Eliminating Concurrency Bugs in Multithreaded Software: A New Approach Based on Discrete-Event Control

Hongwei Liao, *Student Member, IEEE,* Yin Wang, *Member, IEEE,* Jason Stanley, Stéphane Lafortune, *Fellow, IEEE,* Spyros Reveliotis, *Senior Member, IEEE,* Terence Kelly, *Senior Member, IEEE,* and Scott Mahlke, *Member, IEEE*

***Abstract*—Computer hardware is moving from uniprocessor to multicore architectures. One problem arising in this evolution is that only parallel software can exploit the full performance potential of multicore architectures, and parallel software is far harder to write than conventional serial software. One important class of failures arising in parallel software is circular-wait deadlock in multithreaded programs. In our on-going Gadara project, we use a special class of Petri nets, called Gadara nets, to systematically model multithreaded programs with lock allocation and release operations. In this paper, we propose an efficient optimal control synthesis methodology for ordinary Gadara nets that exploits the structural properties of Gadara nets via siphon analysis. Optimality in this context refers to the elimination of deadlocks in the program with minimally restrictive control logic. We formally establish a set of important properties of the proposed control synthesis methodology, and show that our algorithms never synthesize redundant control logic. We conduct experiments to evaluate the efficiency and scalability of the proposed methodology, and discuss the application of our results to real-world concurrent software.**

I. Introduction

A fundamental revolution has taken place in the computer industry in the past decade. The mainstream computer CPUs used to have only a single processor core capable of executing a single task at a time, and CPU speeds doubled roughly every 18 months according to Moore's law. Processor core speed cannot increase indefinitely, however, because faster cores would generate excessive heat. Successive CPU generations therefore now provide more processor cores rather than a faster single core and can execute several tasks at once. The problem is that only parallel software can exploit the full performance potential of multicore architectures, and parallel software is far harder to write than conventional serial software. Choreographing a productive and harmonious interplay among concurrent tasks is a very difficult task because reasoning about concurrency is very challenging for human programmers. Multicore architectures are making parallel programming unavoidable but concurrency bugs are making it costly and error-prone. Significant effort has been spent to eliminate several types of concurrency bugs; see, e.g., [25], [24], [23], [26].

In our on-going Gadara project [16], we are interested in shared-memory multithreaded software, a very common computing paradigm in which concurrent tasks share access to a pool of computer memory. Mutual exclusion locks (or "mutexes") prevent tasks from accessing the same memory concurrently, thus allowing tasks to update shared memory in an orderly way, because at most one task may hold a given lock at any moment. However, it is easy for situations to arise in which, e.g., task 1 has acquired lock A and needs lock B, while task 2 holds B but requires A; these tasks are deadlocked and neither can perform useful work. Such type of deadlock is called *circular-mutex-wait (CMW) deadlock* in the literature, where a set of threads are waiting indefinitely for one another and none of them can proceed. In this paper, we focus on CMW deadlocks, an important class of concurrency bugs. Variants of the Banker's Algorithm [6] provide a principled approach to dynamic deadlock avoidance for concurrent software. The algorithm, however, requires a central controller that can potentially impose a global serial bottleneck on the software it governs. Deadlock "Healing" [24] addresses potential deadlocks by adding "gate locks" that prevent out-of-order lock acquisitions from causing deadlocks. At runtime, actual deadlocks are detected and remedied by adding further gate locks, gradually eliminating deadlocks from programs. Healing is more practical than the Banker's Algorithm because its runtime checks are efficient and because it does not introduce a global serialization point into the software that it controls.

In the Gadara project, we adopt a model-based approach to systematically model, analyze, and control multithreaded software for the purpose of deadlock avoidance [17]. Our results on the first two steps, namely modeling and analysis, have been reported in [19]; the third step, control, is the focus of the present paper. More specifically, we employ modeling and control techniques from discrete-event systems (DES), which have discrete state spaces and event-driven dynamics. While classical control theory, which focuses on time-driven systems, has been successfully applied to computer systems [12], the application of DES to computer

H. Liao, J. Stanley, S. Lafortune, and S. Mahlke are with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109, USA, {hwliao, jasonsta, stephane, mahlke}@umich.edu.

Y. Wang and T. Kelly are with HP Labs, Palo Alto, CA 94304, USA, {yin.wang, terence.p.kelly}@hp.com.

S. Reveliotis is with the School of Industrial & Systems Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA, spyros@isye.gatech.edu.

This work was partially supported by NSF grant CCF-0819882 and an award from HP Labs Innovation Research Program (University of Michigan) and by NSF grants CMMI-0619978 and CMMI-0928231 (Georgia Institute of Technology).

systems is more recent; see, e.g., [29], [27], [20], [7], [3], [9], [15], [5]. Concurrent software is a typical example of a DES. Petri nets, a commonly used modeling formalism in DES, are employed in our project to model multithreaded programs. Reference [14] provides a review of the application of Petri nets to computer programming. There are at least three advantages of using Petri nets in this application context: (i) Petri nets provide a compact, graphical representation of a concurrent program's inherent dynamics, without explicitly enumerating its state space. (ii) The Petri net models enable formal analysis and verification of important properties of their associated programs via efficient structural analysis. (iii) The models also make possible the synthesis of *provably* correct and optimal control logic that can be instrumented in the original programs for deadlock avoidance at run-time. In this regard, we defined a special class of Petri nets, called Gadara nets, to systematically model multithreaded C programs with lock allocation and release operations [19].

The special features of Gadara nets enable the mapping of the desired property of programs (e.g, deadlock-freeness) to some structural properties of their corresponding Gadara net models. More specifically, we have formally established that a multithreaded program that can be exactly modeled as a Gadara net is deadlock-free (a *behavioral* property) if and only if a certain type of siphon (a *structural* property) cannot be reached in its associated Gadara net [19]. Therefore, once we have obtained a Gadara net model of the program, we can focus on detecting the aforementioned siphons in the net. If no such siphon is detected, then this verifies that the underlying program is deadlock-free; otherwise, we synthesize control logic to prevent the above siphons from becoming reachable, thereby avoiding their associated deadlocks. As we will discuss in Section II-B, when it is not possible to build an exact Gadara net model of a program due to modeling constraints, a conservatively-built Gadara net model is needed, which is certain to include all possible execution paths of the program (and possibly some infeasible paths as well). In this case, the absence of the aforementioned siphons is a sufficient condition for CMW-deadlock-freeness of the program; the rest of the discussion in this paper still applies for the conservative model.

In control synthesis, we employ a common control technique for Petri nets, called Supervision Based on Place Invariants (SBPI) [11], [10], [33], [13]. The control logic synthesized by SBPI is in the form of *monitor places* that augment the original net. An original Gadara net model of a concurrent program is *ordinary* by definition, i.e., all the arcs in the net have unit weights. However, after a net is augmented by monitor places and their associated arcs, the resulting *controlled Gadara net* is not necessarily ordinary in general. Moreover, Petri net models for some other application may also belong to the class of controlled Gadara nets and contain arcs with non-unit weights. In [18], we have developed a general methodology of *optimal* control synthesis for controlled Gadara nets that need not be ordinary. Technically, this proposed control methodology is also called a *maximally-permissive liveness-enforcing (MPLE)* control policy, since the synthesized control logic will provably eliminate deadlocks while otherwise minimally constraining program behavior, and the resulting controlled Gadara net is live.

The control synthesis algorithm proposed in [18] prevents a special type of siphons, termed *Resource-Induced Deadly Marked (RIDM) siphons* [28], from becoming reachable in the net. This algorithm possesses a very nice property that for any monitor place synthesized by the algorithm, its associated arcs always have unit weights. In other words, the algorithm will never introduce additional non-ordinariness to a controlled Gadara net. As a result, if our control synthesis starts with a Gadara net model of a concurrent program, then the original net is ordinary, and the subsequent controlled nets will remain ordinary as well. This motivates us to investigate in this paper the customization of the general algorithm in [18], and concentrate on the ordinary case of controlled Gadara nets, where only *resource-induced empty siphons* need to be considered. A resource-induced empty siphon is a special case of a RIDM siphon. Thus, all the properties of the general algorithm will be preserved in the customized algorithm.

The main contributions of this paper are as follows. (i) We propose an iterative control synthesis methodology for ordinary Gadara nets, which customizes the general methodology presented in [18] and preserves the properties of correctness and maximal permissiveness. (ii) We formally establish a set of important properties of the proposed control synthesis methodology, and show that our algorithms never synthesize redundant control logic. (iii) We conduct experiments to evaluate the efficiency and scalability of the proposed methodology.

The paper is organized as follows. Section II reviews the class of Gadara nets and its relevant properties. Section III provides an overview of the proposed methodology. The customized methodology for the optimal control of ordinary Gadara nets is presented in Section IV. We investigate some important properties of the proposed methodology in Section V and report on the results of its experimental evaluation in Section VI. We discuss the application of our results to real-world software in Section VII and conclude in Section VIII.

II. Gadara Project and Gadara Petri Nets

The objective of the Gadara project [1], [16] is to develop a software tool that takes as input a deadlock-prone multithreaded C program and outputs a modified version of the program that is guaranteed to run deadlock-free without affecting any of the functionalities of the program. The system architecture of the Gadara project is shown in Figure 1, which includes four stages. (i) We extract from the program source code a standard graphical representation, called Control Flow Graph (CFG), which captures the execution paths of the program. (ii) The CFG is translated into a Petri net model of the program, formally defined as a Gadara net, based on which potential deadlocks in the program can be mapped to structural features in the net. (iii) Optimal control logic is synthesized for the Gadara net using the method presented in this paper. (iv) The control logic is used to instrument

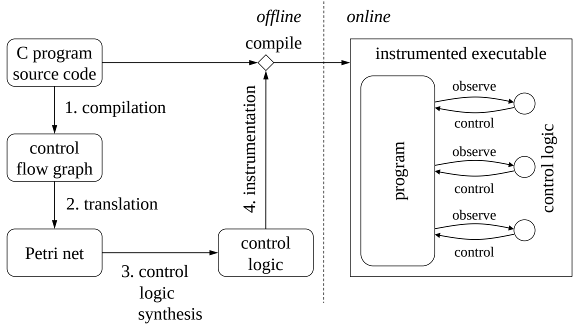


Fig. 1: The Gadara architecture

the source code and manage lock allocation and release at run-time to avoid deadlocks.

Our publications in computer science venues [31], [32] have addressed Steps 1 and 4 in detail. Moreover, the details about Step 2, i.e., the modeling and analysis of multithreaded programs using Gadara nets, are systematically studied in [19]. The results from [19] lay the foundation for the development of the results in this paper where we focus on Step 3. In this section, we review the definitions and properties of Gadara nets. We assume readers are familiar with standard Petri net definitions; see the appendix and [22] for necessary background.

A. Definitions of Gadara nets

Gadara nets are a special class of Petri nets that are employed to systematically model multithreaded C programs with lock allocation and release operations, for the purpose of deadlock analysis and resolution. The class of Gadara nets is formally defined in [19]; we review the relevant results for the sake of completeness.

