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 net models 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 μ 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

M. V. Iordache is with the School of Engineering & Engineering Technology, LeTourneau University, Longview, TX 75607, USA MarianIordache@letu.edu

P. J. Antsaklis is with the Department of Electrical Engineering, University of Notre Dame, Notre Dame, IN 46556, USA antsaklis.1@nd.edu

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_i, q_i indicates how many times t_i is fired at the firing instance. In particular, assuming no concurrency, only one transition t_i 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 ρ denote the labeling function that associates with each transition t one event $\rho(t)$. Note that under supervision, a transition t of the plant may be fired only if there is an enabled transition t_s of the supervisor such that $\rho(t) = \rho(t_s)$.

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 μ 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

$$L\mu + Hq + Cv \leq b \tag{1}$$

where L, H, C, and b are integer matrices of appropriate dimensions. Note that (1) requires that all reachable states of the plant satisfy $L\mu + Cv \leq b$ and that a firing vector q' is fired only if (1) is satisfied for all q in the range $0 \leq q \leq q'$.[1] 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 $q_3 + q_9 + \mu_1 \leq 1$ and $q_{10} + q_{11} \leq \mu_4 + \mu_5$, respectively.

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

$$\bigvee_{k=1}^{n} [L_k\mu + H_kq + C_kv \leq b_k] \tag{2}$$

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

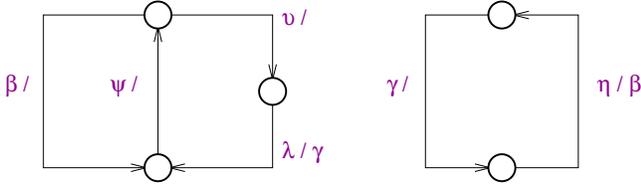[1]Given two vectors x and y, $x \leq y$ means $x_i \leq y_i$ for all indices i.

Fig. 1. A network of CFSMs.

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 η/β occurs when the event η is present and upon its occurrence it generates the event β. The transition labeled by $\beta/$ does not fire at the same time as the one that generates the event β. Rather, the CFSM containing the transition $\beta/$ checks whether the event β 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_β and p_γ indicate the presence of the events β and γ when they contain at least one token. However, for consistency with the CFSM models, the places p_β and p_γ 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 $\gamma\alpha + \overline{\beta}$ 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
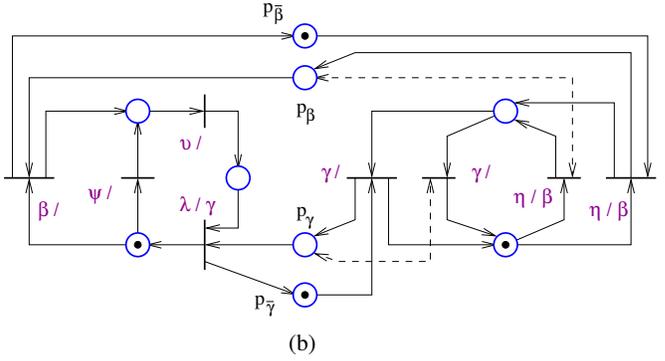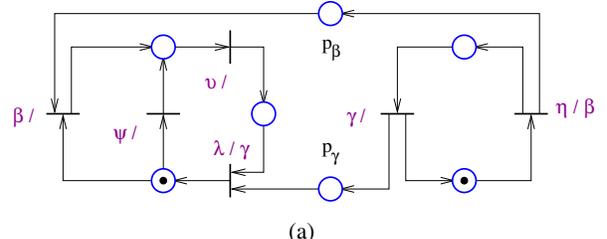

(a)

(b)

Fig. 2. PNs representing the CFSMs of Figure 1. A dashed bidirectional arrow between a place p and a transition t indicates a self-loop, that is, $p \in \bullet t \cap t\bullet$.
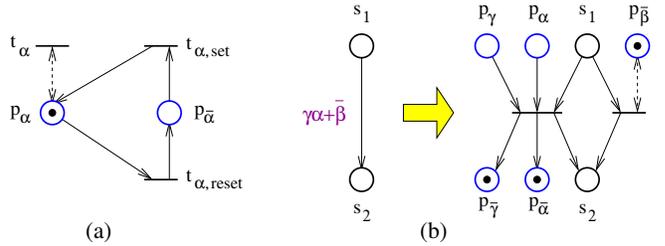

Fig. 3. (a) PN component for the transmission of an event α. $\mu(p_\alpha) = 1$ when the event α is present and else $\mu(p_{\overline{\alpha}}) = 1$. t_α and $t_{\alpha,set}$ have the same label. Note that when α is absent and α occurs, $t_{\alpha,set}$ is fired. Further, when α is present and α occurs again, t_α is fired. (b) Illustration of the PN implementation of a transition that is fireable when a logic expression is true.

