The ACTS Software and its Supervisory Control Framework

Marian V. Iordache and Panos J. Antsaklis

Abstract— **In recent work we have developed software implementing a supervisory control approach to concurrent programming. Starting with a specification describing the concurrency constraints, the software generates automatically the concurrency control code implementing the specification. The paper describes the approach with an emphasis on the supervisory control aspects. Included are also results pertaining to limitations and future extensions of the supervisory control approach.**

I. Introduction

In recent work we have completed the first fully functional version of a concurrency tool suite (ACTS) for concurrent programming [1]. In this approach, instead of writing manually the code that ensures requirements such as mutual exclusion and fairness, a specification file is written, describing the coordination requirements. Then, the coordination code is generated automatically based on the requirements described in the specification. In order to generate the code, the specification is processed in three stages (Figure 1). First, the specification is converted to a discrete event representation using Petri nets. Second, a supervisor is designed in accordance with the specification requirements. The role of the supervisor is to restrict the operation of the Petri net such that the requirements of the specification are satisfied. Once the supervisor has been designed, the coordination code is generated.

In the literature, applications of supervisory control to software engineering have been proposed also in [2], [3], [4], [5], [6]. Moreover, certain methods that have been applied in the context of software engineering are closely related to supervisory control, as shown in the survey [7]. To our knowledge, our approach differs considerably from prior work involving methods related to supervisory control. The most closely related work is as follows. The application of supervisory control for concurrent programming has been proposed also in [4], [5]. In comparison with [4], [5], one difference is that we consider arbitrary synchronizations, including synchronizations that cannot be implemented (exclusively) by locks[1]. Moreover, our methods rely on Petri nets instead of automata. Related is also the work on the GADARA software [6], [8]. There, Petri net methods are used to correct programs by inserting code that prevents potential deadlock situations. However, note that our approach deals with a different problem, that of generating the concurrency control code of concurrent programs.

In computer science and engineering there has been a considerable amount of work on automating concurrent programming. We mention here Intel's Concurrent Collections for C++ [9], [10] and the work on irregular parallelism, such as in [11], [12]. While there are various classes of problems on which our supervisory control approach cannot be as efficient as other methods, the main benefit of our approach is the ability to deal with complex coordination constraints.

The contribution of this paper could be described as follows. The general approach of our work was introduced in [13]. This paper describes the implementation of the approach of [13], with emphasis on supervisory control aspects. Additionally, we consider the performance of conventional supervisory control methods in the context of software systems and present several new results. To our knowledge, ACTS is the first software to implement a supervisory control approach to the synthesis of concurrency control code.

The paper is organized as follows. The software features are outlined in section II. An example illustrating supervisory control constraints is included in section III. The program synthesis procedure is described in section IV. Finally, theoretical guarantees and limitations are considered in section V.

II. Software Features

The input of the software is a specification describing the concurrent entities that should be coordinated and the concurrency constraints. Each user defined entity consists of a DES description and user code associated with the states and transitions of the DES. Based on the given specification, the software generates a program consisting of the user code and the implementation of the concurrency constraints. The program is generated in the C language. The concurrent entities of the program are implemented as processes or POSIX threads, depending on the user choice.

Each entity is described as a state machine in which the states and transitions are associated with user code. Each state of the state machine represents a stage of execution. Thus, the code associated with a state of the state machine defines the operations of a stage of execution. Moreover, the code associated with transitions is used in order to

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).

[1]A lock is a synchronization mechanism in concurrent programming.

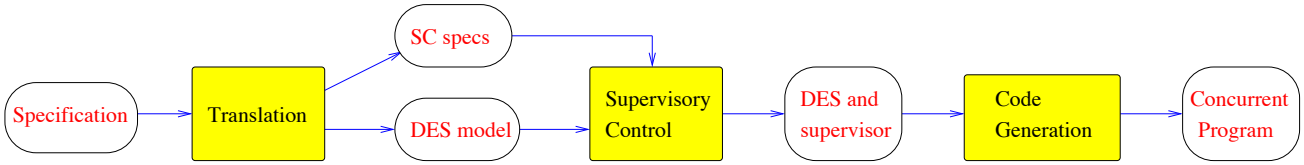

Fig. 1. The program synthesis approach.

determine which transition to fire when there is a choice. The state machines may have source and sink transitions. A source (sink) transition represents process/thread creation (termination).

