain extensions to [1] that aid in the handling of PrT nets. Our approach transforms a simple Petri net into a PrT net and derives assertions for the new net. The

first challenge is to create and display PrT net. Another one is to collapse a Petri net into a PrT net. Following this, the process for computing invariants is automated. Then, we prove safety and liveness properties of Petri nets (but not PrT nets).

Section 2 defines PrT nets and provides a motivation for our work. In Section 3, we give an overview of our extensions. Section 4 addresses invariant and other properties. Concluding remarks are found in Section 5.

2 PrT Nets

A PrT net is a tuple (P, T, F, Σ, L, φ, M_0) [13], where:

- P is a finite set of predicates (first order places), T is a finite set of transitions ($P \cap T = \emptyset$, $P \cup T \neq \emptyset$), and F $\subseteq (P \times T) \cup (T \times P)$ is a set of arcs. (P, T, F) forms a directed net.
- $\sum$ is a structure consisting of some sorts of individuals (constants) together with some operations and relations.
- L is a labeling function on arcs.
- φ is a mapping from a set of inscription formulae to transitions. The inscription on transition $t \in T$, φ(t), is a logical formula built from variables and the individuals, operations, and relations in structure $\sum$; Variables occurring free in a formula have to occur at an adjacent input arc of the transition.
- M_0 is the initial or current marking. $M_0 = \__P p\ p\ M \in)\ (\ 0$, where M_0 (p) is the set of tokens residing in predicate p.

The above PrT nets have simplified general PrT nets [7] in two ways: 1) an arc labeling is a set of tuples (labels) $\{l_i\}$ rather than a formal sum $c_1l_1+c_2l_2+..+c_nl_n$ (i.e. coefficient or arc weight c_i of arc label l_i is 1 for all $1 \leq i \leq n$). 2) Accordingly, the marking of a predicate under a certain state is a set of tokens (i.e. items in [6]) instead of a formal sum of tokens. These nets are more or less like first-order logic programs. From the perspective of formal verification, the reachability analysis of these PrT nets is made more efficient.

PrT nets are a high-level formalism of Petri nets. We comment on the reachability analysis techniques for general PrT nets. Parameterized reachability trees [8] exploit parameterized markings as a means for folding reachability trees of PrT nets so that a number of concrete states can be condensed into a generic state. In particular, parameterized reachability trees may take extra time to instantiate the parameters for generating successor states and for comparing states, though it does save a large amount of space.

The Petri Net Kernel (PNK) [1] provides an infrastructure for handling Petri nets and integrating the net into a tool. This kernel provides a rich environment to draw, edit and represent Petri nets and to write applications for these nets. It is a free software and is supported under the terms of the GNU Library General Public License. PNK is available in python and Java. We chose the Java implementation for our work.

One difficulty with this kernel, however, is the lack of support for PrT nets. We have designed new classes that extend PNK in order to work with prT nets. With our extensions, one can manipulate PrT nets just as easily as PNK manipulates Petri nets. In addition, we designed two new functionalities: 1) Collapsing technique - allows a large Petri net to fold into a, smaller, PrT net. 2) Invariant computation - automatic detection of properties that remain unchanged (invariant) under a firing sequence.

3 PNK and Our Extensions

We briefly dwell into some design issues of PNK. PNK has two classes, *Extendable* and *Extension*. An object of the class Extension is connected to one object of the class Extendable. An Extendable object can be connected to multiple Extension objects.

We focus our attention on four important subclasses of Extendable: *Place*, *Transition*, *Arc* and *Net*. The Extension represents special properties of an Extendable, like Marking for a Place. A Petri net can be built from Places, Transitions and Arcs, for example, by connecting a Place with a Transition. A top level layout of PNK is shown in Figure 2.

An xml file (netTypeSpecifications) specifies what Extensions are allowed for each Extendable in a specific net formulation of a problem. The Extensions are specified with a name and a class as illustrated in Figure 3.

Modification to Subclasses of Extendable

The Place class represents a place in both type of nets. It can contain a number of Tokens (for Petri nets) represented by an Integer number, or a set of variables, individuals, and tuple of variables and individuals (for PrT nets) represented by the PrTrNetMarking class. The PrTrNetMarking contains a Vector of String objects (represent the individuals and variables) and of Tupel objects. The Tupel object again contains a Vector of String objects representing variables and individuals. (For a definition of some of these terms, please see Section 2, paragraph 1.)

