*Proceedings of the 2011 Winter Simulation Conference*
*S. Jain, R. R. Creasey, J. Himmelspach, K. P. White, and M. Fu, eds.*

# SIMULATION ANALYSIS OF MULTITHREADED PROGRAMS UNDER DEADLOCK-AVOIDANCE CONTROL

Hongwei Liao
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor
MI 48109, USA

Hao Zhou
Department of Industrial and Operations Engineering
University of Michigan, Ann Arbor
MI 48109, USA

Stéphane Lafortune

Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor
MI 48109, USA

## ABSTRACT

We employ discrete event simulation to evaluate the performance of deadlock-prone multithreaded programs, either general-purpose software or parallel simulators, under a novel technique for deadlock-avoidance control recently proposed in the literature. The programs are modeled by a special class of Petri nets, called Gadara nets. We propose a formal simulation methodology for Gadara nets. We then use simulation to analyze two deadlock-prone multithreaded programs, where we study system performance in terms of safety, efficiency, and activity level, both before and after deadlock-avoidance control is applied. We further conduct a sensitivity analysis to investigate the effect of key parameters on the program's performance. We discuss the implications of the above results on the practical implementation of control strategies that prevent deadlocks in multithreaded programs.

## 1 INTRODUCTION

The past decade has witnessed a fundamental revolution in the computer industry. Uniprocessors in computers have now been gradually supplanted by multicore processors. In order to exploit the full capability of multicore architectures, performance-conscious developers are spending significant efforts to parallelize applications. This trend has also been witnessed in the field of simulation, where parallel simulation techniques are being deployed to enhance computational capacity and efficiency (Heidelberger and Nicol 1996, Mutschler 2006, Zuberek 2009). With the increasing complexity of simulation models driven by the demand for higher accuracy, and the prevalence of multicore architectures in computer hardware, computer programmers and simulation practitioners will be expected to become adept at parallel programming in the near future. Parallel programming, however, is very hard, as reasoning about concurrency is extremely challenging for human programmers. We are facing a problem: multicore architectures are making parallel programming unavoidable but concurrency bugs are making it costly and error-prone. Ensuring failure-free execution of concurrent programs is a notoriously difficult problem, but an increasingly important one.

978-1-4577-2109-0/11/$26.00 ©2011 IEEE 

An important class of failure in concurrent software, including parallel simulations, is *circular-wait deadlock*, where a set of tasks are waiting for one another indefinitely and no progress can be made. The deadlock problem incurred by event message processing in parallel discrete event simulation has been investigated previously, and various conservative solutions were proposed; see, e.g, (Fujimoto 1990). Conservative algorithms were further evaluated and compared in (Bagrodia and Takai 2000). A circular-wait deadlock can also be induced by lack of resources due to contention among concurrent tasks. For instance, let us consider two tasks, 1 and 2, where both require *exclusively* holding resources A and B to complete their jobs. It is easy for situations to arise in which, e.g., task 1 has acquired A and needs B, while task 2 holds B but requires A; these tasks are deadlocked and neither can perform useful work. (Krishnamurthi et al. 1994) presented a deadlock detection algorithm in simulation models by using a linked list structure of resources and tasks. Various algorithms have been reported for deadlock analysis using graph theoretic models; see, e.g., (Woodward and Mackulak 1997, Venkatesh et al. 1998, Cheung et al. 2009).