Note that we view state machines as a special class of Petri nets in which each transition has at most one input place and at most one output place. Each state machine represents the structure of a concurrent entity (a process or a thread). Each token represents a concurrent entity. The place containing the token denotes the stage of execution of the entity. Any number of identical entities may be associated with the same state machine. The number of tokens of a state machine represents the number of entities associated with the state machine.

Arbitrary synchronizations between state machine transitions can be created by listing the transitions with the same label. Two or more transitions with the same label are not synchronized when they belong to the same state machine.

Specifications may declare also uncontrollable and/or unobservable transitions. It is possible to declare also a list of transitions that should be live. This constrains the supervisory control methods to guarantee liveness for the specified transitions. Concurrency constraints can be expressed by means of inequalities of the form

$$\sum l_i \mu_i + \sum h_i q_i + \sum c_i v_i \leq d \qquad (1)$$

where the coefficients d, l_i, h_i, and c_i represent integer constants, μ_i is the marking of the place i, q_i is a variable denoting whether a transition t_i is fired (the element i of the firing vector), and v_i indicates how many times t_i has fired (the element i of the Parikh vector). The constraint (1) requires that all reachable states satisfy $\sum l_i \mu_i + \sum c_i v_i \leq d$ and that a transition t_j is fired only if $h_j \leq d - \sum l_i \mu_i - \sum c_i v_i$ (where μ_i and v_i are the values *before* t_j is fired).

The specification may describe also state machines that are not associated with a concurrent entity. These can be used to express P-type language constraints. They are called *supervisor components*, since they are incorporated in the supervision policy generated by the software. For instance, the example of section III introduces the Petri net of Figure 3 as a supervisory component in order to describe constraints that cannot be expressed by inequalities (1) alone.

Note that specifications are written in a custom specification language. For syntax details we refer the reader to [1].

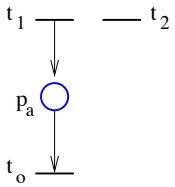

Fig. 3. Additional net used to express the fairness constraint.

III. Example

Consider a problem involving three thread types: readers, writers, and inserters, where the threads work on a shared region of memory. Assume five readers, three deleters, and three inserters with the state machine structure shown in Figure 2. Assume also that p_c, p'_c, and p''_c denote critical states of the three thread types. The constraint that only one inserter thread may be in the critical section could be written as $\mu''_c \leq 1$ where μ''_c denotes the marking of the place p''_c. Moreover, the requirement that only one deleter may be in the critical section and no reader or inserter may be in the critical section at the same time could be written as

$$\mu'_c \leq 1 \wedge (\mu'_c = 0 \vee \mu_c + \mu''_c = 0) \qquad (2)$$

Note that $\wedge$ denotes conjunction (logic "and") and $\vee$ denotes disjunction (logic "or"). Thus, (2) requires $\mu'_c \leq 1$ and either of $\mu'_c = 0$ or $\mu_c + \mu''_c = 0$. Since the maximum number of readers and inserters in the critical section is six, (2) can be written more compactly as

$$6\mu'_c + \mu_c + \mu''_c \leq 6 \qquad (3)$$

Now, if readers are continuously present in the critical section, no deleter can enter because of constraint (3). A possible fairness constraint would be to limit the number of readers that can enter the critical section while a deleter is waiting. This constraint can be expressed by means of an additional place p_a and three additional transitions t_1, t_2, and t_o (Figure 3) as follows.

- Synchronize t_1 and t_2 with t_v. This means that when t_v fires, either t_1 or t_2 must be fired.
- Require $q_2 \leq \mu'_v$, expressing that t_2 may not fire when $\mu'_v = 0$ (i.e. when no deleter is waiting).
- Require $3q_1 + 3q_o \leq 3 - \mu'_v + 3\mu'_c$, expressing that t_1 and t_o may not fire when a deleter is waiting and no deleter is in the critical section.