Definition 1: [19] Let $I_{\mathcal{N}} = \{1, 2, ..., m\}$ be a finite set of process subnet indices. A Gadara net is an ordinary, self-loop-free Petri net $\mathcal{N}_G = (P, T, A, M_0)$ where

1) $P = P_0 \cup P_S \cup P_R$ is a partition such that: a) $P_S = \bigcup_{i \in I_{\mathcal{N}}} P_{S_i}, P_{S_i} \neq \emptyset$, and $P_{S_i} \cap P_{S_j} = \emptyset$, for all $i \neq j$; b) $P_0 = \bigcup_{i \in I_{\mathcal{N}}} P_{0_i}$, where $P_{0_i} = \{p_{0_i}\}$; and c) $P_R = \{r_1, r_2, ..., r_n\}$, $n > 0$.
2) $T = \bigcup_{i \in I_{\mathcal{N}}} T_i, T_i \neq \emptyset, T_i \cap T_j = \emptyset$, for all $i \neq j$.
3) For all $i \in I_{\mathcal{N}}$, the subnet $\mathcal{N}_i$ generated by $P_{S_i} \cup \{p_{0_i}\} \cup T_i$ is a strongly connected state machine. There are no direct connections between the elements of $P_{S_i} \cup \{p_{0_i}\}$ and T_j for any pair (i, j) with $i \neq j$.
4) $\forall p \in P_S$, if $|p \bullet| > 1$, then $\forall t \in p\bullet, \bullet t \cap P_R = \emptyset$.
5) For each $r \in P_R$, there exists a unique minimal-support P-semiflow, Y_r, such that $\{r\} = \| Y_r \| \cap P_R$, $(\forall p \in \|Y_r\|)(Y_r(p) = 1)$, $P_0 \cap \|Y_r\| = \emptyset$, and $P_S \cap \|Y_r\| \neq \emptyset$.
6) $\forall r \in P_R, M_0(r) = 1, \forall p \in P_S, M_0(p) = 0$, and $\forall p_0 \in P_0, M_0(p_0) \geq 1$.
7) $P_S = \bigcup_{r \in P_R} (\|Y_r\| \setminus \{r\})$.

$\mathcal{N}_G$ is defined to be an ordinary Petri net that consists of three types of places: (i) P_0 is the set of *idle places* that model the operational "boundary" of the net processes and, from a more technical standpoint, are used to facilitate the discussion

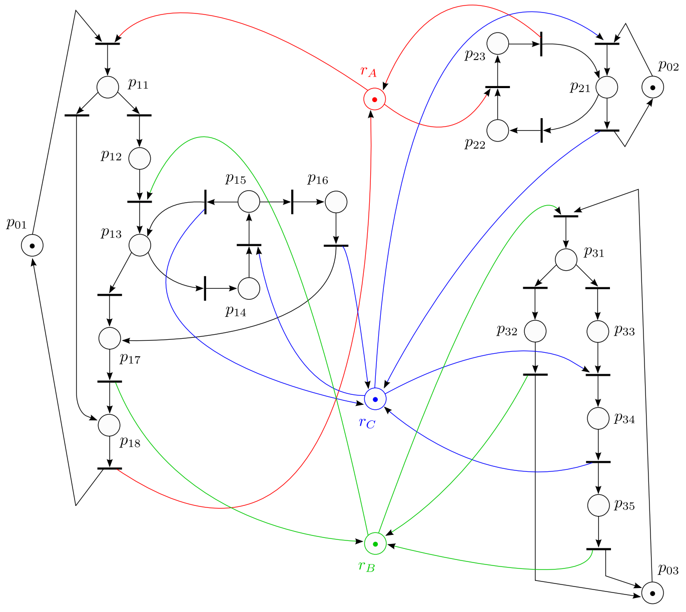


Fig. 2: A deadlock example in the Linux kernel: Gadara net model

of liveness and other properties; (ii) P_S is the set of *operation places*, each of which models a basic block of the program in its critical section; and (iii) P_R is the set of *resource places* that model mutexes and are shared among the threads. Each resource, modeled by a place in P_R, must satisfy the semiflow requirement as specified in Condition 5, which implies that a lock acquired by a thread will always be released later. A detailed discussion about Definition 1 is presented in [19].

Example 1: The Gadara net model $\mathcal{N}_G$ of a deadlock bug in version 2.5.62 of the Linux kernel is shown in Figure 2. This model, together with the source code involved in the deadlock, is presented in [19] (without control). The model and our analysis to be presented later in this paper capture the real deadlock bug of the underlying program that is studied in [8]. We will use it as a running example, and demonstrate the relevant control synthesis throughout this paper. $\square$

Given a Gadara net model of a deadlock-prone program, we employ SBPI to synthesize control logic and augment the original Gadara net by the obtained monitor places. The augmented net is called a controlled Gadara net, denoted as $\mathcal{N}_G^c$; the definition of the class of $\mathcal{N}_G^c$ is given in [19]. In general, a controlled Gadara net is not necessarily ordinary, due to the addition of monitor places and their associated arcs. As a special case, if all the arcs associated with the monitor places in the net have unit weights, then the resulting controlled Gadara net is an ordinary net. Next, we define this special subclass of controlled Gadara nets, which, as discussed in Section I, is the focus of the control synthesis presented in this paper.

Definition 2: [19] Let $\mathcal{N}_G = (P, T, A, M_0)$ be a Gadara net. A controlled Gadara net $\mathcal{N}_{G1}^c = (P \cup P_C, T, A \cup A_C, M_0^c)$ is an *ordinary*, self-loop-free Petri net such that, in addition to all conditions in Definition 1 for $\mathcal{N}_G$, we have

8) For each $p_c \in P_C$, there exists a unique minimal-support P-semiflow, Y_{p_c}, such that $\{p_c\} = \|Y_{p_c}\| \cap P_C$, $P_0 \cap$

$||Y_{p_c}|| = \emptyset$, $P_R \cap \| Y_{p_c} \| = \emptyset$, $P_S \cap \| Y_{p_c} \| \neq \emptyset$, and $Y_{p_c}(p_c) = 1$.

9) For each $p_c \in P_C$, $M_0^c(p_c) \geq \max_{p \in P_S} Y_{p_c}(p)$.

From Definition 2, we can see that $\mathcal{N}_{G1}^c$ preserves the net structure of $\mathcal{N}_G$. A monitor place $p_c \in P_C$ can be considered as a *generalized* resource place, i.e., p_c must also satisfy a semiflow requirement that is specified in Condition 8, but is weaker than that in Condition 5. Due to the similarity between the original resource places and the synthesized monitor places, we will use the term "generalized resource place" to refer to any place $p \in P_R \cup P_C$. By Definition 2, a monitor place can have multiple initial tokens.

As we mentioned in Section I and will further elaborate later, the control synthesis algorithm to be presented next guarantees that for any monitor place synthesized by this algorithm, its associated arcs always have unit weights. In the particular application of concurrent software, we always start with a Gadara net model $\mathcal{N}_G$ of the software, which is ordinary. Thus, by applying the aforementioned control synthesis algorithm, the resulting controlled Gadara nets will remain within the class of $\mathcal{N}_{G1}^c$. Consequently, we can restrict our attention to $\mathcal{N}_G$ and $\mathcal{N}_{G1}^c$ in the following development of the control synthesis algorithm.[1]

Remark 1: We observe that $\mathcal{N}_G$ is a special subclass of $\mathcal{N}_{G1}^c$, where $P_C = \emptyset$ and $A_C = \emptyset$. Therefore, any property that we derive for $\mathcal{N}_{G1}^c$ holds for $\mathcal{N}_G$ as well.

According to the semantics of the program modeled by Gadara nets, branching transitions[2], such as those corresponding to `if/else`, should not be constrained by any resource place, which is stated in Condition 4 of Definition 1. Any monitor place, as a generalized resource place, is also desired to satisfy a similar condition, so that the corresponding control logic can be properly instrumented in the program. Technically, the branching transitions in Gadara nets are said to be *uncontrollable*, as they cannot be disabled by any generalized resource place; if a controlled Gadara net satisfies that no monitor place in the net will attempt to disable any uncontrollable transition, then the net is said to be *admissible*. In the remainder of this paper, we only consider admissible $\mathcal{N}_{G1}^c$.

Assumption 1: $\mathcal{N}_{G1}^c$ is admissible.
The control synthesis algorithm to be presented guarantees the satisfaction of Assumption 1 in the considered application context.

B. Properties of Gadara nets

The main properties of Gadara nets are formally established in [19]. We introduce some relevant definitions, and discuss the properties that will be used in the following control synthesis.

A Petri net is *live* if for any transition t in the net and any reachable marking M, there exists another marking M' that is reachable from M, such that t is enabled under M'. In other words, in a *live* Petri net, starting from any reachable marking, any given transition can always be enabled in some future reachable marking.

Perfect static analysis of program behavior is undecidable. A conservative model is almost needed due to the control flow obtained with limited static analysis. We do our best to build an accurate model but we err on the side of being "conservative" when uncertain, e.g., some paths in the model may not be feasible in the program. When there is uncertainty with static analysis, we conservatively approximate the model in order to capture the CMW deadlock [31]. Based on the above discussion, and Proposition 2 and Theorem 3 established in [19], we have the following proposition that bridges a multithreaded program and its associated model.

Proposition 1: Given a conservatively-built Gadara net model of a multithreaded program, the program is CMW-deadlock-free if the Gadara net model is live.

Example 2: Referring to Figure 2, let us consider the reachable marking M_{u1}, where there is one token in p_{14}, one in p_{22}, and one in p_{03}, while all other places are empty. At marking M_{u1}, all the transitions in the net are disabled, i.e., the net is in a *total deadlock*. □

Proposition 1 implies that the goal of deadlock-avoidance of a program can be achieved by *liveness-enforcing* control of its corresponding Gadara net model. Our main intention for modeling a program via a Gadara net is to enable the analysis (and control) of the *behavioral* properties of the program through the analysis of the *structural* properties of the corresponding Gadara net. Here, the behavioral property of interest is deadlock-freeness of the program (and liveness of its associated Gadara net). Yet, as we will show in the following, we only need to focus on the structure of the Gadara net to detect the program's potential deadlocks, without exhaustively enumerating all the possible behaviors of the program. In this regard, we introduce the notion of siphon, which is a well-defined structural construct in Petri nets.

Definition 3: A *siphon* is a nonempty set of places S, such that $\bullet S \subseteq S \bullet$.

Definition 4: In Gadara nets, a siphon S is said to be a *resource-induced (RI)* siphon, if S contains at least one generalized resource place, i.e., $S \cap (P_R \cup P_C) \neq \emptyset$.

We further introduce the notion of modified marking [28] to facilitate our discussion.

Definition 5: Given $\mathcal{N}_{G1}^c$ and a reachable marking M, the *modified marking* $\overline{M}$ is defined by

$$\overline{M}(p) = \begin{cases} M(p), & \text{if } p \notin P_0; \\ 0, & \text{if } p \in P_0. \end{cases} \tag{1}$$

Modified markings essentially "erase" the tokens in idle places. The number of tokens in idle place p_{0_i} can always be uniquely recovered from the invariant implied by the structure of subnet $\mathcal{N}_i$, i.e., $M_1 = M_2$ if and only if $\overline{M}_1 = \overline{M}_2$. We use $R(\mathcal{N}_{G1}^c, M)$ to denote the set of reachable markings of $\mathcal{N}_{G1}^c$ starting from M. The set of modified markings induced by the set of reachable markings is defined by $\overline{R}(\mathcal{N}_{G1}^c, M_0^c) = \{\overline{M} | M \in R(\mathcal{N}_{G1}^c, M_0^c)\}$.

The following theorem relates the liveness property of a Gadara net to its structural properties in terms of siphons. This

[1]The optimal liveness-enforcing control synthesis for $\mathcal{N}_G^c$ is treated in detail in [18].

[2]The set of branching transitions is formally defined as: $\{t \in T : (\exists p \in P_S), (|p \bullet| > 1) \wedge (t \in p\bullet)\}$.

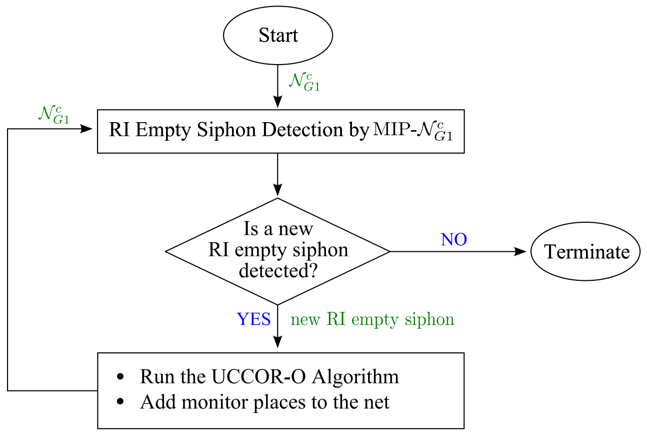

Start
$\mathcal{N}_{G1}^c$
$\mathcal{N}_{G1}^c$
RI Empty Siphon Detection by MIP-$\mathcal{N}_{G1}^c$
Is a new RI empty siphon detected?
NO
Terminate
YES
new RI empty siphon
Run the UCCOR-O Algorithm
Add monitor places to the net

Fig. 3: Iterative control of (controlled) Gadara nets: Ordinary case (ICOG-O)

theorem is a direct result of Theorem 1 presented in [19].[3] According to Remark 1, this theorem also holds for $\mathcal{N}_G$.

