2010 American Control Conference
Marriott Waterfront, Baltimore, MD, USA
June 30-July 02, 2010

Concurrent Program Synthesis Based on Supervisory Control

Marian V. Iordache and Panos J. Antsaklis

Abstract— **The paper introduces a new area of application of the supervisory control (SC) methods and a project dealing with this research topic. Based on the observation that various constraints on the operation and synchronization of concurrent processes can be expressed in terms of SC specifications, the paper proposes the application of SC to the automation of concurrent program synthesis. Specifically, the paper proposes a three-stage approach allowing to generate automatically the part of the programs that deals with the coordination of concurrent processes. In a first stage, Petri net models are extracted from a high level specification. An SC specification is also extracted. Then, SC is applied to generate the supervisor enforcing the specification. Finally, the programs representing the processes and the supervisor are generated. This work is motivated by the difficulty of writing correctly concurrent programs. Since this difficulty is due to the constraints on the operation and synchronization of concurrent processes, research in this area has the potential of simplifying the development of concurrent programs.**

I. Introduction

The development of correct software can be difficult and expensive, especially in the context of concurrency. Thus, tools that can automate software development 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. In the context of concurrency, the main difficulty is writing the program segments that ensure that the concurrent processes satisfy desired operation and synchronization constraints. This paper considers the automatic synthesis of these program segments and introduces an approach based on Supervisory Control (SC). This work is part of a project for the development of a concurrency tool suite (ACTS) for the synthesis of concurrent programs [1], [17]. The goal is to design software that based on a high level specification can generate concurrent programs.

The ACTS architecture is outlined in Figure 1. Given is a specification written in a high level specification language (HLL). Next, an analysis tool is applied to the high level specification in order to extract a plant model of the concurrent processes and also a specification for SC. Then, SC is applied to generate a supervisor. Finally, the plant and the supervisor are translated to programming code. All these steps are carried out transparently and automatically, based on the high level specification. The benefit of this approach is that the programmer would focus on a concise high level description, instead 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.

The paper is organized as follows. First, an illustrative example is given in section II. Next, the Petri net representation of programs is described in section III. The code generation approach is described in section IV. Issues related to the design of the HLL are addressed in section V. The application of SC methods is discussed in section VI. Related literature results are discussed in section VII. Additional information about this project can be found in [1].

II. Illustrative Example

Here we consider the problem of designing control software for an assembly operation in a manufacturing line. 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.

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 2 and the SC specification given in the inequalities (1)–(4). In the plant model of Figure 2, the processing sequence is shown to the left and the states of the conveyor to the right. To incorporate the effect of processing delays, a processing step is modeled by a controllable transition, an uncontrollable transition, and a place, as shown in Figure 3. The controllable transition is fired when the command is issued and the uncontrollable transition is fired after the

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

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

The authors gratefully acknowledge the support of the National Science Foundation (NSF CNS-0834057).

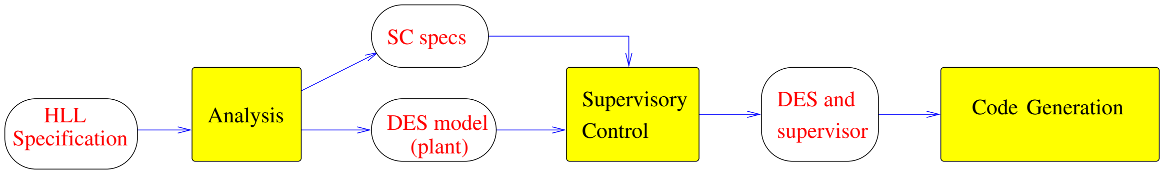

Fig. 1. Outline of the program synthesis approach.