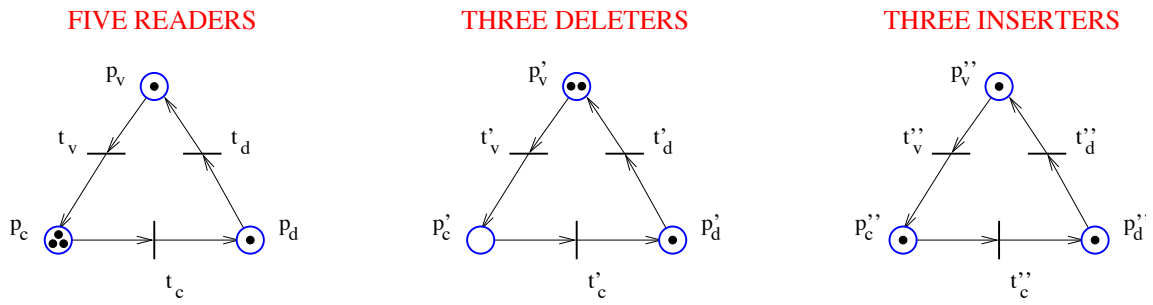

Fig. 2. The example of section III.

- Require $\mu_a \leq 5$, which will limit to 5 the number of reader threads that can enter the critical section while a deleter thread is waiting.

Note that transitions are fired as soon as enabled. Thus, t_o will remove all tokens of p_a as soon as it is allowed to fire. The supervisor enforcing the constraints is shown in Figure 4.

IV. Program Synthesis and Operation

The synthesis process is outlined in Figure 1. In the first stage the specification is parsed. As already mentioned, each concurrent entity of the specification corresponds to a state machine and is represented by a token of that state machine. Multiple tokens of the same state machine represent entities of the same type, that is, identical entities executed concurrently. Now, in the supervisory control setting, the plant is a Petri net (PN) representing the parallel composition of the state machines describing the execution stages of the concurrent entities. The conventional parallel composition algorithm is used [14] in which the PN transitions represent the possible synchronizations between the state machine transitions with the same label. For instance, in Figure 4, note that $t_{v,1}$ and $t_{v,2}$ are the synchronizations of t_v with t_1 and t_2, respectively.

Supervisory control is applied to the PN representing the parallel composition of the state machines. Note that the transitions of the PN may not be the same as the transitions of the state machines. Thus, the q and v terms of the specifications (1) are converted to account for the different transitions of the PN. For instance, in the context of section III, if a constraint involved q_v or v_v, then q_v (v_v) would be replaced by $q_{v,1} + q_{v,2}$ ($v_{v_1} + v_{v,2}$), since firing t_v in its state machine corresponds to firing either $t_{v,1}$ or $t_{v,2}$ in the PN. Now, the supervisory control methods are applied first in order to enforce the constraints (1). Then, if the specification requests explicitly T-liveness enforcement, the T-liveness enforcement procedure of [14] is applied. The end result of the supervisory control stage is a PN representing the supervisor.

The last stage of the synthesis process (Figure 1) consists of the code generation. In this stage the program implementing the specification is generated. This stage relies on the state machine representation of each concurrent entity and the supervisor determined in the previous stage. First, the state machine transitions are classified into several categories. A transition is **controlled** if there are instances in which it should be (temporarily) disabled. There are two types of situations in which a transition t may have to be disabled: when t participates in a synchronization with transitions of other state machines and/or when firing t is not permitted by the supervisor. For instance, in section III, in view of Figure 4, it is clear that t_v, t'_v, and t''_v are controlled. Moreover, a transition that is not controlled is said to be **observed** if the supervisor should be notified when it fires. For instance, referring to the same example, note that t_c is not controlled. However, its firing affects the marking of a supervisor place. Thus, t_c is observed.

In code generation, each state machine is converted to a thread or a process (depending on the specification), in which the instructions consist of the user code associated with places and transitions and of communication code. The communication code allows a coordinator process (corresponding to the supervisor) to coordinate the execution of the concurrent entities. An entity will ask permission to fire a transition t if t is controlled or if t participates in a synchronization. An entity will notify the coordinator that a transition t was fired if t is observed. Now, a place of a state machine may have several output transitions. Then, the choice of the next transition is made as follows. The fireable transitions are first ranked (the ranking is random if the user code does not rank the transitions). If the transition of highest rank is not controlled or synchronized with other transitions, it is fired immediately. Otherwise, the list of possible next transitions is sent to the coordinator. When the coordinator selects the next transition, the entity fires the transition immediately and continues with the user code of the next place.