Theorem 1: (Liveness of Gadara nets) $\mathcal{N}_{G1}^c$ is live *iff* there does not exist a modified marking $\overline{M} \in \overline{R}(\mathcal{N}_{G1}^c, M_0^c)$ and a siphon S such that S is an RI empty siphon at $\overline{M}$.

Example 3: We know from Example 2 that the net $\mathcal{N}_G$ shown in Figure 2 is not live. Let $\overline{M}_{u1}$ be the modified marking that is induced by the marking M_{u1} defined in Example 2. At $\overline{M}_{u1}$, there is one token in p_{14} and one in p_{22}, while all other places are empty. Let S_1 be the set of all empty places in the net at $\overline{M}_{u1}$. Then, S_1 is an RI empty siphon at $\overline{M}_{u1}$. □

Proposition 1 and Theorem 1 together imply that the goal of deadlock-avoidance of a program can be achieved by preventing RI empty siphons from becoming reachable in its associated Gadara net model. They serve as a foundation for the control synthesis to be carried out next.

III. Overall Methodology

In this section, we first present an overview of our control synthesis methodology based on RI empty siphons. Then, we review an efficient method for detecting RI empty siphons using mathematical programming, which is discussed in detail in [19].

A. Iterative control of Gadara nets

Our overall strategy for control synthesis is shown in Figure 3 and described as follows. Given a multithreaded program and its associated Gadara net model $\mathcal{N}_G$, we first detect if there is a potential RI empty siphon that can be reached under the modified markings of $\mathcal{N}_G$. For the detected RI empty siphon, we synthesize control logic to prevent it from becoming reachable, and obtain a controlled Gadara net $\mathcal{N}_{G1}^c$. Then, we detect again, over the modified markings of $\mathcal{N}_{G1}^c$, if there is a new RI empty siphon; and synthesize control logic to prevent it, if any. The above process continues, until there is no new RI empty siphon being detected. According to Proposition 1 and Theorem 1, upon termination, the resulting Gadara net is live, and its corresponding program is deadlock-free.

We see that the proposed methodology is an iterative process, because (i) there may be some RI empty siphons that have not been identified in the previous iterations and need further consideration, and (ii) the synthesized monitor places are generalized resource places, so that they may introduce new potential RI empty siphons in the controlled net. We refer to the above process as the ICOG-O Methodology, which stands for "Iterative Control Of (controlled) Gadara nets: Ordinary case"; the general ICOG Methodology that works for both ordinary and non-ordinary cases is presented in [18]. While ICOG in [18] is based on exploiting RIDM siphons [28], [19] (which can be considered as a generalization of the notion of RI empty siphons for non-ordinary nets), ICOG-O presented in this paper customizes ICOG and only considers RI empty siphons, resulting in lower analytical complexity and some interesting properties. In particular, bookkeeping of prevented states, which is required in ICOG in [18], is no longer necessary in ICOG-O.

The main features of the proposed ICOG-O Methodology to be presented are summarized as follows. (i) ICOG-O is based on structural analysis (using RI empty siphons), and does not require the construction of the reachability space of the net. (ii) ICOG-O is correct and maximally permissive with respect to the goal of liveness enforcement. (iii) ICOG-O is guaranteed to terminate in a finite number of iterations.

There are two major tasks in ICOG-O: detecting RI empty siphons and rendering them unreachable. For the first task, the potential RI empty siphon is detected in each iteration by a Mixed Integer Programming (MIP) formulation we proposed in [19]. We will briefly review this formulation in Section III-B. For the second task, the detected RI empty siphon is prevented by the UCCOR-O Algorithm, which will be presented in Section IV.

B. Detection of RI empty siphons [19]

In [19], we have developed a customized and efficient MIP formulation for the detection of RI empty siphons in $\mathcal{N}_G$ and $\mathcal{N}_{G1}^c$. The formulation exploits the following important property of Gadara nets: If a Gadara net is not live, then the net will always reach a total-deadlock modified-marking $\overline{M}$ (with M being different from the initial marking), i.e., a modified marking $\overline{M}$ where all the transitions in the net are disabled. Moreover, if we let S be the set of all empty places at $\overline{M}$, then S is an RI empty siphon [19]. Using this property, the problem of detecting an RI empty siphon in $\mathcal{N}_G$ and $\mathcal{N}_{G1}^c$ can be reduced to the problem of finding a total-deadlock modified-marking that is potentially reachable in the net. The latter one can be solved by the MIP formulation (2)–(10), denoted as MIP-$\mathcal{N}_{G1}^c$, which is briefly reviewed as follows.

In the formulation MIP-$\mathcal{N}_{G1}^c$, $\overline{M}(p)$ is a *binary indicator variable* associated with any place $p \in P$, such that if p is not an empty place at $\overline{M}$, then $\overline{M}(p) = 1$; otherwise,

[3]Liveness of $\mathcal{N}_{G1}^c$ is also equivalent to the absence of any empty siphon in the original reachable markings of the net. But we have opted to use the result of Theorem 1 in order to stay close to the developments of the results in [18].

$\overline{M}(p) = 0$. In fact, for any $p \in P_0 \cup P_S \cup P_R$, $\overline{M}(p)$ represents both its associated binary indicator variable and its modified marking; however, for any $p \in P_C$, $\overline{M}(p)$ only represents its associated binary indicator variable, but not necessarily its modified marking (a slight abuse of notation).

MIP-$\mathcal{N}^c_{G1}$:

$$\min \sum_{p \in P_S} \overline{M}(p) \tag{2}$$

$$\begin{aligned}
\text{s.t.} \quad & M = M_0 + D\sigma && (3)\\
& \overline{M}(p) = M(p), \forall p \in P_S \cup P_R; && \\
& \overline{M}(p) = 0, \forall p \in P_0 && (4)\\
& \overline{M}(p) = 0, \forall p \in Q, \text{where} && \\
& \quad Q = \{q \in P : (\exists t \in T), (\bullet t = \{q\}) && \\
& \qquad \wedge (q \in P_S)\} && (5)\\
& \sum_{p \in \bullet t} \overline{M}(p) - |\bullet t| + 1 \leq 0, && \\
& \quad \forall t \text{ s.t. } |\bullet t| > 1 && (6)\\
& \sum_{p \in P_S} \overline{M}(p) \geq 2 && (7)\\
& M \geq 0; \sigma \in \mathbb{Z}^+_0 && (8)\\
& M(p) \geq \overline{M}(p) \geq \frac{M(p)}{SB(p)}, \forall p \in P_C && (9)\\
& \overline{M}(p) \in \{0, 1\}, \forall p \in P_C && (10)
\end{aligned}$$

The objective function (2) seeks to minimize the number of marked operation places in the detected total-deadlock modified-marking. The selection of such an objective function will produce siphons that are efficient for control synthesis using ICOG-O. Some resulting interesting properties will be presented in Section V.

We briefly explain the constraints (3) – (10); see [19] for a detailed discussion. Constraint (3) is the state equation of the net. Constraint (4) connects an original marking with its associated modified marking based on Definition 5. Constraints (4), (5), and (6) enforce that all the transitions in the net are disabled at $\overline{M}$. Constraint (7) follows from the fact that at least two threads must be involved in a CMW deadlock. Constraint (8) specifies the sign restrictions for the variables M and σ. Constraints (9) and (10) are specifically formulated for $\mathcal{N}^c_{G1}$, i.e., the siphon detection in $\mathcal{N}_G$ will not involve these two constraints since P_C is empty in that case. These two constraints set the values of the indicator variables associated with any monitor place. In particular, Constraint (10) specifies that $\overline{M}(p)$ is used as an indicator variable in the context of this formulation (and not as the modified marking of the corresponding monitor place p). On the other hand, the parameter $SB(p)$ that appears in Constraint (9) denotes a structural bound for the marking of place p. In Gadara nets, we can set: $SB(p) = M^c_0(p)$, $\forall p \in P_0 \cup P_C$, and $SB(p) = 1$, $\forall p \in P_S \cup P_R$.

IV. Optimal Control Algorithm Based on RI Empty Siphons

Once an RI empty siphon is detected, we input it to the control synthesis algorithm, called UCCOR-O, which customizes the general algorithm UCCOR presented in [18]. The abbreviation UCCOR-O stands for "Unsafe-Covering-based Control Of RIDM siphons: Ordinary case". In UCCOR-O, we focus on a special type of RIDM siphons in ordinary nets, namely RI empty siphons. UCCOR-O synthesizes control logic based on the notion of unsafe covering, which is introduced next.

Similar to the modified-marking defined in Definition 5, we further define the notion of P_S-marking to facilitate the discussion.

Definition 6: Given $\mathcal{N}^c_{G1}$ and $M \in R(\mathcal{N}^c_{G1}, M^c_0)$, the *$P_S$-marking* $\overline{\overline{M}}$ is defined by

$$\overline{\overline{M}}(p) = \begin{cases} M(p), & \text{if } p \in P_S; \\ 0, & \text{if } p \notin P_S. \end{cases} \tag{11}$$

P_S-markings essentially "erase" the tokens in idle places and generalized resource places, retaining only tokens in operation places. Given the P_S-marking $\overline{\overline{M}}$ corresponding to the original marking M, the number of tokens in places P_R and P_C under M can be uniquely recovered from their associated semiflows; the number of tokens in places P_0 can also be uniquely recovered similar to the case of modified marking. In other words, P_S-markings do not introduce any ambiguity, i.e., there is a one-to-one mapping between the original marking and the P_S-marking, such that $M_1 = M_2$ if and only if $\overline{\overline{M}}_1 = \overline{\overline{M}}_2$. Therefore, we can restrict out attention to P_S-markings in the following discussion, which greatly facilitates the control synthesis. Furthermore, from Condition 6 of Definition 1, we know that a P_S-marking is always a *binary vector*, i.e., any component of a P_S-marking is either 0 or 1.

In view of the above discussion, when focusing on P_S-markings, we "don't care" the number of tokens in $P_0 \cup P_R \cup P_C$. We introduce the notation "χ" for the value of a P_S-marking component, where "χ" stands for "0 or 1". The notion of covering is introduced below.

Definition 7: In Gadara nets, a *covering* C is a generalized P_S-marking, whose components can be $0, 1$, or χ.

For any place $p \in P_S$, $C(p)$ represents the covering component value on p. This notation can be extended to a set of places $Q \subseteq P_S$ in a natural way. Furthermore, we extend the notion of covering so that it encompasses any place $p \in P$ by setting $C(p) = \chi$, $\forall p \in P_0 \cup P_R \cup P_C$.

Given two coverings C_1 and C_2, we say that C_1 *covers* C_2, if $\forall p \in P_S$ such that $C_1(p) \neq C_2(p)$, $C_1(p) = \chi$. The "cover" relationship between a covering and a P_S-marking is defined in a similar way. Note that as ICOG-O evolves, new monitor places will be added to the net throughout the iterations. In the rest of this paper, when comparing two coverings (or, a covering and a P_S-marking) with different dimensions, and the difference is due to the synthesized monitor places, we assume the one with a lower dimension is padded by χ's for those monitor places.