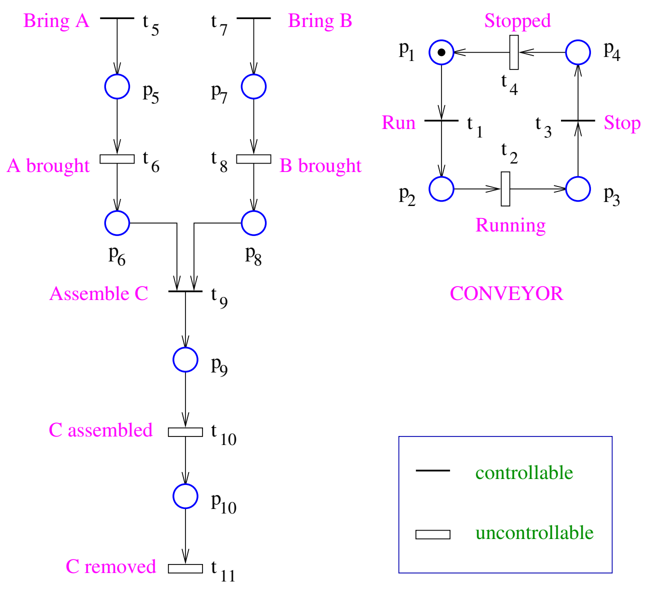

Fig. 2. Plant model.

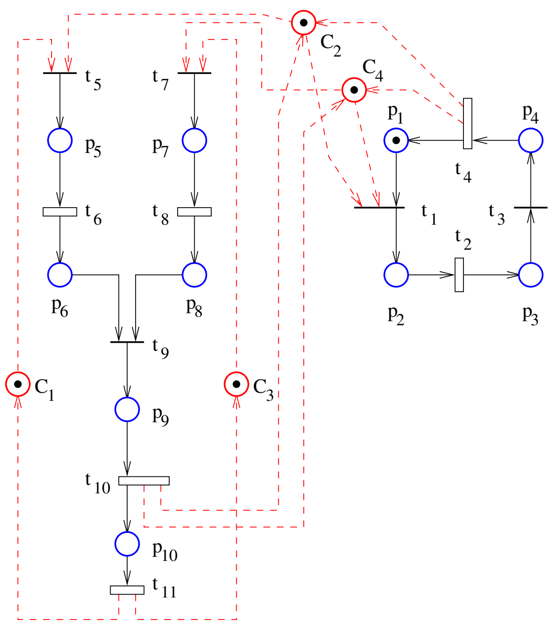

Fig. 4. Plant and supervisor.

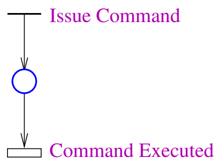

Fig. 3. A possible way to model processing delays.

command has been executed. For instance, in Figure 2, t_1 is fired when the command to turn on the conveyor is issued and t_2 is fired when the conveyor is on. Finally, the following inequalities on the marking of the PN express the remaining requirements of the specification. Note that μ_i denotes the marking of the place p_i.

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

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.

The supervisor enforcing (1)–(4) could be designed following any of the literature SC approaches. The procedure presented in this paper is not limited to a particular set of SC methods. If [25] and [14] are followed, the inequalities (1)–(4) are transformed to an admissible form that accounts for the partial controllability of the plant:

$$\mu_5 + \mu_6 + \mu_9 + \mu_{10} \leq 1 \quad (5)$$
$$\mu_5 + \mu_6 + \mu_9 + \mu_2 + \mu_3 + \mu_4 \leq 1 \quad (6)$$
$$\mu_7 + \mu_8 + \mu_9 + \mu_{10} \leq 1 \quad (7)$$
$$\mu_7 + \mu_8 + \mu_9 + \mu_2 + \mu_3 + \mu_4 \leq 1 \quad (8)$$

The inequalities (5)–(8) are implemented by the places $C_1, \ldots, C_4$ shown in Figure 4.

In this example we could associate software processes with the operation sequences of the parts A, B, and C and with the conveyor. SC provides a way to generate a coordination strategy of these processes such that the specified outcome is achieved. While here we have used a manufacturing example, coordination needs arise also in purely software applications.

III. Petri Net Representation of Programs

In this project, a program consists of a number of processes running concurrently. The structure of each process