The Transition class represents a transition in a Petri net. It has the function of transferring markings from one place to another. The Transition of the Petri net does not have an Extension.

For the PrT net we wrote an Extension which implements the given interface Mode (see [1] for a discussion of Mode). In a PrT net Variables always belong to a Transition. The class Variable contains the variables, its possible values and its actual value. The function *evaluate* replaces Variables in a PrTrNetMarking with the actual Values of the Variables.

The Arc class represents the connection between transitions and places. It specifies the number of tokens or marking that are transferred by the firing of a transition. Its Extension is called Inscription. For the Petri net this Inscription represents a Integer Number represented by the class llNet.NaturalNumber1 (please see [1] for a description of this object). Following this approach, we developed a new class, PrTrNetInscription for PrT net. It is a subclass of PrTrNetMarking.

One can write applications for a net using PNK. The net for the application is described in a file. The application is invoked with a call to start which then calls the run method of the application. Additional menu entries may be added by overwriting the getMenus method. To act on the selection of a new entry, we wrote a class that implements the ActionListener interface and listens on that special menu entry. Finally, we note that any application is a subclass of the class MetaApplication**.**

Collapse Places and Transitions

CollapsingRule is an Extension that belongs to a PrT net and includes the collapsing functionality. We collapse a Petri net and display it in a PrT net structure before collapsing to a PrT net. The CollapsingRule contains three main methods: collapseWithUserInteraction, collapseAutomated and checkTrans. CollapseWithUserInteraction lets the user decide which places to collapse. CollapseAutomated prepares a preselection for the user from which the user can choose the places to collapse. CheckTrans examines transitions that are collapsible into one transition. As compared to collapsing places the user does not have to choose anything, instead collapsing of transitions happens without any user intervention. A set of transitions can be collapsed if for all transitions in the set the preset and the postset of places is the same. This means that all the transitions have the same arcs to the same places. If the software can find this set of transitions, it sends them to the reduceTransition method. Some of these activities are summarized in Figure 4**.**

Computation of Invariants

The reader is referred to [9] for a definition of invariants. Integers are used to compute invariants of a Petri net. However, polynomials are used in computing invariants of a PrT net. Invariants are computed using a rather involved algorithm from [10]. For brevity, we do not discuss their approach.

To calculate the invariants of a net we wrote an abstract class GetInvariants and specific subclasses, one for Petri net and the other for PrT net, as shown in Figure 5. Note that GetInvariants is a subclass of Extension. The figure captures only the essential public methods that are important for the calculation of invariants.

To keep the computation of the invariants generic we used the interface mechanism of Java. The interfaces provide all the method signatures that are necessary to calculate the invariants. For each of the two types of nets we have a different class that implements the Element interface (IntegerElement for Petri net and PolynomElement for PrT net). Again, to stay generic during the calculation process we needed a method to

create Elements without knowing which type of Elements we are working with. This was achieved using an ElementFactory interface which is based on java factory methods.

The calculate method (of GetPTInvariants) first fills the incidence matrix of the Petri net with IntegerElements. Then it creates a calcInvariants object with this matrix and calls the calcInvariants method. Finally, the invariants are used to prove liveness and safety of the Petri net (see Section 4 below). Similarly, the calculate method (of GetPrTrInvariants) fills the incidence matrix with PolynomElements. Again it creates a calcInvariants object and calls the calcInvariants method which is then used to construct the invariants for the PrT net. However, as noted in Section 1, we have not been able to extend this work to prove liveness and safety of PrT nets.

4 Properties of Petri Nets

Two important properties of Petri nets are safety and liveness. A Petri net is safe if no place has more than one token at any time. At the beginning the net is represented by an initial state, i.e. each place holds its initial marking. The state of a net at any time is represented by the markings of all places at that time. A state Y is reachable from state X if there exist a possible firing sequence of transitions that transfer the net from X to Y. [M0] represents the set of Markings that are reachable from the initial marking M0. A Petri-Net is live if for every marking M of [M0], M0 is in [M], i.e. M0 is reachable from M through a series of firings.

Following [9], that M can be reached from M' can be represented by the equation M' = M + C.f where f represents the firing sequence and C, the incidence matrix, represents the mapping of places to transitions. f is a vector that represents the number of times each transition is fired. Then, it can be shown that if *i* is an invariant, then $i.C = 0$ and hence $i.M' = i.M$, for all M, M' in [M0]. Using the initial marking M0, we derive a set of equations (see [9]**).** As an example of a place invariant equation, consider P1 + P2 = 1. This means that at any time place P1 or P2 will hold a token and the total number of tokens between the two places is exactly 1. We have been able to automate the computation of such invariants for both Petri net and PrT net.

