Synthesis of Concurrent Programs Based on Supervisory Control

Marian V. Iordache
School of Engineering and Eng. Tech.
LeTourneau University
Longview, TX 75607, USA
E-mail: MarianIordache@letu.edu

Panos J. Antsaklis
Department of Electrical Engineering
University of Notre Dame
Notre Dame, IN 46556, USA
E-mail: Antsaklis.1@nd.edu

Technical Report
September 2009

Abstract–This document describes an application of the supervisory control (SC) methods to the synthesis of concurrent programs and presents current work on this topic of research. In particular, special attention is given to the development of software that applies SC to program synthesis. This work is motivated by the difficulties encountered in writing correct programs in the context of concurrency. Writing correct programs is essential for the development of software applications as well as for all other engineering applications in which formal languages are used for system design. In the context of concurrency, SC can be help by addressing issues such as mutual exclusion, liveness, and fairness. In the approach proposed here, SC is applied to Petri net (PN) models of concurrent processes. Then, the resulting control logic is converted to code. PNs are formal models developed in Computer Science for the modeling of concurrent systems. In Control Systems, PNs have been used in the context of the SC of discrete event systems and powerful theoretical results have been developed. However, these results have not yet been applied to Computer Science problems for which PNs were created. The main objective of this research work is to apply SC tools to the automatic synthesis of programming code based on a high-level program specification. The goal is to reduce the programming effort by having more of the higher level requirements implemented automatically. On one hand, the automatically generated code is correct by construction and on the other hand, the programmer has only to manage simpler high-level specifications.

1 Overview

Programming is essential in the design of contemporary engineering system. However, the development of correct programs is known to be difficult and expensive, especially in the context of concurrency. Thus, tools that can automate the programming process to a higher degree are of interest, in order to reduce the programming effort and increase the number of features of the product that are correct by construction. This document introduces a project dealing with the application of supervisory control methods to the synthesis of programs that are correct by construction.

The supervisory control (SC) theory has been developed in the context of control systems, for discrete event systems (DESs). Given a DES model, called *the plant*, and a specification, SC methods are applied to design a *supervisor* that restricts the operation of the plant so that the specification is enforced. The supervisor ensures that the operation of the plant satisfies at all times the specification, subject to the constraints imposed by the plant. Typically, these constraints refer to the events that can be controlled or observed by the supervisor.

Petri nets (PNs) represent a class of DES models, that along with automata have been used in the context of the SC. PNs are used here as the DES models of the specifications and of the plant. PNs are a natural choice for the modeling of concurrency and there are numerous results on the SC of PNs. Moreover, it is also possible to benefit from the SC results obtained for automata, as automata are a special case of PNs.

The approach of this project is to apply the SC in order to generate automatically segments of programming code. The approach is illustrated in Figure 1. First, the plant and specification of the SC are extracted from a higher level specification, written in a high level specification language (HLL). Next, the SC is applied to generate the supervisor. Finally, the plant and the supervisor are translated to programming code. All these steps are carried out transparently and automatically, based on the higher level specification. The benefit of this approach is that the programmer would focus on concise high level descriptions, instead of the details of the more complex lower level implementation. Of course, not every type of specifications can be handled by a SC approach. However, the problem of automating to a higher degree program synthesis raises issues intrinsically related to SC, since various high level requirements, such as fairness, absence of deadlocks, and mutual exclusion, can be seen as SC specifications.

Concerning the application of SC methods, we note that many of the problems that SC addresses are of considerable difficulty. Moreover, sometimes there are no known algorithms that are satisfactory in every respect. For instance, by searching all states reached by a PN it is possible to design supervisors that are optimal with respect to various criteria. However, optimality comes to the expense of high computational complexity, due to the fact that the number of reachable states could be extremely large, if finite. Thus, it may be of interest to use instead methods that do not search the reachable states but analyze the structure of PNs. However, such methods may be suboptimal. Moreover, they may only be applicable to certain special classes of PNs. Thus, one of the long term objectives of this project is to implement various SC methods and determine criteria to select between methods depending on context. In spite of such difficulties, in view of the relation of SC type problems to program synthesis, this work appears to be necessary in order to achieve a high degree of automation in the development of software.

The paper is organized as follows. Related literature results are discussed in section 2. The PN representation of programs is described in section 3. Issues related to the design of the high level

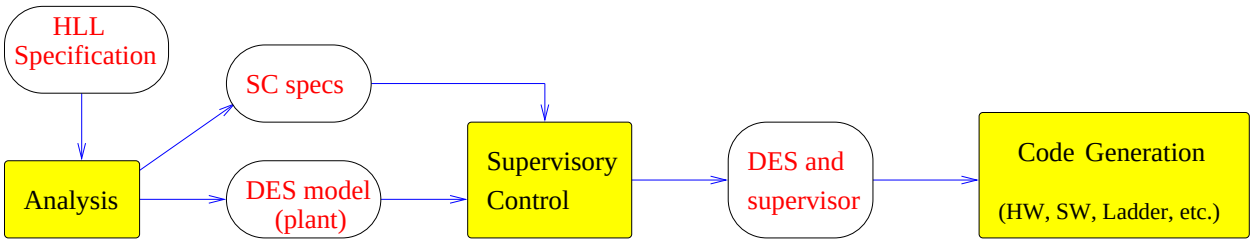

Figure 1: A DES plant and an SC specification are extracted from the user specification. The SC specification describes the objective of the supervision. After the SC tools are applied, the resulting closed-loop DES can be transformed into programming code. While focusing here on program synthesis, the same approach could also be used in other contexts, such as to generate code for hardware implementation or ladder logic for manufacturing applications.

specification language are addressed in section 4. An example is given in section 5. Supervisory control methods are discussed in section 6. Finally, an approach to code generation is described in section 7. Additional information on this project can be found in [1]. The reader is referred to the appendix for an introduction to PNs.

2 Literature Review

Due to the popularity of multicore microprocessors, concurrent programming is a topic of growing importance. There are tools that can be used to accelerate code execution on multicore processors, such as [44]. Nonetheless, the development of concurrent programs is still considered to be difficult. The application of the SC, as proposed here, could help by automating certain aspects of the development of concurrent programs, especially the aspects related to the coordination of concurrent tasks. Note that the development of concurrent specifications remains necessary even when there are good tools for the efficient execution of sequential code on multiprocessor systems. To illustrate this necessity, consider the example of the dining philosophers[1]. A possible sequential solution would be to implement a token ring protocol in which the philosopher that has the token may eat. A tool converting sequential code to parallel code could make very efficient implementations of the processes associated with a philosopher taking the forks or eating or thinking. However, it will not change the protocol: the generated program would still allow only one philosopher to eat at a time. In contrast, if a parallel specification is developed, two philosophers may be allowed to eat at the same time. Further, tools converting sequential code to parallel code would remain very useful, as they could improve the execution time of the sequential segments of code (such as those dealing with picking up the forks, eating, and thinking). Note that the our project deals with the development of concurrent specifications and not with the conversion of sequential code to parallel code.

[1]In this classic synchronization problem five philosophers sit around a circular table. A philosopher can either eat, think, or be hungry. There is a fork between each two adjacent philosophers. In order to eat, each philosopher needs the two forks at his left and right. He may take a fork only if it is not already taken by a neighbor. When hungry, a philosopher may begin to eat only when he has both forks. When he is done eating, he puts down the forks to his right and left. The problem is to find a strategy allowing the philosophers to think and eat without reaching deadlock or starvation.

In the literature, closely related to the SC theory is the approach for program synthesis for reactive systems, as in [69, 90] and the references therein. The problem is to synthesize a program based on a specification described in temporal logic. The specification must be satisfied for all possible inputs of the environment. In terms of the SC terminology, a program would correspond to a supervisor. For the most part, this work on program synthesis has not yet been reduced to practice [68]. In this project we use PNs instead of temporal logic for the SC specifications. However, in the longer term, other types of methods could also be incorporated, as appropriate, to increase the area of applicability of this project.

A significant amount of work has been done on the modeling and analysis of concurrent programs using PNs, such as in [14] and references therein. A software tool PEP has been created for the development, verification, and simulation of parallel programs [2, 45, 99]. The structure of the PEP tool has been described at three levels: the development level, the net level, and the analysis level. At the development level user input is accepted in the form of programs written in one of the languages B(PN)^2 [16] and SDL [39] or as a parallel finite automata [46]. At the net level, specifications are represented as high level nets, Petri box calculus expressions [14], and low-level PNs. M-nets [15, 17] represent the high level nets and safe[2] PNs the low-level PNs. At the analysis level, model checking and other verification methods can be applied to the low-level PNs. Comparing our approach with the approach of the PEP tool, note that the input is described by a specification language in our work and by a low-level language in PEP. Our approach could be used to assist the programmer in writing a low-level specification, while the PEP tool can be used to verify a low-level specification.

The scheduling problem, dealing with the execution order of tasks based on a concurrent specification, has been approached based on PN models in [33, 32, 55, 73, 77, 78, 95, 96, 103, 114]. Typically the results are on the sequential execution of concurrent programs, as it is the case in platforms with a single execution resource. However, the parallel execution has also been considered [33, 73]. There are also results for real-time specifications, based on an approach for hybrid controller synthesis [7] and heuristics [55]. Typically reachability analysis is used for synthesis, though there are also structural results, such as in [78]. Note that a schedule can be seen as the supervisory policy enforced by a DES supervisor, such as in [100]. Thus, these results could be applied in the SC framework of our project. However, note that in our project SC is to be used to assist the programmer in writing concurrent programs, not just in solving the scheduling problem based on a *given* concurrent program. Further, the intent is to focus on structural SC methods, in an attempt to avoid the state explosion problem of reachability based methods.

There is also work on the tasking analysis of programs written in the Ada language. Verification based on PN modeling appears in [13, 20, 82]. There are both structural methods [13] as well as reachability based approaches [82, 83, 108]. Methods that could be used to model concurrent programs with PNs appear in [97, 109]. Note that our project is on correct-by-construction synthesis, not on verification.

Related is also the work on hardware/software codesign of [9, 10]. There, the specification is written in a language such as Esterel [49], from which a network of codesign finite state machines (CFSMs) [29] is extracted. It is interesting to notice that networks of CFSMs could be modeled by safe PNs. Compared to our project, while individual processes will be modeled by state machines, the parallel composition of the process models will be a PN not necessarily safe. Moreover, we intend

[2] A safe PN is a PN in which all markings are less or equal to one.

to use specifications at a higher level. In our project, the specification language is to allow some declarative language features, since part of the lower level code is to be generated automatically.

Recently, research work has been done on detecting and correcting deadlock situations in software based on PN models and methods for liveness enforcement in PNs [110]. The approach can be described as follows. Given a program, a PN model is extracted. Then, a liveness enforcing supervisor is generated. Finally, the liveness enforcement supervisor is implemented by additional lines of code in the original program. This approach has been implemented in the software tool GADARA. Just as in our approach, SC methods for liveness enforcement are applied. However, in our project we deal with program synthesis instead of programs that are already written. Thus, in our project SC is applied not only for liveness enforcement but also to automate code generation for other requirements that can be expressed in terms of SC specifications.

While program synthesis is in general a difficult topic, real-time constraints present an additional challenge for embedded applications. The time issue is perceived as a potential obstacle in the development of the next generation of networked embedded systems [72]. The SC framework of our work appears to be suitable for programming with real-time constraints. However, we currently focus on untimed specifications and postpone real-time specifications for future work. In the literature, timed PNs have been used for verification [7, 21, 41] and also for SC [27, 42, 94, 93].

Numerous SC methods have been proposed for the supervision of PNs. Survey papers have been published [52, 61]. The SC methods differ in the assumptions they make, the type of specifications they consider, and the complexity of the computations. Thus, the context determines which are the most appropriate methods. Therefore, in order to better exploit the power of the SC results, the SC step in our approach (Figure 1) will not be limited to only one SC approach. While our current focus is on the SC approaches of [62], incorporating also other SC methods is of interest for the future.

To some extent, the application of SC methods to software engineering has been considered for some time [74, 75]. Some of the methods proposed in CS, such as predicate control for distributed computations [106] and the aforementioned scheduling approaches, can be seen as SC methods [64]. Moreover, some of the approaches used to generate control software are related to the SC. Thus, an approach for the generation of control software based on condition system models appears in [8, 51, 98]. Given a condition system model and a specification language describing a sequence of states that the system should follow, control software is automatically generated [4]. Specific SC problems in this context are addressed in [48]. Control software can also be generated using the tool Supremica [65, 5] based on finite automata specifications and methods. Note that in our project, by using PN models, it is possible to take advantage of both PN methods and automata methods, as automata represent the reachability space of PNs. Further, compared to [8, 51, 98], we intend to use more general specifications.

Among the software tools developed for SC, the following could be mentioned. Of special interest for this project is the SC toolbox [57] developed in Matlab, containing SC methods for PNs. This toolbox has been converted to C and has been included in the software developed for this project. There is also other work on software tools for the SC of PNs [3, 40]. Further, the condition system tool of [51, 4] can also be seen as a PN software tool, due to the relation of condition systems to P/T nets. Some of the SC tools relying on automata models are TCT [111], UMDES [70], and Supremica [5]. Controller synthesis based on temporal logic specifications can also be done [47].