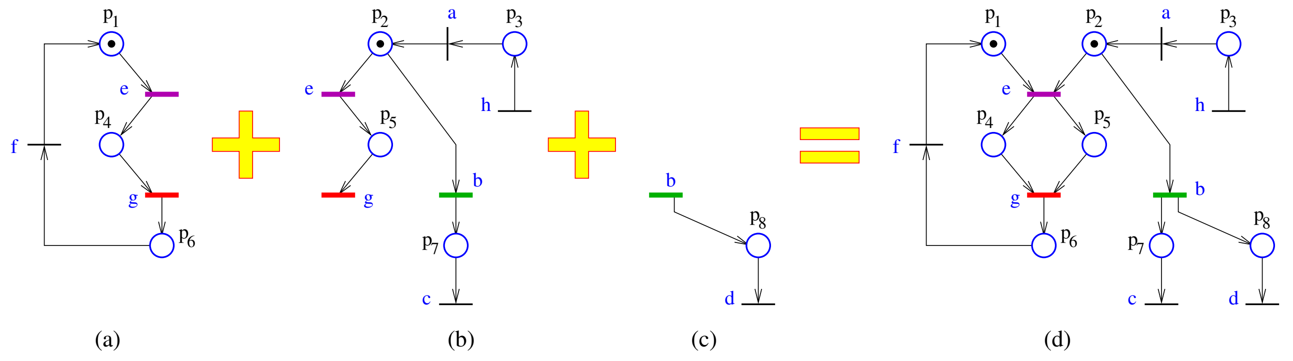

Fig. 5. PNs that represent programs are a composition of state machine components. Note that the transitions with the same label are composed.

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. As a token moves from one place to another, the execution of the corresponding 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. We will denote by HPN (high level PN) a PN in which places are labeled with instructions and transitions with conditions.

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.

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 choice of the next transition may be 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.

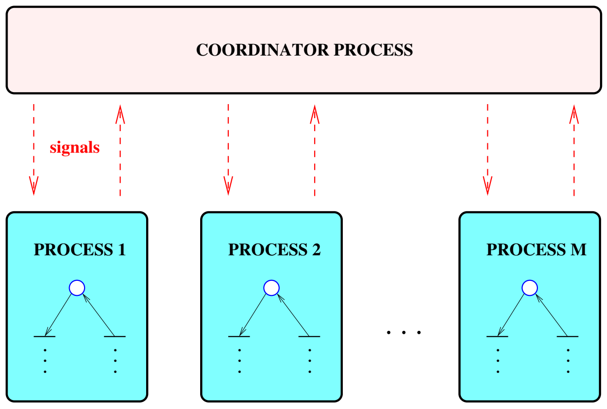

Fig. 6. Implementation of the specification.

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 apparent that the reachable stages and transitions of any software 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. An example of composition of state machine components into a PN is shown in Figure 5. Therefore, without loss of generality, state machines rather than general PNs can be associated with processes. However, unlike to the typical definition of state machines, note that here arbitrary markings and arbitrary arc weights are allowed. Moreover, note that associating state machines with processes does not simplify the SC problem, since the parallel composition of state machines can result in arbitrary PNs that are not necessarily state machines.

IV. Code Generation

Code is generated according to the coordinator architecture shown in Figure 6. Thus, the result of software

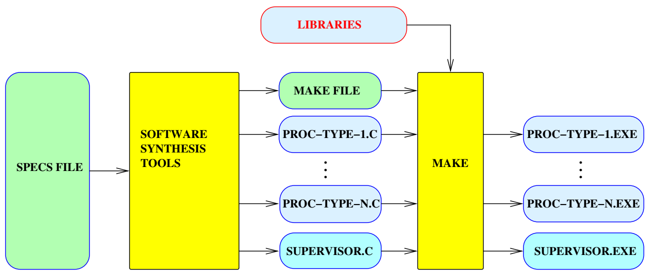

Fig. 7. How the software synthesis tools are applied.

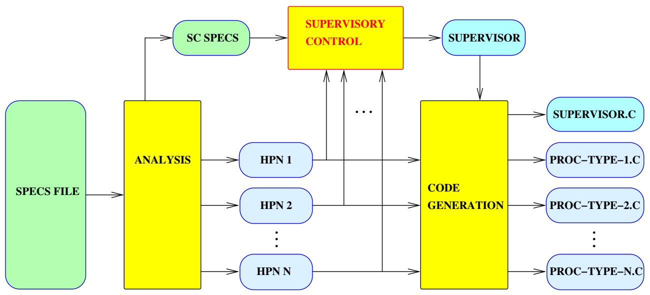

Fig. 8. The software synthesis procedure. HPN stands for high level PN.

synthesis consists of a number of *application processes* and a *coordinator process*. The application processes correspond to processes defined in the specification. The coordinator process corresponds to the software implementation of the constraints given in the specification. Thus, the coordinator process represents the supervisor generated by means of SC. The supervisor (coordinator) exchanges messages with the other processes to ensure that their operation respects the constraints given in the specification. Note that the number of processes is variable. Processes may terminate and new processes may be created, as described in the specification. While Figure 6 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 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 7, where the software synthesis procedure is outlined in Figure 8.