If the marking of every place appears in at least one of the equations, we can use this result to prove the safety of the net. When the right side of the place invariant equation is 1 for each of the equations we can immediately conclude that the net is safe. If for one or more equations the right side is greater than 1 we have to do a more complicated proof which is based on proof by contradiction.

To prove liveness we use the approach outlined in [12]. Suppose a function, v(M), can be found such that

- $v(M) = 0 \Leftrightarrow M = M0$
- $\forall$ M that are reachable from M0 and $v(M) \neq 0$ there exist an M' that is reachable from M0 such that $v(M') < v(M)$

If so, M0 is a home state. In other words, for all M of [M0], we can get back to home, i.e. M0 is a member of [M].

[9, 11, and 12] do not provide a rule for deriving the function v(M). In our model, the user is asked to input the function v. The software then checks if those two conditions hold for the user specified v(M) and the initial state M0. First the program checks if v(M0) = 0. Then it finds all Markings that satisfy the set of equations we got from the invariants. For each Marking M that satisfies all the equations the program calculates v(M). If v(M) = 0, M must be M0. If v(M) > 0 the program checks if there exists a Marking M' reachable from M such that v(M') < v(M). At the present time, our algorithm only works if for each M we find a M' reachable by firing of only *one* Transition such that v(M') < v(M).

Figure 6 shows a screen snapshot of computing invariants and proving the safety and liveness properties for the Petri net model of Figure 1, the alternating communication bit protocol.

5 Conclusion

In this paper we provided an overview of our extension to a software from [1]. Our extensions handle predicate transition nets that are more generalized than Petri nets. We implemented an automated way to collapse Petri nets and PrT nets to reduce their complexity. We computed incidence matrix and place invariants. While we have been able to use them to prove liveness and safety property of Petri nets we were unable to do so for predicate transition nets.

Reference

[1] Petri Net Kernel, version 2.2. http://www.informatik.hu-berlin.de/top/pnk/index.html Humboldt-Universität zu Berlin Germany.

[2] D. Bjorner, "Finite State Automaton - Definition of Data Communication Line Control Procedures," FJCC, vol. 37, Houston, TX, Nov 1970, pp. 477-491.

[3] R.W. Stutzman, "Data Communication Control Procedures," Computing Survey, vol 4. December 1972, pp 197-220.[11]

[4] J. Peterson, "Petri Nets," Associations for Computing Machinery, Computing Surveys, volume 9. pp. 223-251, September 1977.

[5] International Standardization Organization (ISO). ISO/IEC JTC1/SC7/WG11

[6] P. Merlin, "A Methodology for the Design and Implementation of Communication Protocols," IEEE Trans. On Communications, vol. 24, pp. 614-621. June 1976.

[7] H. J. Genrich, "Predicate/Transition Nets" In: K. Jensen and G. Rozenberg (eds.): *High-level Petri Nets. Theory and Application*. pp 3-43 Springer-Verlag, 1991

[8] M. Lindqvist, "Parameterized Reachability Trees for Predicate/Transition Nets," Proceedings of the 11th International Conference on Applications and Theory of Petri Nets, Paris, 1990.

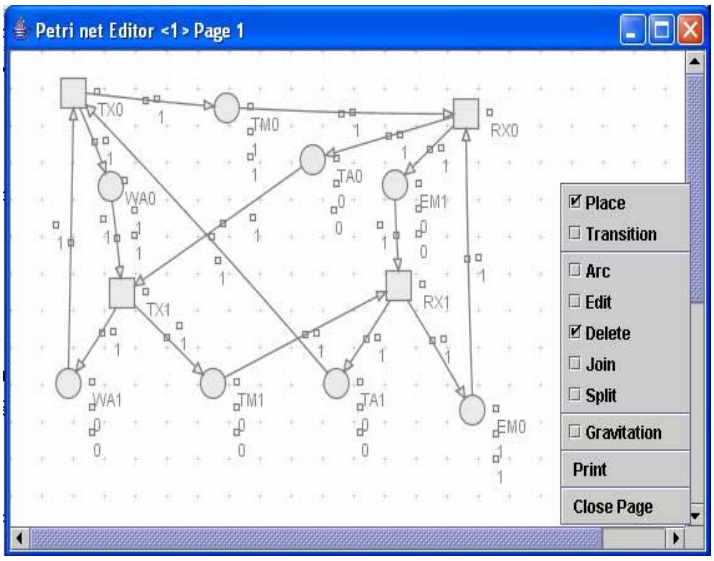
[9] G. Berthelot and R. Terrat, "Petri Nets Theory for the Correctness of Protocols" In: *IEEE Transactions on Communications, Vol. COM-30, No. 12*, pages 2497-2505. December 1982

