49th IEEE Conference on Decision and Control
December 15-17, 2010
Hilton Atlanta Hotel, Atlanta, GA, USA

Limitations of Liveness in Concurrent Software Systems

Marian V. Iordache and Panos J. Antsaklis

Abstract— **A desirable property of software is that from any reachable state any transition of interest will eventually take place. In this paper, software satisfying this property will be said to be responsive. Responsiveness can be studied on untimed DES models of the software. The paper shows that DES liveness is not sufficient to guarantee that the software will be responsive. Two causes of this problem are identified, namely transition determinism and operation timing. Transition determinism refers to the fact that not any DES enabled transition may be fired, but only the specific transition selected by the software. Operation timing refers to the times at which software execution stages take place. Conditions required to fire a transition may become unlikely or impossible due to operation timing. To address these issues, explicit modeling of deterministic choice is proposed and a special DES structure is introduced. Then, a sufficient condition is formulated under which DES liveness implies software responsiveness. This sufficient condition is then applied to the supervisory control problem in order to identify a class of liveness enforcing supervisors that ensure responsiveness. While Petri nets are used here to represent the DES models, the results are also relevant in the automata framework.**

I. Introduction

Difficulties encountered in the context of concurrency have motivated the use of DES models for analysis, verification, and synthesis of software systems. One of the basic properties that software systems should satisfy is that from any reachable state any desirable transition should eventually take place. Systems satisfying this property will be said to be *responsive*. At a first look, responsiveness and liveness may appear to be identical. However, this is not the case. As we show in this paper, due to certain aspects of software systems, unresponsive software systems may have live DES models. Here, by DES models we mean classic untimed Petri net and automata models. Since liveness has received considerable attention in the technical literature, it is of interest to determine conditions under which liveness guarantees responsiveness. The paper addresses this topic as follows. First, Petri net (PN) models are defined that are appropriate for the study of responsiveness. Second, a sufficient condition is proven under which liveness guarantees responsiveness. Note that responsiveness implies that there are no reachable states in which *starvation* takes place, since starvation occurs when a process can no longer receive resources necessary for its execution. Note also that in our setting the absence of starvation does not imply responsiveness. Indeed, we consider concurrent software systems in which processes may not only access common resources but also synchronize their execution on certain operations.

This work is part of a project for the development of a concurrency tool suite (ACTS) for the synthesis of concurrent programs [1], [7], [8]. The goal is to design software that based on a high level specification can generate concurrent programs. The general approach of the project can be outlined as follows. Based on a specification written in a high level specification language, a PN model and a supervisory control (SC) specification is extracted. Then, SC methods are applied. Finally, low level code is generated that combines low level user code and concurrency control code implementing the SC policy. Thus, the project involves developing a high level specification language, the software tools for compilation and SC, and theoretical work on the required methods. Of special interest are theoretical results that can guarantee the performance of the generated code. While numerous SC results have been developed and rigorously proven, there are still certain aspects that need to be considered when applying them to software systems. Thus, the work presented in this paper is motivated by the need to offer guarantees that liveness enforcement methods will ensure that the generated programming code will be responsive.

Related work includes the following papers. Deadlock analysis of programs based on PN models has been used in papers such as [2], [11], [12], [13]. The application of SC to software systems has been proposed in references such as [4], [8], [9], [10]. A literature survey on PN applications to software systems has appeared in [6]. Related work includes also [3], dealing with conditions that ensure that the implementation of automata-based supervisors is nonblocking.

The paper is organized as follows. An introduction to the notation, concepts, and setting of the paper is given in section II. Then, section III describes limitations of liveness in the context of software systems. Finally, section IV presents sufficient conditions under which liveness implies responsiveness and liveness enforcement implies enforcement of responsiveness. The results of section IV represent the main contribution of the paper. Much of the material of section III could also be new, since we are not aware of prior work related to the observation that determinism and timing limit liveness.

II. Preliminaries

In this paper a PN denotes a place/transition (P/T) net. Given a PN and a place or transition x, $\bullet x$ and $x\bullet$ will

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).

denote the preset and postset of x, respectively. Given a set X, let $|X|$ denote the number of elements of X. A **state machine** is a PN in which $|\bullet t| \leq 1$ and $|t \bullet| \leq 1$ for all transitions t. Note that we do not assume that a state machine will have only one token. Rather, in this paper, state machines may have an arbitrary number of tokens.