The coordinator process waits for messages from the concurrent entities. When a firing request is received, the request is placed into a queue. When a firing notification is received, the program updates the marking of the supervisor. After a firing notification or a firing request, the program searches the queue for requests that can be granted. A request may be granted if the transition is supervisor

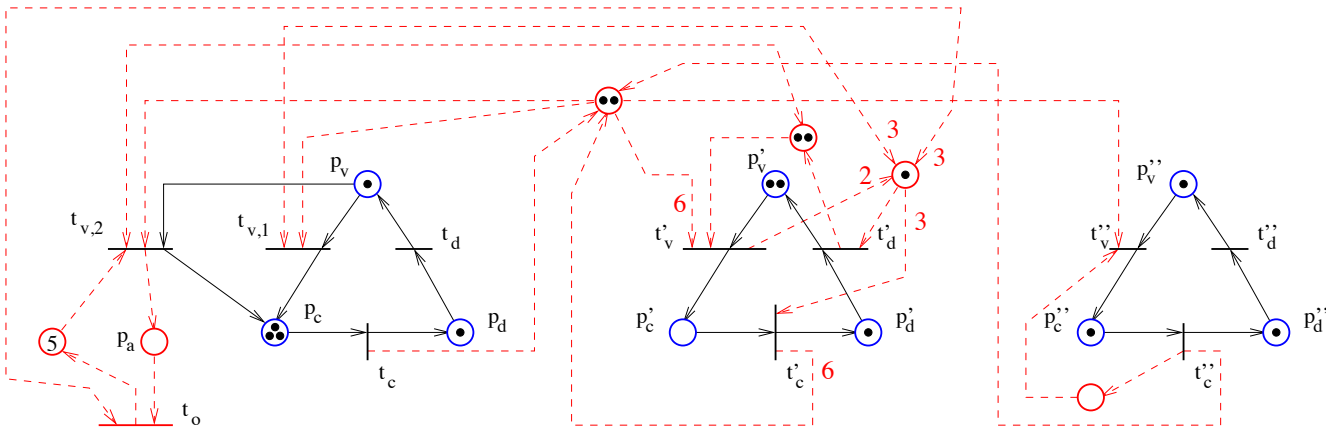

Fig. 4. Supervision in the reader, deleter, and inserter example. A bidirectional arrow between a place p and a transition t represents a pair of arcs (p,t) and (t,p) of equal weight. A number inside of a place indicates the number of tokens.

enabled. Additionally, for transitions participating in synchronizations, a request to fire a transition is granted when there is a synchronization involving that transition such that for all transitions t in the synchronization there are $W(p,t)$ waiting entities that can fire t, where $W(p,t)$ denotes the weight of the input arc of t. Requests in the queue are examined in the order they are received. If a firing request can be granted, the supervisor marking and the queue are updated and notification messages are sent to all involved entities. Due to transition synchronizations, more than one entity may be involved in a transition firing.

When the generated program begins its execution, it starts a number of entities in accordance with the initial marking given in the specification. Entities can be created or terminated also during the execution of the program: when entities fire sink transitions they terminate and when source transitions are fired new entities are created. Unless terminated by the user, the program terminates when all entities terminate.

V. Guarantees and Limitations

There are several kinds of limitations. First, from a programming perspective, the latency of concurrency control could be considerable. This would mean that specifications should be written such as to avoid executing the concurrency control code extremely often. Second, the time required to generate the code could be considerable, due to the computational complexity of supervisory control methods. While specifications (1) on a fully controllable and observable system involve low polynomial complexity, the computational complexity can be increased dramatically by the presence of uncontrollable and/or unobservable transitions. Moreover, the software includes an implementation of the T-liveness enforcement procedure of [14]. While this procedure can be applied to arbitrary PNs, it does not have guaranteed convergence and some of its operations can have exponential complexity (such as the identification of minimal siphons). Thus, the T-liveness procedure is not applied unless the user requests it in the specification. Finally, there are also limitations due to the difference between the supervisory control contexts of conventional PNs and PNs representing software systems. We address this issue in the remaining part of the paper.