[10] H. Mevissen, “Algebraische Bestimmung von S-Invarianten in Praedikat/Transitions-Netzen”, Gesellschaft fuer Mathematik und Datenverabeitung mbH Bonn, ISF-Report 81.02 (Maerz, 1985)

[11] H. J. Genrich and K. Lautenbach, "The Analysis of Distributed Systems by Means of Predicate/ Transition-Nets" In: Kahn, G.: *Lecture Notes in Computer Science, Vol. 70: Semantics of Concurrent Computation*, pages 123-146. Berlin: Springer-Verlag, 1979.

[12] R.M. Keller, "Formal Verification of Parallel Programs". In: *Communications of the ACM*, 19, 7, 371-384 (July 1976).

[13] D. Xu, J. Yin, Y. Deng and J. Ding: A Formal Architecture Model for Logical Agent Mobility. *IEEE Trans. on Software Engineering*. vol. 29, No. 1, pp. 31-45, Jan. 2003



- WA0/1 … Wait for Acknowledgement of message 0/1
- TM0/1 … Channel has message 0/1
- TA0/1 … Acknowledgment of message 0/1 is in the Channel
- EM0/1 …Expect message 0/1

Figure 1: Petri Net Representation of the Alternating Bit Protocol

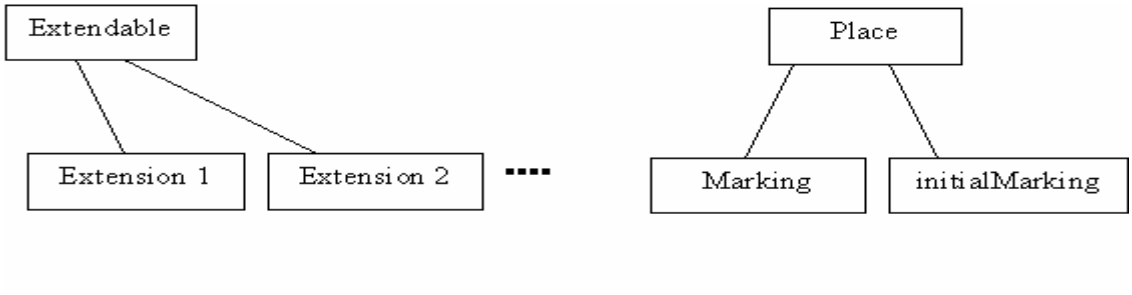


Figure 2: PNK Design Layout

Extension	Name and Class
Net	"firingRule": de.huberlin.informatik.pnk.netElementExtensions.llNet.SimpleRule
Place	"marking": de.huberlin.informatik.pnk.netElementExtensions.llNet.NaturalNumber "initialMarking": de.huberlin.informatik.pnk.netElementExtensions.llNet.NaturalNumber
Arc	"inscription": de.huberlin.informatik.pnk.netElementExtensions.llNet.NaturalNumber1