3 Petri Net Representation of Programs

Since this project involves SC methods for PNs, of special interest here is the precise way in which programs are represented using PNs. Subsection 3.1 describes several ways in which PNs could represent programs. A comparison of PNs and automata is included in subsection 3.2. Then the specific way in which programs are represented in our project is described in section 3.3.

3.1 Possible Approaches to PN Modeling

There are several ways in which PN models could be obtained.

1. Extract a PN model of a low-level program.
   (a) Extract a complete PN model of a program.
   (b) Extract only the structure of the program as a PN structure.
2. Start with a PN model.
   (a) Start with a PN structure in which program segments are associated with each place.
   (b) **Develop a specification language compatible with a PN structure.**

The approach 2(b) is the one we intend to follow. The approach 2(a) is also of interest, as will be seen later. Next, we discuss in more detail these four approaches and their relationship to this project.

Concerning the approach 1(a), note that finite PNs cannot completely model arbitrary programs, as they do not have the power of a Turing machine. Programs with a bounded number of states, as well as some programs with an unbounded number of states, can be modeled by finite PNs. For certain programming languages, an approach allowing to convert programs to safe PNs is available [14] and implemented in the PEP tool [2]. The PN is finite if the variables are defined on a finite domain [16]. The approach 1(a) should not be confused with the approach 2(b), which deals with a high level specification language (HLL), not with a low-level programming language. Recalling the distinction between the two, an HLL describes objectives (the what), while a low-level language describes objective implementations (the how to).

Less complex PN models are obtained if only the structure of the program is extracted. This is the approach 1(b). In this approach, we deal with a high-level PN (HPN) in which the places and/or transitions are labeled by the operations they represent. Both sequential and concurrent execution of programs could be represented, where the latter may be obtained from the precedence graph. (Precedence graphs are discussed, for instance, in [89].) This modeling approach has been used to represent concurrent C programs in [77] and the tasking behavior of Ada programs in [13, 82, 97]. An example of an HPN is shown in Figure 2.

Still another possibility is to write from the beginning the program in an HPN format in which the places are associated with program segments. This is the approach 2(a). For state machines, this idea has been implemented to some extent in tools such as Stateflow of Matlab and in specification languages such as SDL and UML. This idea is attractive because some problems are easily solved by first defining states and transitions between states. For instance, such problems may arise in

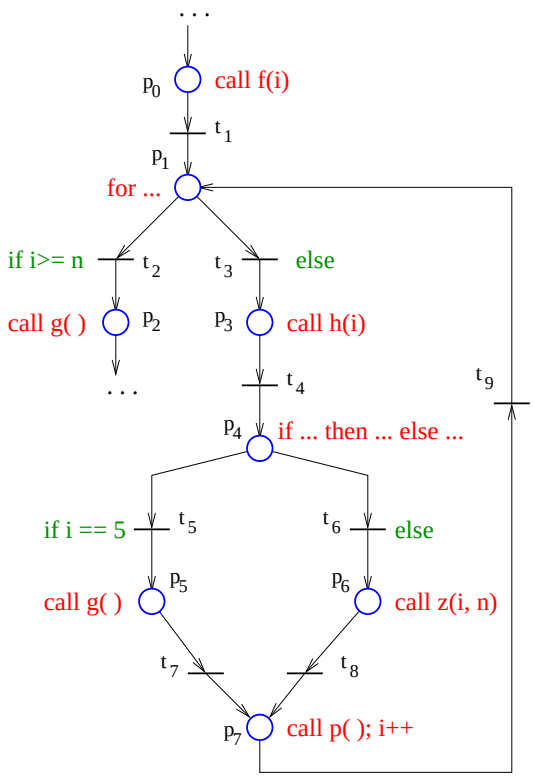

Figure 2: Example of an HPN.

the context of communication protocols. Note that not only state machines but also PNs are of interest in communication protocol problems [34, 105, 87, 18].

This project uses the approach 2(b). Based on a specification written in an HLL, a PN model and SC specification are extracted (Figure 1). As mentioned in section 4, the HLL may contain segments of lower-level code. Such segments of code will be associated with places of the extracted PN. In this sense, the PN model can be seen as one of the HPNs of the approach 2(a). A detailed description of the PN representation used in this project is given in section 3.3.

3.2 Other DES Models

This project involves using PN models for the SC. Most literature on the SC uses automata for the DES models. The choice of PNs for this project has several advantages. First, PNs are natural models of concurrent processes. Further, by using PNs, it is possible to take advantage not only of efficient automata methods but also of PN methods. A PN model can be converted to an equivalent automaton by means of the *reachability graph*. Conversely, automata can be seen as a particular type of *safe* PNs. However, PNs typically are considerably more compact representations of systems than automata. Indeed, for some PNs the equivalent automata are not even finite. Thus, in principle, it may be possible to have an exponential complexity PN method that is faster than a polynomial complexity method on the equivalent automaton. It should also be mentioned that typically PN models are easier to obtain than automata models, as the composition of automata components could be seen as the generation of the reachability graph of a PN.

Safe PNs are PNs for which the the marking vector is a binary vector (has only 0 and 1 elements). Any automata or collection of automata can be seen as a safe PN. Rather than using safe PNs, the more general P/T nets are used in this project. In this way the marking of the PN could be used to model nonnegative integer variables, where the variables do not have to be bounded. In safe PNs the modeling of bounded variables is more complex and the modeling of unbounded variables is not possible. For instance, a variable that equals the difference between the number of parts entering a manufacturing system and the number of parts *successfully* processed is unbounded. Such a variable cannot be modeled in a safe PN or a finite automaton, though it can be easily represented by the marking of a place of a PN.

3.3 Internal Representation of Programs

This subsection describes the precise way in which PNs are used to represent programs in this project. In this project, a program consists of a number of processes running concurrently. The structure of each process is represented by a PN. The places of the PN correspond to operations performed by the process. The transitions may be labeled by conditions, indicating which transition should be taken when there is a choice. Each PN token corresponds to a process and indicates the current state of the process. As a token moves from one place to another, the execution of the process progresses from one set of operations to another. Thus, the various places of the PN correspond to different stages in the execution of the process. Throughout this paper an HPN (high level PN) will denote a PN in which places are labeled with instructions and transitions with conditions. An illustration of an HPN is shown in Figure 2.

In general, PN transitions may have multiple input places and multiple output places. The effect of firing such transitions is made precise by describing the PN structure by means of tuples of the form (p_1, t, p_2), (p, t), and (t, p), where p_1, p_2, and p stand for places and t for a transition.

- A (p_1, t, p_2) tuple indicates that the PN has one arc from p_1 to t and of one arc from t to p_2. Further, when the transition t is fired, a process in the stage p_1 continues with the stage p_2.
- A (p, t) pair indicates that the PN has one arc from p to t. Further, when the transition t is fired, a process in the stage p terminates.
- A (t, p) pair indicates that the PN has one arc from t to p. Further, when the transition t is fired, a new process is created and the process begins in the stage p.

The description above can be applied to PNs with arbitrary weights, since repeated arcs could be used to indicate weights greater than one. For instance, if a transition t involves the tuples (p_1, t), (p_1, t, p_2), (p_1, t, p_2), then the arc (p_1, t) has the weight 3 and the arc (t, p_2) has the weight 2.

Note that when a place p has multiple output transitions, if the transitions are labeled with conditions, a process in the stage p will select the next transition to be fired based on the conditions labeling the transitions. On the contrary, if the transitions do not have conditions and there is no code associated with p to select the next transition, the place p is said to be *nondeterministic*. For a nondeterministic place the choice of the next transition is made by the supervisor.

Transitions with a single input place are fired immediately, unless controlled by a supervisor process. However, transitions controlled by a supervisor or involving more than one input place cannot be fired immediately. Rather, a process sends a request to fire such a transition and then waits for permission. After permission is granted, the process goes on with the next stage.

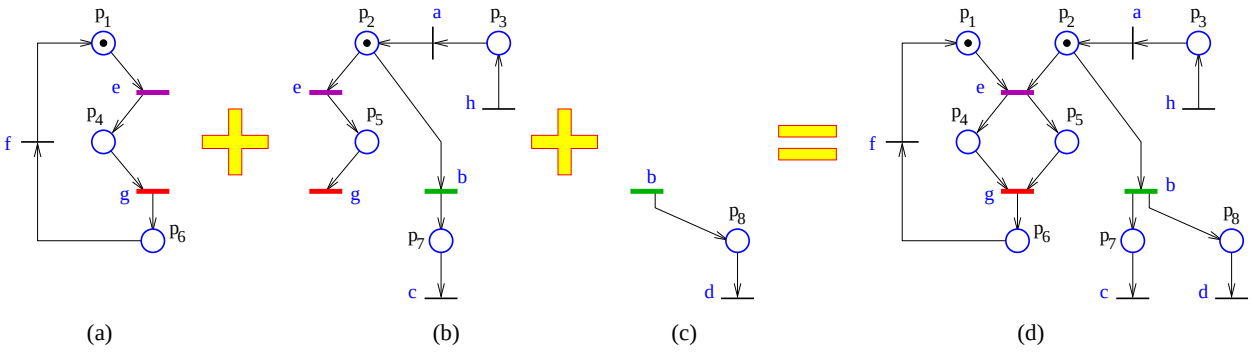

Figure 3: PNs that represent programs are a composition of state machine components. Note that the transitions with the same label are composed.

Note that a PN may have more than one token. Each token of the PN corresponds to a different instance of the program associated with the PN. Note also that multiple tokens in the same place are allowed. This situation corresponds to multiple processes in the same execution stage.

By examining the way PNs are used to represent programs, it becomes clear that the possible stages and transitions of any process form a state machine. That is, for any given initial position of a token in the PN, a state machine will describe the possible stages and transitions of the process associated with the token. Moreover, the PN can be seen as a parallel composition of state machines. For a formal definition of parallel composition the reader is referred to the appendix A, at page 29. An example of composition of state machine components into a PN is shown in Figure 3.

Without loss of generality, it will be assumed that the PNs describing the processes have a state machine structure. Unlike to the typical definition of state machines, note that here arbitrary markings and arbitrary arc weights are allowed. Moreover, note that the parallel composition of state machines can result in an arbitrary PN, which is not necessarily a state machine.

4 The Specification Language

As previously mentioned, in our approach (Figure 1) the specification is given in a high level specification language (HLL). This section describes objectives, constraints, and work related to the design of an HLL.

First, note that the result of software synthesis consists of a number of application processes and a coordinator process (Figure 4). In Figure 4, the *coordinator process* represents the supervisor generated by means of SC. Further, the M processes correspond to process definitions given in the specification. The supervisor (coordinator) exchanges messages with the other processes to ensure that their operation respects the constraints given in the specification file. The number of processes is variable. Some processes may terminate and new processes may be created, as described in the specification. The application is started by starting the supervisor. Then, the supervisor starts other child processes, as determined by the specification. While Figure 4 shows a single coordinator process, a decentralized or distributed approach is possible by using the corresponding SC methods.

Note that a distinction is made here between *processes* and *process types*. Several processes may

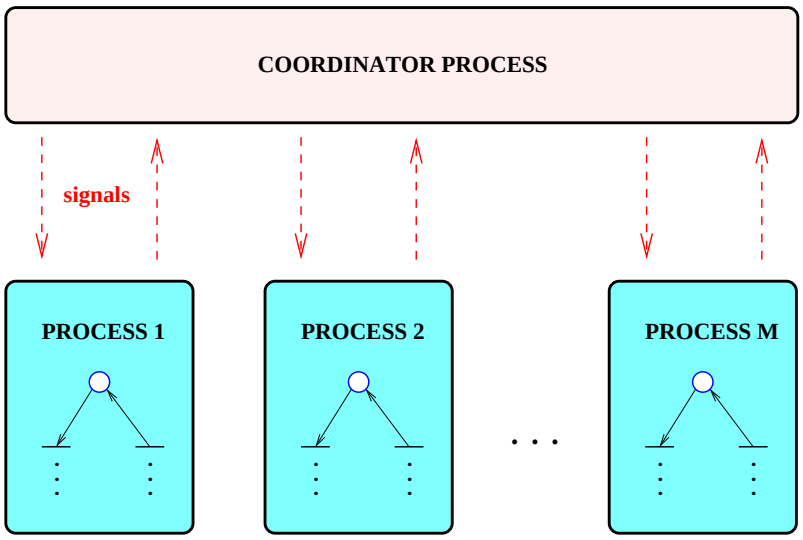

Figure 4: Implementation of the specification.

have the same process type, that is, the same executable code. The code of the supervisor and the code of the process types is generated as shown in Figure 5. While the approach proposed here is not limited to a particular operating system or low level language, the programs are currently developed for Unix using C for the low level language. Note that the files are generated using the architecture shown in Figure 6.

As shown in Figure 6, based on the specification, a number of high level PNs (HPNs) and a supervisory control (SC) specification are extracted. Note that using HPNs instead of place transition nets (P/T nets) is necessary due to the fact that the latter do not have the power of Turing machines. Thus, processes are represented by PNs in which places are associated with low level code and transitions with conditions. However, this means that only the part of the specification expressed by PNs is addressed by the synthesis tools. Thus, the SC tools would only guarantee correctness for the subproblem associated with the PN structure extracted from the HLL program. This is because the SC tools do not take in account the low level code sections. The low level code sections embedded in the specification are simply copied, as appropriate, to the output files. In this respect our approach resembles the approach taken in other program synthesis tools, such as lexical analyzer generators and parser generators [6, 76].