We are interested in multithreaded programs that use shared data, a common paradigm for general-purpose concurrent software. This programming technique has also been employed in multithreaded simulators (Hsu, Pino, and Bhattacharyya 2008, Mutschler 2006, Zuberek 2009). In this programming paradigm, programmers use lock primitives, such as mutual exclusion locks (or "mutexes"), to protect shared data. Inappropriate use of locks can lead to the circular-mutex-wait (CMW) deadlock problem described above. It is useful to delineate the two levels at which CMW deadlocks may arise in the context of this paper: *the program source code* itself, and *the system being simulated* in the case of simulation analysis. Our investigations pertain to deadlocks that may arise at either of these two levels. At the *program* level, the aforementioned tasks are the concurrent threads that are executing, and the resources are sections of shared data in memory. In this case, mutexes prevent threads from accessing the same data concurrently, thus allowing threads to update the shared data in an orderly manner. Misuse of mutexes by programmers can lead to CMW deadlocks. Case Study 1, presented in Section 4.1, addresses this type of program-level deadlock. At the *system simulation* level, the aforementioned tasks and resources can model various entities in different contexts. For example, in flexible manufacturing system simulation, the tasks can be parallel assembly lines, and the resources can be machines processing the parts of a product; in healthcare system simulation, the tasks can be concurrent patient flows, and the resources can be physicians, nurses, or medical equipment. In multithreaded simulations, these shared resources are protected via mutexes by the tasks that first acquire them. If the system design is such that deadlocks due to shared resources can arise, and no proper efforts are made to avoid them, then these deadlocks can potentially manifest themselves as CMW deadlocks when the system is being simulated. Case Study 2, presented in Section 4.2, addresses the deadlock problem at this level. A similar analytical approach has also been employed to study software contention in computer systems (Roy et al. 2008).

In our ongoing Gadara project, we employ Petri nets to systematically model, analyze, and control multithreaded programs with lock allocation and release operations (Wang et al. 2008, Liao et al. 2010). The architecture of the Gadara methodology is shown in Figure 1, and involves the novel paradigm of *controlling* the execution of the multithreaded program by instrumenting it with *optimal* (i.e., maximally permissive) control logic that is synthesized using recent results from the control theory of discrete event systems (Cassandras and Lafortune 2008). Petri nets are a commonly used modeling formalism for discrete event systems; a Petri net model captures the concurrency of the dynamics of a multithreaded program while avoiding enumerating its state space. The Gadara methodology works for general-purpose multithreaded software; in particular, it can be applied to multithreaded simulators. In this latter context, the deadlock-avoidance control logic synthesized in the Gadara methodology applies to

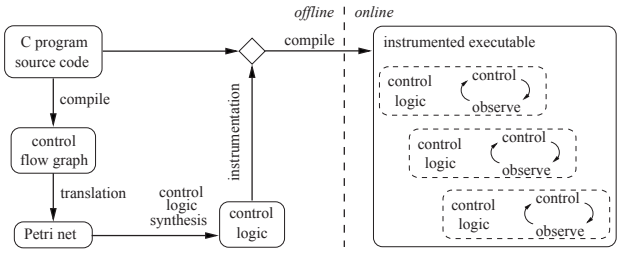



Figure 1: Gadara architecture

CMW deadlocks at both the aforementioned program and system levels. In the Gadara project, we have used Petri nets to model, analyze, and control several general-purpose large software programs, such as OpenLDAP, BIND, and Apache (Wang et al. 2008). We will study a nontrivial deadlock example from OpenLDAP in Section 4.1.

In the context of discrete event simulation, in contrast to (Mueller, Alexopoulos, and McGinnis 2007) where Petri nets have been used in the simulation modeling phase to prevent deadlocks, we focus on existing simulation models and their associated multithreaded programs, which may be deadlock-prone. Also, in contrast to (Woodward and Mackulak 1997, Venkatesh et al. 1998), where graph theoretic models are used for deadlock detection and resolution, we not only detect deadlocks via formal methods in Petri nets, but also synthesize *maximally permissive* control logic to *provably* prevent potential CMW deadlocks. Here, maximal permissiveness means that the control logic will restrict concurrency only when necessary to eliminate deadlock. The development of this control technique has been reported in (Liao et al. 2010). Our prior works have focused on the class of *untimed* Petri nets and mainly studied *logical* level properties, e.g., deadlock-freeness. In this paper, we extend our models to the class of *stochastic timed* Petri nets and use them to investigate the *quantitative* performance of programs, before and after control, via discrete event simulation. Our simulation analysis principally investigates: the impact of the synthesized control logic on program's performance; the effect of key parameters on the program before and after control; and the tradeoff between aggressive and conservative deadlock-avoidance control strategies. More importantly, our study provides a general simulation-based methodology to evaluate the performance of multithreaded programs, when deadlock-avoidance control logic is applied. Therefore, our contributions are both in terms of modeling methodology and analysis methodology for the problem under consideration.