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, and $down$, 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.
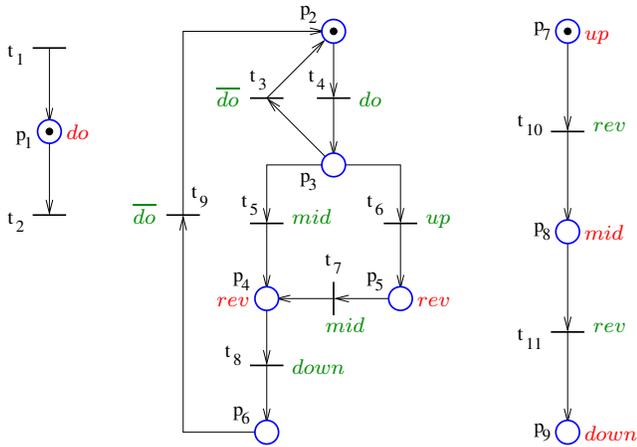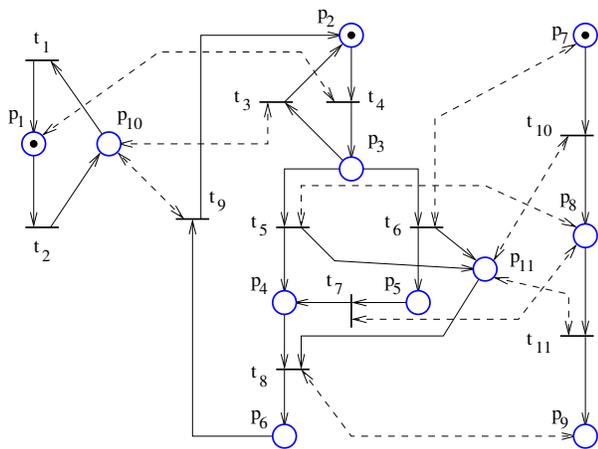
Fig. 4. Condition system example.

Fig. 5. PN representing the condition system. Bidirectional dashed arrows indicate self-loops.

transition cannot fire unless condition enabled. For instance, a self-loop including t_6 and p_7 guarantees that t_6 fires only if the condition up is satisfied. A more complex construction is necessary in order to implement the requirement that t_{10} and t_{11} fire only if p_4 or p_5 is marked. The requirement can be expressed by $q_{10} + q_{11} \leq \mu_4 + \mu_5$. By implementing this constraint we obtain the place p_{11} of Figure 5. Another situation that requires special attention is when a transition is labeled by the complement of a condition. For instance, t_3 and t_9 are labeled by $\overline{do}$. However, this enabling constraint can be expressed by $q_3 + q_9 + \mu_1 \leq 1$, from which we obtain the place p_{10} of Figure 5. Note that the solution would be more complex if more than one place would generate the condition do. For instance, if there were another place p_x generating do, the enabling constraint would be expressed by a predicate $[q_3 + q_9 + \mu_1 \leq 1] \vee [q_3 + q_9 + \mu_x \leq 1]$. The construction of an equivalent PN is still possible by using the method for disjunctive constraints of [5].

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].
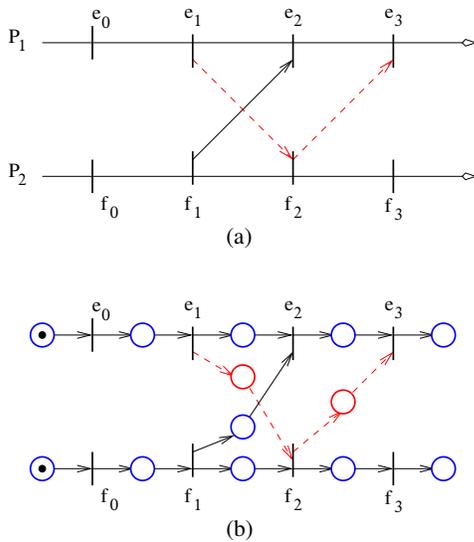
Fig. 6. (a) Two processes and their synchronization; (b) PN model.