While the HLL will allow sections of low level code, the HLL has to provide other ways to specify the software parts that are difficult to write manually, so that they are generated automatically. Thus, in the context of concurrent programming, the HLL has to address the various synchronization constraints that may be needed.

The role of the HLL is to allow for programs that are both compact and very readable. The HLL should allow users not familiar with PNs to easily generate correct code. Further, a specification written in the HLL is expected to be considerably more compact than the PN representation of the specification and much more compact than the result of the SC and code generation steps. Indeed, the high level specification would not detail how to implement requirements such as mutual exclusion or liveness. Such details would be handled by the SC tools. Thus, the user would focus more on what needs to be done and less on how it should be done. Moreover, since the high level

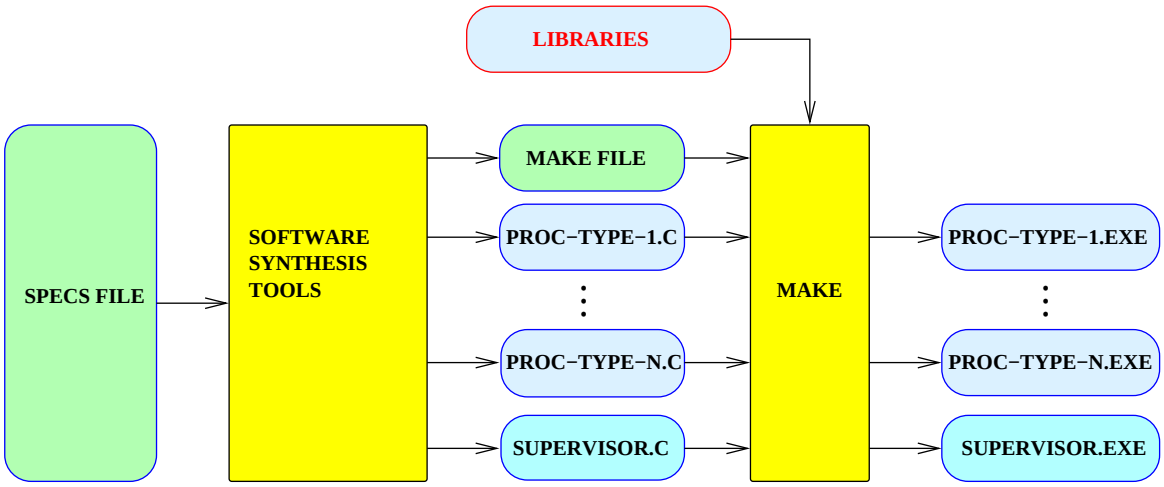

Figure 5: How the software synthesis tools are applied.

specification is more compact, the programmer would have less code to check for errors.

To describe explicitly the PN representation of a program, a low level specification language (LLS) was defined. The LLS describes how specification programs are internally represented in our software. Moreover, the LLS is important for debugging purposes. It also provides a way to include low level descriptions in HLL specifications. Compared to the HLL, the LLS provides an explicit description of the HPNs representing the specification and an explicit description of the constraints. Thus, an HLL specification should be considerably more compact than an LLS specification. Likely, HLL specifications will be translated to LLS specifications. The relationship between the HLL and the LLS is similar to the relationship between a high level language and assembly language.

A simplified description of the LLS is shown in Figure 7. The user defines the process types, where a process type may be external or internal. Note that code is generated only for internal processes. External processes describe constraints on the operation of the application, such as constraints imposed by hardware. Defining all constraints is important. On one hand, this can simplify the operations performed by the SC methods. On the other hand, it enables SC to provide solutions that avoid all deadlock possibilities, as it is known that constraints can create deadlock states.

Each process type is described by a high level Petri net (HPN) structure. In this structure, places are associated with *operations* (such as function calls). Moreover, when a place has multiple output transitions, the transitions are associated with *conditions* (such as conditions in an if-then-else statement).

The LLS allows processes or process groups to be declared. In a process group, each process shares the same HPN. Note that each process of a process group is a token in the HPN. Thus, a process declaration involves an HPN with a single token and a process group declaration involves an HPN with several tokens. The initial markings of the HPNs are defined explicitly. While threads are not used in this paper, it is possible to implement a process group as a single process consisting of several threads.

As previously mentioned, the PN structure of a process type is described by an enumeration of tuples of the form (p_1, t, p_2), (p, t), and (t, p). An example of a specification described in the low

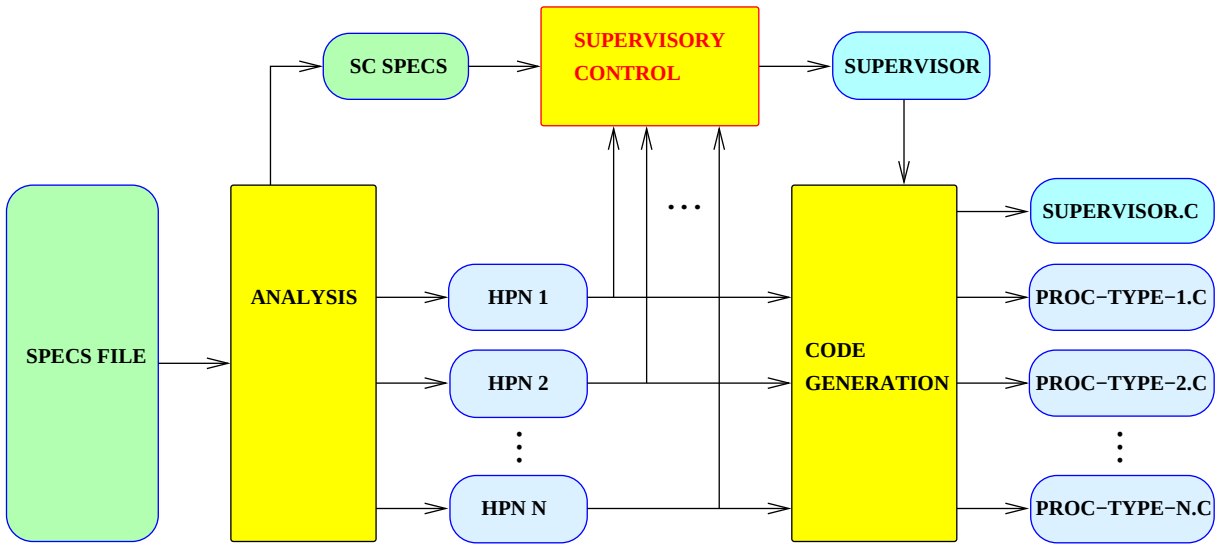

Figure 6: Internal structure. HPN stands for high level PN.

level language is given in the Appendix B at page 31. Note that by describing PNs structures as an enumeration of tuples (p_1, t, p_2), (p, t), and (t, p), the PN is described as a composition of state machine components (e.g. Figure 3).

5 Example

Assume that the HLL describes control software for an assembly process in a manufacturing application. In this process two components A and B are assembled into a component C, as follows. A robot takes a part A and places it on a conveyor, if the conveyor is stopped and no other part A is on the conveyor. Another robot takes a part B and places it on the conveyor at the same location if the conveyor is stopped and no other part B is there. Then, the two parts A and B are assembled. Then, after the conveyor is turned on and the assembled product is removed, a new cycle may begin. The conveyor should not move from the time a part A or B is placed until the time when the parts are assembled.

Referring to Figure 1, the SC specification corresponds to the requirements that only one part A (B) is placed on the conveyor, that the parts are placed when the conveyor is stopped, and that the conveyor should not move from the time a part A or B is placed until the time when the parts are assembled. For the rest, the specification describes the processing sequence and corresponds to the description of the plant.

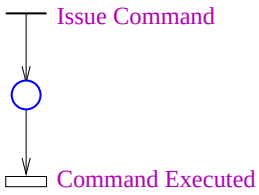

Figure 9: A possible way to model processing delays.

The role of the analysis tool is to extract a PN model of the plant and the SC specification based on a formal description of the specification above. A possible solution is the PN model of the plant is shown in Figure 8 and the SC specification given in the inequalities (1)–(4). In the plant model of Figure 8, the processing sequence is shown to the left and the states of the conveyor to the right. To incorporate the effect of processing delays,

```
<specs> ::= <types> <definitions> <processes> <constraints>

  <types> ::= <process type> | <types> <process type>

    <process type> ::= <external type> | <internal type>

  <definitions> ::= <HPN> | <definitions> <HPN>

    <HPN> ::= <proc-type name> <places> <transitions> <place code> <arcs>

      <arcs> ::= | <arcs> <arc> <condition>

         <arc> ::= <place> <tran> | <tran> <place> | <place> <tran> <place>

  <constraints> ::= <synchronizations> <group constraints>

    <group constraints> ::= |
                         <group constraints> <process group> <local constraints>

      <local constraints> ::= | <local constraints> <constr>

        <constr> ::= <controllability> | <observability> | <liveness> |
                     <inequalities>
```

Figure 7: Simplified BNF of the LLS.

a processing step is modeled by one controllable transition, one uncontrollable transition, and one place, as shown in Figure 9. The controllable transition is fired when the command is issued and the uncontrollable transition is fired after the command has been executed. For instance, in Figure 8, t_1 is fired when the command to turn on the conveyor is issued and t_2 is fired when the conveyor is on.

The following inequalities on the marking of the PN express the remaining requirements of the specification.

$$\mu_6 + \mu_9 + \mu_{10} \leq 1 \tag{1}$$
$$\mu_6 + \mu_9 + \mu_2 + \mu_3 + \mu_4 \leq 1 \tag{2}$$
$$\mu_8 + \mu_9 + \mu_{10} \leq 1 \tag{3}$$
$$\mu_8 + \mu_9 + \mu_2 + \mu_3 + \mu_4 \leq 1 \tag{4}$$

The inequality (1) expresses the requirement that only one part A should be placed on the conveyor. Further, (2) describes the requirement that the conveyor should not move from the time a part A is placed until the time when the parts A and B are assembled. The inequalities (3) and (4) expresses the similar requirements for the parts B.

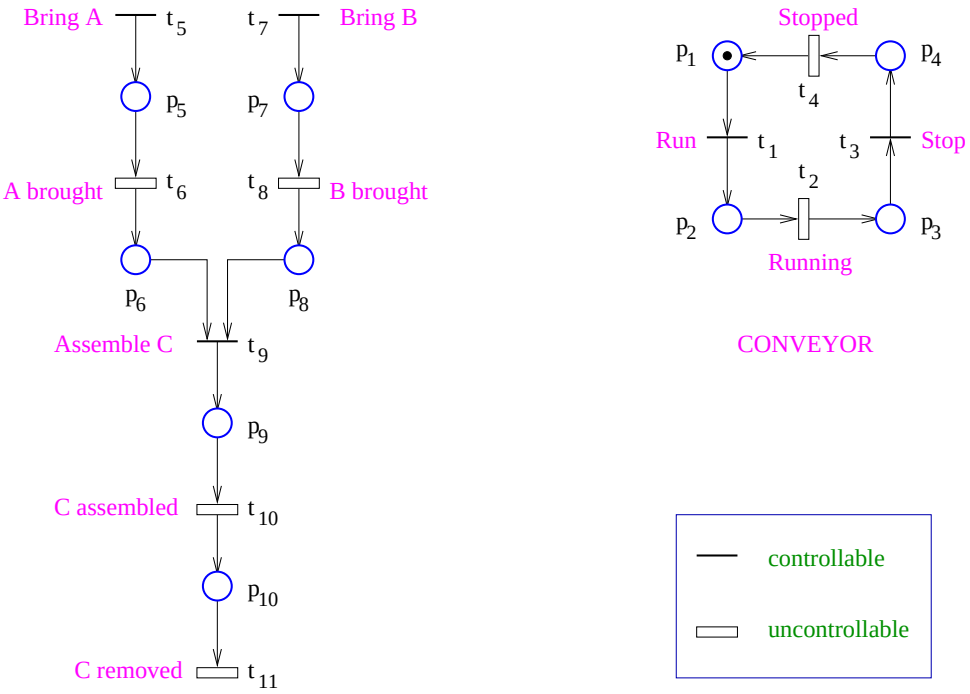

Figure 8: Plant model.

The inequalities (1)–(4) correspond to the PN specification shown in Figure 10. The PN representing the specification has the property that the parallel composition of the plant with the specification satisfies the desired requirements. Note that the transitions of the specification and the plant that are marked with the same symbol correspond to the same event.

Note that the inequalities (1)–(4) are not the only way to express the SC specification. The following system of inequalities is equivalent. However, it is more complex and results in a considerably more complex PN representation, due to disjunctions [63].

$$\mu_6 + \mu_9 + \mu_{10} \leq 1 \tag{5}$$

$$\mu_8 + \mu_9 + \mu_{10} \leq 1 \tag{6}$$

$$[\mu_6 + \mu_9 \leq 0] \ \vee \ [\mu_1 \geq 1] \tag{7}$$

$$[\mu_8 + \mu_9 \leq 0] \ \vee \ [\mu_1 \geq 1] \tag{8}$$

A long term direction of research in the development of the analysis tool is on how to obtain the most efficient representation of the SC specifications.

An LLS description of a version of this example is given in Appendix B at page 31. There, the plant is described as the parallel composition of five state-machine components, each corresponding to one process.