A. PN Representation of Software

We consider programs consisting of entities that run concurrently. Each such entity will be called process, though it may be implemented as a thread. Each process is modeled by a state machine. The places of the state machine represent stages in the operation of the process. Each place is associated with the segment of code of the corresponding process stage. Transitions between places correspond to transitions of the process from one stage to another. Transitions t with $\bullet t = \emptyset$ model process creation and transitions t with $t\bullet = \emptyset$ model process termination. The process itself is represented by a token of the state machine. The place in which the token is present indicates the current stage of the process. Identical processes may be modeled by distinct tokens of the same state machine.

The DES model of the program is obtained by composing the models of its processes. The method used to compose the process models is the parallel composition of PNs [5]. Now, even though processes are modeled by state machines, the result of the composition is usually not a state machine. This is due to the fact that processes are allowed to synchronize their operation. In particular, when processes share resources, a process accessing a shared resource has to synchronize its operation with the process managing the access to the shared resources. Since any PN can be seen as the composition of a number of state machines, it follows that in general there is nothing that can be assumed about the structure of the PN representing the program.

The PN representing a program together with the blocks of code associated with each place forms a high level PN (HPN). Note that a transition t of the HPN may fire when enabled by the underlying PN and by the code associated with the places $p \in \bullet t$. Enabled transitions are fired immediately.

In an HPN, the tokens involved in a transition firing have special significance, since tokens represent processes. Multiple tokens in the same place are possible. They represent multiple identical processes in the same stage of execution. In principle, an HPN might reach a state in which the number of tokens of a place exceeds the number of tokens necessary to fire a transition. When there is a choice concerning which processes should participate in a transition firing, processes are considered in the order in which they made the request to fire the transition.

B. Determinism

In the state machine representing a process, each place corresponds to a process stage. In each stage a process executes a block of code. At the end, the process executes code that selects the transition to the next place. The choice of the transition is **deterministic** if the choice is completely determined by the internal code of the process. Once a transition has been chosen, the process waits until the transition may be fired. Thus, the choice cannot be revoked. Now, the choice of a transition is **nondeterministic** if the process lets external factors (such as a supervisor process) select which transition should be fired.

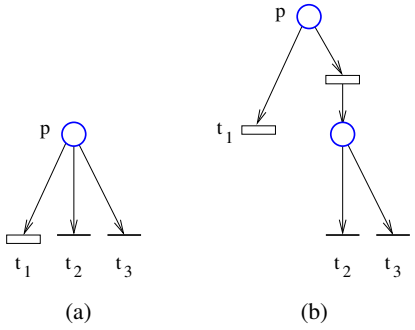

Fig. 1. A place involving a combination of deterministic and nondeterministic choice can be decomposed into two or more places, one place involving deterministic choice and the other places nondeterministic choice.

Without loss of generality, we will assume that at any place the choice is either completely deterministic or completely nondeterministic. For instance, for a place p having three output transitions t_1, t_2, and t_3, a possibility might be that p chooses deterministically between firing t_1 or allowing a supervisor to select one of t_2 and t_3. Without loss of generality, this possibility will be ignored. Indeed, it is possible to separate deterministic choice from nondeterministic choice by means of additional places, as illustrated by the PN transformation of Figure 1.

Places modeling resources provide an example of nondeterministic choice. By itself, a resource place does not determine which process should use the resource. Rather, a supervisor process determines the process that may use the resource according to some rule, such as according to the order in which requests are made.

Given a place p and a transition $t \in p\bullet$, the transition arc (p, t) is said to be **deterministic** if the choice of transitions at the place p is deterministic. Otherwise, the arc (p, t) is said to be **nondeterministic**.

III. Liveness

A process of a software application may have to wait, such as for a shared resource. A desirable property of the software would be that the process will eventually be able to resume its execution. An indirect approach to address this problem is to ensure liveness.

A transition t of a PN is **live** if for any reachable marking there is an enabled firing sequence that includes t. A PN is **live** if all its transitions are live. An HPN is **live** if its underlying PN is live. As shown in this section, liveness does not address all deadlock possibilities of an HPN. Therefore, we define a stronger requirement called *responsiveness*.