The coordinator approach is illustrated here on the example of section II. The coordinator will use four variables $m_1 \ldots m_4$, each corresponding to the marking of the supervisor places $C_1 \ldots C_4$. The coordinator process is notified by the conveyor process each time t_4 is fired and by the assembly process each time t_{10} and t_{11} are fired. Further, before firing one of t_1, t_5, or t_7, the conveyor and the assembly processes request permission from the coordinator. The coordinator determines whether t_1, t_5, and t_7 are enabled or disabled based on the solution shown in Figure 4. For instance, the coordinator enables t_1 if $m_2 \geq 1$ 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.

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.

Transitions are classified as follows. A transition is *controlled* if the supervisory policy disables it in certain circumstances. A transition is *observed* if its firings must be communicated to the supervisor. Further, t is a *synchronization transition* if $\bullet t \neq \emptyset$ and there is t' of a different process type that has the same label and satisfies $\bullet t' \neq \emptyset$. Moreover, if two transitions of two different process types 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*. For example, in Figure 5(a)–(c) the transitions of label e are synchronization transitions and the transitions of label b 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 is notified in order to generate the corresponding new processes. Based on the communications needs for each transition and the supervisory policy, code generation algorithms can be developed [17].

V. The Specification Language

As previously mentioned, in our approach (Figure 1) the specification is given in a high level specification language (HLL). As shown in Figure 8, 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.

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

VI. Supervisory Control

In the context of concurrent programs, SC specifications could involve constraints expressed by inequalities, such as (1)–(4), constraints involving disjunctions of inequalities [15], and constraints described in terms of languages. The ability to enforce liveness or reversibility is also of great importance. Specific to concurrent programming applications is also the fairness requirement. In its most basic form, it requires that regardless of external events, a process waiting for a resource will eventually get access to it. Some of the fairness constraints can be expressed in terms of marking inequalities or Parikh vector inequalities [13].

While some of the aforementioned specifications have simple solutions when the plant is fully controllable and observable, note that partial controllability and observability can arise in certain contexts. 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 [8]).
- For certain SC problems (such as liveness enforcement), transitions labeled by conditions have to be considered uncontrollable.

Currently, the SC methods used in this project are those implemented in [12]. However, all available SC methods are of interest, including automata based methods. Of special interest for future work are algorithms and heuristics to find the methods that suit best a given plant and specification.

VII. Literature Review

Due to the advent of multicore microprocessors, software tools that convert sequential code to parallel code have become increasingly important. However, such tools do not affect the need for concurrent programming. Excepting special cases, they cannot convert a sequential algorithm to a parallel algorithm. Thus, concurrent programming is still necessary and remains 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. When this approach is applied, tools converting sequential code to parallel code remain very useful, as they could improve the execution time of the sequential segments of code of the concurrent processes. Note that our project deals with the development of concurrent specifications and not with the conversion of sequential code to parallel code.

Related to the SC theory is the approach for program synthesis for reactive systems [19], [28]. The problem is to synthesize a program based on a specification described in temporal logic. In terms of the SC terminology, a program would correspond to a supervisor. While currently our project does not consider temporal logic specifications, these and other classes of specifications could be incorporated in the future in order to increase the area of applications.

The modeling and analysis of concurrent programs using PNs has been considered before, such as in [7] and references therein. A software tool PEP has been created for the development, verification, and simulation of parallel programs [2], [9]. 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. Among other PN based verification approaches, we mention [6], [26], [27] for Ada programs. Note that our project is on correct-by-construction synthesis, not on verification.

The scheduling problem, dealing with the execution order of concurrent tasks, has been approached based on PN models in references such as [8], [20], [23], [24], [30]. Typically, the results are on the sequential execution of concurrent programs on hardware with a single computational resource. Most often reachability analysis is used for synthesis, though there are also structural results, such as in [24]. 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.