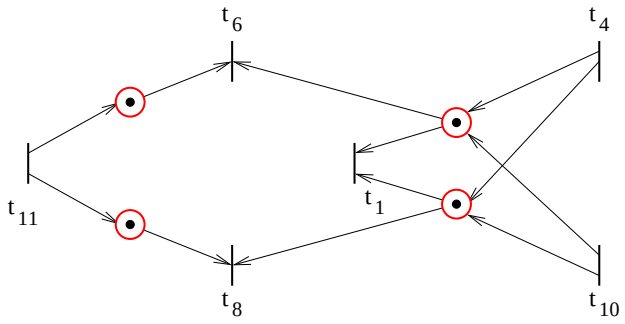

Figure 10: PN representing the SC specification.

6 Supervisory Control

In supervisory control (SC), a supervisor is designed based on a plant model and a specification. The supervisor ensures that the operation of the plant satisfies at all times the specification, subject to the constraints imposed by the plant. In its simplest form, the supervision of PNs involves a plant PN, such as the PN of Figure 8, and a specification given in terms of marking inequalities, such as the inequalities (9)–(12). The specification is written compactly in the form $L\mu \leq d$, where L is a matrix, μ is the marking vector, and d is a column vector. Then, if D is the incidence matrix of the PN, the supervisor is a PN of incidence matrix $D_s = -LD$. For instance, assuming the plant of Figure 8 and the specification given by the inequalities (9)–(12), the supervisor implementing the specification is shown in Figure 11, where the supervisor consists of the places C_1, C_2, C_3, and C_4. In general, the SC problem is considerably more difficult due to constraints imposed by the plant and complex specifications. Typically, the constraints imposed by the plant refer to the events that can be controlled or observed by the supervisor.

The dining philosophers problem could be used as an example. From the viewpoint of an agent controlling the access to the forks, a transition between the states "think" and "hungry" is uncontrollable and unobservable. Further, a transition between "eat" and "think" is uncontrollable but observable. Moreover, a transition modeling access to a "fork" can be seen as controllable and observable. The specification to the dining philosophers problem could be that "starvation" is impossible, that is, any hungry philosopher eventually gains access to the forks.

Uncontrollable and/or unobservable transitions may be needed in any of the following contexts:

- A decentralized environment, in which the transitions of one entity are unobservable and uncontrollable to the other entities.
- An embedded system environment in which transitions are controllable when they can be controlled by actuators and observable when they can be detected based on sensor information.
- A transition associated with an interrupt can be considered uncontrollable (such as in [31]).
- For certain SC problems (such as liveness enforcement), transitions labeled by conditions have to be considered uncontrollable.

Various types of specifications may be necessary, such as enforcing liveness or reversibility, ensuring mutual exclusion, formal language constraints, and others. The SC problem is usually easy to solve

when all transitions are controllable and observable and no liveness or reversibility requirements are given. Therefore, most research effort has been directed towards the SC problems involving liveness specifications and/or partial controllability and partial observability.

6.1 Supervisory Control Methods

Significant effort has been spent in developing methods for the SC of PNs and automata. Note that finite automata are a special case of PNs and that the reachability graph of a PN is a (not necessarily finite) equivalent automaton. Thus, methods developed for automata are important for the reachability analysis of PNs. The reachability graph of a PN may not be finite and when finite it may have a size that is exponentially related to the size of the PN. For this reason, much work has been done on SC methods that avoid direct reachability analysis of PNs. Such SC methods are said to be structural, as they rely on the structure of the PN rather than the reachability graph. A description of available structural methods can be found in the survey papers [52, 61]. For this class of methods, there are results on the following types of specifications:

- liveness (at any reachable state any transition should be eventually fireable)
- safety constraints: (generalized) mutual exclusion, language constraints (requiring the words generated by the plant to belong to the specified language), state-based constraints (a forbidden set of states is to be avoided)
- certain types of safety constraints for decentralized/distributed supervision.

Concerning liveness, there is a considerable amount of work on methods for deadlock avoidance, prevention, and recovery. Several of the main approaches appear in [113]. In the context of PNs, most methods for liveness enforcement were proposed for special classes of PNs, modeling resource allocation systems, such as in [71, 104, 11, 112, 37, 85, 86, 107, 92]. Among these, [86] can also be used for partial controllability. For general PN models and/or partial controllability and partial observability the method of [62, 58] is of interest. This approach consists of a procedure that iteratively identifies and removes deadlock situations. However, its termination is not guaranteed.

The research on safety constraints has resulted in methods for the representation of the specifications by PNs and in methods of enforcement of safety constraints under partial controllability and partial observability. Many of the methods for the enforcement of safety constraints deal with generalized mutual exclusion specifications. In such specifications the marking μ of a PN is required to satisfy a number of inequalities $l\mu \leq b$, where b is an integer and l is an integer vector. Some methods address the general problem but are suboptimal [79, 80, 26]. Suboptimal methods may generate supervisors that are not least restrictive. Moreover, suboptimal methods may not be able to find solutions for certain problems, even when solutions exist. Then, there are other methods intended for special classes of PNs and specifications, which are optimal, such as in [28, 25, 35, 43, 54, 67, 101, 102]. Among the methods designed for different types of forbidden sets we mention [19, 53]. A suboptimal approach for PN language constraints appears in [61, 62].

For decentralized/distributed supervision, the supervision methods of [62, 60] can be used. Based on the given (centralized) specification, decentralized supervisors are obtained that operate autonomously. The approach can be used for decentralized settings that exclude communication between supervisors and for distributed settings in which communication is allowed. When the

supervisors rely on communication to operate correctly, distributed synchronization is necessary. Nonetheless, a distributed solution may involve less communication than a centralized solution.

6.2 Application of SC methods

Referring to the SC step of our approach (see Figure 1), the implementation work has begun with the family of structural methods of [62]. This family of methods is general in terms of both plant description and class of specifications. Further, the synthesis can be carried out under general settings of uncontrollability and unobservability and many of the results can be adapted to the decentralized/distributed settings. However, note that a long term goal for this project is to take advantage of all available methods, including reachability based methods. Therefore, of special interest for future work is to develop algorithms that identify special cases for which an optimal approach is available. Future work should address also strategies to select between various approaches based on their estimated completion time.

Among the specifications considered in the SC of PNs, language type specifications are especially important, due to their generality. A language specification is represented by a PN that generates the specified language. The specification PN introduces constraints on the sequence in which the plant events may occur. These constraints can be described algebraically. It has been shown [59] that each place of a PN describes a constraint of the form $hq + cv \leq b$, where b is an integer, h and c are integer vectors, and q and v are parameters related to the transition firings. They are called firing vector (q) and Parikh vector (v). It has been shown also that the constraints of the form $l\mu + hq + cv \leq b$ are as expressive as the constraints of the form $hq + cv \leq b$, where l is another integer vector and μ is the marking of the PN. Note that under some boundedness assumptions, PN languages can describe disjunctions $\bigvee_i [l_i\mu + h_i q + c_i v \leq b_i]$ [63].

In [62], which describes the SC approaches we have focused on, the language enforcement problem is seen as the problem of enforcing a set of constraints $l\mu + hq + cv \leq b$. The solution is found based on the solution to a supervision problem involving a transformed PN and a transformed set of constraints $l'\mu \leq b$. Thus, the methods for the constraints $l\mu \leq b$ discussed in the previous subsection are very important for general specifications.

Fairness is an important issue in programming applications. Starvation refers to the situation in which one or more processes may have to wait indefinitely. The constraints $l\mu + hq + cv \leq b$ could be used to describe fairness constraints. Alternatively, fairness could be considered in the lower level code. In this case a lower level code segment included with the HLL specification would describe which of the transitions enabled by the SC code should be fired.

Note that DES controllers, as opposed to supervisors, appear to be of greater interest in programming applications. While a supervisor disables events that may lead to unacceptable behavior of the plant, a controller selects the events that should be fired next, while ensuring the specification stays satisfied. Thus, controllers can be easily derived from supervisors. In this context the issue of whether a supervisor design method is optimal or not is not important, as long as the method is able to find a solution when one exists. Indeed, controllers would enforce a specification regardless of whether they are derived from least restrictive supervisors or from suboptimal supervisors. Work on the problem of obtaining DES controllers appears for instance in [12, 23, 24, 38, 56].

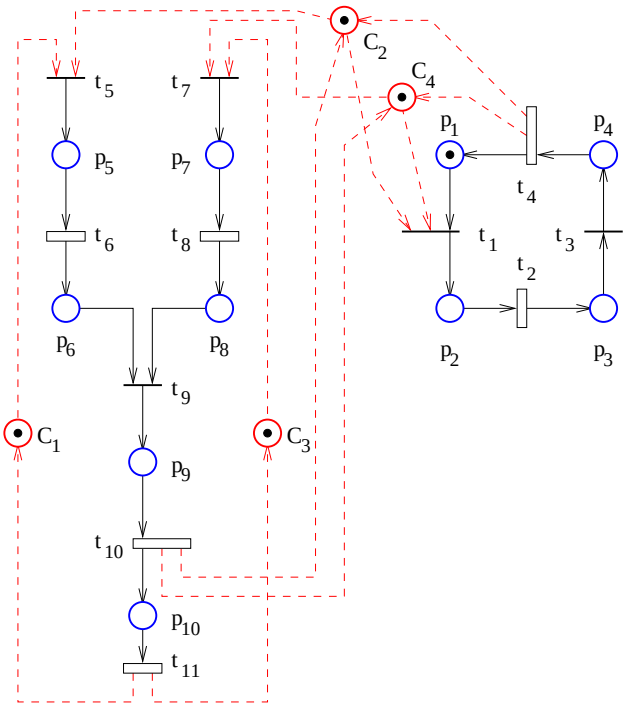

Figure 11: Plant and supervisor.

6.3 Example

Considering the example of section 5, the SC specification of Figure 10 cannot be used as a supervisor, as it does not account for the partial uncontrollability of the plant. By taking in account the uncontrollable transitions of the plant, the SC step changes the inequalities (1)–(4) to (9)–(12), which can be implemented as shown in Figure 11.

$$\mu_5 + \mu_6 + \mu_9 + \mu_{10} \leq 1 \tag{9}$$

$$\mu_5 + \mu_6 + \mu_9 + \mu_2 + \mu_3 + \mu_4 \leq 1 \tag{10}$$

$$\mu_7 + \mu_8 + \mu_9 + \mu_{10} \leq 1 \tag{11}$$

$$\mu_7 + \mu_8 + \mu_9 + \mu_2 + \mu_3 + \mu_4 \leq 1 \tag{12}$$

7 Code Generation

7.1 Problem Formulation and Possible Approaches

Referring to Figure 1, note that the role of code generation is to implement both the plant and the supervisor into code. The implementation of the plant involves the following operations.

- Writing the code associated with the places of the HPN and implementing the if-then-else statements associated with conditions that label transitions.
- Writing code for communication with the supervisor.

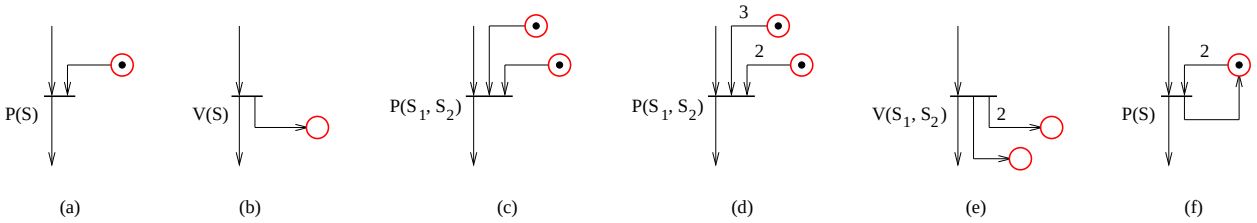

Figure 12: Regular P and V operations represented at (a) and (b). Next, (c) corresponds to a simultaneous P. Simple extensions are necessary for the operations shown at (d–f). Note the integer arc weights.

- Writing code for transitions involving multiple input and output places. Such code may implement synchronization, process creation, and process termination.

The supervisor is used to restrict the operation of a plant PN. The role of the supervisor is to identify transitions that can be fired without violating the given specification. Note that a supervisor does not force transition firings. It only indicates which transitions may be fired. Once a transition is enabled by both plant and supervisor, it can be immediately fired. A software implementation of the supervisor involves writing the code associated with plant events, such as transition firings or requests for permission to fire certain transitions.

The implementation in software of the supervisor could be done in more than one way. One possibility would be to use semaphores or other synchronization mechanisms to implement the supervisory policy. Another possibility would be to implement the supervisory policy as a coordinating task.

Semaphores are a natural choice for the representation of supervisor places [50, 52, 66, 115]. Since the code is automatically generated, difficulties arising in programming with semaphores are not relevant for this application. It should be noticed that some simple extensions of the semaphore operations are necessary, beyond that of the *simultaneous P/AND synchronization* [84]. Some necessary extensions are shown in Figure 12.

Semaphores would allow tasks to decide autonomously which transition to fire and when. However, a supervisor may involve more than just a number of places connected to plant transitions. Thus, a more general solution is to use a coordinating process to implement the supervisor. In this case the supervisor places could be modeled by integer variables. These variables would be tested each time a decision is made to fire a transition. Currently, this project implements the coordinating process approach. Future work may consider also a semaphore approach.

The semaphore and coordinating process approaches are illustrated on an example in subsection 7.2. Then, the details of code generation are considered in subsection 7.3.