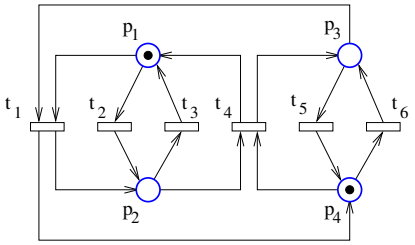

Fig. 2. PN representing the composition of two processes that synchronize their operation on the transitions t_1 and t_4.

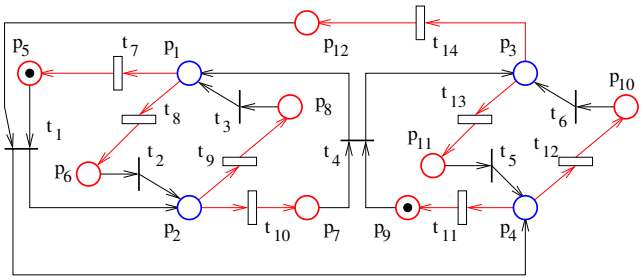

Fig. 3. Enhancement of the PN model of Figure 2 that models explicitly deterministic choice.

An HPN is **responsive** if from any reachable state of the software system any transition can be eventually fired. Note that the state of the software system consists of the value of all program variables, including the information about the stage of execution. Responsiveness ensures that a process waiting for permission to fire a transition will eventually be allowed to fire it. This relies on the fact that when there is a choice concerning which processes should be involved in a transition firing, the "first come first served" policy is applied. Just like liveness, responsiveness is a property of systems that have a repetitive operation. Of special interest here are software systems in which all processes correspond to structurally live state machines.

A. Determinism and Liveness

Deterministic choices are a source of deadlock. However, deadlocks due to deterministic choices may not be apparent from the PN model of the software, unless adequate changes are made. To illustrate this point, consider the PN model of Figure 2. The PN represents two processes that synchronize their operation on the transitions t_1 and t_4. Clearly, the PN is live. However, if choices are taken into account, the model may be in deadlock. Indeed, if the block of code at the place p_1 chooses the transition t_1 and the block of code at p_4 chooses the transition t_4, then the system is deadlocked. A PN model could reveal potential deadlocks caused by deterministic choice if choice is explicitly modeled in the PN (Figure 3).

The PN change done in Figure 3 can be described by the following algorithm. Note that given a transition arc x, $W(x)$ denotes the weight of x.

1) Let A be the initial set of deterministic arcs.
2) For all arcs $(p,t) \in A$ 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. PNs obtained using the algorithm above will be called **normal**. Normal PNs represent deterministic choice explicitly.

Note that deterministic choice is related to uncontrollability [14]. Indeed, a supervisor cannot control the outcome of deterministic choice. For instance, consider the system of Figure 3. Assume that p_1 and p_9 have each one token and that there are no other tokens in the system. A possible approach to prevent deadlock is to disable the transition t_7 in the hope that t_8 will be eventually fired. This approach will not work in our setting, since the decision to fire t_8 or t_7 is determined independently by the code associated with the place p_1. Thus, if the code determines that t_7 should be fired, deadlock is reached, since the supervisor disables t_7. The transitions t_8 and t_7 could be seen as uncontrollable, in the sense that the supervisor cannot select which of the two should fire. All that a supervisor could do is to block the firing of either transition.

While a supervisor cannot determine choice, it can postpone choice and the effects of choice. For instance, assume that the PN of Figure 3 has tokens in the places p_7, p_8, p_9, and p_{10}. Choice at p_1 can be postponed by disabling t_3 and t_4. Moreover, the effect of the choice at p_4 that caused a token to enter p_{10} can be postponed by disabling t_6. A supervisor will be said to be **admissible** if it never disables deterministic transitions.

B. Timing and Liveness

Even in the absence of deterministic choice, the timing of the operations performed by processes can create deadlocks. In terms of PN models, this problem can be noticed when there are transitions with inputs from multiple resource or monitor places. An implication of this observation is that a supervisor could create deadlocks when it is equally permissive to a monitor based supervisor in which two or more monitors have common transitions in their postset.

To illustrate how timing can be an issue, consider three processes P_a, P_b, and P_c that share two resources R_b and R_c.

