Petri Nets and Programming: A Survey

Marian V. Iordache and Panos J. Antsaklis

Abstract—Petri nets and related models have been used for specification, analysis, and synthesis of programs. The paper contains a survey of several literature approaches and an examination of their relationship to Petri net modeling and supervisory control. The discussion is restricted to Petri nets in the class of place/transitions nets and the supervisory control of this class of models.

I. INTRODUCTION

Petri nets (PNs) are formal models developed in computer science (CS) for the modeling of concurrent systems. Since PNs have also been used in control systems in the context of the supervisory control (SC) of discrete event systems, it is interesting to examine the relationship between SC and CS applications.

The contribution of this paper is that it presents a survey of several CS approaches and examines their relationship to PN modeling and SC. The CS applications considered in this paper are those related to the synthesis of computer programs. Further, we restrict our attention to SC methods for PNs. By comparing the settings of specific CS applications and of SC it is possible to identify methods from one setting that could be applicable to the other and to distinguish opportunities for the development of new SC methods.

The paper is organized as follows. In section III we will examine the methods that could be used to extract a PN model from a program specification. Then, in section IV, we will consider how PN models could be used for the implementation of a specification. Some of the literature approaches will be examined in more detail, emphasizing their relationship to SC.

The paper assumes some familiarity of the reader with PNs. For more information on PNs the reader is referred to [1] and to the second chapter of [2], which contains also a detailed comparison of PNs with finite automata. For a survey of SC methods for PNs the reader is referred to the survey papers [3], [4]. A brief introduction to the SC terms and notation used in this paper is presented in section II.

II. PRELIMINARIES

Given a PN, let \( \mu \) denote the marking, \( q \) the firing vector, and \( v \) the Parikh vector. The firing vector represents the transition (or transitions) fired at a firing instance. The Parikh vector is a state vector of the PN recording how many times each transition has been fired. Thus, the Parikh vector equals the sum of the firing vectors. Given a place \( p_i \) and transition \( t_j \), let \( \mu_i = \mu(p_i) \), \( q_j = q(t_j) \), and \( v_j = v(t_j) \).

Note that the firing vector is defined with respect to a firing instance. For all transitions \( t \), \( q_i \) indicates how many times \( t \) is fired at the firing instance. In particular, assuming no concurrency, only one transition \( t \) may be fired at a time.

Then, \( q_i = 1 \) and \( q_j = 0 \) for all \( j \neq i \).

In SC, a supervisor is designed to control the operation of a plant such that a given specification is satisfied. If the supervisor and plant are represented by PNs, the specifications restrict the sequences of events generated by the plant. Let \( \rho \) denote the labeling function that associates each transition \( t \) with events \( \rho(t) \). Note that under supervision, a transition \( t \) of the plant may be fired only if there is an enabled transition \( t \) of the supervisor such that \( \rho(t) = \rho(t) \).

Most often, SC problems addressed in the literature consider the case in which the transitions of the plant are labeled by distinct events and the specification is given as a set of linear inequalities in terms of the marking \( \mu \) or of the Parikh vector \( v \), and sometimes also in terms of the firing vector \( q \). Thus, all these specifications can be described by the general form

\[
L\mu + Hq + Cv \leq b
\]

where \( L, H, C, \) and \( b \) are integer matrices of appropriate dimensions. Note that (1) requires that all reachable states of the plant satisfy \( L\mu + Hq + Cv \leq b \) and that a firing vector \( q' \) is fired only if (1) is satisfied for all \( q \) in the range \( 0 \leq q \leq q' \).

Specifications of the form (1) have the property that the supervisor can be designed such that its parallel composition with the plant results in a PN consisting of the plant and a number of additional places connected to the transitions of the plant. These additional places are known as monitors.

For example, the places \( p_{10} \) and \( p_{11} \) of Figure 5 could be seen as monitors enforcing \( q_{3} + q_{0} + \mu_{1} \leq 1 \) and \( q_{10} + q_{11} \leq \mu_{4} + \mu_{5} \), respectively.