Definition 8: In $\mathcal{N}^c_{G1}$, a marking M is said to be an *RIE-*

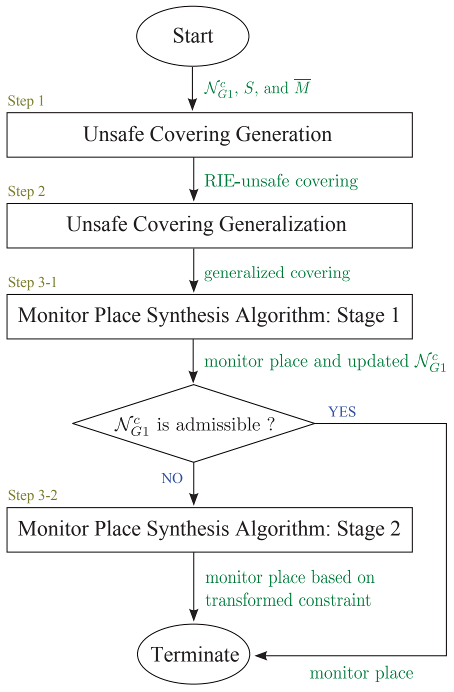

Start
$\mathcal{N}^c_{G1}$, S, and $\overline{M}$
Step 1
Unsafe Covering Generation
RIE-unsafe covering
Step 2
Unsafe Covering Generalization
generalized covering
Step 3-1
Monitor Place Synthesis Algorithm: Stage 1
monitor place and updated $\mathcal{N}^c_{G1}$
$\mathcal{N}^c_{G1}$ is admissible ?
YES
NO
Step 3-2
Monitor Place Synthesis Algorithm: Stage 2
monitor place based on transformed constraint
Terminate
monitor place

Fig. 4: Flowchart of the UCCOR-O Algorithm

unsafe marking, if at its associated modified marking $\overline{M}$, there exists at least one RI empty siphon.

Definition 9: A covering C is said to be an *RIE-unsafe covering*, if for all P_S-markings $\overline{\overline{M}}$ it covers, the corresponding M is an RIE-unsafe marking.

A. The UCCOR-O Algorithm: Overview

We are now ready to present the UCCOR-O Algorithm. We organize our presentation in a top-down manner. We first overview the procedure of UCCOR-O as illustrated in Figure 4, and then explain the three steps of UCCOR-O in subsequent sections. We will apply UCCOR-O to the running example throughout our discussion.

The input to UCCOR-O is $\mathcal{N}^c_{G1}$, an RI empty siphon S, and the associated total-deadlock modified-marking $\overline{M}$ obtained from MIP-$\mathcal{N}^c_{G1}$. The output of UCCOR-O is a monitor place that prevents the RI empty siphon S from becoming reachable. The UCCOR-O Algorithm contains three steps. In Step 1, an RIE-unsafe covering is generated based on the input to the algorithm. This covering captures the RI empty siphon we want to prevent. In Step 2, the obtained RIE-unsafe covering is generalized into a new covering, by exploiting a monotonicity property of Gadara nets. This generalization step enhances the efficiency of the algorithm, in terms of the number of undesirable markings that can be prevented by the final monitor place. In Step 3, a monitor place is synthesized to prevent the covering obtained in Step 2. Step 3 contains two stages in general. Stage 2 is necessary only when the controlled Gadara net obtained in Stage 1 is not admissible. We discuss these three steps in further detail below.

B. Unsafe Covering Generation

Step 1 of UCCOR-O generates an RIE-unsafe covering, denoted as C_{u1}, based on the input to the algorithm. We consider the following set of places:

$$\Lambda_S = \bigcup_{p \in S \cap (P_R \cup P_C)} \|Y_p\| \cup S \tag{12}$$

Intuitively, Λ_S contains the set of *all* places that are relevant to the siphon S. In particular, Λ_S complements S with all those operation places that utilize the generalized resources appearing in S.

Therefore, we can specify the values for the components of C_{u1} that are associated with Λ_S as: $C_{u1}(\Lambda_S) = \overline{M}(\Lambda_S)$; and set $C_{u1}(p) = \chi$, $\forall p \notin \Lambda_S$, since these places are irrelevant to the considered siphon. Moreover, we know from the definition of covering that we can further set $C_{u1}(p) = \chi$, $\forall p \in P_0 \cup P_R \cup P_C$. The resulting C_{u1} is input to Step 2 of UCCOR-O.

Example 4: We continue our discussion on the example in Figure 2. Let $\mathcal{N}_G$, $\overline{M}_{u1}$, and S_1, described in Example 3, be the input to UCCOR-O. After Step 1 of UCCOR-O, the RIE-unsafe covering C_{u1} is specified as follows. $C_{u1}(p_{14}) = C_{u1}(p_{22}) = 1$; $C_{u1}(p) = 0$, $\forall p \in P_S \setminus \{p_{14}, p_{22}\}$; and $C_{u1}(p_{01}) = C_{u1}(p_{02}) = C_{u1}(p_{03}) = C_{u1}(r_A) = C_{u1}(r_B) = C_{u1}(r_C) = \chi$. □

C. Unsafe Covering Generalization

Step 2 of UCCOR-O generalizes the RIE-unsafe covering obtained from Step 1, by exploiting a monotonicity property of Gadara nets, which is formally proved in [18]. The monotonicity property is explained as follows. Let M and M' be two markings of a Gadara net, which satisfy: $M(p) \geq M'(p)$, for all $p \in P_S$, and $M(p) > M'(p)$, for at least some $p \in P_S$. If M' is a marking that needs to be prevented, then M also needs to be prevented. The intuition is that loading a program, which is already in a deadlock or will unavoidably enter a deadlock, with even more active threads will only worsen the deadlock situation, but not cure it.

Based on the above property, for the RIE-unsafe covering C_{u1} obtained in Step 1, if we replace any of its 0 components (associated with operation places) by 1, the resulting covering will only cover reachable P_S-markings that need to be prevented, or non-reachable P_S-markings. Therefore, C_{u1} can be generalized by replacing all of its 0 components by χ, and the resulting covering is denoted as C_{u2}, which is input to Step 3 of UCCOR-O.

By construction, the generalized covering C_{u2} will not "miss" covering any P_S-markings that are covered by C_{u1}. In general, C_{u2} will cover a larger set of P_S-markings than C_{u1}, because the former contains more χ components. So instead of preventing C_{u1}, a monitor place that prevents C_{u2} is more efficient, in the sense that it will prevent a larger set of markings in the controlled net. More importantly, the property of maximal permissiveness is still preserved, i.e., we only prevent reachable markings that need to be prevented, or markings that are not reachable, due to the above discussion.

Example 5: Given the RIE-unsafe covering C_{u1} described in Example 4, Step 2 of UCCOR-O generalizes C_{u1} and obtains C_{u2}, which is specified as follows. $C_{u2}(p_{14}) = C_{u2}(p_{22}) = 1$; and $C_{u2}(p) = \chi$, $\forall p \in P \setminus \{p_{14}, p_{22}\}$. □

D. Monitor Place Synthesis Algorithm

Step 3 of UCCOR-O aims to find an appropriate linear inequality constraint in the form:

$$l^T M \leq b \tag{13}$$

so that SBPI can be employed to synthesize a monitor place, to prevent C_{u2} that is obtained in Step 2; the constraint should also guarantee that the resulting controlled Gadara net is *admissible*. Generally, Step 3 consists of two stages. Stage 2 is necessary only when the controlled Gadara net obtained in Stage 1 is not admissible. For the sake of simplicity and without any confusion, we let $C_u \equiv C_{u2}$ and will use the notation C_u in the following discussion. (Step 3 of UCCOR-O in this paper is similar to the corresponding step of UCCOR that is presented in [18], for which no customization is necessary; we include it here for the sake of completeness. Also note that in the general UCCOR Algorithm, there is a step called "Inter-Iteration Coverability Check", which is eliminated in the customized UCCOR-O Algorithm. The reason of this customization will become clear when we present Theorem 3 in Section V.)

In Stage 1, we specify a linear inequality constraint in the form of (13) for C_u. From the first two steps of UCCOR-O, we know that C_u contains only "1" or "χ" components. The parameters of the constraint associated with C_u are:

$$l_{C_u}(p) = \begin{cases} 1, & \text{if } C_u(p) = 1; \\ 0, & \text{otherwise.} \end{cases} \tag{14}$$

$$b_{C_u} = \left(\sum_{p: p \in \Lambda_S \text{ and } C_u(p)=1} C_u(p) \right) - 1 \tag{15}$$

According to Theorem 3 in [18], this constraint *only* prevents C_u (i.e., any P_S-marking or covering that is covered by C_u). Thus, the corresponding control logic synthesized based on this constraint is maximally permissive. The synthesis of a monitor place based on this constraint can be achieved by SBPI. If the resulting $\mathcal{N}^c_{G1}$ is admissible, then Stage 2 is not necessary and we can continue with the next iteration of ICOG-O; otherwise, we need to proceed to Stage 2, where constraint transformation is carried out to deal with the partial controllability and ensure the admissibility of $\mathcal{N}^c_{G1}$.

Example 6: We illustrate Stage 1 by continuing our discussion on the running example. Given the covering described in Example 5, we specify the following linear inequality constraint according to (14) and (15):

$$M(p_{14}) + M(p_{22}) \leq 1 \tag{16}$$

The monitor place p_c, which enforces (16), is synthesized by SBPI and shown in Figure 5. We see that p_c has two outgoing arcs, both of which connect to branching transitions. In this running example, we define that only the lock acquisition

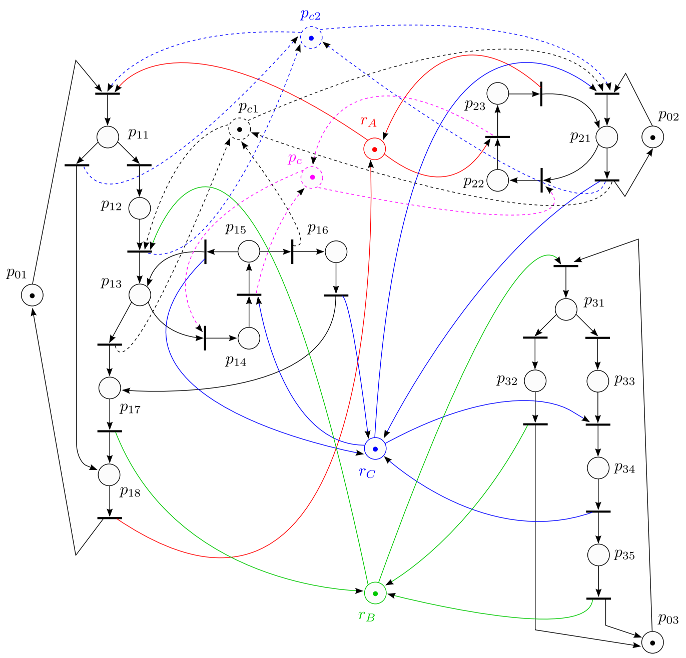


Fig. 5: A deadlock example in the Linux kernel: Controlled Gadara net model

transitions are controllable; and all the other transitions (i.e., those corresponding to branching and lock releases) are uncontrollable. Thus, the controlled net that contains p_c is not admissible. We resolve this problem in Stage 2 of Step 3. □

In Stage 2, the original constraint specified by (14) and (15) is transformed, so that the new constraint, when applied to SBPI, will render a monitor place that leads to an admissible controlled net. For the sake of discussion, the constraint obtained in Stage 1 can be rewritten as:

$$M(p_1) + M(p_2) + ... + M(p_n) \leq n - 1 \tag{17}$$

The key idea of the proposed constraint transformation is the following. If place p_i in (17) can gain tokens through a sequence of uncontrollable transitions, places along the sequence of uncontrollable transitions must be included to the left-hand-side of (17) as we cannot prevent these transitions from firing and populating tokens into p_i. We make two remarks for the above statement: (i) The set of places corresponding to a given sequence of uncontrollable transitions is unique due to the state-machine structure of the process subnet. (ii) The uncontrollable transitions in this sequence are not blocked by any generalized resource place, otherwise they would be controllable. The pseudo-code that implements the constraint transformation for (17) is given in Figure 6. Based on the set of places C obtained above, the new, transformed constraint is:

$$\sum_{p \in C} M(p) \leq n - 1 \tag{18}$$

The important properties of the proposed constraint transformation technique are summarized as follows; see [18] for a formal treatment. (i) The constraint transformation

```
Algorithm: Constraint Transformation
Input: A linear inequality constraint, e.g., (17)
Output: A set of places $C$
Method:
    1. add $p_1, ..., p_n$ in (17) to stack $S$, and to set $C$
    2. while $S$ is not empty
    3.     $p = S.pop()$
    4.     for each uncontrollable $t$ in $\bullet p_i$, if $\bullet t$ is not in $C$,
           add $\bullet t$ to $S$ and $C$
    5. end while
```

Fig. 6: The constraint transformation technique used in Stage 2 of the Monitor Place Synthesis Algorithm

technique guarantees that the resulting controlled Gadara net is admissible. (ii) Any marking prevented by the original constraint is also prevented by the new constraint. (iii) Any reachable marking that is prevented by the new constraint but not by the original constraint, can reach a marking prevented by the original constraint via a sequence of uncontrollable transitions.