1) The process P_a executes an infinite loop consisting of the execution stages a_1 and a_2.
   a) In each iteration, the process executes first a_1 and then a_2.
   b) At the beginning of a_1 the process waits until the resources R_b and R_c are available. When both resources are available, the process acquires them and continues by executing the stage a_1.

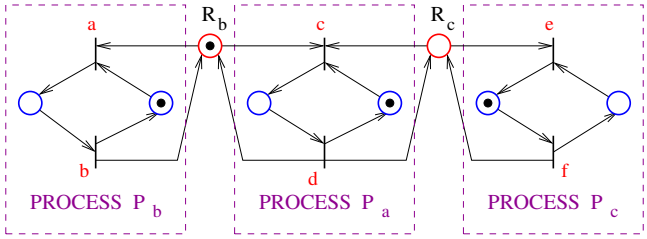

Fig. 4. PN model of three processes with two shared resources.

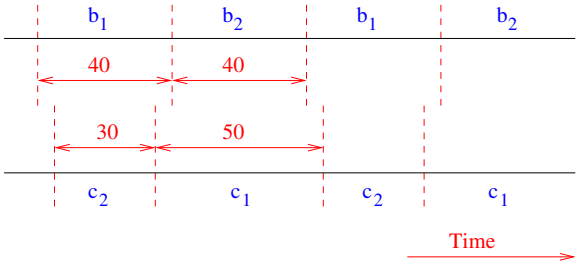

Fig. 5. Example illustrating how timing can prevent shared resources from being available at the same time.

c) At the end of a_1 the process releases the resources.

2) The process P_b has the same description as P_a with the following exceptions: (a) it uses only R_b instead of both R_b and R_c; (b) the stage names are b_1 and b_2 instead of a_1 and a_2.
3) The process P_c has the same description as P_a with the following exceptions: (a) it uses only R_c instead of both R_b and R_c; (b) the stage names are c_1 and c_2 instead of a_1 and a_2.

A PN model of this system is shown in Figure 4. Clearly, the PN is live. However, the process P_a will have to wait forever if the two resources are never available at the same time. For instance, if the execution time of the stages b_1, b_2, c_1, and c_2 is 40, 40, 50, and 30 time units, respectively, and if stage c_1 follows b_1 after 35 time units (Figure 5), then the resources are never available at the same time. In general, the execution time of any stage might not be a constant. Nonetheless, it might still be difficult to guarantee that the two resources will eventually be available at the same time. Even if the two resources are guaranteed to become available at the same time, access to resources could still be unfair, since process P_a might have a much lower likelihood to access them than the other two processes.

The equivalent automaton model of the PN of Figure 4 is shown in Figure 6. Note that the state s_1 is the state in which both resources are available. In our example the two resources are never available at the same time. The states s_1 and s_3 are unreachable because an event b is always followed by an event a and an event f is always followed by an event e.

Figure 7 can also be used to illustrate that timing can lead to deadlock. The places $p_1, \ldots, p_6$ model execution stages

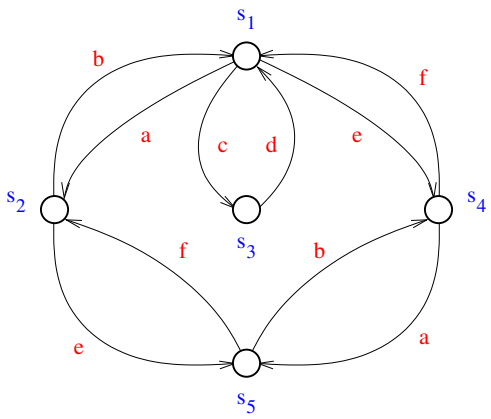

Fig. 6. Equivalent automaton model of the processes of Figure 4.

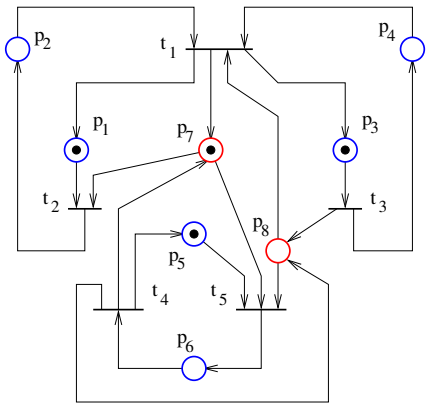

Fig. 7. In this example, if $\tau_1 < \tau_3$ then t_5 cannot fire.

of three processes. The places p_7 and p_8 model resource places or monitor places. Let τ_i be the time necessary to complete the stage p_i. Assuming that a transition fires as soon as possible, t_5 will never fire unless $\tau_1 \geq \tau_3$. Since the PN is live, this example shows that liveness does not guarantee that every transition will eventually fire.