Note that the specifications (1) can be generalized to disjunctions of the form

\[
\bigvee_{k=1}^{n} [L_k \mu + H_k q + C_k v \leq b_k]
\]

A specification (2) requires that at any time, at least one of the terms \( L_k \mu + H_k q + C_k v \leq b_k \) be satisfied. In general, such specifications cannot be implemented by monitors.

1Given two vectors \( x \) and \( y \), \( x \leq y \) means \( x_i \leq y_i \) for all indices \( i \).
Under certain boundedness assumptions, the supervisor can be designed in the form of a labeled PN [5].

III. EXTRACTING PN MODELS

This section deals with the extraction of PN models from higher level specifications of a program. Of special interest here is the extraction of PN models from specifications given in a programming language.

Languages allowing a finite state machine (FSM) representation of the specification could be of interest for PN modeling. Note that FSMs are PNs in which each transition has exactly one input place and one output place. Esterel [6] is a language that has been used for specifications that can be represented by FSMs. For instance, Esterel is used to describe specifications of Polis, a tool for hardware/software codesign of embedded systems [7]. In Polis, the specifications are translated into networks of Codesign Finite State Machines (CFSMs) [8]. Note that CFSM networks can be converted to safe\(^2\) PNs, as shown next.

A network of CFSMs consists of CFSM components interacting asynchronously. Thus, one CFSM may not react instantly to events generated by another CFSM. For instance, in Figure 1, the transition labeled by \(\eta/\beta\) occurs when the event \(\eta\) is present and upon its occurrence it generates the event \(\beta\). The transition labeled by \(\beta/\eta\) does not fire at the same time as the one that generates the event \(\beta\). Rather, the CFSM containing the transition \(\beta/\eta\) checks whether the event \(\beta\) is present. Then, if it is present, it fires the transition. A possible PN implementation of the CFSMs shown in Figure 1 is given in Figure 2(a). The places \(p_\beta\) and \(p_\gamma\) indicate the presence of the events \(\beta\) and \(\gamma\) when they contain at least one token. However, for consistency with the CFSM models, the places \(p_\beta\) and \(p_\gamma\) should not contain more than one token. This fact can be modeled by the PN shown in Figure 2(b), based on the construction shown in Figure 3(a). More complex networks of CFSMs, involving logic expressions on transitions, such as \(\alpha \land \beta\) in Figure 3(b), can also be dealt with, as illustrated in the figure.

Related to CFSMs but in certain respects more general are the condition systems [9]. A condition system is a PN in which transitions are labeled by enabling conditions and places by output conditions. A condition is a signal that may have a “true” or “false” value. A transition may fire only if the marking enables it and the conditions labeling it are satisfied. Further, the output conditions of a place are satisfied when the place is marked. An example is given in Figure 4. For instance, the output conditions of the places \(p_1\), \(p_7\), \(p_8\), and \(p_9\) are \(do\), \(up\), \(mid\), respectively. Moreover, the enabling conditions of the transitions \(t_5\), \(t_6\), and \(t_3\) are \(mid\), \(up\), and \(\overline{do}\) (the complement of \(do\)), respectively. For the marking shown in Figure 4 the conditions \(do\) and \(up\) are signaled. Therefore, the transition \(t_4\) may fire, since it is condition enabled.

As defined in [9], conditions systems allow enabled transitions to fire at the same time. However, note that if we impose the restriction that only one transition may be fired at a time, a condition system can be rather easily converted to a PN. This remark is illustrated on the condition system of Figure 4 under the additional assumption that the marking of any place can only be 0 or 1. An equivalent PN is shown in Figure 5. Note that self-loops can be used to ensure that a

---

\(^2\)A PN is safe if for all reachable markings there is no place containing more than one token.
Condition systems have been applied in [9], [10], [11] to the synthesis and specification of control software. There, condition systems represent the components of the system to be controlled as well as the controller modules, which are called taskblocks. Based on specifications describing sequences of states that the system should follow, taskblocks are synthesized. The controller is obtained by composing the synthesized taskblocks, where taskblocks interact hierarchically and sequentially to achieve the specification goals. The controller is then converted to control software [12]. Some specific SC problems in this context are addressed in [13]. The synthesis algorithms presented in [9] assume each component to be an FSM.

