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 involving Petri nets (PNs). Second, a supervisor is designed in accordance with the specification requirements. The role of the supervisor is to restrict the operation of the PN 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], and [6]. Moreover, certain methods developed 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 to concurrent programming has been proposed also in [4] and [5]. Compared to [4] and [5], one different aspect in our work is that we consider arbitrary synchronizations, including synchronizations that cannot be implemented (exclusively) by locks\(^1\). Furthermore, our methods rely on PNs instead of automata. Related is also the work on the GADARA software [6], [8]. There, PN 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 the software engineering literature there is 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] and [12]. While there are various classes of problems for which our 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 is 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 (SM) in which the states and transitions are associated with user code. Each state of the SM represents a stage of execution. Thus, the code associated with a state of the SM defines the operations of a stage of execution. Moreover, the code associated with transitions determines which transition should fire when

\(^1\)A lock is a synchronization mechanism in concurrent programming.
there is a choice. The SMs may have source and sink transitions. A source (sink) transition represents process/thread creation (termination).

Note that we view SMs as a special class of PNs in which each transition has at most one input place and at most one output place. Each SM represents the structure of a concurrent entity (that is, 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 SM. The number of tokens of a SM represents the number of entities associated with the SM.

Arbitrary synchronizations between SM 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 SM.

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

$$\sum l_i \mu_i + \sum h_i q_i + \sum c_i v_i \leq d$$

where the coefficients $d$, $l_i$, $h_i$, and $c_i$ represent integer constants, $\mu_i$ 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 $\mu_i$ and $v_i$ are the values before $t_j$ is fired).

The specification may describe also SMs 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 PN of Figure 3 as a supervisory component in order to describe constraints that cannot be expressed by inequalities (1) alone.

The parallel composition of the SMs representing the concurrent entities results in a labeled PN. A current limitation of the software is that the constraints (1) have to be admissible with respect to the labeling of the PN. A special case in which admissibility is not a concern is when all transitions of the PN have distinct labels. Note also that the supervisory components do not limit the constraints (1), since they are not part of the plant PN.

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

III. Example

Consider a problem involving three thread types: readers, deleters, and inserters. Assume five readers, three deleters, and three inserters described by the SM structures of Figure 2. Assume also that the threads work on a shared region of memory when they are in the execution stages $p_c$, $p_v'$, and $p''$. The constraint that only one inserter thread may be in the critical section $p_i'$ could be written as $\mu_i' \leq 1$ where $\mu_i'$ denotes the marking of the place $p_i'$. Moreover, the expression (2) could be used to describe that only one deleter may be in the critical section $p_d'$ and that no reader or inserter may be in $p_c$ and $p_i'$ at the same time.

$$\mu_i' \leq 1 \land (\mu_c = 0 \lor \mu_c + \mu_p'' = 0) \quad (2)$$

Note that $\land$ denotes conjunction (logic “and”) and $\lor$ denotes disjunction (logic “or”). Thus, (2) requires $\mu_i' \leq 1$ and either of $\mu_c = 0$ or $\mu_c + \mu_p'' = 0$. Since the maximum number of readers and inserters in the critical section is six, (2) can be written more compactly as

$$6 \mu_c + \mu_p + \mu_p'' \leq 6 \quad (3)$$

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 $p_{ad}$ and three additional transitions $t_1$, $t_2$, and $t_o$ (Figure 3) as follows.

- Synchronize $t_1$ and $t_2$ with $t_{ad}$. This means that when $t_v$ fires, either $t_1$ or $t_2$ must be fired.
- Require $\mu_1' \leq \mu_{1o}'$, expressing that $t_1$ may not fire when $\mu_{1o}' = 0$ (i.e. when no deleter is waiting).
- Require $3 \mu_2 + 3 \mu_p \leq 3 - \mu_{1o}' + 3 \mu_{1o}'$, expressing that $t_2$ and $t_o$ may not fire when a deleter is waiting and no deleter is in the critical section.
- Require $\mu_{1o} \leq 5$, 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 $p_{ad}$ as soon as it is allowed to fire. The supervisor enforcing the constraints is shown in Figure 4.
There are two situations in which a transition \( t \) may have to be disabled: when \( t \) participates in a synchronization involving transitions of other SMs and when the supervisor controls \( t \). For instance, in the context of section III, \( t_v, t'_v, \) and \( t''_v \) are controlled. Now, a transition that is not controlled is said to be observed if the supervisor should be notified when it fires. For instance, in the context of section III, \( t_c \) is not controlled. However, its firing affects the marking of a supervisor place (Figure 4). Thus, \( t_c \) is observed.