Fig. 7. PN model.

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 $e_{i,0} e_{i,1} \cdots$. 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 $|\epsilon_1 + \epsilon_2 + \epsilon_3 - \phi_1 - \phi_2| \leq 1$, where ϵ_i denotes the number of occurrences of the event e_i and ϕ_i the number of occurrences of the event f_i. The predicate is enforced by adding the synchronization arrows drawn with dashed lines.

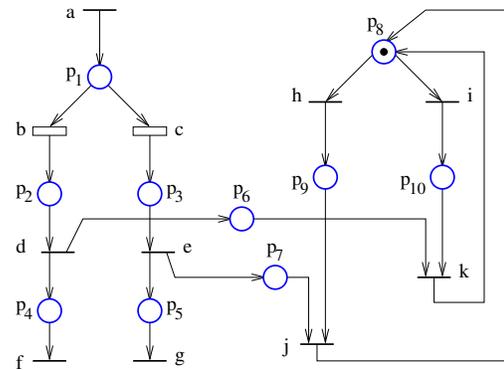The predicate control problem can be stated in terms of PNs, as illustrated in Figure 6(b). A synchronization arrow corresponds to a place connected between the two transitions of the arrow. Thus, generating synchronization arrows corresponds to designing a monitor based supervisor, where each monitor 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 when the schedule is computed, it is not known which transition should be fired.
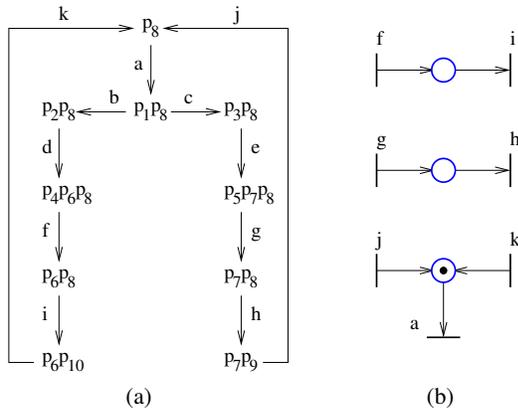
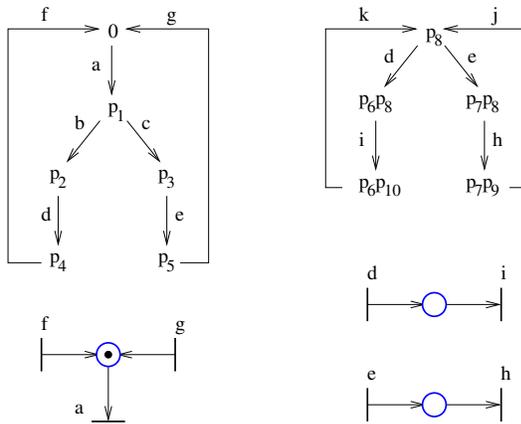**4997**

Fig. 8. Sequential schedule and supervisor.

Fig. 9. Concurrent schedules and supervisors.

In our example, the transitions b and c are uncontrollable. In principle, it may be possible to have a control structure whose outcome can be controlled at the time the schedule is computed. Then, the corresponding transitions would be controllable. This situation is illustrated in our example by the transitions h and i, which are controllable. A possible schedule for our example is shown in Figure 8(a).

Reachability based methods could be used to derive a schedule, such as in [14], [34]. An invariant based approach is considered in [38] and structural conditions for the existence of a schedule are obtained in [37]. There are also results for real-time specifications, based on an approach for hybrid controller synthesis [39] and heuristics [36].

As noticed in the literature [40], a schedule could be seen as a supervisory policy enforced by a supervisor. For instance, if we represent the schedule of Figure 8(a) as a state machine, the state machine would be a supervisor enforcing the schedule on the PN of Figure 7. In general, obtaining a supervisor by representing the schedule as a state machine results in a rather complex supervisor with many places. Indeed, a schedule represents a portion of the reachability tree of the PN model of the system. However,

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

4998

synthesis. Specific needs encountered in the context of program synthesis provide opportunities for the development of new supervisory control methods.

REFERENCES

[1] T. Murata, "Petri nets: Properties, analysis and applications," in *Proceedings of the IEEE*, Apr. 1989, pp. 541–580.