Since PNs do not have the expressiveness of a Turing machine, generally programs cannot be completely represented by finite PNs. Nonetheless, PNs can easily model the control flow and the interaction of the tasks of a program [14], [15], [16]. For example, PNs were used to represent concurrent programs written in a C-like language in [14] and the tasking behavior of Ada programs in [17], [18].

For a complete representation of programs by PNs, high level PNs are to be used. For instance, high level PNs are used for the internal representation of the specification in the PEP tool. PEP is a software tool for the development, verification, and simulation of parallel programs [19]. In PEP, specifications can be described by programs written in B(PN)$^2$ [20] and SDL [21] or by parallel finite automata [22]. Note that parallel finite automata are a collection of finite automata labeled with B(PN)$^2$ instructions. Specifications given in any of these three forms can be translated into high level PNs, Petri box calculus expressions [23], and low-level PNs. M-nets [24] are used for the high level PNs and safe PNs for the low-level PNs. Naturally, finite safe PNs cannot represent the specification unless all variables are defined on finite domains [20].

**IV. PN Based Design**

Once a PN model has been extracted from a program, desirable properties of the program can be examined based on the PN model. While a complete PN model of a program allows extensive verification using model checking, as in the PEP tool [19], only some of the properties of interest can be examined on PN models that are abstractions of software specifications. Of special interest in the literature has been the latter class of PN models for programs written in the Ada language [17], [18], [25]. In this context, the PN models were typically used in order to detect the presence of deadlocks. For the analysis of these PN models both structural methods [17] and reachability based approaches [18], [26], [27] were proposed. The proposed methods take advantage of the particular form of the PNs modeling Ada programs.

For the remaining part of this section we will examine in more detail two topics closely related to SC. Note that to some extent, the application of SC methods to software engineering has been considered in [28], [29]. Further, SC for deadlock prevention has been applied to the execution control of concurrent software in [30].
A. Predicate Control

A literature topic closely related to the SC of PNs is predicate control [31], [32]. The predicate control problem has been defined in the context of distributed systems. The system model used for predicate control can be described as follows. A distributed system consists of several processes, each executing a predefined program. Each process \( P_i \), \( i = 1 \ldots n \), is modeled by a sequence of events \( e_{i,0}e_{i,1} \ldots \). The events \( e_{i,0}, e_{i,1}, \ldots \) are generated in the same order each time \( P_i \) is executed. However, since the processes \( P_i \) are not completely synchronized, an event \( e_{i,k} \) may occur before or after an event \( e_{j,r} \). Thus, the distributed computation may have several executions, differing in the order in which the events are generated. The predicate control problem is to control the processes \( P_i \) such that a given predicate is always satisfied. The processes \( P_i \) can be controlled by adding some synchronization constraints to the system. A synchronization constraint involves two events \( e_{i,k} \) and \( e_{j,r} \) and requires that the process \( P_i \) should not generate \( e_{i,k} \) before \( P_j \) generates \( e_{j,r} \). Synchronization is implemented by a control message sent by \( P_j \) to \( P_i \).

As an example, Figure 6(a) illustrates two processes \( P_1 \) and \( P_2 \). \( P_1 \) consists of the event sequence \( e_0, \ldots, e_3 \) and \( P_2 \) of the event sequence \( f_0, \ldots, f_3 \). Graphically, synchronization is represented by arrows. For instance, in Figure 6(a), the arrow from \( f_1 \) to \( e_2 \) indicates that \( f_1 \) must precede \( e_2 \). Consider the predicate implementing a fairness requirement of the form \( |e_1 + e_2 + e_3 - \phi_1 - \phi_2| \leq 1 \), where \( \epsilon_i \) denotes the number of occurrences of the event \( e_i \) and \( \phi_i \) the number of occurrences of the event \( f_i \). The predicate is enforced by adding the synchronization arrows drawn with dashed lines.

The predicate control problem can be stated in terms of PNs, as illustrated in Figure 6(b). A synchronization arrow corresponds to a place connected between the two transitions of the arrow. Thus, generating synchronization arrows corresponds to designing a monitor based supervisor, where each monitor corresponds to a synchronization arrow. Monitors here are constrained to have exactly one input transition and one output transition. Note that in this problem PN models are safe marked graphs.