A common feature of the examples presented in this section is that in each case there is a transition having among its input places more than one resource place. Resource places differ from other places in that all their output arcs are nondeterministic. Transitions in which multiple input arcs are nondeterministic are essential for this type of deadlock, as will be shown in the following section.

IV. Main Result

This section provides a sufficient condition under which liveness guarantees responsiveness. The result is then applied to supervisors in order to obtain a sufficient condition under which a liveness enforcing supervisor guarantees responsiveness. The results rely on several assumptions. The first two assumptions guarantee that the software system will try to fire all its transitions. The third assumption is that the nondeterministic part of the PN has a special structure.

Assumption 1 For every place p that outputs deterministic transitions, for every transition $t \in p\bullet$, the number of consecutive choices at p that do not select t is finite. Formally, let $\sigma = t_1 t_2 \ldots$ denote a sequence of firing decisions made at the place p (where $t_1, t_2, \ldots \in p\bullet$). Then, the assumption is that for all possible infinite sequences σ, for all $t \in p\bullet$, for all $n \geq 1$, there is $k > n$ such that $t_k = t$.
Assumption 2 The code associated with any place p is executed in finite time.
Assumption 3 Places with nondeterministic choice have "asymmetric choice". That is, for any two nondeterministic arcs (p_1, t) and (p_2, t) connected to the same transition t, either $p_1\bullet \subseteq p_2\bullet$ or $p_1\bullet \supseteq p_2\bullet$.

A PN is said to be **PT-ordinary** if all arcs from a place to a transition have unity weight. Furthermore, given a state s of the software system, let $\mathcal{R}(s)$ denote the set of states reachable from s, including s. Recall that the **state** of the software system consists of the value of all program variables, including the information about the stage of execution. Thus, a state s determines the marking μ of the underlying PN model.

Lemma 4.1 *Assume a PT-ordinary normal PN. Under the assumptions 1–3, if there is a reachable state s from which a transition t can never be fired, then there is a place $p \in \bullet t$ and $s' \in \mathcal{R}(s)$ such that $\mu(p) = 0$ $\forall s'' \in \mathcal{R}(s')$.*

Proof: If $|\bullet t| = 1$, let p be the input place of t. If (p, t) is nondeterministic, it must be that $\mu(p) = 0$ $\forall s' \in \mathcal{R}(s)$, since t cannot be fired. If (p, t) is deterministic, in view of Assumption 1, there is a reachable state $s' \in \mathcal{R}(s)$ at which $\mu(p) = 0$ and p can no longer receive tokens.

If $|\bullet t| \geq 2$, then all input arcs (p, t) are nondeterministic, since the PN is normal. Let $n = |\bullet t|$. If $n = 1$ the conclusion follows, since t cannot be fired. If $n > 1$, in view of assumption 3, there are n places $p_1, p_2, \ldots, p_n$ such that $p_1\bullet \subseteq p_2\bullet \subseteq \ldots \subseteq p_n\bullet$ and (p_1, t), (p_2, t), $\ldots$, (p_n, t) are the input arcs of t. Let k be the least index for which $\exists s'' \in \mathcal{R}(s')$ at which $\mu(p_k) = 0$. Then, $\forall s'' \in \mathcal{R}(s')$, $\mu(p_i) \geq 1$ for $i = 1 \ldots k-1$. Consider the following procedure:

1) Let $j = k$ and $s_{j-1} = s'$.
2) While $\exists s_j \in \mathcal{R}(s_{j-1}) : \mu(p_j) \geq 1$
   a) $j = j + 1$

At step 2) note that after the state s_j is reached, p_j can no longer lose tokens. Indeed, p_j can only lose tokens by firing a transition $t' \in p_j\bullet$ and t' cannot be enabled without t being enabled, since $\mu(p_i) \geq 1$ for $i = 1 \ldots j$ and $t' \in p_i\bullet$ for $i = j+1 \ldots n$. Now, since t will not get enabled, there must be $u \leq n$ such that $\forall s_u \in \mathcal{R}(s_{u-1}) : \mu(p_u) = 0$. Then, the conclusion of the lemma is verified for $s' = s_{u-1}$ and $p = p_u$. ■

A PN $\mathcal{N}'$ is said to be the **normalized** version of a PN $\mathcal{N}$ if it is obtained from $\mathcal{N}$ by means of the normalization algorithm of section III-A.