[2] M. V. Iordache and P. J. Antsaklis, *Supervisory Control of Concurrent Systems: A Petri Net Structural Approach*. Birkhäuser, 2006.

[3] L. E. Holloway, B. H. Krogh, and A. Giua, "A survey of Petri net methods for controlled discrete event systems," *Discrete Event Dynamic Systems*, vol. 7, no. 2, pp. 151–190, 1997.

[4] M. V. Iordache and P. J. Antsaklis, "Supervision based on place invariants: A survey," *Discrete Event Dynamic Systems*, vol. 16, pp. 451–492, 2006.

[5] ——, "Petri net supervisors for disjunctive constraints," in *Proceedings of the 2007 American Control Conference*, 2007, pp. 4951–4956.

[6] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Kluwer Academic, 1993.

[7] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara, *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Publishers, 1997.

[8] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli, "A formal specification model for hardware/software codesign," in *Proceedings of the International Workshop on Hardware-Software Codesign*, 1993.

[9] L. E. Holloway, X. Guan, and R. Sundaravadivelu, "Automated synthesis and composition of taskblocks for control of manufacturing systems," *IEEE Transactions on Systems, Man, and Cybernetics: Part B*, vol. 30, no. 5, pp. 696–712, 2000.

[10] J. Ashley and L. Holloway, "Automated control, observation, and diagnosis of multi-layer condition systems," *Studies in Informatics and Control*, vol. 16, no. 1, 2007.

[11] D. Shewa, J. Ashley, and L. Holloway, "Spectool 2.4 beta: A research tool for modular modeling, analysis, and synthesis of discrete event systems," in *Proceedings of the 8th International Workshop on Discrete Event Systems*, 2006, pp. 477–478.

[12] "Spectool homepage," http://www.engr.uky.edu/~holloway/spectool.

[13] X. Guan and L. Holloway, "Supervisory control of contradictions in hierarchical task controllers," in *Proceedings of the 37th Annual Allerton Conference on Communication, Control, and Computing*, Urbana-Champaign, 1999.

[14] B. Lin, "Software synthesis of process-based concurrent programs," in *DAC '98: Proceedings of the 35th annual conference on Design automation*. ACM Press, 1998, pp. 502–505.

[15] S. Shatz, K. Mai, D. Moorthi, and J. Woodward, "A toolkit for automated support of Ada-tasking analysis," in *Proceedings of the 9th International Conference on Distributed Computing Systems*, 1989, pp. 595–602.

[16] S. Vercauteren, D. Verkest, G. de Jong, and B. Lin, "Derivation of formal representations from process-based specification and implementation models," in *Proceedings of the 10th International Symposium on System Synthesis (ISSS '97)*, 1997, pp. 16–23.

[17] K. Barkaoui and J.-F. Pradat-Peyre, "Verification in concurrent programming with Petri nets structural techniques," in *High-Assurance Systems Engineering Symposium*, 1998, pp. 124–133.

[18] T. Murata, B. Shenker, and S. M. Shatz, "Detection of Ada static deadlocks using Petri net invariants." *IEEE Transactions on Software Engineering*, vol. 15, no. 3, pp. 314–326, 1989.

[19] C. Stehno, "Real-time systems design with PEP," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, Katoen, J.-P. and Stevens P., Eds. Springer-Verlag, 2002, vol. 2280, pp. 476–480.

[20] E. Best and R. Hopkins, "B(PN)2 - a basic Petri net programming notation," in *PARLE*, ser. Lecture Notes in Computer Science, Bode, A., Reeve, M., and Wolf, G., Eds. Springer-Verlag, 1993, vol. 694, pp. 379–390.

[21] H. Fleischhack and B. Grahlmann, "A compositional Petri net semantics for sdl," in *Application and Theory of Petri Nets*, ser. Lecture Notes in Computer Science, Desel J. and Silva M., Eds. Springer-Verlag, 1998, vol. 1420, pp. 144–164.

[22] B. Grahlmann, M. Moeller, and U. Anhalt, "A new interface for the PEP tool - parallel finite automata," in *Workshop Algorithmen und Werkzeuge für Petrinetze*, ser. AIS, Desel, J. and Fleischhack, H. and Oberweis, A. and Sonnenschein, M., Ed., 1995, vol. 22, pp. 21–26.