Example 7: We apply the proposed constraint transformation technique to (16), which is obtained from Stage 1 in Example 6. After Stage 2, the new, transformed constraint is:

$$\begin{aligned}&[M(p_{14}) + M(p_{13}) + M(p_{15})]\\&+[M(p_{22}) + M(p_{21}) + M(p_{23})] \leq 1 \qquad (19)\end{aligned}$$

The monitor place p_{c1}, which enforces (19), is synthesized by SBPI and shown in Figure 5. We see that p_{c1} has two outgoing arcs, both of which connect to controllable transitions. Thus, the controlled net that contains p_{c1} is admissible. We denote the resulting controlled Gadara net as $\mathcal{N}_{G1}^{c(1)}$, which consists of $\mathcal{N}_G$ and p_{c1}. □

Example 7 completes the first iteration of ICOG-O on the running example. We continue our discussion on the second iteration in Example 8.

Example 8: In the second iteration of ICOG-O, we first input the net $\mathcal{N}_{G1}^{c(1)}$ obtained from Example 7 into MIP-$\mathcal{N}_{G1}^c$ for the detection of RI empty siphons. MIP-$\mathcal{N}_{G1}^c$ finds a total-deadlock modified-marking $\overline{M}_{u2}$, where there is one token in p_{12} and one in p_{22}, while all other places are empty. Note that this corresponds to a circular-wait deadlock induced by r_A and p_{c1}. Let S_2 be the set of all empty places in the net at $\overline{M}_{u2}$. Then, S_2 is an RI empty siphon at $\overline{M}_{u2}$.

We input $\mathcal{N}_{G1}^{c(1)}$, S_2, and $\overline{M}_{u2}$ to UCCOR-O. Step 1 of UCCOR-O generates the covering C_{u1}, which is specified as: $C_{u1}(p_{12}) = C_{u1}(p_{22}) = 1$; $C_{u1}(p) = 0, \forall p \in P_S \setminus \{p_{12}, p_{22}\}$; and $C_{u1}(p) = \chi$, $\forall p \in P_0 \cup P_R \cup P_C$. Step 2 of UCCOR-O further generalizes C_{u1} and obtains the covering C_{u2}, which is specified as: $C_{u2}(p_{12}) = C_{u2}(p_{22}) = 1$; and $C_{u2}(p) = \chi$, $\forall p \in P \setminus \{p_{12}, p_{22}\}$.

Based on C_{u2}, Stage 1 of Step 3 of UCCOR-O constructs the following constraint:

$$M(p_{12}) + M(p_{22}) \leq 1 \qquad (20)$$

Similar to the situation encountered in Example 6, the monitor place that is synthesized by SBPI and enforces (20), will attempt to disable uncontrollable transitions. Thus, the resulting controlled net would not be admissible, which necessitates Stage 2 of Step 3.

In Stage 2, the original constraint in (20) is transformed into:

$$[M(p_{12})+M(p_{11})]+[M(p_{22})+M(p_{21})+M(p_{23})] \leq 1 \quad (21)$$

The monitor place p_{c2}, which enforces (21), is synthesized by SBPI and shown in Figure 5. We denote the resulting controlled net as $\mathcal{N}_{G1}^{c(2)}$, which consists of $\mathcal{N}_G$, p_{c1}, and p_{c2}. The controlled Gadara net $\mathcal{N}_{G1}^{c(2)}$ is admissible.

In the third iteration of ICOG-O, we input $\mathcal{N}_{G1}^{c(2)}$ into MIP-$\mathcal{N}_{G1}^c$, and no solution is found. Therefore, no new RI empty siphon can be detected in $\mathcal{N}_{G1}^{c(2)}$, and ICOG-O terminates. □

V. Properties of the Proposed Control Synthesis Methodology

The general ICOG Methodology and UCCOR Algorithm proposed in [18] are shown to be both correct and maximally permissive, with respect to the goal of liveness enforcement of Gadara nets via siphon-based control. Moreover, ICOG is guaranteed to terminate in a finite number of iterations. The general ICOG Methodology is developed independent of the method used to detect siphons. Thus, ICOG-O and UCCOR-O presented in this paper, which are customized versions of ICOG and UCCOR respectively, still preserve the aforementioned properties. Moreover, the customization possesses some new properties that are formally established below.

A. Properties of the UCCOR-O Algorithm

Theorem 2: In $\mathcal{N}_{G1}^c$, for any monitor place $p_c \in P_C$ synthesized by UCCOR-O, any process subnet will never have two consecutive resource acquisitions from p_c without a resource release to p_c in between. Also, any process subnet will never have two consecutive resource releases to p_c without a resource acquisition from p_c in between. Moreover, all the arcs associated with p_c have unit arc weights.[4]

Proof: Since any covering C_u considered in UCCOR-O is a generalized P_S-marking, the linear constraint generated in Step 3 of UCCOR-O will only involve operation places. That is, for any p such that $l_{C_u}(p) = 1$, p must be an operation place; further, $l_{C_u}(p) = 0, \forall p \in P_0 \cup P_R \cup P_C$. Given C_u, let p_c be the corresponding monitor place synthesized by UCCOR-O using SBPI. Also, let Q be the set of places that are involved in the linear constraint, i.e., $Q = \{p \in P_S : l_{C_u}(p) = 1\}$.

Consider an arbitrary transition $t \in T$. We discuss the connectivity of the monitor place p_c to t in four cases, as shown in Figure 7, where the places that belong to Q are highlighted. Due to the aforementioned property of l_{C_u}, we can focus on process subnets (since the generalized resource places will not affect the connectivity of p_c to t in terms of l_{C_u}). Recall Condition 3 of Definition 1 that, in the process subnet, t has only one input place, denoted as p_{11}, and one output place, denoted as p_{12}.

[4]This theorem also applies to UCCOR developed for the control synthesis for $\mathcal{N}_G^c$, as presented in [18]. However, this result was not presented in [18].

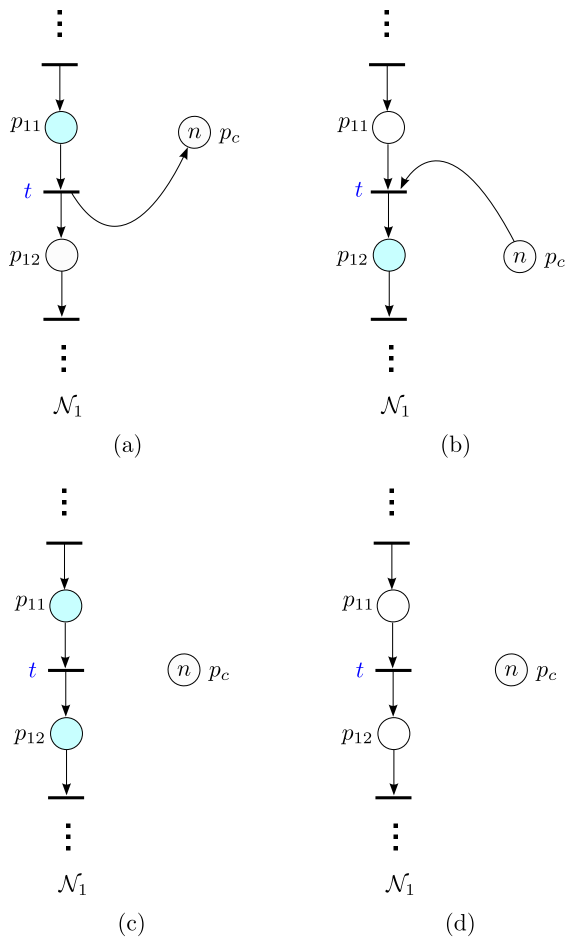


Fig. 7: Cases considered in the proof: (a) Case 1; (b) Case 2; (c) Case 3; (d) Case 4

Case 1: $p_{11} \in Q$ and $p_{12} \notin Q$, as shown in Figure 7(a). In this case, we have: $l_{C_u}(p_{11}) = 1$ and $l_{C_u}(p_{12}) = 0$. The state machine structure of the process subnets leads to the following feature of the incidence matrix D of $\mathcal{N}^c_{G1}$: if we only consider the rows associated with the places in all the process subnets, then in the column corresponding to t, there are *only* two nonzero entries, i.e., $D_{p_{11},t} = -1$ and $D_{p_{12},t} = 1$. Therefore, the algebraic calculation of SBPI [13] will result in one arc connecting t to p_c, whose weight equals to 1.

Case 2: $p_{11} \notin Q$ and $p_{12} \in Q$, as shown in Figure 7(b). In this case, we have: $l_{C_u}(p_{11}) = 0$ and $l_{C_u}(p_{12}) = 1$. Similar to the analysis in Case 1, the calculation results in one arc connecting p_c to t, whose weight equals to 1.

Case 3: $p_{11} \in Q$ and $p_{12} \in Q$, as shown in Figure 7(c). In this case, we have: $l_{C_u}(p_{11}) = l_{C_u}(p_{12}) = 1$. According to the calculation of SBPI, no arc will be synthesized between p_c and t.

Case 4: $p_{11} \notin Q$ and $p_{12} \notin Q$, as shown in Figure 7(d). In this case, we have: $l_{C_u}(p_{11}) = l_{C_u}(p_{12}) = 0$. Similar to Case 3, no arc will be synthesized between p_c and t.

Note that Cases 1 and 4 also apply to the situation when t is a terminating transition of the process subnet and p_{12} is an idle place. Similarly, Cases 2 and 4 also apply to the situation when t is an initiating transition of the process subnet and p_{11} is an idle place. Thus, the above four cases cover all the possibilities of the connectivity of p_c to an arbitrary transition t.

As a result, if we traverse from the upstream to the downstream of a process subnet, it is impossible for the subnet to have two consecutive resource acquisitions from (or resource releases to) p_c. ■

As a consequence of Theorem 2, we have the following corollary, which can be considered as a special case of Condition 8 of Definition 2 when UCCOR-O is employed to synthesize monitor places.

Corollary 1: In Gadara nets, for each $p_c \in P_C$ synthesized by the UCCOR-O Algorithm, there exists a unique minimal-support P-semiflow, Y_{p_c}, such that $\{p_c\} = \|Y_{p_c}\| \cap P_C$, $(\forall p \in \|Y_{p_c}\|)(Y_{p_c}(p) = 1)$, $P_0 \cap \|Y_{p_c}\| = \emptyset$, $P_R \cap \|Y_{p_c}\| = \emptyset$, and $P_S \cap \|Y_{p_c}\| \neq \emptyset$.

B. Properties of the ICOG-O Methodology

Define $C_u^{(i)}$ to be the covering input to Step 3 of UCCOR-O in the i-th iteration of ICOG-O; and define

$$K^{(i)} = \sum_{p:p\in\Lambda_S \text{ and } C_u^{(i)}(p)=1} C_u^{(i)}(p) \tag{22}$$

namely, $K^{(i)}$ is the total number of 1's in $C_u^{(i)}$ that is induced by the siphon S under consideration.

Lemma 1: In ICOG-O, $K^{(i)}$ is non-decreasing with respect to i. That is, the total number of 1's in the covering considered in Step 3 of UCCOR-O is non-decreasing, throughout the iterations of ICOG-O.

Proof: Consider an arbitrary $i \geq 1$, and let p_c be the monitor place synthesized in the i-th iteration of ICOG-O that prevents $C_u^{(i)}$. According to Step 3 of UCCOR-O, the initial marking of p_c is $M_0(p_c) = K^{(i)} - 1$.