Our main contributions are as follows. (i) We propose a simulation methodology for a special class of stochastic timed Petri nets, which model multithreaded programs (including multithreaded simulators) with lock allocation and release operations, and provide a systematic framework for the synthesis of optimal deadlock-avoidance control logic. (ii) We conduct simulation analysis on the Petri net models of programs before and after deadlock-avoidance control, and study the performance metrics related to safety, efficiency, and activity level via output data analysis. (iii) We further conduct sensitivity analysis on these Petri net models, and investigate the effect of key parameters on the programs and the implication for deadlock-avoidance control. The paper is organized as follows. We introduce relevant background in Section 2. The proposed discrete event simulation model is described in Section 3. We present the simulation results and analysis of two cases of deadlock-prone programs in Section 4, and conclude in Section 5.

## 2 PRELIMINARIES

We briefly overview the Gadara project and then introduce background material and relevant definitions.

### 2.1 Overview of the Gadara Project

As shown in Figure 1, the Gadara methodology contains four stages. (i) The program source code is compiled into a graphical model, the control flow graph (CFG), that captures execution paths. (ii) The CFG is translated into a Petri net model of the whole program, called Gadara net (to be introduced shortly), which maps potential deadlocks in the program to a structural feature called *resource-induced deadly-marked siphons* in the net. Therefore, potential deadlocks can be automatically detected and prevented, by searching for these structural features in the net (a problem previously investigated), and seeking their elimination by control at the next stage. (iii) Maximally permissive control logic is synthesized for the obtained Gadara net (Liao et al. 2010), so that the aforementioned structural features will *not* be reachable in the controlled Gadara net. (iv) The control logic is used to instrument the source code and manage lock allocation and release at run-time to avoid deadlocks.

### 2.2 Standard Definitions of Petri Nets

Due to space limitations, we assume readers are familiar with standard Petri net definitions, as in (Cassandras and Lafortune 2008). We only present the definitions that are relevant to our discussion.

**Definition 1** 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* (a.k.a. the *state*) of a Petri net $\mathcal{N}$ is a column vector $M$ of $n$ entries corresponding to the $n$ places. As defined above, $M_0$ is the initial marking. We use $M(p)$ to denote the number of tokens in place $p$ under marking $M$. A transition $t$ is *enabled* or *fireable* at $M$, if for any input place $p$ of $t$, $M(p) \geq W(p, t)$. Based on Definition 1, we can further define a stochastic timed-place Petri net as follows.

**Definition 2** A stochastic timed-place Petri net is a six-tuple $\mathcal{N} = (P, T, A, W, M_0, \mathbf{V})$, where $\mathcal{N} = (P, T, A, W, M_0)$ is a Petri net, and, $\mathbf{V} : P \to \mathbb{R}^+$ is a timing structure that associates places with stochastic time delays.