When the program is generated, the supervisor is converted to a coordinator process and the SMs to code executed by the concurrent entities. Each SM is converted to code by putting together the user code associated with places and transitions and the communication code. The communication code allows the concurrent entities to communicate with the coordinator. Communication consists of messages requesting permission to fire controlled transitions, messages granting permission to fire controlled transitions, firing notifications of observed transitions, and notifications of process or thread termination.

If a place of a SM has several output transitions, the next transition is chosen based on transition ranks. Ranks are determined when transitions get enabled based on user code or a default ranking method assigning random ranks. If the transition of highest rank is not controlled it is fired immediately. Otherwise, the list of possible next transitions is searched. For instance, \( t_v, t'_v, \) and \( t''_v \) are controlled. Now, a transition that is not controlled is said to be observed if the supervisor should be notified when it fires. For instance, in the context of section III, \( t_c \) is not controlled. However, its firing affects the marking of a supervisor place (Figure 4). Thus, \( t_c \) is observed.

In its normal operation the coordinator waits for messages from the concurrent entities. Firing requests are queued. Firing notifications result in an update of the supervisor marking. After a firing request or notification, the firing-request queue is searched for requests that can be granted. A request can be granted when it involves a supervisor-enabled transition that satisfies the synchronization constraints. A transition participating in synchronizations satisfies the synchronization constraints if 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 \( t \) and \( t \) is supervisor enabled. Note that \( W(p, t) \)
denotes the weight of the input arc of \( t \). Firing requests are examined in the order they are received. If a firing request is granted, the supervisor marking and the firing-request queue are updated and all involved entities are notified. 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. The program ends 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 implies 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 \( p \) may fire. 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 underlying PN. Then, the set of reachable states \((\mu, v)\) of the HPN is a subset of the set of reachable states of the PN. Therefore, the conclusion follows.

**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, $\mu_2 = 1$) 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, $\mu_2 = 1$) and the entity waits for permission to fire $t_2$.

A possible approach to the modeling of deterministic choice was included 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) $p' = \{t\}$, $p \rightarrow t' = \{t\}$, $W(p', t) = W(p, t)$, and $W(t', p') = 1$.
   c) $p \rightarrow t' = (p \bullet \{t\}) \cup \{t'\}$ 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 unobservable 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 PN. 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. For instance, consider the constraint $\mu_3 \leq 1$ on the PN of Figure 5(b). Assume that the place $p_1$ involves deterministic choice. Assuming all transitions observable, $t_2$ and $t_3$ uncontrollable, and $t_1$, $t_4$, and $t_5$ observable, the least restrictive supervisory policy would be to disable $t_1$ when $\mu_3 = 1$. However, user code could ensure that the concurrent entities synchronize themselves such that an entity will select $t_2$ (and not $t_3$) when $\mu_3 = 1$. Thus, in the context of the HPN, the user code policy would be less restrictive, since it would allow the sequence $t_1t_2t_4$ when $\mu_3 = 1$. 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 may be fired. 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.

\footnote{Note that [16] describes a class of applications in which transitions cannot be disabled due to uncontrollability and yet it is possible to select which uncontrollable transition will be fired.}
Indeed, consider the PN of Figure 6. Assume that all 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 $p_3$ 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. A supervisory policy preventing deadlock in the underlying PN will prevent deadlock also in the HPN.

**Proof:** Deadlock in the HPN implies that no user code is executed and all entities wait for permission to fire transitions. Since the underlying PN is normal, deadlock implies that 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 assumption 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. Note that the results of this section apply under very general circumstances, since no assumptions are made about the PN structure and marking. As can be seen in related work, stronger results can be obtained by restricting the PN structure and marking.

Ideally, the transformation to normal PNs would be applied first and then the supervisory control methods. As of 2012, the software does not implement the transformation to normal PNs. Thus, Proposition 5.4 describes a limitation of the software as well as a direction of future work. Moreover, as pointed out by one anonymous reviewer of the paper, our framework is reminiscent of the hierarchical supervision setting of discrete-event systems. A future direction of work would be to explore the possibility of applying hierarchical supervision results to our setting.

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 included together with results describing the performance of conventional methods in this framework.

REFERENCES