Related is also the work on hardware/software codesign of [5]. There, the specification is written in a language such as Esterel [10], from which a network of codesign finite state machines (CFSMs) is extracted. Note that networks of CFSMs correspond to safe PNs [16]. Compared to our project, while individual processes are modeled by state machines, the parallel composition of the process models results in PNs that are typically not safe. Moreover, we intend to use specifications at a higher level.

An approach for finding and correcting potential deadlock situations in software appears in [32]. Given a program, a PN model is extracted first. 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. Compared to our project, we deal with program synthesis instead of programs that are already written. 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.

The application of SC methods to software engineering has also been considered in [21], [22]. Moreover, certain computer science methods, such as predicate control for distributed computations [31] and the aforementioned scheduling approaches, can be seen as SC methods [16]. The approaches used to generate control software are also related to the SC. In [11], [29], control software is obtained based on condition system models. Given a condition system model and a specification language describing a sequence of states that the system should follow, control software is automatically generated [3]. Control software can also be generated using the tool Supremica [18], [4] 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 [11], [29], we intend to use more general specifications.

VIII. Conclusion

This paper proposes the application of supervisory control (SC) for the automatic synthesis of concurrent programs. The proposed approach involves three steps. Starting with a program description written in a high level specification language, a Petri net model and an SC specification is extracted. Then, a supervisor is generated using SC methods. Finally, the supervisor is converted to low level code. 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.

References

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

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

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

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

[5] F. Balarin et al. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Publishers, 1997.

[6] 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.

[7] E. Best, R. Devillers, and M. Koutny. Petri nets, process algebras and concurrent programming languages. In *Lectures on Petri Nets II: Applications*, vol. 1492 of *LNCS*, pages 1–84. Springer, 1998.

[8] J. Cortadella et al. Task generation and compile-time scheduling for mixed data-control embedded software. In *Proc. Design Automation Conf.*, pages 489–494, 2000.

[9] 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.

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

[11] L. E. Holloway, X. Guan, and R. Sundaravadivelu. Automated synthesis and composition of taskblocks for control of manufacturing systems. *IEEE Trans. Syst. Man Cyber. Part B*, 30(5):696–712, 2000.

[12] 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.

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

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

[15] M. V. Iordache and P. J. Antsaklis. Petri net supervisors for disjunctive constraints. In *Proc. 2007 ACC*, pages 4951–4956, 2007.

[16] M. V. Iordache and P. J. Antsaklis. Petri nets and programming: A survey. In *Proc. 2009 ACC*, pages 4994–4999, 2009.

[17] M. V. Iordache and P. J. Antsaklis. Synthesis of concurrent programs based on supervisory control. Technical report isis-2009-005, University of Notre Dame, September 2009.

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

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

[20] 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.

[21] M. Lemmon and K. He. Supervisory plug-ins for distributed software. In *Proc. Workshop on Software Engineering and Petri Nets*, pages 155–172. University of Aarhus, 2000.

[22] M. Lemmon, K. He, and S. Shatz. Dynamic reconfiguration of software objects using Petri nets and network unfolding. In *Proc. IEEE Internat. Conf. Syst. Man Cyber.*, pages 3069–3074, 2000.

[23] B. Lin. Software synthesis of process-based concurrent programs. In *Proc. Design Automation Conf.*, pages 502–505, 1998.

[24] C. Liu et al. Schedulability analysis of Petri nets based on structural properties. In *IEEE Internat. Conf. on Application of Concurrency to System Design*, 2006.

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

[26] 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.

[27] 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.

[28] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. ACM Symp. Principles of Programming Languages*, pages 179–190, 1989.

[29] 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 *Proc. 8th Internat. Workshop on Discrete Event Systems*, pages 477–478, 2006.

[30] F.-S. Su and P.-A. Hsiung. Extended quasi-static scheduling for formal synthesis and code generation of embedded software. In *Proc. Internat. Symp. on Hardware/Software Codesign*, pages 211–216, 2002.

[31] 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.

[32] Y. Wang, T. Kelly, M. Kudlur, S. Mahlke, and S. Lafortune. The application of supervisory control to deadlock avoidance in concurrent software. In *Proc. 9th Internat. Workshop on Discrete Event Systems*, pages 287–292, 2008.