Theorem 4.1 *Consider a PT-ordinary PN. Under the assumptions 1–3, if there is a reachable state from which a transition can never be fired, then the normalized PN is not live.*

Proof: Note that assumptions 1–3 are not affected by normalization. Moreover, reaching a state in which a transition can never be fired implies that the normalized PN has a reachable state in which a transition can never be fired. The proof considers the normalized PN and shows that if there is a reachable state from which a transition t can never be fired then there is a reachable state for which a siphon is empty. The siphon is constructed according to the following algorithm.

1) Let s_0 be a state from which a transition t can no longer be fired.
2) By Lemma 4.1, there is a state s_1 reachable from s_0 and there is a place $p \in \bullet t$ such that p has no tokens in the state s_1 and in all states reachable from s_1. Let $k = 1$, $A_0 = \emptyset$, and $A_1 = \{p\}$.
3) While $A_k \setminus A_{k-1} \neq \emptyset$ do
   a) In view of Lemma 4.1, let s_{k+1} be a state reachable from s_k such that $\forall p \in A_k \setminus A_{k-1}$, $\forall t \in \bullet p$, $\exists p' \in \bullet t$ such that $\mu(p') = 0$ in s_{k+1} and all states reachable from s_{k+1}. Let A be the set of all such places p'.
   b) Let $A_{k+1} = A_k \cup A$.
   c) $k = k + 1$.

   End while.

Let S be the last set A_k. By construction, S is a siphon and S is empty at the state s_k. Therefore, the PN is not live. ■

Based on Theorem 4.1 it is possible to obtain a sufficient condition under which liveness enforcing supervisors ensure responsiveness. Now, when a supervisor is represented by a PN, the total system consisting of the plant and supervisor is represented by the closed-loop PN, where the closed-loop PN is the parallel composition of the plant and supervisor PNs. Note that the closed-loop PN is the union of the plant and supervisor PNs with the following modifications.

1) Let I and O be the input and output matrices of the closed-loop PN. Let I_p, O_p, I_s, and O_s be the corresponding matrices of the plant and supervisor.
2) Let $T_p = \emptyset$ and $T_s = \emptyset$.
3) For every plant transition t_x and every supervisor transition t^y that have the same label:
   a) Create a new transition t_x^y such that t_x^y has copies of the arcs of t_x and t^y (that is, $I(p, t_x^y) = I_p(p, t_x)$ and $O(p, t_x^y) = O_p(p, t_x)$ for all plant places p and $I(p, t_x^y) = I_s(p, t^y)$ and $O(p, t_x^y) = O_s(p, t^y)$ for all supervisor places p).
   b) $T_p = T_p \cup \{t_x\}$ and $T_s = T_s \cup \{t^y\}$.
4) Remove all transitions in T_p and T_s.

By definition, an admissible supervisor will not attempt to disable deterministic transitions. This does not mean that the places of an admissible supervisor may not be connected to deterministic transitions. Then, note that the step 3 of the algorithm can create multiple copies of a single deterministic arc of the plant. Multiple copies of a deterministic

arc are neither deterministic nor nondeterministic. Thus, the following closed-loop normalization algorithm can be used in order to ensure that all transitions are either deterministic or nondeterministic. The algorithm also ensures that no supervisor places are connected to deterministic transitions.

1) For all deterministic arcs (p, t_x) of the plant that are copied to one or more closed-loop transition t_x^y, perform the following changes to the closed-loop PN:
   a) Let T_x be the set of transitions t_x^y obtained by composing t_x with supervisor transitions t^y.
   b) Add a new transition t and a new place p_x. The function of t is as follows. In the plant, the code associated with the place p contains a request to fire t_x. This request to fire t_x is replaced in the closed-loop with a request to fire t.
   c) Let $p\bullet = (p\bullet \setminus T_x) \cup \{t\}$, $t\bullet = \{p_x\}$, $p_x\bullet = T_x$.