7.2 Example

Referring to the example of section 6.3, the code generation step is to generate the programming code based on the PN plant and supervisor shown in Figure 11. A semaphore implementation could be done as follows. Four semaphores $S_1 \ldots S_4$ are needed, one for each of the four supervisor places $C_1 \ldots C_4$. P-type operations on semaphores would be executed before firing any of t_1, t_5, and t_7, and V-type operations after firing t_4, t_{10}, and t_{11}. For instance, $P(S_1, S_2)$ is executed before firing

t_5 and $V(S_1)$ and $V(S_3)$ after firing t_{11}. Note that the simultaneous P/AND synchronization [84] is used for the P operation.

Alternatively, a coordinator process could be used to implement the supervisor instead of the semaphores. The coordinator would use four variables, $m_1 \ldots m_4$, each corresponding to the marking of the supervisor places $C_1 \ldots C_4$. In this approach the conveyor process notifies the coordinator each time t_4 is fired and the assembly process each time t_{10} and t_{11} are fired. The conveyor and assembly processes request permission to fire t_1, t_5, and t_7 to the coordinator. The coordinator enables/disables t_1, t_5, and t_7 based on the supervisory solution shown in Figure 11. For instance, t_1 is enabled by the coordinator when $m_2 \geq 1$ and $m_4 \geq 1$. Further, if permission to fire t_1 is granted, m_2 and m_4 are decremented. Similarly, if the firing of t_4 or t_{10} is announced, m_2 and m_4 are incremented.

7.3 A Coordinator Based Approach

Here we present a possible code generation approach involving a coordinator process. Recall that in this project the specification describes a number of processes as well as constraints on the operation of the processes. The processes defined in the specification are called *application processes*. PNs are used to model the operation sequence of application processes, where a process is represented by a PN token and stages in the operation of the process are represented by PN places. Further, the operation of application processes is controlled by a supervisor process, which may inhibit or enable various PN transitions. The objective of code generation is to obtain the programs implementing the application processes as well as the coordination code.

The code generation algorithms described here are developed under the following assumptions. While these assumptions do not seem to be restrictive for practical applications, they can simplify the code generation algorithms.

A1. The PNs associated with application processes are state machines. (However, their parallel composition does not have to be a state machine.)

A2. The PNs associated with application processes have distinct labels. (However, PNs of different processes may have common labels and the supervisor process does not have to have distinct labels.)

As mentioned in the first assumption, the PNs used to represent the structure of a program are a composition of state machine (SM) components. Each SM component corresponds to the possible stages of a process. For an example, consider Figure 3. In the figure, by composing the transitions with the same label of the SMs (a), (b), and (c), the PN (d) is obtained. Each SM component represents a process type, where the number of tokens of an SM component equals the number of processes of that type. Transitions are used to move a process from one stage to another, to terminate a process, or to begin a process. For instance, when the transition of label b is fired, a new process of type (c) is created. Moreover, when the transition of label d is fired, a process of type (c) terminates.

In code generation, special attention is given to transition synchronization. Synchronization is implemented by a coordinator process. When an application process is ready to fire a transition t that is synchronized with other transitions, the process requests the coordinator permission to fire

t. Permission is granted when all other processes involved in the synchronization are ready. Each synchronization involves two or more processes, where one of the processes may be the supervisor process.

Transitions that are not synchronized can be fired immediately. However, it may still be necessary to notify the supervisor process when they occur. Depending on their relationship to the supervisor, transitions are classified as follows.

- A transition is *controlled* if the supervisor may inhibit its firing. A process cannot fire a controlled transition without permission from the supervisor.
- A transition is *observed* if the supervisor monitors its firings in order to update internal variables. A process will notify the supervisor after firing an observed transition.

Note that the two properties defined above are independent. However, most often a controlled transition will be also observed. In terms of the PN representation of supervisors, the controlled transitions have one or more supervisor places as inputs and the observed transitions are connected to one or more supervisor places. However, a transition with input supervisor places is not necessarily controlled. This happens when for all reachable states it is never the case that the transition is plant enabled and supervisor disabled. Moreover, a transition connected to supervisor places may not be observed. This may happen when the supervisor places are controlled by means of self-loops. A reachability based algorithm could be used to determine the transitions that are controlled and observed. However, this would be rather computationally expensive. By overapproximating the sets of controlled and observed transitions, the performance of the resulting code might be somewhat affected, since the amount of communication between processes and supervisor is increased. However, since no significant performance degradation is expected, Algorithm 7.1 will be used to overapproximate the sets of controlled and observed transition. The algorithm is correct because a supervisor would never attempt to control an uncontrolled transition or to observe an unobservable transition.

The following PN notation is convenient for the further developments. If x is a place or a transition, let $x\bullet$ be the set of output places or output transitions of x and let $\bullet x$ be the set of input places or input transitions of x. Moreover, let $|X|$ denote the number of elements of the set X. Since the PNs representing processes are state machines, for all transitions $|\bullet t| \leq 1$ and $|t \bullet| \leq 1$. Note that while state machines are usually represented by ordinary PNs (PNs in which all arcs have unity weight), here we do allow arbitrary integer weights.

In addition to controlled and observed transitions, the following classes of transitions are defined. A transition t of a process type is a **synchronization transition** if it satisfies at least one of the following properties.

- t has an input arc of weight greater than one.
- t has an input place and there is a transition t' of a different process type such that t' has an input place and t and t' have the same label.

Moreover, if two transitions of two application processes have the same label, one of them has one output place but no input place, and none is a synchronization transition, then they are **action transitions**. Note that transitions with one output place and no input place create new processes. Thus, when an action transition takes place, the coordinator process should be notified in order to generate the corresponding new processes.

Algorithm 7.1 ***Finding Controlled and Observed Transitions***

Input: The PNs $\mathcal{N}_i = (P_i, T_i, D_i^+, D_i^-, \rho_i)$, $i = 1 \ldots n$, associated with each of the n processes; the supervisor PN $\mathcal{N}_s = (P_s, T_s, D_s^+, D_s^-, \rho_s)$.

Output: T_c, the set of controlled transitions, and T_o, the set of observed transitions.

/ P_i (P_s) and T_i (T_s) denote the sets of places and transitions, respectively; D_i^+ (D_s^+) and D_i^- (D_s^-) denote the input and output matrices, respectively; $\rho_i : T_i \to \Sigma$ ($\rho_s : T_s \to \Sigma$) denotes the labeling function. */*

1. $T_c = \emptyset$ and $T_o = \emptyset$.
2. For $i = 1 \ldots n$
3. For all transitions $t \in T_i$ {
4. Let $T_s(t) = \{t_s \in T_s : \rho_s(t_s) = \rho_i(t)\}$. */* This is the set of supervisor transitions with the same label as t. */*
5. If $T_s(t) \neq \emptyset$ {
6. If $D_s^-(\cdot, t_s) \neq 0$ for all $t_s \in T_s(t)$ */* $D_s^-(\cdot, t_s) \neq 0$ means that the column t_s of D_s^- has at least one nonzero element. */*
7. $T_c = T_c \cup \{t\}$.
8. If $D_s^-(\cdot, t_s) = D_s^+(\cdot, t_s)$ for all $t_s \in T_s(t)$
9. $T_o = T_o \cup \{t\}$.
10. }
11. }

An outline of the operations performed when a process (token) enters the stage (place) p is as follows.

1. The functions associated with p are executed.
2. The next transition t is selected.
3. If applicable, the process requests and waits for permission to fire t.
4. If applicable, the process notifies the supervisor that t is fired.
5. The transition t is fired. There are two possible outcomes:
   - The process terminates if t has no output place.
   - The process continues with the next stage p', where p' is an output place of t.

Note that if a place p has several output transitions, the next transition t can be selected based on conditions associated with the transitions or based on internal operations performed by the functions associated with the place p. If permission to fire t is to be obtained from a supervisor, a function call of the form

```
RequestToFire(t)
```

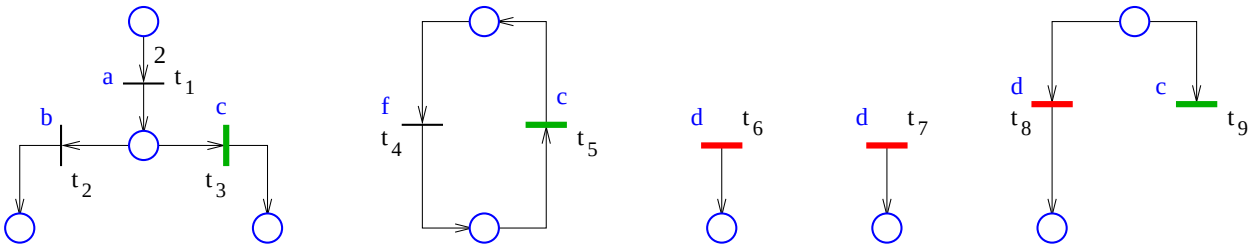

Figure 13: Examples of synchronization and action transitions. t_1 is a synchronization transition since it has an input arc of weight 2. t_3, t_5, and t_9 are also synchronization transitions. t_6, t_7, and t_8 are action transitions.

is made. After a `RequestToFire` call, the process has to wait until permission is granted. Now, the call `RequestToFire(t)` addresses the case when the process selects only one transition t. However, it may be that a process could continue by firing any of n transitions t_1, t_2, ..., t_n. An example in which this possibility might arise is when a process may have to complete n operations and the order in which it performs them is not important. Then, each of the operations might be associated with a place reached by firing one of t_1, t_2, ..., t_n. Thus, the `RequestToFire` function can be called with more than one parameter:

```
RequestToFire(t1, t2, ..., tn)
```

In a call involving more than one parameter, the transition that is fired is selected by the supervisor process. Typically, the supervisor will select the first transition in the list that is enabled or, if all transitions are disabled, the first transition that gets enabled. Note that the order of the parameters indicates the order of preference of the transitions. Just as in the case with only one parameter, the process calling `RequestToFire` will have to wait until the supervisor gives permission to fire one of the transitions. A nonblocking version of `RequestToFire` could be

```
Try(t)
```

This function would obtain permission to fire t if t is enabled or else report that t is disabled. While this function would provide additional flexibility for code that is manually written, the `RequestToFire` function is sufficient for automatically generated code. By using a call of the form `RequestToFire(t1, t2, ..., tn, t)` in which the last transition t is always enabled, the call will not get blocked. An example of a transition t that is always enabled is a transition implementing a self-loop.

A **nondeterministic** place is a place p in which the next transition is selected by means of a call `RequestToFire(t1, t2, ..., tn)` in which *all* output transitions of p are present. Note that a nondeterministic place is a place in which the process lets the supervisor make the selection of the next transition. The ability to control the selection of the next transition is important for liveness enforcing supervisors.

A description of the code structure of an application process is given in Algorithm 7.2. A description of the coordinator is given in the Algorithm 7.3. For simplicity, all coordination functions are assigned to the supervisor process. Thus, the supervisor process and the coordinator process are the same. For simplicity, implementation details have been omitted from the algorithm descriptions. Various steps of the algorithms could be done more efficiently.

Algorithm 7.2 ***Structure of the Application Process***

Input: p – the place where the token is located initially.

- Go to stage p. */* There are $p_1, p_2, \ldots, p_m$ stages, one for each place of the underlying state machine. */*

  ⋮

  / The code for stage p_i. */*

1. Initialize $trans\_list = p_i\bullet$. */* $p_i\bullet$ is the set of output transitions of p_i. This step is necessary only if the transitions of $p_i\bullet$ do not have conditions. */*
2. Run the code associated with place p_i. */* In the LLS, this code is defined with the p_i.code command. */*
3. Select the next transition. */* This operation determines trans_list. The operation is performed only when the transitions of $p_i\bullet$ have conditions. */*
4. Let t be the first transition of $trans\_list$.
5. If $t \in T_c \cup T_{sync}$ */* T_c, T_{sync}: the sets of controlled and synchronization transitions. */*
6. $t = RequestToFire(trans\_list)$; */* Requests and waits for permission to fire. The function returns the transition in the list that may be fired. */*
7. Else if $t \in T_o \cup T_{act}$ */* T_o, T_{act}: the sets of observed and action transitions. */*
8. Notify coordinator that the transition t is fired.
9. If $t\bullet == \emptyset$ */* $t\bullet$ is the set of output places of t. */*
10. Exit. */* The process terminates. */*
11. Go to the stage $p' = t\bullet$. */* In a state machine $t\bullet$ has at most one element. */*

    ⋮

The following remarks could be made about the algorithms.

- A process type represents the structure of a process. This includes the PN structure, the code associated with each place, and the conditions associated with the transitions. Note that a process type does not define the labels of the PN structure.
- For any process, the PN representing its structure has distinct labels. However, two different processes may share common labels. Moreover, the PN associated with a process is a state machine, that is, a transition may have at most one input place and at most one output place.
- The algorithms take in account that the PNs representing processes are state machines and have distinct labels. An exception is Algorithm 7.6 which supports also PNs that do not have distinct labels.

Algorithm 7.3 ***The Supervisor (Coordinator) Process***

Input: The n processes to be controlled, each being described by a PN $\mathcal{N}_i = (P_i, T_i, D_i^+, D_i^-, \rho_i)$ of initial marking $\mu_{0,i}$, for $i = 1 \ldots n$; the supervisor PN $\mathcal{N}_s = (P_s, T_s, D_s^+, D_s^-, \rho_s)$ and its initial marking $\mu_{0,s}$.