We mentioned above that a monitor place is essentially a generalized resource place and may introduce new potential deadlocks in the controlled net. More specifically, the monitor place p_c can *directly induce* a new circular-wait deadlock, if in the controlled net, (i) there exists a total-deadlock modified-marking $\overline{M}$ ($M \neq M_0^c$), such that p_c is empty at $\overline{M}$, and (ii) p_c blocks at least one thread that is involved in a circular-wait deadlock at $\overline{M}$, i.e., the thread is waiting for the resource from p_c while holding some other resources involved in the deadlock.

Let $C_u^{(i+1)}$ be the covering that corresponds to the optimal solution of MIP-$\mathcal{N}^c_{G1}$ in the $(i+1)$-st iteration of ICOG-O. We consider the following two cases.

Case 1: p_c does not directly induce the deadlock involved in $C_u^{(i+1)}$, i.e., p_c is not part of the deadlock. In this case, the optimal solution of MIP-$\mathcal{N}^c_{G1}$ in the $(i+1)$-st iteration of ICOG-O must also be a feasible solution in the i-th iteration, because, by the assumption of Case 1, this optimal solution is not a new feasible solution induced by p_c. Therefore, MIP-$\mathcal{N}^c_{G1}$ guarantees that the number of 1's contained in $C_u^{(i+1)}$ will be greater than or equal to that in $C_u^{(i)}$; otherwise, $C_u^{(i+1)}$ would have been exploited in earlier iterations.

Case 2: p_c directly induces the deadlock involved in $C_u^{(i+1)}$, i.e., p_c is part of the deadlock. In this case, we show that at least $K^{(i)}$ operation places must be marked at $C_u^{(i+1)}$. Since p_c directly induces the deadlock, p_c is empty at $C_u^{(i+1)}$ (Condition (i) mentioned above). Thus, according to Theorem 2, there must be $M_0(p_c) = K^{(i)} - 1$ different operation places in $\parallel Y_{p_c} \parallel$ that are marked at $C_u^{(i+1)}$ in order to empty p_c. Moreover, we know that p_c blocks at least one thread that is involved in the deadlock at $C_u^{(i+1)}$ (Condition (ii) mentioned above). Then, there exists an output transition t of p_c, such that the (unique) input operation place of t (denoted as q_1) is marked at $C_u^{(i+1)}$, which corresponds to a thread blocked by p_c. We argue that $q_1 \notin \parallel Y_{p_c} \parallel$. If $q_1 \in \parallel Y_{p_c} \parallel$, then the (unique) output operation place of t (denoted as q_2), which belongs to $\parallel Y_{p_c} \parallel$ by definition, must satisfy $Y_{p_c}(q_2) > 1$. This contradicts Corollary 1. Thus, the marked operation place q_1 is different from the aforementioned $K^{(i)} - 1$ marked operation places in $\|Y_{p_c}\|$. As a result, at least $K^{(i)}$ operation places are marked at $C_u^{(i+1)}$, and the number of 1's contained in $C_u^{(i+1)}$, $K^{(i+1)}$, is at least $K^{(i)}$. ■

We conclude this section with an important property of ICOG-O that need not be true for ICOG.

Theorem 3: ICOG-O will not synthesize redundant monitor places. That is, there does not exist a pair of monitor places p_{ci} and p_{cj} synthesized by ICOG-O, such that the covering prevented by p_{ci} covers the covering prevented by p_{cj}.

Proof: For the sake of discussion, let p_{ci} and p_{cj} be the monitor places synthesized in the i-th and j-th iterations of ICOG-O, respectively. Correspondingly, let $C_u^{(i)}$ and $C_u^{(j)}$ be the coverings considered in Step 3 of UCCOR-O in the i-th and j-th iterations of ICOG-O, respectively. That is, p_{ci} is synthesized to prevent $C_u^{(i)}$ and p_{cj} is synthesized to prevent $C_u^{(j)}$.

If $i > j$, then according to Lemma 1 and the fact that $C_u^{(i)} \neq C_u^{(j)}$, we know that $C_u^{(i)}$ cannot cover $C_u^{(j)}$.

If $i < j$, we want to show that $C_u^{(i)}$ cannot cover $C_u^{(j)}$ either. In the i-th iteration of ICOG-O, p_{ci} is synthesized to prevent $C_u^{(i)}$; hence, any marking covered by $C_u^{(i)}$ will not be reachable in the net considered in the j-th iteration of ICOG-O. As a result, in the j-th iteration, any marking covered by $C_u^{(i)}$ will not be a feasible solution to the state equation of the net, and hence will not be a feasible solution to MIP-$\mathcal{N}_{G1}^c$. In other words, in the j-th iteration, the solution of MIP-$\mathcal{N}_{G1}^c$ and the corresponding $C_u^{(j)}$ cannot be covered by $C_u^{(i)}$. ■

VI. Experimental Evaluation

We discussed above that the development of the control synthesis methodology and the validity of the associated properties are independent of the method used to detect RI empty siphons. However, we observe that the RI empty siphon detection algorithm does play an important role in the efficiency of control synthesis; it is in fact the computational bottleneck of ICOG-O. This motivated us to develop the customized formulation, MIP-$\mathcal{N}_{G1}^c$, for efficient siphon detection in Gadara nets, which we reviewed in Section III-B. While MIP-$\mathcal{N}_{G1}^c$ is specifically designed for Gadara nets, siphon detection algorithms for more general classes of Petri nets have been extensively studied in the literature. A generic MIP formulation is presented in [4] for the detection of maximal empty siphons in ordinary, structurally bounded Petri nets, and it is one of the most widely used empty siphon detection algorithms in the literature; we refer to this formulation as MIP-ES hereafter. Our customized algorithm, MIP-$\mathcal{N}_{G1}^c$, is inspired by MIP-ES, and further incorporates the special properties of Gadara nets.

A. Objective and setup of the experiments

In this section, we investigate the performance of two versions of ICOG-O: (i) the original ICOG-O that uses MIP-$\mathcal{N}_{G1}^c$ for siphon detection, and (ii) a modified version of ICOG-O, denoted as ICOG-O-ES, that uses MIP-ES for siphon detection. Since MIP-$\mathcal{N}_{G1}^c$ is customized for Gadara nets, while MIP-ES is formulated for general, ordinary bounded Petri nets, we of course expect ICOG-O to be more efficient than ICOG-O-ES in the context of the Gadara nets. Thus, in the following experiments, we use ICOG-O-ES as the baseline for assessing and concretizing this attained efficiency by ICOG-O. We also report a sample of experimental results that demonstrate the scalability of ICOG-O.

Our experiments were completed on a Mac OS X laptop with a 2.4 GHz Intel Core2Duo processor and 2 GB of RAM. Both ICOG-O and ICOG-O-ES are implemented in C++ and compiled under the GNU gcc compiler. The MIP formulations are solved using Gurobi 3.0.1 [2]. Random Gadara nets for these experiments are generated by a random-walk-style algorithm. At each step, the program randomly decides either to grab a lock or to release one already held, according to the input parameters. Additional logic was applied to ensure valid behavior. The random Gadara net generator (available at `http://gadara.eecs.umich.edu/software.html`) is based on our experience modeling real concurrent programs [30]. The input parameters of the generator are further explained in Section VI-B and Table I.

In our experiments, for each set of parameters (each row in Table I), 150 samples of random Gadara nets are generated. The generated nets with no unsafe states[5] are removed from the samples. We set a time-out threshold of 10 seconds for the stage of RI empty siphon detection in ICOG-O and ICOG-O-ES. A net times out if it cannot be solved by either MIP-$\mathcal{N}_{G1}^c$ or MIP-ES in less than 10 seconds. Unless otherwise specified, all statistical results reported below are calculated over the sample nets where both ICOG-O and ICOG-O-ES did not time out.

B. Comparative analysis of ICOG-O and ICOG-O-ES

Figure 8 shows the time to converge (TTC) of ICOG-O and ICOG-O-ES. Figure 8(a) shows the Normalized Cumulative

[5]A state is said to be *unsafe* if (i) at this state, there exists a deadlock in the corresponding program, or (ii) starting from this state the net will unavoidably or uncontrollably reach a state, where there exists a deadlock in the corresponding program; otherwise it is said to be *safe*.

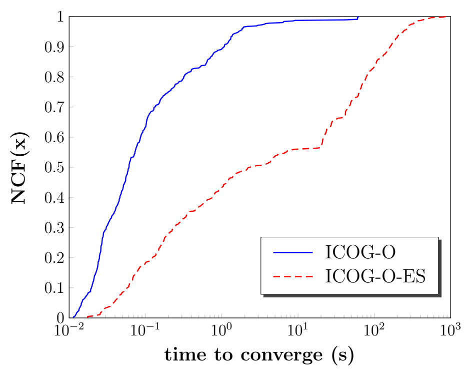
TABLE I: Comparative analysis between ICOG-O and ICOG-O-ES

s	**a**	**TLE**	$\mathbf{SS_1}$	$\mathbf{US_1}$	**n**	**P**	**T**	$\mathbf{SS_2}$	$\mathbf{US_2}$	**time (s)**		**iterations**		$\frac{\textbf{time}}{\textbf{iteration}}$
										μ	σ	μ	σ	μ
6	6	0.00	4,202	969	16	35.62	29.00	1,441	200	0.10	0.21	5.25	7.22	0.02
		0.11	*1,441*	*200*						*19.89*	*52.63*	*6.38*	*7.10*	*3.12*
6	7	0.05	3,341	1,017	27	41.46	34.69	1,612	251	0.19	0.40	8.00	8.98	0.02
		0.31	*1,553*	*242*						*41.50*	*78.72*	*11.31*	*11.96*	*3.67*
6	8	0.14	4,244	958	30	42.93	35.48	2,293	276	0.23	0.50	8.07	10.18	0.03
		0.32	*2,216*	*267*						*30.70*	*51.92*	*10.17*	*12.01*	*3.02*
7	6	0.05	3,146	449	19	38.95	31.89	2,077	230	0.08	0.16	4.84	6.67	0.02
		0.10	*2,077*	*230*						*17.48*	*49.09*	5.79	6.64	3.02
7	7	0.07	7,831	3,030	29	46.79	39.69	3,818	697	0.17	0.33	7.83	8.30	0.02
		0.34	*3,818*	*697*						*70.04*	*160.09*	*13.35*	*15.65*	*5.25*
7	8	0.13	8,969	2,833	36	46.89	39.22	3,746	481	1.89	10.00	8.50	10.08	0.22
		0.35	*3,746*	*481*						*32.75*	*51.58*	*11.22*	*11.67*	*2.92*
8	6	0.00	8,750	1,280	21	43.19	35.76	5,716	483	0.08	0.16	5.14	6.51	0.02
		0.16	*5,716*	*483*						*19.21*	*54.44*	5.95	6.22	3.23
8	7	0.06	12,375	4,484	35	48.14	40.54	5,340	855	0.22	0.40	8.66	9.40	0.03
		0.31	*5,340*	*855*						*46.19*	*69.40*	*12.86*	*12.46*	*3.59*
8	8	0.24	10,413	1,384	37	49.23	41.57	5,731	612	1.95	10.15	8.49	9.71	0.23
		0.40	*5,421*	*579*						*38.13*	*59.51*	*11.86*	*11.60*	*3.22*
8	9	0.17	17,558	4,755	27	48.96	41.00	5,101	789	2.62	12.30	9.35	10.57	0.28
		0.59	*4,912*	*760*						*58.55*	*108.88*	*13.96*	*14.91*	*4.19*
8	10	0.18	12,261	4,155	30	55.90	47.66	8,890	1,895	0.66	1.33	14.52	15.48	0.05
		0.58	*8,594*	*1,832*						*79.34*	*126.98*	*19.62*	*21.32*	*4.04*
9	8	0.14	20,871	5,841	41	54.02	46.05	11,062	1,472	1.85	9.36	11.61	12.87	0.16
		0.41	*11,062*	*1,472*						*70.60*	*141.38*	*16.07*	*16.25*	*4.39*
9	9	0.22	21,314	4,481	30	52.83	44.45	8,791	1,049	1.60	7.66	8.76	9.28	0.18
		0.61	*8,498*	*1,014*						*35.63*	*79.70*	*12.07*	*12.41*	*2.95*
9	10	0.23	19,039	5,091	33	58.67	50.24	10,597	1,763	1.06	2.03	17.70	19.74	0.06
		0.58	*10,597*	*1,763*						*104.72*	*143.20*	*22.64*	*23.43*	*4.63*
10	8	0.15	31,562	6,733	47	55.63	47.28	17,848	1,970	1.51	8.85	8.50	9.03	0.18
		0.40	*17,469*	*1,929*						*42.80*	*60.19*	*14.33*	*12.36*	*2.99*
10	9	0.22	39,690	9,206	37	56.27	47.95	14,721	1,761	0.28	0.45	9.59	9.63	0.03
		0.56	*14,721*	*1,761*						*46.00*	*71.70*	*14.19*	*12.72*	*3.24*
10	10	0.21	34,488	9,676	31	60.03	51.57	14,439	1,319	0.82	1.92	14.27	18.96	0.06
		0.64	*13,973*	*1,277*						*80.94*	*134.43*	*18.37*	*22.05*	*4.41*