A PN representing a program is a high level Petri net (HPN), since the fireable transitions are determined not only by the marking but also by the user code associated with places and transitions. Let $W(p,t)$ denote the weight of an arc (p,t) from a place p to a transition t. Given a transition t, the number of input places of t represents the number of types of concurrent entities that are synchronized by means of t. Thus, each place $p \in {\bullet}t$ represents an execution stage for a different type of entities. Now, a place $p \in {\bullet}t$ may have other output transitions besides t. Then, it is possible to encounter a situation in which all places $p \in {\bullet}t$ have at least $W(p,t)$ tokens and yet t is not fireable. This would happen when some of the tokens of the places $p \in {\bullet}t$ correspond to entities that do not attempt to fire t but some other transitions of $p{\bullet}$. Thus, in the HPN, a transition t is **enabled** if for each place $p \in {\bullet}t$, at least $W(p,t)$ entities can fire t. For a place p with multiple output transitions, if the user code determines the choice of the next transition, the choice is said to be **deterministic**. If the user code does not describe how to select the next transition, the choice is **nondeterministic**.

In our approach, supervisory control methods are applied to the underlying PN of an HPN without accounting for the user code associated with its places and transitions. On the positive side, this simplifies considerably the supervisory control problem. On the negative side, supervisors designed this way may have certain performance limitations when applied to the HPNs.

Proposition 5.1 *A supervisory policy enforcing a set of constraints (1) on the underlying PN will enforce the constraints also when applied to the HPN.*

Proof: Apart from restricting transition firing, user

code has no effect on the operation of the PN. Then, the set of reachable states (μ, v) of the HPN is a subset of the set of reachable set of the underlying PN. Therefore, if the reachable states of the underlying PN satisfy the constraints (1), the reachable states of the HPN will satisfy them also. ■

Proposition 5.2 *Assuming no uncontrollable transitions and no unobservable transitions, a least restrictive supervisory policy enforcing a set of constraints (1) on the underlying PN will be least restrictive also when applied to the HPN.*

Proof: Since the least restrictive supervisory policy disables a transition t only when firing t would break one of the constraints (1), it remains least restrictive when applied to the HPN. ■

In the presence of uncontrollable and/or unobservable transitions, a supervisory policy may be too restrictive unless deterministic choice is modeled in the underlying PN. This could be seen on the following examples. Consider the PN shown in Figure 5(a). Assume that the choice between firing t_2 or t_3 is deterministic. Assume also that t_3 is unobservable and controllable, all other transitions are controllable and observable, and the supervision objective is $\mu_2 \leq 1$. If the supervisor cannot know which of t_2 or t_3 is chosen to fire, the least restrictive supervisory policy is $v_1 \leq v_2 + v_4 + 1$, requiring that once t_1 has been fired, t_1 may not be fired again until either t_2 or t_4 fire. However, if the choice to fire t_2 or t_3 is known, the supervisory policy is too restrictive, for it will not allow t_1 to fire when there is one concurrent entity in the stage p_1 (that is, $\mu_1 = 1$) and the entity waits for permission to fire t_2. Now, considering the same example with t_3 uncontrollable and observable, the least restrictive policy is $v_1 \leq v_2 + v_4 + 1$. However, as before, if the choice to fire t_2 or t_3 is known, the supervisory policy is too restrictive, for it will not allow t_1 to fire when there is one concurrent entity in the stage p_1 (that is, $\mu_1 = 1$) and the entity waits for permission to fire t_2.

A possible approach to the modeling of deterministic choice has appeared in [15]. The approach is described by the following algorithm.

1) Let D be the set of places with deterministic choice.
2) For all arcs (p, t) with $p \in D$ do:
   a) Let p' and t' be a new place and a new transition.
   b) $p'\bullet = \{t\}$, $\bullet p' = \{t'\}$, $W(p', t) = W(p, t)$, and $W(t', p') = 1$.
   c) $p\bullet = (p \bullet \setminus \{t\}) \cup \{t'\}$ and $W(p, t') = 1$.
   d) Note that the block of code associated with the place p contains a request to fire t. This request to fire t is replaced with a request to fire t'.

In the algorithm above note that (p, t') is deterministic and (p', t) is nondeterministic. Note also that the transitions t' are uncontrollable to the supervisory control algorithms. PNs obtained using the algorithm above will be called **normal**. Normal PNs represent deterministic choice explicitly.