### 2.3 The Gadara Net Model

In (Wang et al. 2009), we formally define a special class of Petri nets, called Gadara nets, to systematically model multithreaded programs with lock allocation and release operations. We use $\mathcal{N}_G$ to denote a Gadara net. See (Wang et al. 2009) for the detailed definition and analysis of $\mathcal{N}_G$. We discuss its key features using the example in Figure 2. $\mathcal{N}_G$ in Figure 2 contains two *process subnets*: $\mathcal{N}_1$ and $\mathcal{N}_2$. Each process subnet models a work process of the program. In $\mathcal{N}_G$, there are three types of places: (i) The set of *idle places*, denoted as $P_0$ and shown in purple; these places contain tokens that represent "idle" threads waiting for future execution. At the initial state, $M_0(p) > 0, \forall p \in P_0$. (ii) The set of *operation places*, denoted as $P_S$ and shown in black; these places model the main body of the program. Each operation place represents a set of lines of code executed by a thread. A token in an operation place represents a thread executing these lines of code. At the initial state, $M_0(p) = 0, \forall p \in P_S$. (iii) The set of *resource places*, denoted as $P_R$ and shown in blue, red, and green, model mutexes in the program. A token in a resource place represents the availability of the lock. At the initial state, $M_0(p) = 1, \forall p \in P_R$. Any transition in $\mathcal{N}_G$ either models a lock allocation or release operation, or models a branch selection (e.g., `if/else` as shown in Figure 4) in the program. Further, if all the resource places were removed in $\mathcal{N}_G$, then any transition will only have one input place and one output place, which is due to the nature of the program we model.

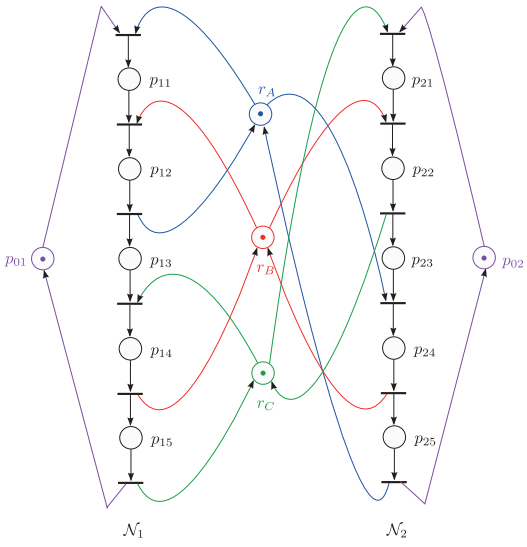



Figure 2: A Gadara net model of two threads sharing three resources

In order to prevent deadlock states from being reachable, we synthesize *monitor places* (a.k.a. *control places*) to augment the original Gadara net, and obtain the *controlled Gadara net* (Liao et al. 2010). The set of monitor places is denoted as $P_C$. In essence, monitor places are generalized resource places. The details about control synthesis are beyond the scope of this paper; see (Liao et al. 2010) for further discussion.

### 2.4 Stochastic Timed Gadara Net Model for Simulation Analysis

From Section 2.3, we know that the tokens in $P_0 \cup P_S$ represent the threads and drive the dynamics of the net, while the tokens in $P_R \cup P_C$ represent the availability of (generalized) resources. This implies that for the purpose of simulation of $\mathcal{N}_G$, we *only* need to consider delay times associated with idle or operation places, which represent thread waiting or execution times. More specifically, based on Definition 2, in a *stochastic timed-place Gadara net*, the timing structure is: $\mathbf{V} : P_0 \cup P_S \to \mathbb{R}^+$.

In general, timed Petri nets can have delays associated with transitions or places, which results in *timed-transition Petri nets* or *timed-place Petri nets*, respectively. From a theoretical viewpoint, these two subclasses of nets are expressively equivalent (Sifakis 1980). In timed-transition Petri net models of physical systems, transitions usually represent actions in the system that take time to complete, while places and tokens usually represent the pre-conditions of the execution of the actions, which are not directly related to time. In contrast, in the proposed timed-place Gadara net models of multithreaded programs, we only need to assign time delays to idle and operation places, because these places and tokens therein are used to model thread executions, which take time to complete; transitions are used to model branches or lock allocations and releases, which can be completed in the program instantaneously (when enabled). In this paper, we assume that all the delay times follow exponential distribution. However, our proposed simulation model can be applied to any general distribution.

## 3 THE DISCRETE EVENT SIMULATION MODEL

We first define our simulation model, and then introduce the performance metrics in the simulation analysis.