B. Scheduling

PNs have been applied to scheduling in the context of embedded software. In this context, often applications consist of multiple tasks and are naturally expressed by concurrent programs. Therefore, it is often the case that several tasks are to be run on a single computational resource (processor). Since the computational resource can execute only one instruction at a time, it is necessary to decide the sequence in which the instructions should be executed. Scheduling deals with creating a correct schedule of operations, that is, a correct sequence of instructions. In the context of data flow applications, which are common in digital signal processing, it has been noticed that scheduling does not have to be done at run time [33]. However, in more general applications, scheduling depends on data computed at run time. Of special interest has been the problem of deriving schedules in which regardless of the decisions made at run time, certain properties are satisfied. Properties of interest include liveness and that the program should be executable with bounded memory.

PNs have been used for the software model in papers such as [14], [34], [35], [36], [37], [38]. As an example, consider the PN of Figure 7 representing two processes of a software application. One process consists of the transitions \( a \ldots g \) and the places \( p_1 \ldots p_5 \). The other process consists of the transitions \( h \ldots k \) and the places \( p_6 \ldots p_{10} \). Transitions model internal operations. In particular, source transitions are associated with the reading of external inputs. For instance, in Figure 7, when the transition \( a \) is fired, input data is read. Further, transitions in conflict, such as \( h \) and \( i \) in the figure, model the possible choices of a control structure, such as an if-then-else statement. If the control structure depends on internal data computed at run time, then the transitions are uncontrollable. Indeed, at the time the schedule is computed, it is not known which transition should be fired.
it is interesting to notice that it is possible to obtain simpler supervisors enforcing the same schedule. For instance, in our example, the supervisor of Figure 8(b) enforces the schedule of Figure 8(a) and is considerably simpler than the state machine representing that schedule.

The relationship between supervision and scheduling could be exploited in two directions. On one hand, supervisory control methods could be applied to obtain schedulers. A scheduler would correspond to a supervisor and would run in real time. Its function would be to enable a set of operations that should be executed at a given time. On the other hand, scheduling methods could be used as supervision methods for PNs.

In the literature, the schedulability problem is typically approached for programs running on a single computational resource (processor). The schedulability problem for multiple computational resources is considered in [33], [34]. In this case each resource is allocated a number of processes and the problem is to find an appropriate schedule for each resource. Thus, instead of a single sequential schedule, there are several concurrent schedules, one for each resource. The approach proposed in [34] is to search for a sequential schedule that produces an appropriate concurrent schedule when projected on the subsystems associated with each resource. To illustrate this idea, consider the example of Figure 7 representing a program consisting of two processes. One process is defined by the transitions \( a, \ldots, g \) and the places \( p_1, \ldots, p_5 \), and the other process by the transitions \( d, e, h, i, j, k \) and the places \( p_6, \ldots, p_{10} \). Assuming that each process runs on a different resource, by projecting the sequential schedule of Figure 8(a) on the set of transitions of each process, we obtain the two schedules of Figure 9. In SC terms, the problem corresponds to decentralized SC [2], [41]. For instance, in the example of Figure 7, the subsystems for decentralized control would be defined as follows. For the first subsystem, the sets of controllable and observable transitions would be \( T_{c1} = \{a, d, \ldots, g\} \) and \( T_{o1} = \{a, \ldots, g\} \), respectively. For the second subsystem, the sets would be \( T_{c2} = \{h, \ldots, k\} \) and \( T_{o2} = \{d, e, h, \ldots, k\} \). Then, the scheduling problem would be similar to the problem of finding a decentralized supervisor enforcing liveness and boundedness. Note that decentralized supervision of PNs for liveness enforcement is an interesting direction of research. Figure 9 shows a possible decentralized PN supervisor implementing the two schedules shown in the figure.

**V. CONCLUSION**

Petri nets and related models have been used for the modeling of program specifications. Moreover, various problems related to the synthesis of programs can be considered in the setting of supervisory control. On one hand, some of the methods developed in the context of computer science could be of interest for supervisory control. On the other hand, supervisory control methods could be adapted for program
synthesis. Specific needs encountered in the context of program synthesis provide opportunities for the development of new supervisory control methods.

REFERENCES