As shown above, in the presence of uncontrollable and/or unobservable transitions, if deterministic choice is not modeled explicitly in the underlying PN of an HPN, then a least restrictive supervisory policy of the PN may not be least restrictive in the HPN. We show now that even if the underlying PN is normal, a least restrictive supervisory policy of the PN may be suboptimal in the HPN. The reason for this is that the user code could implement the supervisory control specification in a less restrictive fashion, since it may not be subject to the same constraints as the supervisory control methods applied to the underlying Petri net. Two situations that could make user code solutions more permissive are as follows. First, it may be that some of the transitions declared as uncontrollable and/or unobservable in the specification are not truly uncontrollable and unobservable, and thus the user code could control and observe them. This could happen when certain transitions are declared uncontrollable or unobservable in order to prevent the supervisor from delaying their execution. Second, a situation in which the user code may have more control options than the supervisor is when a place p outputs several uncontrollable transitions. While neither the supervisor nor the user code could disable uncontrollable transitions, the user code might be able to select the uncontrollable transition that will be fired[2]. For instance, consider the constraint $\mu_3 \leq 1$ on the PN of Figure 5(b). Assuming that the place p_1 involves deterministic choice, the PN is not normal, since choice is not represented explicitly. Assuming also all transitions observable, t_2 and t_3 uncontrollable, and t_1, t_4, and t_5 controllable, the least restrictive supervisory policy would be to disable t_1 when $\mu_3 \geq 1$. However, user code could ensure that the concurrent entities synchronize themselves such that an entity will select t_2 (and not t_3) when $\mu_3 \geq 1$. Thus, in the context of the HPN, the user code policy would be less restrictive, since it would allow the sequence $t_1 t_2 t_3$ when $\mu_3 \geq 1$. Note that this permissiveness issue is not eliminated by applying the normalization algorithm to the PN, since the transitions modeling deterministic choice are uncontrollable.

Proposition 5.3 *If the HPN has uncontrollable and/or unobservable transitions, a least restrictive supervisory policy enforcing a set of constraints (1) on the underlying PN may not be least restrictive when applied to the HPN.*

In the following we will distinguish between deadlock prevention and liveness enforcement as follows. A supervisor preventing deadlock ensures that a PN does not reach a state from which no transition is enabled. A supervisor enforcing T-liveness ensures that for every transition $t \in T$ and for every reachable state there is an enabled firing sequence that includes t.

Note that if the underlying PN of an HPN is not normal, a supervision policy preventing deadlock or enforcing T-liveness in the PN may not prevent deadlock in the HPN. Indeed, consider the PN of Figure 6. Assume that all

[2]Note that [16] describes a class of applications in which transitions may not be disabled due to uncontrollability and yet it is possible to select which uncontrollable transition will be fired.

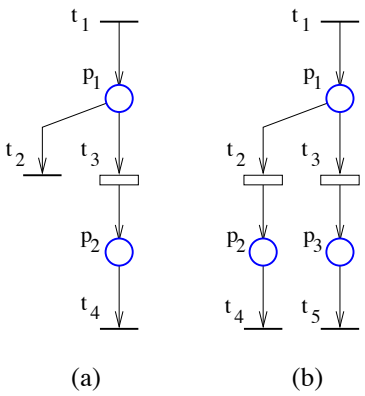

Fig. 5. PNs illustrating permissiveness issues.

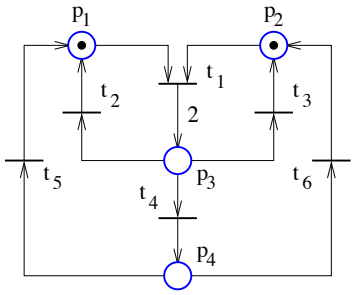

Fig. 6. PN illustrating deadlock issues.

transitions are controllable and observable and that p_4 is the only place involving deterministic choice. Note that the PN is not normal. A supervisory policy for deadlock prevention or liveness enforcement would ensure that $\mu_1 \leq 1$ and $\mu_2 \leq 1$. However, such a policy does not prevent deadlock in the HPN, since it will allow reaching a state in which p_2 and p_4 have each one token and the user code of place p_4 selects the transition t_6. This would be a deadlock state, since the supervisor will continuously disable t_6, which is the only transition that the HPN can fire.