### 3.1 Basic Components

We follow the simulation framework proposed in Section 1.3.2 of (Law 2007), with necessary extensions to incorporate the special features in Petri nets. The basic components of our simulation model are defined as follows. The *system* being simulated is the Gadara net $\mathcal{N}_G$, which is obtained from a multithreaded program, possibly a multithreaded simulation program for an underlying physical system of interest (recall the discussion in Section 1 about the two levels of deadlocks considered in this paper). The *system state* is the marking (or state) of $\mathcal{N}_G$. An *event* is the firing of a transition in $\mathcal{N}_G$. An event is denoted as a two-tuple $(\tau, t)$, where $\tau$ is the scheduled event time, and $t$ is the transition scheduled to fire at time $\tau$.

We maintain an *enhanced event list* in our simulation. This is due to a special dynamic feature in Petri nets: a transition, which is scheduled to fire at some time, say $\tau$, can be *disabled* in the net at $\tau$ due to lack of tokens in its input place. Moreover, since it corresponds to a pending action of the multithreaded program we study, this scheduled event cannot be discarded, but rather, it should be "backlogged" and fired whenever it becomes enabled after $\tau$. Any element in the enhanced event list is an event and is denoted as a two-tuple $(\tau, t)$ as defined above. As shown in Figure 3, the proposed enhanced event list is divided into two parts: (i) *Future Event List (FEL)*, that records the scheduled events for future execution; (ii) *Backlogged Event List (BEL)*, that records the backlogged events to be executed once they become enabled under a "First-Come First-Served" discipline. Other relevant approaches dealing with this situation include rescheduling a new time for the disabled transition once it becomes enabled, and maintaining different event chains as in General Purpose Simulation System (GPSS).

The *main function* invokes several routines throughout the simulation, including *initialization routine*, *timing routine*, *event routine*, and *report generator*. Due to space limitations, we have to omit their specifics. Here, we further describe the *event-scheduling* scheme that is customized for the special features of $\mathcal{N}_G$ as discussed in Sections 2.3 and 2.4. At the beginning of the simulation, for

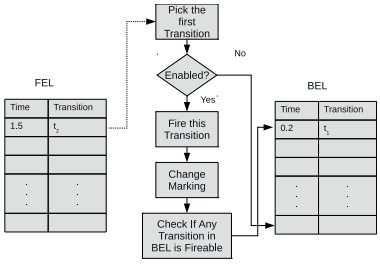



Figure 3: Enhanced event list

each token in any idle place $p \in P_0$, the simulation program schedules a (future) time to fire the output transition of $p$. During the simulation, when a token enters an operation place $p \in P_S$, the simulation program immediately schedules a (future) time to fire the output transition of $p$. In both cases, the time is scheduled according to the delay time distribution associated with $p$. When $p$ has more than one output transition (e.g., when the output transitions of $p$ model `if/else`), the simulation program first selects one of the output transitions of $p$, according to some pre-specified probability distribution. Then, the simulation program schedules a time to fire the selected transition according to the delay time distribution associated with $p$. Due to the nature of the problem, our simulation falls into the class of *terminating simulations*. There are two alternative *terminating events*: (i) $E_1 =$\{a transition is fired so that the system goes back to the initial state\}; (ii) $E_2 =$\{a transition is fired so that the system goes into a deadlock state\}. The simulation terminates if either $E_1$ or $E_2$ occurs. The set of potential deadlock states is computed off-line in Stage (ii) of the Gadara methodology described in Section 2.1. This set is input to our simulation program for deciding the occurrence of event $E_2$.

### 3.2 Performance Metrics

We are interested in the following three performance metrics. These metrics are measured for program models before and after deadlock-avoidance control is applied. A comparison analysis between the original uncontrolled program model and the controlled program model is conducted based on these metrics.