Figure 3: XML File and Extensions

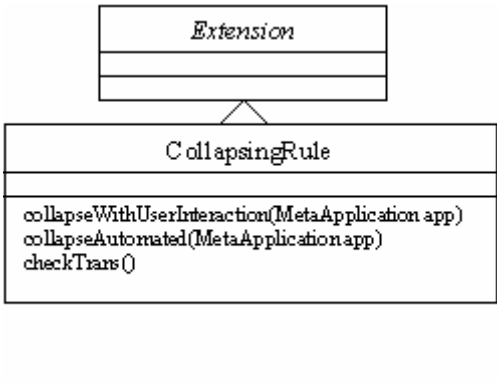


Figure 4: Collapsing a Petri net to a PrT net

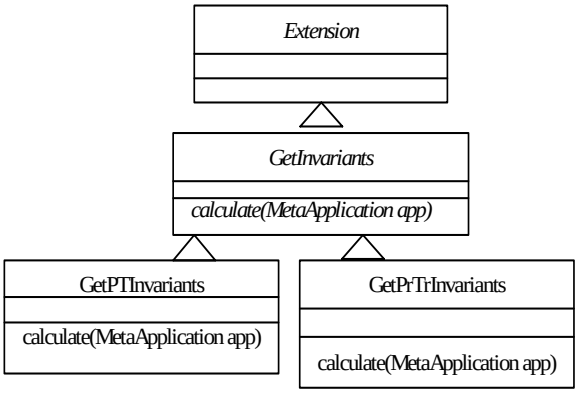


Figure 5: Our Extensions for Invariants

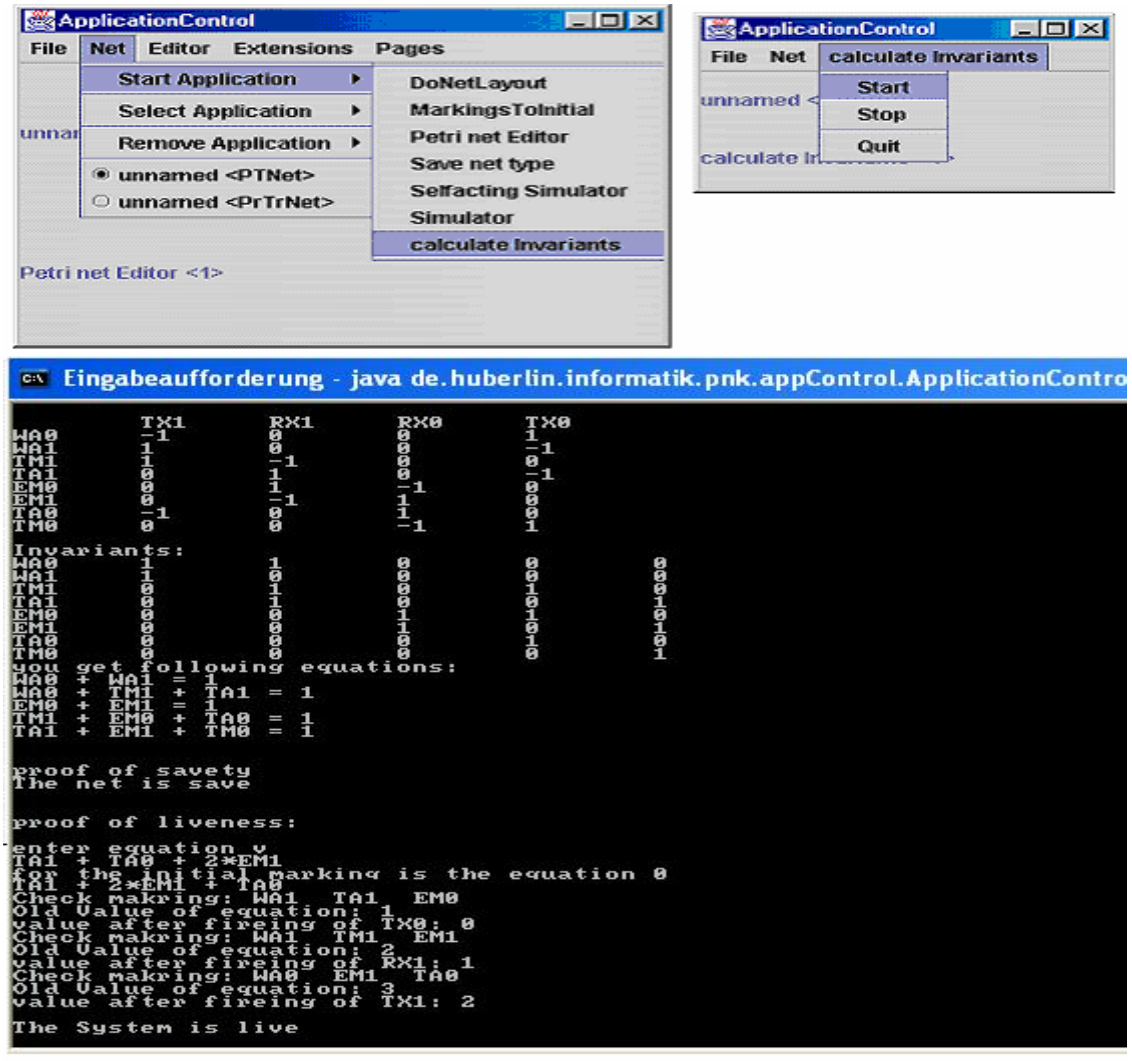


Figure 6: Petri net - Computation of Invariants and Proof of Safety and Liveness