Proposition 5.4 *If the underlying PN of an HPN is not normal, a supervisory policy preventing deadlock or enforcing T-liveness in the underlying PN may not prevent deadlock in the HPN.*

Provided that the user code is correct, a deadlock prevention policy for a normal PN will prevent deadlock also in the HPN.

Proposition 5.5 *Consider an HPN in which the underlying PN is normal. Assuming that for all HPN places the execution of the user code takes a finite amount of time, a supervisory policy preventing deadlock in the underlying PN will prevent deadlock also in the HPN.*

Proof: In the HPN, deadlock implies that no user code is executed and all entities wait for permission to fire certain transitions. Since the underlying PN is normal, for any place p, the set of transitions enabled by p is the same in the HPN and its underlying PN. Thus, a deadlock in the HPN corresponds to a deadlock in its underlying PN. It follows that any supervisory policy preventing deadlock in the PN will prevent deadlock also in the HPN. ■

Under the assumptions of Proposition 5.5, a T-liveness enforcing supervisor of the underlying PN will prevent deadlock in the HPN. However, it would be useful to guarantee not only the absence of total deadlock but also that from any reachable state, any transition in the set T can eventually be fired. As indicated in [15], additional assumptions are needed in order to guarantee that all transitions of interest can eventually be fired. Extensions of the results of [15] could be obtained in future work.

The 2011 version of the software does not implement the transformation to normal PNs. Thus, Proposition 5.4 describe a limitation of the software as well as a direction of future work.

VI. Conclusion

The development of concurrent programs can be simplified by generating automatically the concurrency control code. This paper has presented a supervisory control approach to the synthesis of the concurrency control code. The features and algorithms of a software implementation of this approach were outlined. A formal characterization of the supervisory control framework was also included as well as results describing the performance of conventional methods in the context of software systems.

References

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

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

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

[4] A. Auer, J. Dingel, and K. Rudie, "Concurrency control generation for dynamic threads using discrete-event systems," in *Proceedings of the 47th Annual Allerton Conference on Communication, Control, and Computing*, 2009, pp. 927–934.

[5] J. Dingel, K. Rudie, and C. Dragert, "Bridging the gap: Discrete-event systems for software engineering," in *Proceedings of the Canadian Conference on Computer Science and Software Engineering*. ACM, 2009, pp. 66–71.

[6] T. Kelly, Y. Wang, S. Lafortune, and S. Mahlke, "Eliminating concurrency bugs with control engineering," *Computer*, vol. 42, no. 12, pp. 52–60, 2009.

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

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

[9] K. Knobe, "Ease of use with concurrent collections (cnc)," in *Proceedings of the First USENIX conference on Hot topics in parallelism*. USENIX Association, 2009.

[10] Z. Budimlic, A. Chandramowlishwaran, K. Knobe, G. Lowney, V. Sarkar, and L. Treggiari, "Multi-core Implementations of the Concurrent Collections Programming Model," in *Workshop on Compilers for Parallel Computing*, Jan. 2009.

[11] R. Lublinerman, S. Chaudhuri, and P. Cerny, "Parallel programming with object assemblies," in *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, ser. OOPSLA '09. ACM, 2009, pp. 61–80.

[12] M. Méndez-Lojo, D. Nguyen, D. Prountzos, X. Sui, M. A. Hassaan, M. Kulkarni, M. Burtscher, and K. Pingali, "Structure-driven optimizations for amorphous data-parallel programs," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2010, pp. 3–14.

[13] M. V. Iordache and P. J. Antsaklis, "Concurrent program synthesis based on supervisory control," in *Proceedings of the 2010 American Control Conference*, 2010, pp. 3378–3383.

[14] ——, "Supervision based on place invariants: A survey," *Discrete Event Dynamic Systems*, vol. 16, pp. 451–492, 2006.

[15] ——, "Limitations of liveness in concurrent software systems," in *Proceedings of the 49th International Conference on Decision and Control*, 2010.

[16] ——, "Des abstractions for the supervisory control of hybrid systems," *Transactions of the Institute of Measurement and Control*, vol. 32, no. 5, pp. 468–486, 2010.