[23] E. Best, R. Devillers, and M. Koutny, "Petri nets, process algebras and concurrent programming languages," in *Lectures on Petri Nets II: Applications*, ser. Lecture Notes in Computer Science, W. and Rozenberg, G., Eds. Springer-Verlag, 1998, vol. 1492, pp. 1–84.

[24] E. Best, W. Fraczak, R. Hopkins, H. Klaudel, and E. Pelz, "M-nets: an algebra of high-level petri nets, with an application to the semantics of concurrent programming languages," *Acta Informatica*, vol. 35, no. 10, pp. 813–857, 1998.

[25] A. Burns, A. J. Wellings, F. Burns, A. M. Koelmans, M. Koutny, A. Romanovsky, and A. Yakovlev, "Towards modelling and verification of concurrent ada programs using petri nets," *Computer Systems Science and Engineering*, vol. 16, no. 3, pp. 173–182, 2001.

[26] M. Notomi and T. Murata, "Hierarchical reachability graph of bounded petri nets for concurrent-software analysis," *IEEE Transactions on Software Engineering*, vol. 20, no. 5, pp. 325–336, 1994.

[27] S. Shatz, S. Tu, T. Murata, and S. Duri, "An application of Petri net reduction for Ada tasking deadlock analysis," *IEEE Trans. on Parallel and Distributed Systems*, vol. 7, no. 12, pp. 1307–1322, 1996.

[28] M. Lemmon and K. He, "Supervisory plug-ins for distributed software." in *Proceedings of the Workshop on Software Engineering and Petri Nets*, Pezze, M. and Shatz, M., Eds. University of Aarhus, Department of Computer Science, 2000, pp. 155–172.

[29] M. Lemmon, K. He, and S. Shatz, "Dynamic reconfiguration of software objects using Petri nets and network unfolding," in *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, 2000, pp. 3069–3074.

[30] Y. Wang, T. Kelly, M. Kudlur, S. Mahlke, and S. Lafortune, "The application of supervisory control to deadlock avoidance in concurrent software," in *Proceedings of the 9th International Workshop on Discrete Event Systems*, 2008, pp. 287–292.

[31] N. Mittal and V. Garg, "Debugging distributed programs using controlled re-execution," University of Texas at Austin, Tech. Rep., 2000, technical report TR-PDS-2000-002 of the Parallel & Distributed Systems group.

[32] A. Tarafdar and V. K. Garg, "Predicate control: synchronization in distributed computations with look-ahead," *Journal of Parallel and Distributed Computing*, pp. 219–237, 2004.

[33] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. 36, no. 1, pp. 24–35, 1987.

[34] J. Cortadella, A. Kondratyev, L. Lavagno, and Y. Watanabe, "Quasi-static scheduling for concurrent architectures," in *Third International Conference on Application of Concurrency to System Design (ACSD 2003)*. IEEE Computer Society, June 2003, pp. 29–40.

[35] J. Cortadella, A. Kondratyev, L. Lavagno, C. Passerone, and Y. Watanabe, "Quasi-static scheduling of independent tasks for reactive systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2005.

[36] P.-A. Hsiung, "Formal synthesis and code generation of embedded real-time software," in *CODES '01: Proceedings of the 9th International Symposium on Hardware/Software Codesign*. ACM Press, 2001, pp. 208–213.

[37] C. Liu, A. Kondratyev, Y. Watanabe, J. Desel, and A. Sangiovanni-Vincentelli, "Schedulability analysis of Petri nets based on structural properties," in *IEEE International Conference on Application of Concurrency to System Design*, 2006.

[38] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli, "Synthesis of embedded software using free-choice Petri nets," in *Proceedings of the 36th Design Automation Conference (DAC-99)*, 1999, pp. 805–810.

[39] K. Altisen, G. Gosler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine, "A framework for scheduler synthesis," in *20th IEEE Real-Time Systems Symposium*. IEEE Computer Society, 1999, pp. 154–163.

[40] K. Strehl, L. Thiele, D. Ziegenbein, R. Ernst, and J. Teich, "Scheduling hardware/software systems using symbolic techniques," in *Internat. Workshop on Hardware/Software Codesign*, 1999, pp. 173–177.

[41] M. V. Iordache and P. J. Antsaklis, "Decentralized supervision of Petri nets," *IEEE Transactions on Automatic Control*, vol. 51, no. 2, pp. 376–381, 2006.

4999