In the previous algorithm note that the arc (p, t) is deterministic and the arcs (p_x, t_x^y) are nondeterministic for all $t_x^y \in T_x$. In the following, the **normalized closed-loop PN** will denote the PN obtained by applying the algorithm above to the result of the parallel composition of the supervisor PN and the normalized plant PN.

Corollary 4.1 *A supervisor enforces responsiveness if all of the following conditions are met.*

1) *The plant satisfies the assumptions 1 and 2.*
2) *The supervisor can be represented by a PN.*
3) *The normalized closed-loop PN is live.*
4) *The normalized closed-loop PN is PT-ordinary and satisfies the assumption 3.*

Proof: If the plant satisfies the assumptions 1 and 2, then the closed-loop will satisfy them also. The normalized closed-loop has all the properties of a PN normalized using the algorithm of section III-A. Thus, Theorem 4.1 can be applied to the normalized closed-loop to guarantee responsiveness. ■

Note that the corollary does not assume admissible supervisors. Admissibility as defined in section III-A is not a feasibility issue but a design issue. A design approach that does not consider admissibility ignores a source of deadlocks and thus is likely to fail.

The first three conditions of the corollary are critical for responsiveness. However, the fourth condition is not necessary. A special case in which assumption 3 is satisfied is when all nondeterministic transitions of the plant have at most one input place, the supervisor is a state machine, and no two transitions of the supervisor have the same label. In particular, supervisors obtained using liveness enforcement methods that rely on control places [5] can be in this category. Such supervisors have distinct labels and are state machines when no transition has more than one input control place. Now, if arbitrarily labeled state machines are considered, assumption 3 may not be satisfied even when both the supervisor and the plant are state machines.

A possible supervisor design approach would be to generate a liveness enforcing supervisor such that assumption 3 is satisfied. This approach would result in supervisors in which typically a transition has at most one input place. A transition having more than one input supervisor place might be anyways undesirable, since it might be unlikely to have all input places marked at the same time.

Another possible supervisor design approach would be to enforce first fairness constraints and then design a liveness enforcing supervisor. The fairness constraints would exclude the type of deadlock illustrated in section III-B. This approach would not rely on assumption 3. For an example of fairness constraints, consider Figure 7. An appropriate fairness constraint would be $v_2 \leq v_5 + 1$, requiring that the number of firings of t_2 does not exceed by more than one the number of firings of t_5. This constraint would in fact ensure responsiveness.

V. Conclusions

In general, liveness is not sufficient for software-system responsiveness. However, it is possible to obtain sufficient conditions under which liveness can guarantee that the system will be responsive. In the context of supervisory control, a possible approach to supervision would be to design liveness enforcing supervisors that satisfy such sufficient conditions. In particular, an approach of special interest for future work is to use fairness constraints to guarantee that liveness enforcement will ensure responsiveness.

References

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

[2] 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.

[3] P. Dietrich, R. Malik, W.M. Wonham, and B.A. Brandin. Implementation considerations in supervisory control. In *Synthesis and control of discrete event systems*, pages 185–201, Kluwer, 2002.

[4] J. Dingel, K. Rudie, and C. Dragert. Bridging the gap: Discrete-event systems for software engineering. In *Proc. Canadian Conf. on Comp. Sci. and Soft. Eng.*, pages 66–71. ACM, 2009.

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

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

[7] 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.

[8] M. V. Iordache and P. J. Antsaklis. Concurrent program synthesis based on supervisory control. In *Proc. 2010 Amer. Control Conf.*, pages 3378–3383, 2010.

[9] T. Kelly, Y. Wang, S. Lafortune, and S. Mahlke. Eliminating concurrency bugs with control engineering. *Computer*, 42(12):52–60, 2009.

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

[11] 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.

[12] 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.

[13] S. Shatz, S. Tu, T. Murata, and S. Duri. An application of Petri net reduction for Ada tasking deadlock analysis. *IEEE Transactions on Parallel and Distributed Systems*, 7(12):1307–1322, 1996.

[14] Y. Wang. *Software Failure Avoidance Using Discrete Control Theory*. PhD thesis, University of Michigan, 2009.