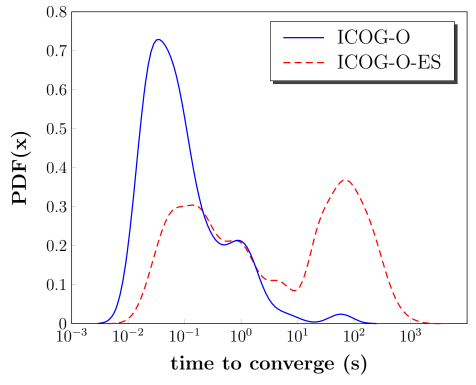
Frequency (NCF, a.k.a. the empirical cumulative distribution function). The x-axis is the TTC (in seconds), and the y-axis is the NCF, which is the cumulative number of samples normalized by the sample size. A point (x, y) on the graph means that a fraction of y samples have a TTC that is less than x seconds. From Figure 8(a), we observe that using ICOG-O, 64% of the samples can be completed within 0.1 second, while using ICOG-O-ES, 18% of the samples can be completed within 0.1 second. Moreover, using ICOG-O, 89% of the samples can be completed within 1 second, while using ICOG-O-ES, 43% of the samples can be completed within 1 second. Figure 8(b) is the empirical probability distribution function obtained by kernel density estimation. The x-axis is the TTC, and the y-axis is the probability. We see that using ICOG-O, the majority of the samples can be completed between 0.01 second and 0.1 second, while using ICOG-O-ES, the majority of the sample completion times span a wider range from 0.1 second to 100 seconds.

Figure 8(c) is the NCF graph for the difference of the number of iterations of ICOG-O-ES and ICOG-O. The x-axis is the extra number of iterations required by ICOG-O-ES as compared to ICOG-O. The y-axis is the NCF. Note that for *all* the samples we tested, ICOG-O always requires fewer or equal number of iterations than ICOG-O-ES; and correspondingly, ICOG-O always synthesizes fewer or equal number of monitor places than ICOG-O-ES. From Figure 8(c), we see that ICOG-O requires fewer iterations (and synthesizes fewer monitor places) than ICOG-O-ES for 43% of the samples.

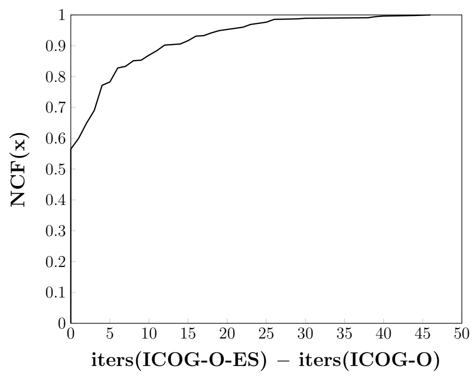
Table I presents a summary of the experimental results of the comparative analysis between the performance of ICOG-O and that of ICOG-O-ES. For each row of the table, the sub-row with italics corresponds to the performance of ICOG-O-ES, and the counterpart without italics corresponds to the performance of ICOG-O. The first two columns correspond to the parameters used to generate the random sample Gadara nets. The first (s) and second (a) columns are the number of process subnets and the number of resource acquisitions per subnet. In generating the random nets, the number of resources (locks) in the original Gadara net is set to be 11, the probability of acquiring a new resource before releasing one already held is 0.2, and the branching probability is 0.1. The third column (TLE) shows the ratio of sample nets that timed out in any iteration of ICOG-O and ICOG-O-ES. The fourth (SS_1) and fifth (US_1) columns describe the state space complexity. The sub-row without italics (resp., with italics) shows the average number of safe and unsafe states that are reachable by the



(a)



(b)



(c)

Fig. 8: (a) TTC of ICOG-O and ICOG-O-ES: Normalized cumulative frequency; (b) TTC of ICOG-O and ICOG-O-ES: Estimated probability density function; (c) Difference of the number of iterations of ICOG-O-ES and ICOG-O

original nets, where ICOG-O (resp., ICOG-O-ES) did not ever time out. *Note that ICOG-O and ICOG-O-ES do not construct the state space, since they exploit structural properties of Gadara nets; these numbers were generated separately for the sake of scalability assessment.* The sixth column (n) is the number of generated Gadara nets, where both ICOG-O and ICOG-O-ES did not ever time out throughout the iterations. The seventh (P) and eighth (T) columns correspond to the average number of places and transitions in the original Gadara nets. The ninth (SS_2) and tenth (US_2) columns shows the average number of safe and unsafe states that are reachable by the original nets, where both ICOG-O and ICOG-O-ES did not ever time out. The eleventh column (time (s)) shows the average and standard deviation of the time (in seconds) the entire ICOG-O and ICOG-O-ES processes took until they converged. The twelfth column (iterations) shows the average and standard deviation of the number of iterations for ICOG-O and ICOG-O-ES to converge. Since for any sample net, the number of synthesized monitor places is always 1 less than the number of total iterations, we have not included the number of monitor places in the table. The last column (time/iteration) is the average time per iteration of ICOG-O and ICOG-O-ES.

In the experiments, we observed that the majority of time spent by ICOG-O or ICOG-O-ES is on the stage of RI empty siphon detection. This is precisely why we developed a customized MIP formulation for RI empty siphon detection in Gadara nets. Compared to the baseline performance of ICOG-O-ES, the data above show the efficiency attained by ICOG-O – the improvement in average time ranges from 17 to 404 times faster. In addition, the average number of iterations of ICOG-O is smaller than that of ICOG-O-ES for all the cases. From the second to fourth columns, we see that ICOG-O timed out on much fewer nets; and, on average, ICOG-O is able to handle much larger nets than ICOG-O-ES.

TABLE II: Scalability study of ICOG-O

SS	US	time (s)	iters
786,430	487,990	46.05	102
727,240	295,290	2.17	48
532,630	233,800	46.05	61
373,700	136,260	18.45	91
354,270	64,488	25.92	29
336,250	200,370	8.35	83
320,180	118,470	18.35	91
290,970	50,002	3.54	51
285,700	64,386	0.34	13
271,780	64,488	46.21	29
247,920	84,502	26.41	57
226,330	28,242	0.50	20
176,960	26,788	0.42	13
176,920	22,392	0.44	20

C. Scalability study of ICOG-O

Table II presents a sample of experimental results that highlight the scalability of ICOG-O. The first (SS) and second (US) columns are the number of safe and unsafe states. (Again, ICOG-O does not expand these states; these numbers were generated separately.) The third column (time (s)) is the total time (in seconds) for ICOG-O to converge. The fourth column (iters) is the number of iterations until convergence. We set a time-out threshold of 6000 seconds for these experiments. Table II shows that ICOG-O is very scalable even on a modest computer set up.

VII. Discussion

In the analysis of multithreaded programs, our approach fully exploits the structural properties of the proposed Petri net models, without explicitly constructing the reachability space of the programs [19]. Our choice of Petri nets is also supported by the implementation of control logic. The overhead of controlling software can be generally attributed to two aspects: (i) control logic runtime decisions, and (ii) transitions blocked as a result of the control decisions. With an automaton model, the control decision is based on the global state of the program. In contrast, the control logic in a Petri net model is expressed as a set of decentralized monitor places, which only locally intervene the critical regions that are involved in the potential deadlocks, thus avoiding a global bottleneck for control decisions. A synthesized monitor place is essentially a generalized resource place, whose outgoing arc effectively delays the target lock acquisition action that will otherwise lead to a CMW-deadlock [32]. The similarity between the monitor places and resource places (that model locks) implies that the synthesized control logic can be implemented with primitives supplied by standard multithreading libraries, e.g., `libpthread`. The framework of Gadara nets enables the synthesis of correct and maximally permissive control logic that provably prevents all the potential CMW-deadlocks in the program and will delay a lock acquisition only when necessary [18]. The customized methodology developed in the current paper further guarantees that no redundant control logic is synthesized throughout the iteration process (Theorem 3).

VIII. Conclusion

We proposed an iterative control synthesis methodology for ordinary Gadara nets, called ICOG-O, based on structural analysis in terms of siphons. The control logic synthesized by ICOG-O enforces liveness in Gadara nets and provably eliminates all the potential CMW-deadlocks in the corresponding multithreaded programs. ICOG-O customizes the general control synthesis algorithm, ICOG, presented in [18], from which the properties of correctness and maximal permissiveness are preserved. In addition, we formally established a set of important properties of the proposed methodology, and showed that ICOG-O never synthesizes redundant control logic. Compared to the general ICOG and UCCOR [18], the customized ICOG-O and UCCOR-O presented in this paper focus on ordinary Gadara nets, and thus enable us to implement control synthesis based on a type of empty siphons. The customization permits a conceptually simpler process for the control of ordinary Gadara nets. It also simplifies some steps in the general ICOG and UCCOR. Due to Theorem 3, in the control of ordinary Gadara nets, ICOG-O requires a fewer number of iterations than ICOG in general, and ICOG-O never synthesizes redundant control logic even without the bookkeeping of prevented states (that is required in ICOG). Our experimental results showed that ICOG-O is very efficient in terms of time and the number of synthesized monitor places. The results also demonstrated the scalability of our approach to large-scale real-world software. From a more general perspective, the results in the Gadara project illustrate that software failure avoidance is a fertile application area for discrete-event control, and moreover, special features from this application area are motivating further theoretical developments on the control of discrete-event systems.

Appendix

Definition 10: A Petri net dynamic system $\mathcal{N} = (P, T, A, W, M_0)$ is a bipartite graph (P, T, A, W) with an initial number of tokens. Specifically, $P = \{p_1, p_2, ..., p_n\}$ is the set of places, $T = \{t_1, t_2, ..., t_m\}$ is the set of transitions, $A \subseteq (P \times T) \cup (T \times P)$ is the set of arcs, $W : A \to \{0, 1, 2, ...\}$ is the arc weight function, and for each $p \in P$, $M_0(p)$ is the initial number of tokens in p.

The *marking* of a Petri net $\mathcal{N}$ is a column vector M of n entries corresponding to the n places. M_0 is the initial marking. We use $M(p)$ to denote the (partial) marking on a place p, which is a scalar. The notation $\bullet p$ denotes the set of input transitions of place p: $\bullet p = \{t|(t, p) \in A\}$. Similarly, $p\bullet$ denotes the set of output transitions of p. The sets of input and output places of transition t are similarly defined by $\bullet t$ and $t\bullet$. This notation is extended to sets of places or transitions in a natural way. A transition t is *enabled* or *fireable* at a marking M, if $\forall p \in \bullet t$, $M(p) \geq W(p, t)$. A pair (p, t) is called a *self-loop* if p is both an input and output place of t. We consider only *self-loop-free* Petri nets in this paper. Our Petri net models of multithreaded programs have unit arc weights. Such Petri nets are called *ordinary*. However, addition of monitor places may render them *non-ordinary*. The *incidence matrix* D of a Petri net is an integer matrix $D \in \mathbb{Z}^{n \times m}$, where $D_{ij} = W(t_j, p_i) - W(p_i, t_j)$ represents the net change in the number of tokens in place p_i when transition t_j fires. A *state machine* is an ordinary Petri net such that each transition t has exactly one input place and exactly one output place, i.e., $\forall t \in T, |\bullet t| = |t \bullet| = 1$.