1. For all PN components $\mathcal{N}_i$
2. For all $p \in P_i$
3. If $\mu_{0,i}(p) \neq 0$ */* $\mu_{0,i}$ *is the initial marking of* $\mathcal{N}_i$. **/*
4. Start $\mu_{0,i}(p)$ processes of type i in stage p.
5. While $terminate == FALSE$ {
6. For all new messages requiring permission to fire a transition
7. Place request in the queue;
8. For all new messages notifying the firing of a transition t {
9. If $t \in T_o$ */* T_o *is the set of observed transitions. */*
10. UpdateSupervisorMarking(t); */* Algorithm 7.5. */*
11. If $t \in T_{act}$ */* T_{act} *is the set of action transitions. */*
12. PerformAction(t); */* Algorithm 7.4. */*
13. }
14. For all queue entries q {
15. $EntryList = IsPermissible(q)$; */* Algorithm 7.6; identifies processes that are granted permission to fire. */*
16. If $EntryList \neq NULL$ {
17. Grant permission to fire to the processes in $EntryList$.
18. Remove from queue the entries in $EntryList$.
19. Let $t = q.fireable$. */* This is the transition that is fired. */*
20. UpdateSupervisorMarking(t); */* Algorithm 7.5. */*
21. PerformAction(t); */* Algorithm 7.4. */*
22. }
23. }
24. }
25. Terminate application processes.

Algorithm 7.4 ***The*** **PerformAction** ***Method of the Supervisor Process***

```
    PerformAction(t) {
1.      For all PN components N_i
2.          For all transitions t' with the same label as t
3.              If t'• ≠ 0 and •t' == 0 {
4.                  Let w be the weight of the output arc of t'.
5.                  Start w processes of type i in the stage p = t'•.
6.              }
7.  }
```

Algorithm 7.5 ***The*** **UpdateSupervisorMarking** ***Method of the Supervisor Process***

```
    UpdateSupervisorMarking(t) {
1.      For all transitions t_s ∈ T_s /* T_s is the set of supervisor transitions. */
2.          If t_s and t have the same label
3.              If t_s is enabled {
4.                  Fire t_s.
5.                  Break.  /* Exit loop. */
6.              }
7.      flag = 1;
8.      While flag == 1 {
9.          flag = 0;
10.         For all t_s ∈ T_s
11.             If t_s is unlabeled and enabled {
                    /* A supervisor transition is labeled if and only if it is to be synchronized
                    with one or more plant transitions. */
12.                 Fire t_s.
13.                 flag = 1;
14.             }
15.     }
16. }
```

Algorithm 7.6 ***The* IsPermissible *Method of the Supervisor Process***

```
    IsPermissible(q) {
1.      For all transitions t listed in the queue entry q
2.          If t is supervisor enabled {
3.              /* t is supervisor enabled if either no supervisor transition has the same label
                as t or a supervisor transition with the same label as t is enabled. */
4.              permissible = 1;
5.              q.fireable = t;  /* which transition to fire if permission is granted */
6.              QueueEntryList = {q};
7.              For all process types {
8.                  For all transitions t' with the same label as t {
9.                      count = 0;
10.                     If t' ≠ t and t' and t belong to the same process type
11.                         continue;  /* go to the next iteration of the loop */
12.                     If t' == t {
13.                         If W(t) == 1  /* W(t) is the weight of the input arc of t */
14.                             break;  /* exit loop */
15.                         count = 1;
16.                     }
17.                     QL = SearchQueue(t', count, QueueEntryList); /* Algorithm 7.7 */
18.                     If QL ≠ ∅ {
19.                         QueueEntryList = QL;
20.                         permissible = 1;
21.                         break;  /* exit loop */
22.                     }
23.                     permissible = 0;  /* the line is reached if QL = ∅ */
24.                 }
25.                 If permissible == 0
26.                     break;  /* exit loop: no process of this process type allows firing t */
27.             }
28.             If permissible == 1
29.                 Return QueueEntryList;
30.         }
31.     Return NULL;  /* that is, request not permissible. */
32. }
```

Algorithm 7.7 ***The*** **SearchQueue** ***Procedure of*** **IsPermissible**

```
    SearchQueue(t', count, QueueEntryList) {
1.      If •t' ≠ ∅ {
2.          QL = QueueEntryList;
3.          For all queue entries q' ∉ QueueEntryList
4.              If t' listed in q' {
5.                  q'.fireable = t';
6.                  QL = {q'} ∪ QL.
7.                  count = count + 1;
8.                  If count ≥ W(t') /* W(t') is the weight of the input arc of t' */
9.                      Return QL;
10.             }
11.         Return NULL;
12.     }
13.     Return QueueEntryList;
14. }
```

8 Conclusion

Notoriously difficult [22], the development of concurrent programs could be simplified by using tools that generate automatically part of the required code. This paper proposes the application of supervisory control (SC) for the automatic synthesis of concurrent programs. SC is of interest because various high level requirements can be seen as supervisory control (SC) specifications. Thus, SC methods or similar methods from related areas of research have to be applied in order to achieve a high degree of automation of the programming process. The SC problem, in its general form, is of considerable difficulty. Nonetheless, significant progress has been made and powerful results are already available. The project described in this paper aims to develop software for program synthesis that exploits available SC methods. In this project, based on a specification written in a high level specification language (HLL), an SC problem is formulated and then solved using SC methods. The result of the SC step is then converted to low level code. The project involves work on the HLL, the translation of HLL specifications to SC specifications, the translation of supervisory policies to low level code, and the SC methods.

Three of the novel features of this project are as follows. First, we propose to apply SC results developed for Petri net (PN) models to program synthesis. Note that by using PN models it is possible to resort also to automata results. Second, the translation of a high level programming specification into a SC specification is also a new topic. Third, for the SC we propose a framework in which multiple methods can be used, based on the context, in an effort to exploit the strengths of each method.

9 Acknowledgments

The support of the National Science Foundation (NSF CNS-0834057) is gratefully acknowledged. A number of students have done programming work for this project. Bion Oren has written the translator for LLS specifications. Drew Crawford and Kristopher Eggert have worked on the code generation tool. Stephen Camp has implemented in C the toolbox [57].

A Appendix A: Basic Petri Net Concepts

A **Petri net structure** is a tuple $\mathcal{N} = (P, T, F, W)$ where P is the **set of places**, T the **set of transitions**, $F \subseteq (P \times T) \cup (T \times P)$ is the set of **transition arcs** and $W : F \to \mathbb{N} \setminus \{0\}$ is a **weight function**. Note that $\mathbb{N}$ is the set of nonnegative integers. A **marking** μ of the Petri net structure is a map $\mu : P \to \mathbb{N}$. A Petri net structure $\mathcal{N}$ with **initial marking** μ_0 is called a **Petri net**, and will be denoted by $(\mathcal{N}, \mu_0)$. Often, a Petri net structure is also called a Petri net. Further, there are many classes of nets generically called Petri nets. In fact, in the general context of Petri nets, the nets defined above are known as P/T nets [91]. However, since their definition is sufficient to represent the most common varieties of Petri nets, following the survey paper of Murata [81], we simply call them Petri nets.

The marking represents the state of a Petri net. The marking is often represented as a vector $[\mu(p_1), \mu(p_2), \ldots \mu(p_n)]^T$, where $p_1, p_2, \ldots p_n$ are the places of the net enumerated in a chosen (but fixed) order. The same symbol μ is used to denote this vector.

Petri nets have a convenient graphical representation, useful for tutorial purposes. This representation is illustrated in Figure 14. Places are represented by circles, transitions by thick lines and transition arcs by arrows. Arc weights greater than one are indicated close to the arcs. For instance, in Figure 14(c) $W(p_3, t_1) = 2$ and $W(t_2, p_2) = 4$. The figures also show a marking for each Petri net. The marking of each place consists of the number of tokens inside the circle. (A token is represented by a small dark filled circle.) For instance, the marking vector in Figure 14(c) is $[0, 1, 1]^T$.

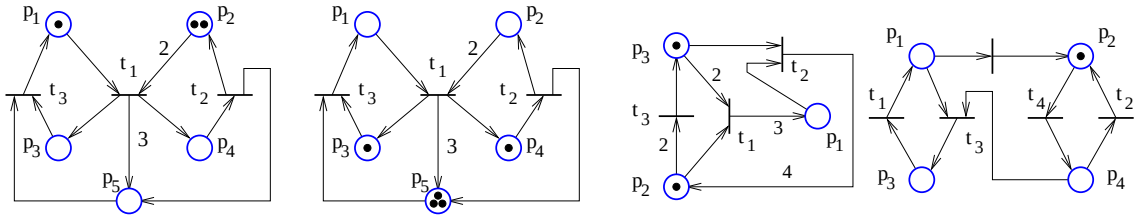

Figure 14: Petri net examples.

The **preset** of a transition t is the set of input places of t: $\bullet t = \{p \in P : (p, t) \in F\}$. The **postset** of a transition t is the set of output places of t: $t\bullet = \{p \in P : (t, p) \in F\}$. Note that $\bullet p$ and $p\bullet$ are similarly defined. For instance, in Figure 14(c): $\bullet t_2 = \{p_1, p_3\}$, $t_3\bullet = \{p_3\}$, and $\bullet p_2 = \{t_2\}$.

The marking μ **enables** the transition t if $\forall p \in \bullet t$: $\mu(p) \geq W(p, t)$. When μ enables t and t **fires**, the marking is changed. Let μ' be the next reached marking; we formally express this by $\mu \xrightarrow{t} \mu'$. The marking μ' satisfies:

$$\mu'(p) = \begin{cases} \mu(p) & \text{if } p \notin \bullet t \cup t\bullet \\ \mu(p) + W(t, p) & \text{if } p \in t\bullet \setminus \bullet t \\ \mu(p) - W(p, t) & \text{if } p \in \bullet t \setminus t\bullet \\ \mu(p) - W(p, t) + W(t, p) & \text{if } p \in \bullet t \cap t\bullet \end{cases}$$

For instance, firing t_1 in Figure 14(a) produces the marking shown in Figure 14(b). A sequence of transitions $\sigma = t_1 t_2 \dots t_k$ is **enabled** at the marking μ if μ enables t_1 and $\mu \xrightarrow{t_1} \mu_1$, μ_1 enables t_2 and $\mu_1 \xrightarrow{t_2} \mu_2$, and so on. The marking μ' is **reachable** from μ if there is a sequence of markings $\mu_1, \dots \mu_k$, $\mu_k = \mu'$, and a sequence of transitions $\sigma = t_{i_1}, \dots t_{i_k}$ such that $\mu \xrightarrow{t_{i_1}} \mu_1 \dots \xrightarrow{t_{i_k}} \mu'$. This is also written as $\mu \xrightarrow{\sigma} \mu'$. The **set of reachable markings** of a Petri net $(\mathcal{N}, \mu_0)$ is the set of markings reachable from the initial marking μ_0. It is denoted by $\mathcal{R}(\mathcal{N}, \mu_0)$. The **reachability graph** of a Petri net is obtained by associating a node with each marking $\mu \in \mathcal{R}(\mathcal{N}, \mu_0)$; there is an arc from the node of μ_1 to the node of μ_2, if μ_2 is reached from μ_1 by firing a transition t: $\mu_1 \xrightarrow{t} \mu_2$.

In a Petri net $\mathcal{N} = (P, T, F, W)$ with m places and n transitions, the **incidence matrix** is an $m \times n$ matrix defined by $D = D^+ - D^-$, where the elements d_{ij}^+ and d_{ij}^- of D^+ and D^- are

$d_{ij}^+ = W(t_j, p_i)$ if $(t_j, p_i) \in F$ and $d_{ij}^+ = 0$ otherwise;

$d_{ij}^- = W(p_i, t_j)$ if $(p_i, t_j) \in F$ and $d_{ij}^- = 0$ otherwise.

The matrices D^+ and D^- are called the **input matrix** and the **output matrix**, respectively.

The incidence matrix allows an algebraic description of the marking change of a Petri net:

$$\mu_k = \mu_{k-1} + D q_k \tag{13}$$

where q_k is called **firing vector**, and its elements are all zero excepting $q_{k,i} = 1$, where i corresponds to the transition t_i that fired. Note that when multiple transitions are allowed to fire at the same time, q_k is an integer vector in which for all i, $q_{k,i}$ indicates how many times the transition t_i fires at the instant k. Further, q_k is enabled (may be fired) when

$$\mu_{k-1} \geq D^- q_k \tag{14}$$

We will also denote by **Parikh vector** a vector v associated with a sequence of transitions that have fired, whose entries record how many times each transition appears in the sequence. If v is the Parikh vector of the transition sequence that led the Petri net from the marking vector μ_0 to μ_k:

$$\mu_k = \mu_0 + Dv \tag{15}$$

A Petri net is a **state machine** if all transitions t have at most one input place and at most one output place (that is, $\forall t \in T : |\bullet t| \leq 1$ and $|t \bullet| \leq 1$). Just as automata can have transitions labeled by events, the transitions of Petri nets can also be labeled by events. A **labeled Petri net** is a Petri net enhanced with a labeling function $\rho : T \to 2^{\Sigma} \cup \{\lambda\}$, where Σ is the set of events, ρ the labeling function, and λ the null event. There are several ways in which Petri net languages can be defined [88]. The P-type languages of labeled Petri nets are of special interest in the context of the structural methods of supervision. The **P-type language** of a labeled Petri net $(\mathcal{N}, \rho, \mu_0)$ consists of all sequences of events $w = \rho(\sigma)$ generated by the firing sequences σ enabled at μ_0. Note that for $\sigma = t_1 t_2 t_3 \dots$, $\rho(\sigma)$ denotes the sequence $\rho(t_1)\rho(t_2)\rho(t_3)\dots$.