Let D be the incidence matrix of a Petri net $\mathcal{N}$. Any non-zero integer vector y such that $D^T y = 0$, is called a *P-invariant* of $\mathcal{N}$. Further, P-invariant y is called a *P-semiflow* if all the elements of y are non-negative. By definition, P-semiflow is a special case of P-invariant. A straightforward property of P-invariants is given by the following well known result [22]: If a vector y is a P-invariant of Petri net $\mathcal{N} = (P, T, A, M_0)$, then we have $M^T y = M_0^T y$ for any reachable marking $M \in R(\mathcal{N}, M_0)$. The *support* of P-semiflow y, denoted as $\|y\|$, is defined to be the set of places that correspond to nonzero entries in y. A support $\| y \|$ is said to be *minimal* if there does not exist another nonempty support $\|y'\|$, for some other P-semiflow y', such that $\| y' \| \subset \| y \|$. A P-semiflow y is said to be *minimal* if there does not exist another P-semiflow y' such that $y'(p) \leq y(p)$, $\forall p$. For a given minimal support of a P-semiflow, there exists a unique minimal P-semiflow, which we call the *minimal-support P-semiflow* [22].

Supervision Based on Place Invariants (SBPI) [13] provides an efficient algebraic technique for control logic synthesis by introducing a monitor place, which essentially enforces a P-invariant so as to achieve a given linear inequality constraint of the form: $l^T M \leq b$, where M is the marking vector of the net under control, l is a weight (column) vector, and b is a scalar. All entries of l and b are integers. The main result of SBPI is as follows.

Theorem 4: [13], [21] Consider a Petri net $\mathcal{N}$, with incidence matrix D and initial marking M_0. If M_0 satisfies $b - l^T M_0 \geq 0$, then a monitor place, p_c, with incidence matrix $D_{p_c} = -l^T D$, and initial marking $M_0(p_c) = b - l^T M_0$, enforces the constraint $l^T M \leq b$ when included in the closed-loop system. This supervision is maximally permissive, i.e., a transition in the net is disabled by the monitor place only if its firing leads to a marking where the given linear constraint $l^T M \leq b$ is violated.

References

[1] The Gadara Project. http://gadara.eecs.umich.edu/.
[2] Gurobi Optimizer. http://www.gurobi.com/.

[3] A. Auer, J. Dingel, and K. Rudie. Concurrency control generation for dynamic threads using discrete-event systems. In *Proc. Allerton Conference on Communication, Control and Computing*, 2009.

[4] F. Chu and X.-L. Xie. Deadlock analysis of Petri nets using siphons and mathematical programming. *IEEE Transactions on Robotics and Automation*, 13(6):793–804, December 1997.

[5] G. Delaval, H. Marchand, and E. Rutten. Contracts for modular discrete controller synthesis. In *Proc. ACM Conference on Languages, Compilers and Tools for Embedded Systems*, 2010.

[6] E. W. Dijkstra. *Selected Writings on Computing*, chapter The Mathematics Behind the Banker's Algorithm, pages 308–312. Springer-Verlag, 1982.

[7] C. Dragert, J. Dingel, and K. Rudie. Generation of concurrency control code using discrete-event systems theory. In *Proc. ACM International Symposium on Foundations of Software Engineering*, 2008.

[8] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proc. the 19th ACM Symposium on Operating Systems Principles*, 2003.

[9] A. Gamatie, H. Yu, G. Delaval, and E. Rutten. A case study on controller synthesis for data-intensive embedded system. In *Proc. International Conference on Embedded Software and Systems*, 2009.

[10] A. Giua. *Petri nets as discrete event models for supervisory control*. PhD thesis, Rensselaer Polytechnic Institute, 1992.

[11] A. Giua, F. DiCesare, and M. Silva. Generalized mutual exclusion constraints on nets with uncontrollable transitions. In *Proc. 1992 IEEE International Conference on Systems, Man, and Cybernetics*, pages 974–979, 1992.

[12] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. Wiley, 2004.

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

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

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

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

[17] H. Liao. *Modeling, Analysis, and Control of a Class of Resource Allocation Systems Arising in Concurrent Software*. PhD thesis, University of Michigan, Ann Arbor, 2012.

[18] H. Liao, S. Lafortune, S. Reveliotis, Y. Wang, and S. Mahlke. Optimal liveness-enforcing control for a class of Petri nets arising in multithreaded software. *IEEE Transactions on Automatic Control*, 2013 (in print).

[19] H. Liao, Y. Wang, H. K. Cho, J. Stanley, T. Kelly, S. Lafortune, S. Mahlke, and S. Reveliotis. Concurrency bugs in multithreaded software: Modeling and analysis using Petri nets. *Journal of Discrete Event Dynamic Systems*, 2012 (in print).

[20] C. Liu, A. Kondratyev, Y. Watanabe, J. Desel, and A. Sangiovanni-Vincentelli. Schedulability analysis of Petri nets based on structural properties. In *Proc. International Conference on Application of Concurrency to System Design*, 2006.

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

[22] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[23] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Proc. the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[24] Y. Nir-Buchbinder, R. Tzoref, and S. Ur. Deadlocks: From exhibiting to healing. In *Proc. Workshop on Runtime Verification*, 2008.

[25] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically correcting memory errors with high probability. *Communications of the ACM*, 51(12):87–95, December 2008.

[26] S. Park, S. Lu, and Y. Zhou. Ctrigger: Exposing atomicity violation bugs from their hiding places. In *Proc. 14th International Conference on Architecture Support for Programming Languages and Operating Systems*, 2009.

[27] V. V. Phoha, A. U. Nadgar, A. Ray, and S. Phoha. Supervisory control of software systems. *IEEE Transactions on Computers*, 53(9):1187–1199, September 2004.

[28] S. A. Reveliotis. *Real-Time Management of Resource Allocation Systems: A Discrete-Event Systems Approach*. Springer, New York, NY, 2005.

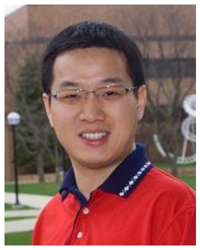
[29] C. Wallace, P. Jensen, and N. Soparkar. Supervisory control of workflow scheduling. In *Proc. International Workshop on Advanced Transaction Models and Architectures*, 1996.

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

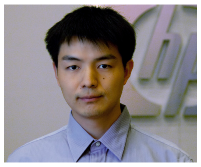
[31] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. A. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proc. the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 281–294, 2008.

[32] Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. Mahlke. The theory of deadlock avoidance via discrete control. In *Proc. the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 252–263, 2009.

[33] K. Yamalidou, J. Moody, M. Lemmon, and P. Antsaklis. Feedback control of Petri nets based on place invariants. *Automatica*, 32(1):15–28, January 1996.



Hongwei Liao received the Ph.D. degree (2012) in Electrical Engineering-Systems from the University of Michigan, Ann Arbor, where he also received the M.Sc. degree (2009) in Electrical Engineering-Systems, and the M.S.E. degree (2011) in Industrial and Operations Engineering. He received his B.Eng. degree (2007) in Electrical Engineering and Dual B.Mgt. degree (2007) in Business Administration with Honors from Shanghai Jiao Tong University, Shanghai, China. His research interests include discrete event systems, operations research, and wireless communications. Dr. Liao joined the Operations Research Group at US Airways, Tempe, Arizona, in 2012, where he is an Analyst and Technologies Lead. He was an intern at General Electric Global Research, Niskayuna, New York, in summer 2010, and an intern at General Motors Global Research & Development, Warren, Michigan, in summer 2011. He has been the recipient of a number of awards, including the Rackham Predoctoral Fellowship Award (2011) and the College of Engineering Distinguished Achievement Award (2011) from the University of Michigan, Ann Arbor. He is a member of Phi Kappa Phi and Tau Beta Pi.



Yin Wang received Bachelor's (2000) and Master's (2003) degrees from the Shanghai Jiao Tong University, Department of Automation. He earned his Ph.D. at the University of Michigan Electrical Engineering and Computer Science department and joined Hewlett-Packard Laboratories in early 2009. While a graduate student, Wang interned at Microsoft Shanghai, IBM Almaden Research Center, and HP Labs.

Jason Stanley received the B.Eng. degree in Computer Science from the University of Michigan, Ann Arbor, in 2012. As an undergraduate, he worked as a research assistant in the Electrical Engineering and Computer Science department. After graduation, he began working at Citadel Investment Group. His research interests lie in computer science theory and machine learning.

Stéphane Lafortune received the B. Eng degree from Ecole Polytechnique de Montréal in 1980, the M. Eng. degree from McGill University in 1982, and the Ph.D. degree from the University of California at Berkeley in 1986, all in electrical engineering. Since September 1986, he has been with the University of Michigan, Ann Arbor, where he is a Professor of Electrical Engineering and Computer Science. Dr. Lafortune is a Fellow of the IEEE (1999). He received the Presidential Young Investigator Award from the National Science Foundation in 1990 and the George S. Axelby Outstanding Paper Award from the Control Systems Society of the IEEE in 1994 (for a paper co-authored with S. L. Chung and F. Lin) and in 2001 (for a paper co-authored with G. Barrett). Dr. Lafortune's research interests are in discrete event systems and include multiple problem domains: modeling, diagnosis, control, optimization, and applications to computer systems. He is the lead developer of the software package UMDES and co-developer of DESUMA with L. Ricker. He co-authored, with C. Cassandras, the textbook *Introduction to Discrete Event Systems - Second Edition* (Springer, 2008). Dr. Lafortune is a member of the editorial boards of the Journal of Discrete Event Dynamic Systems: Theory and Applications and of the International Journal of Control.

Spyros Reveliotis is a Professor in the School of Industrial & Systems Engineering, at the Georgia Institute of Technology. He holds a Diploma in Electrical Engineering from the National Technical University of Athens, Greece, an M.Sc. degree in Computer Systems Engineering from Northeastern University, Boston, and a Ph.D. degree in Industrial Engineering from the University of Illinois at Urbana-Champaign. Dr. Reveliotis' research interests are in the area of Discrete Event Systems theory and its applications. He is a Senior member of IEEE and a member of INFORMS.

Dr. Reveliotis currently serves as associate editor for the IEEE Trans. on Automatic Control and as department editor for IIE Transactions. He has also been an associate editor for IEEE Trans. on Robotics and Automation and for IEEE Trans. on Automation Science and Engineering, and a senior editor in the Conference Editorial Board for the IEEE Intl. Conference on Robotics & Automation. In 2009 he was the Program Chair for the IEEE Conference on Automation Science and Engineering, and currently he serves as a member of the Steering Commitee for this conference. Dr. Reveliotis has been the recipient of a number of awards, including the 1998 IEEE Intl. Conf. on Robotics & Automation Kayamori Best Paper Award.

Terence Kelly is a senior researcher in the Intelligent Infrastructure Lab at Hewlett-Packard Laboratories. His research applies discrete control theory to failure avoidance/elimination in computing systems. Kelly received a Ph.D. in computer science from the University of Michigan. He is a senior member of the IEEE and the ACM.

Scott Mahlke is a Professor in the Electrical Engineering and Computer Science Department at the University of Michigan where he leads the Compilers Creating Custom Processors group (http://cccp.eecs.umich.edu). The CCCP group delivers technologies in the areas of compilers for multicore processors, application-specific processors for mobile computing, and reliable system design. Mahlke received the Ph.D. degree in Electrical Engineering from the University of Illinois at Urbana-Champaign in 1997. Mahlke's achievements were recognized by being named the Morris Wellman Assistant Professor in 2004 and being awarded the Most Influential Paper Award from the Intl. Symposium on Computer Architecture in 2007. He is a member of the IEEE Computer Society and the ACM.