Note that the labeling function can also be defined as $\rho : T \to \Sigma \cup \{\lambda\}$, since a transition with n labels can be replaced by n identical transition, each labeled with one of the n labels. We will use this definition of the labeling function in order to introduce the parallel composition of two labeled Petri nets. Let $\mathcal{N}_i = (P_i, T_i, F_i, W_i, \rho_i)$, $i = 1, 2$, be two Petri nets, where $\rho_i : T_i \to \Sigma_i \cup \{\lambda\}$

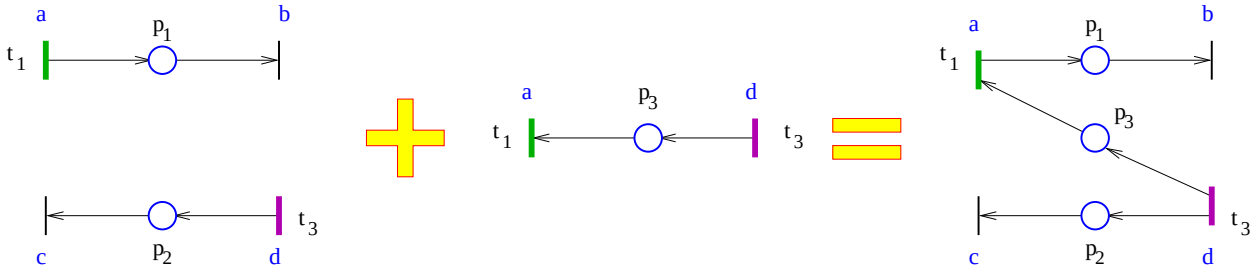

Figure 15: Parallel composition example. The letters a, b, c, and d, denote the labels.

and Σ_i is defined such that for all $e \in \Sigma_i$ there is a transition $t \in T_i$ labeled by e. The **parallel composition** of $\mathcal{N}_1$ and $\mathcal{N}_2$ is a Petri net $\mathcal{N} = (P, T, F, W, \rho)$ obtained as follows.

1. $P = P_1 \cup P_2$.
2. For all $t_1 \in T_1$ and $t_2 \in T_2$ such that $\rho_1(t_1) = \rho_2(t_2) \neq \{\lambda\}$, create a transition $t_{1,2}$ such that $\rho(t_{1,2}) = \rho_1(t_1)$, $\bullet t_{1,2} = \bullet t_1 \cup \bullet t_2$, and $t_{1,2}\bullet = t_1 \bullet \cup t_2\bullet$.
3. For all $t \in T_1$ with $\rho_1(t) \in (\Sigma_1 \setminus \Sigma_2) \cup \{\lambda\}$, create a transition t' of $\mathcal{N}$ such that $\rho(t') = \rho_1(t)$, $t'\bullet = t\bullet$, and $\bullet t' = \bullet t$.
4. For all $t \in T_2$ with $\rho_2(t) \in (\Sigma_2 \setminus \Sigma_1) \cup \{\lambda\}$, create a transition t' of $\mathcal{N}$ such that $\rho(t') = \rho_2(t)$, $t'\bullet = t\bullet$, and $\bullet t' = \bullet t$.

Note that when the parallel composition is performed, each transition of one Petri net is synchronized with each transition of the other Petri net that has the same label. The remaining transitions are copied without change. Two examples are shown in Figures 15 and 16.

B Appendix B: A Specification Written in the Low Level Specification Language

The following is a description of the manufacturing example presented in this report.

```
// Manufacturing example

// 1. PROCESS-TYPE DECLARATIONS

process  CONVEYOR;

build: {make -f conveyor.mak};

include: {#include "conveyor.h"};



process  MANFLINE_A;
```

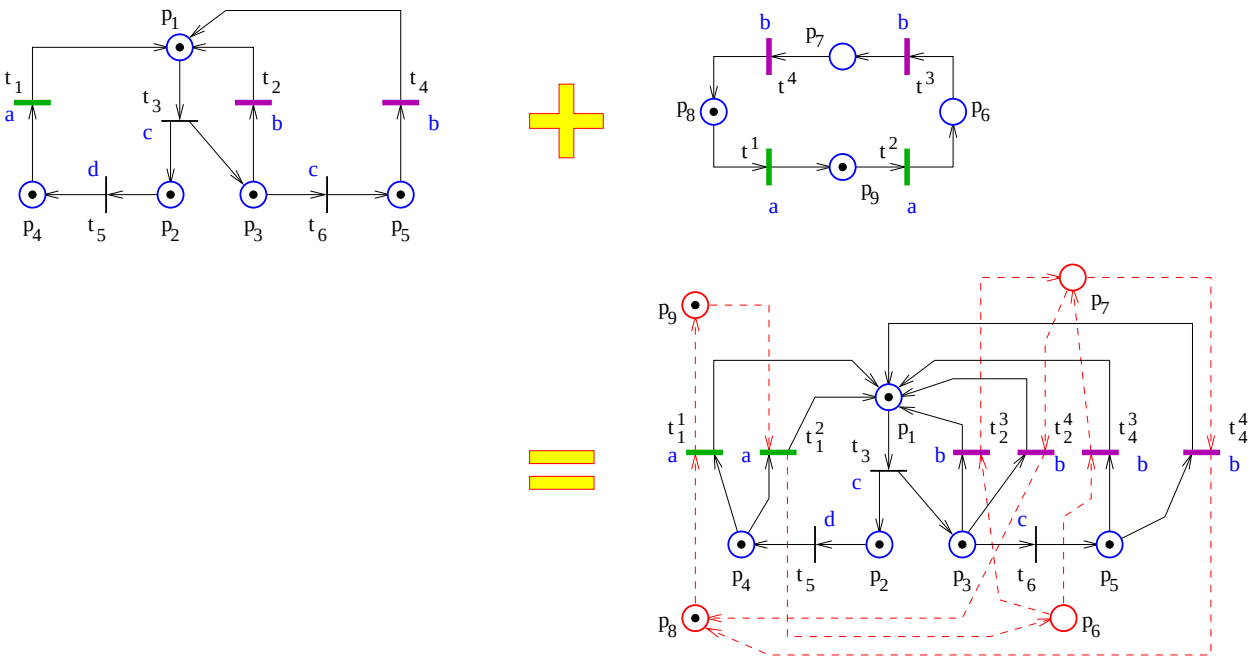

Figure 16: Parallel composition example. The letters a, b, c, and d, denote the labels.

```
build: {make -f manfline_A.mak};

include: {
  #include "manfline.h"
};



process  MANFLINE_B;

build: {make -f manfline_B.mak};

include: {
  #include "manfline.h"
};



process  FETCH_A;

build: {make -f fetch_a.mak};

include: {
  #include "fetch_a.h"
};
```

```
process  FETCH_B;

build: {make -f fetch_b.mak};

include: {
  #include "fetch_b.h"
};


// 2. DEFINITIONS

CONVEYOR.PN:

places: p1 p2 p3 p4;

transitions: t1 t2 t3 t4;

p1.code: { wait_run_command(); };

p2.code: { start_conv();
           wait_until_conv_ready();
         };

p3.code: { wait_stop_command(); };

p4.code: { stop_conv();
           wait_until_stopped();
         };

(p1, t1, p2);
(p2, t2, p3);
(p3, t3, p4);
(p4, t4, p1);


MANFLINE_A.PN:

places: p5 p6 p9 p10;

transitions: t5 t6 t9 t10 t11;

p6.code: { begin_assemble1();};

p9.code: { end_assemble(); };
```

```
p10.code: { request_move();
            wait_until_removed();
          };

(t5, p5);
(p5, t6, p6);
(p6, t9, p9);
(p9, t10, p10);
(p10, t11);


MANFLINE_B.PN:

places: p7 p8;

transitions: t7 t8 t9;

p8.code: { begin_assemble2();};

(t7, p7);
(p7, t8, p8);
(p8, t9);



FETCH_A.PN:

places: p_a;

transitions: t_a;

p_a.code: { fetch_part_A(); };

(p_a, t_a, p_a);


FETCH_B.PN:

places: p_b;

transitions: t_b;

p_b.code: {fetch_part_B(); };

(p_b, t_b, p_b);
```

```
// 3. PROCESS DECLARATIONS

FETCH_A fpa(p_a:1); // initial marking = 1
FETCH_B fpb(p_b:1); // initial marking = 1
MANFLINE_A manf1;      // initial marking = [0 0 ... 0]
MANFLINE_B manf2;      // initial marking = [0 0 ... 0]
CONVEYOR conv(p1:1);// initial marking = [1 0 0 0]

// DEPENDENCIES

sync fpa.t_a manf1.t5;
sync fpb.t_b manf2.t7;
sync manf1.t9 manf2.t9;

// 4. INSTRUCTIONS ON THE DESIGN OF THE SUPERVISOR

conv.constraints:

uncontrollable: t2 t4;

manf1.constraints:

uncontrollable: t6 t10 t11;

live: t11;

manf2.constraints:

uncontrollable: t8;

global.constraints:

manf1.p6 + manf1.p10 + manf1.p9 <= 1;

manf1.p6 + manf1.p9 + conv.p2  + conv.p3 + conv.p4 <= 1;

manf2.p8 + manf1.p9 + manf1.p10 <= 1;

manf2.p8 + manf1.p9 + conv.p2 + conv.p3 + conv.p4 <= 1;
```

References

[1] A Concurrency Tool Suite. http://www.letu.edu/people/marianiordache/acts.

[2] PEP homepage. http://parsys.informatik.uni-oldenburg.de/∼pep.

[3] PNetLab homepage. http://www.prisma.unina.it/automation/frame.htm.

[4] Spectool homepage. http://www.engr.uky.edu/∼holloway/spectool.

[5] Supremica homepage. http://www.supremica.org.

[6] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[7] 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 (RTSS'99)*, pages 154–163. IEEE Computer Society, 1999.

[8] J. Ashley and L.E. Holloway. Automated control, observation, and diagnosis of multi-layer condition systems. *Studies in Informatics and Control*, 16(1), 2007.

[9] 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.

[10] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, E. Sentovich, and K. Suzuki. Synthesis of software programs for embedded control applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):834–849, 1999.

[11] Z. Banaszak and B. Krogh. Deadlock avoidance in flexible manufacturing systems with concurrently competing process flows. *IEEE Transactions on Robotics and Automation*, 6(6):724–734, 1990.

[12] M. Barbeau, M. Frappier, F. Kabaza, and R. St.-Denis. A supervisory control synthesis case study: The antenna control system. In *Proceedings of 1997 Allerton Conference on Communication, Control, and Computing*, pages 533–542, 1997.

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

[14] E. Best, R. Devillers, and M. Koutny. Petri nets, process algebras and concurrent programming languages. In Reisig, W. and Rozenberg, G., editors, *Lectures on Petri Nets II: Applications*, volume 1492 of *Lecture Notes in Computer Science*, pages 1–84. Springer-Verlag, 1998.

[15] 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*, 35(10):813–857, 1998.

[16] E. Best and R. Hopkins. B(PN)2 - a basic Petri net programming notation. In Bode, A., Reeve, M., and Wolf, G., editors, *PARLE*, volume 694 of *Lecture Notes in Computer Science*, pages 379–390. Springer-Verlag, 1993.

[17] E. Best and C. Palamidessi. Linear constraint systems as high-level nets. In Montanary, U. and Sassone, V., editors, *CONCUR*, volume 1119 of *Lecture Notes in Computer Science*, pages 498–513. Springer-Verlag, 1996.

[18] J. Billington. Protocol specification using P-graphs, a technique based on coloured Petri nets. In Reisig, W. and Rozenberg, G., editors, *Lectures on Petri Nets II: Applications*, volume 1492 of *Lecture Notes in Computer Science*, pages 293–330. Springer-Verlag, 1998.

[19] R. K. Boel, L. Ben-Naoum, and V. Van Breusegem. On forbidden state problems for a class of controlled Petri nets. *IEEE Transactions on Automatic Control*, 40(1):1717–1731, 1995.

[20] 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. In Pezz, M. and Shatz, M., editors, *Workshop Proceedings Software Engineering and Petri Nets*, pages 115–134, June 2000.

[21] U. A. Buy and R. H. Sloan. Automatic real-time analysis of reactive systems with the PARTS toolset. *Automated Software Engineering*, 23(4):227–273, 2001.

[22] D. Callahan. Design considerations for parallel programming. *MSDN Magazine*, October 2008.

[23] V. Chandra, Z. Huang, and R. Kumar. Automated control syntesis for and assembly line using discrete event system control theory. *IEEE Transactions on Systems, Man, and Cybernetics: Part C*, 33(2):284–289, 2003.

[24] F. Charbonnier, H. Alla, and R. David. The supervised control of discrete-event dynamic systems. *IEEE Transactions on Control Systems Technology*, 7:175–187, 2003.

[25] H. Chen. Net structure and control logic synthesis of controlled Petri nets. *IEEE Transactions on Automatic Control*, 43(10):1446–1450, 1998.

[26] H. Chen. Control synthesis of Petri nets based on s-decreases. *Discrete Event Dynamic Systems: Theory and Applications*, 10(3):233–250, 2000.

[27] H. Chen and H.-M. Hanish. Control synthesis of timed discrete event systems based on predicate invariance. *IEEE Transactions on Systems Man and Cybernetics*, 30(5):713–724, 2000.

[28] H. Chen and B. Hu. Monitor-based control of a class of controlled Petri nets. In *Proceedings of the 3rd International Conference on Automation, Robotics and Computer Vision*, 1994.

[29] 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.

[30] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[31] J. Cortadella, A. Kondratyev, L. Lavagno, M. Massot, S. Moral, C. Passerone, Y. Watanabe, and A. Sangiovanni-Vincentelli. Task generation and compile-time scheduling for mixed data-control embedded software. In *Proceedings of the Design Automation Conference*, pages 489–494, 2000.

[32] 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.

[33] 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)*, pages 29–40. IEEE Computer Society, June 2003.

[34] J.-P. Courtiat, J. M. Ayache, and B. Algayres. Petri nets are good for protocols. In *ACM, SIGCOMM'84 Tutorials and Symposium, Communications Architectures and Protocols*, pages 66–74, 1984.

[35] P. Darondeau and X. Xie. Linear control of live marked graphs. *Automatica*, 39(3):429–440, 2003.

[36] G. de Jong and B. Lin. A communicating Petri net model for the design of concurrent asynchronous modules. In *Proceedings of the 31st Annual Conference on Design Automation (DAC '94)*, pages 49–55. ACM Press, 1994.

[37] J. Ezpeleta, J. M. Colom, and J. Martínez. A Petri net based deadlock prevention policy for flexible manufacturing systems. *IEEE Transactions on Robotics and Automation*, 11(2):173–184, 1995.

[38] M. Fabian and A. Hellgren. PLC-based implementation of supervisory control for discrete event systems. In *Proceedings of the 37th IEEE Conference on Decision and Control*, pages 3305–3310, 1998.

[39] H. Fleischhack and B. Grahlmann. A compositional Petri net semantics for sdl. In Desel J. and Silva M., editors, *Application and Theory of Petri Nets*, volume 1420 of *Lecture Notes in Computer Science*, pages 144–164. Springer-Verlag, 1998.

[40] J. Flochova, F. Auxt, M. Radakovic, and O. Jombik. PNDesigner–a tool designed for model based diagnosis and supervisory control of DES. In *Proceedings of the 8th International Workshop on Discrete Event Systems*, pages 471–472, 2006.

[41] G. Gardey, D. Lime, M. Magnin, and O. H. Roux. Romeo: A tool for analyzing time Petri nets. In *Proceedings of the 17th International Conference on Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 418–423, 2005.

[42] G. Gardey, O. F. Roux, and O. H. Roux. Safety control synthesis of time Petri nets. In *Proceedings of the 8th International Workshop on Discrete Event Systems*, pages 222–228, 2006.

[43] A. Ghaffari, N. Rezg, and X. Xie. Feedback control logic for forbidden-state problems of marked graphs: application to a real manufacturing system. *IEEE Transactions on Automatic Control*, 48(1):2–17, 2003.

[44] H. Goldstein. Cure for the multicore blues. *IEEE Spectrum*, 44(1):41–43, 2007.

[45] B. Grahlmann. The PEP tool. In Grumberg, O., editor, *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 440–443. Springer-Verlag, 1997.

[46] B. Grahlmann, M. Moeller, and U. Anhalt. A new interface for the PEP tool - parallel finite automata. In Desel, J. and Fleischhack, H. and Oberweis, A. and Sonnenschein, M., editor, *Workshop Algorithmen und Werkzeuge für Petrinetze*, volume 22 of *AIS*, pages 21–26. 1995.

[47] A. Gromyko, M. Pistore, and P. Traverso. A tool for controller synthesis via symbolic model checking. In *Proceedings of the 8th International Workshop on Discrete Event Systems*, pages 475–476, 2006.

[48] X. Guan and L.E. 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.

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

[50] K.X. He and M.D. Lemmon. On the transformation of maximally permissive marking-based supervisors into monitor supervisors. In *Proceedings of the IEEE Conference on Decision and Control*, pages 2657–2662, December 2000.

[51] 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*, 30(5):696–712, 2000.

[52] L. E. Holloway, B. H. Krogh, and A. Giua. A survey of Petri net methods for controlled discrete event systems. *Discrete Event Dynamic Systems*, 7(2):151–190, 1997.

[53] L.E. Holloway, X. Guan, and L. Zhang. A generalization of state avoidance policies for controlled Petri nets. *IEEE Transactions on Automatic Control*, 41(6):804–816, 1996.

[54] L.E. Holloway and B.H. Krogh. Synthesis of feedback control logic for a class of controlled Petri nets. *IEEE Transactions on Automatic Control*, 35(5):514–523, 1990.

[55] 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*, pages 208–213. ACM Press, 2001.

[56] J. Huang and R. Kumar. Nonblocking directed control of discrete event systems. In *Proceeding of the 2005 IEEE Conference on Decision and Control*, 2005.

[57] M. V. Iordache and P. J. Antsaklis. Software tools for the supervisory control of Petri nets based on place invariants. Technical report isis-2002-003, University of Notre Dame, April 2002.

[58] M. V. Iordache and P. J. Antsaklis. Design of T-liveness enforcing supervisors in Petri nets. *IEEE Transactions on Automatic Control*, 48(11):1962–1974, 2003.

[59] M. V. Iordache and P. J. Antsaklis. Synthesis of supervisors enforcing general linear vector constraints in Petri nets. *IEEE Transactions on Automatic Control*, 48(11):2036–2039, 2003.

[60] M. V. Iordache and P. J. Antsaklis. Decentralized supervision of Petri nets. *IEEE Transactions on Automatic Control*, 51(2):376–381, 2006.

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

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

[63] M. V. Iordache and P. J. Antsaklis. Petri net supervisors for disjunctive constraints. In *Proceedings of the 2007 American Control Conference*, pages 4951–4956, 2007.

[64] M. V. Iordache and P. J. Antsaklis. Petri nets and programming: A survey. In *Proceedings of the 2009 American Control Conference*, pages 4994–4999, 2009.

[65] K. Åkesson, M. Fabian, H. Flordal, and R. Malik. Supremica – an integrated environment for verification, synthesis and simulation of discrete event systems. In *Proceedings of the 8th International Workshop on Discrete Event Systems*, pages 384–385, 2006.

[66] S. R. Kosaraju. Limitations of Dijkstra's semaphore primitives and Petri nets. *Operating Systems Review*, 7(4):122–126, 1973.

[67] B.H. Krogh and L.E. Holloway. Synthesis of feedback control logic for manufacturing systems. *Automatica*, 27(4):641–651, 1991.

[68] O. Kupferman, N. Piterman, and M.Y. Vardi. Safraless compositional synthesis. In *Proceedings of the 18th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

[69] O. Kupferman and M. Vardi. Church's problem revisited. *The Bulletin of Symbolic Logic*, 5(2):245–263, 1999.

[70] S. Lafortune. Umdes-lib software library. http://www.eecs.umich.edu/umdes/toolboxes.html.

[71] K. Lautenbach and P. S. Thiagarajan. Analysis of a resource allocation problem using Petri nets. In *Proceedings of the 1st European Conference on Parallel and Distributed Processing*, pages 260–266. Cepadues Editions, 1979.

[72] E. A. Lee. Cyber-physical systems – are computing foundations adequate? In *NSF Workshop On Cyber-Physical Systems: Research Motivation, Techniques and Roadmap*, Austin, TX, October 2006.

[73] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1):24–35, 1987.

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

[75] 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*, pages 3069–3074, 2000.

[76] J. R. Levine, T. Mason, and D. Brown. *Lex and Yacc*. O'Reilly & Associates, second edition, 1992.

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

[78] 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.

[79] J. O. Moody and P. J. Antsaklis. *Supervisory Control of Discrete Event Systems Using Petri Nets*. Kluwer Academic Publishers, 1998.

[80] J. O. Moody and P. J. Antsaklis. Petri net supervisors for DES with uncontrollable and unobservable transitions. *IEEE Transactions on Automatic Control*, 45(3):462–476, 2000.

[81] T. Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, pages 541–580, April 1989.

[82] T. Murata, B. Shenker, and S. M. Shatz. Detection of Ada static deadlocks using Petri net invariants. *IEEE Transactions on Software Engineering*, 15(3):314–326, 1989.

[83] M. Notomi and T. Murata. Hierarchical reachability graph of bounded petri nets for concurrent-software analysis. *IEEE Transactions on Software Engineering*, 20(5):325–336, 1994.

[84] G. Nutt. *Operating Systems*. Addison Wesley, 2003.

[85] J. Park and S. Reveliotis. Deadlock avoidance in sequential resource allocation systems with multiple resource acquisitions and flexible routings. *IEEE Transactions on Automatic Control*, 46(10):1572–1583, 2001.

[86] J. Park and S. Reveliotis. Liveness-enforcing supervision for resource allocation systems with uncontrollable behavior and forbidden states. *IEEE Transactions on Robotics and Automation*, 18(2):234–240, 2002.

[87] E. Pastor, O. Roig, J. Cortadella, and R. M. Badia. Petri net analysis using boolean manipulation. In *Application and Theory of Petri Nets '94*, volume 815 of *Lecture Notes in Computer Science*, pages 416–435. Springer-Verlag, 1994.

[88] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, Englewood Cliffs, NJ, 1981.

[89] J. L. Peterson and A. Silberschatz. *Operating Systems Concepts*. Addison-Wesley, 1985.

[90] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 179–190, 1989.

[91] W. Reisig. *Petri Nets*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[92] S. Reveliotis. *Real-Time Management of Resource Allocation Systems: A Discrete Event Systems Approach*. Springer-Verlag, 2005.

[93] A. Sathaye. Synthesis of real-time supervisors for controlled time Petri nets. In *Proceedings of the 32nd International Conference on Decision and Control*, pages 235–236, 1993.

[94] A. Sathaye and B. Krogh. Logical analysis and control of time Petri nets. In *Proceedings of the 31st International Conference on Decision and Control*, pages 1198–1203, 1992.

[95] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli. Quasi-static scheduling of embedded software using equal conflict nets. In Donatelli, Susanna and Kleijn, Jetty, editors, *20th International Conference on Application and Theory of Petri Nets 1999 (ICATPN'99)*, volume 1630 of *Lecture Notes in Computer Science*, pages 208–227. Springer-Verlag, 1999.

[96] 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)*, pages 805–810, 1999.

[97] 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*, pages 595–602, 1989.

[98] 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*, pages 477–478, 2006.

[99] C. Stehno. Real-time systems design with PEP. In Katoen, J.-P. and Stevens P., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 476–480. Springer-Verlag, 2002.

[100] K. Strehl, L. Thiele, D. Ziegenbein, R. Ernst, and J. Teich. Scheduling hardware/software systems using symbolic techniques. In *International Workshop on Hardware/Software Codesign*, pages 173–177, 1999.

[101] G. Stremersch. *Supervision of Petri Nets*. Kluwer Academic Publishers, 2001.

[102] G. Stremersch and R. K. Boel. Structuring acyclic Petri nets for reachability analysis and control. *Discrete Event Dynamic Systems*, 12(1):7–41, 2002.

[103] F.-S. Su and P.-A. Hsiung. Extended quasi-static scheduling for formal synthesis and code generation of embedded software. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 211–216. ACM Press, 2002.

[104] Z. Suraj. Resource allocation problem. In *Proc. of the 3rd Symp. on Math. Foundations of Comput. Science, Zaborow 1980, ICS PAS Reports*, pages 83–86, 1980.

[105] T. Suzuki, S. M. Shatz, and T. Murata. A protocol modeling and verification approach based on a specification language and Petri nets. *IEEE Transactions on Software Engineering*, 16(5):523–536, 1990.

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

[107] F. Tricas, F. Garcia-Valles, J. M. Colom, and J. Ezpeleta. New methods for deadlock prevention and avoidance in concurrent systems. *Actas de las Jornadas de Concurrencia 2000*, pages 97–110, June 2000.

[108] S. Tu, S. Shatz, and T. Murata. Applying Petri nets reduction to support Ada-tasking deadlock detection. In *Proceedings of the 10th IEEE International Conference on Distributed Computing Systems*, pages 96–102, 1990.

[109] 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)*, pages 16–23, 1997.

[110] 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*, pages 287–292, 2008.

[111] W. M. Wonham. Supervisory control of discrete event systems and design software. Department of Electrical and Computer Engineering, University of Toronto, http://www.control.toronto.edu/DES.

[112] K. Xing, B. Hu, and H. Chen. Deadlock avoidance policy for Petri net modeling of flexible manufacturing systems with shared resources. *IEEE Transactions on Automatic Control*, 41(2):289–295, February 1996.

[113] M. Zhou and M. P. Fanti. *Deadlock Resolution in Computer-Integrated Systems*. Marcel Dekker, Inc., 2005.

[114] X. Zhu and B. Lin. Compositional software synthesis of communication processes. In *Proceedings of the International Conference on Computer Design*, pages 646–651, 1999.

[115] W. M. Zuberek. Petri net models of process synchronization mechanisms. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, pages 841–847, 1999.