#### 3.2.1 Measure of Safety

A program is *safe* if no deadlock ever occurs; otherwise, it is *unsafe*. Given a Gadara net model of a deadlock-prone program, we want to measure the *deadlock probability*, $P_d$, of this program. Assume we make $n$ independent replications of the simulation. Let $X_i$ be the random variable associated with the $i$-th replication, which is defined as follows:

$$X_i = \begin{cases} 0, & \text{if the } i\text{-th replication terminates with event } E_1; \\ 1, & \text{if the } i\text{-th replication terminates with event } E_2. \end{cases} \tag{1}$$

According to Section 9.4.2 of (Law 2007), we know that an unbiased point estimator for $P_d$ is: $\hat{P}_d = \frac{\sum_{i=1}^{n} X_i}{n}$.

#### 3.2.2 Measure of Efficiency

We use the average total time the threads take to complete their tasks to characterize the *efficiency* of the program. Thus, we measure the *Mean Time To Finish (MTTF)*, given that no deadlock occurs in the program. MTTF is estimated by mean termination times of the replications that terminate with event $E_1$.

#### 3.2.3 Measure of Activity Level

We use the average number of threads, which are simultaneously executing in the critical region, to characterize the *activity level* of the program, which also reflects the program's concurrency level. According to the definition of Gadara nets, the critical region is captured by the set of operation places. Therefore, the activity level, $\beta$, can be estimated as follows.

$$\hat{\beta} = \sum_{p \in P_0} M_0(p) - \sum_{p \in P_0} \hat{U}(p) \tag{2}$$

where $M_0(p)$ is the initial number of tokens in place $p$, and where $\hat{U}(p)$ is the expected time-average number of tokens in place $p$ given that no deadlock occurs in the program. In (2), the first term represents, at the

initial state, the total number of threads waiting in the idle places; the second term represents, throughout the simulation, the average number of threads waiting in the idle places. The difference of these two terms, $\hat{\beta}$, represents the average number of threads executing in the operation places throughout the simulation. The measure $\hat{U}(p)$, also known as the *utilization* of place $p$, can be estimated as: $\hat{U}(p) = E\left[\frac{\int_0^T M_\tau(p)d\tau}{T}\right]$, where $M_\tau(p)$ is the number of tokens in place $p$ at time $\tau$ and $T$ is the termination time of a replication that terminates with event $E_1$. Note that $T$ is a random variable, and the expectation is with respect to $T$. $\hat{U}(p)$ can be measured by standard techniques employed in the simulation analysis of queueing systems for estimating average queue length or average server utilization.

## 4 CASE STUDIES

We present our simulation study for two deadlock-prone Gadara nets, both before and after control. The sensitivity analysis reports the performance metrics introduced in Section 3.2 for a range of values of key parameters. For each case, we carried out 20,000 replications. When comparing the before-control and after-control nets in each example, we employ the *Common Random Number (CRN)* technique to facilitate the comparison analysis. Due to space limitations, we focus on simulation analysis, and have to omit the details about deadlock detection and control logic synthesis; see (Liao et al. 2010) for further discussion.

### 4.1 Case Study 1: A Deadlock Scenario in the OpenLDAP Software

OpenLDAP is a popular open-source implementation of the Lightweight Directory Access Protocol (LDAP). We built the Gadara net model of version 2.2.20 of `slapd`, which is a high-performance multithreaded network server program of OpenLDAP, and has a confirmed CMW deadlock bug. For the sake of discussion, we focus on the critical region involved in this deadlock, and study its associated Gadara net model, shown in solid lines in Figure 4. A deadlock will occur if one token (representing thread 1) is in place $p_5$ and another token (representing thread 2) is in place $p_1$. In this scenario, thread 1 is holding lock B and waiting for lock A, while thread 2 is holding A and waiting for B. The program model after control is the entire net shown in Figure 4, where the synthesized monitor place is shown with a dashed line. In presence of the monitor place, the aforementioned deadlock will not be reachable in the controlled model.

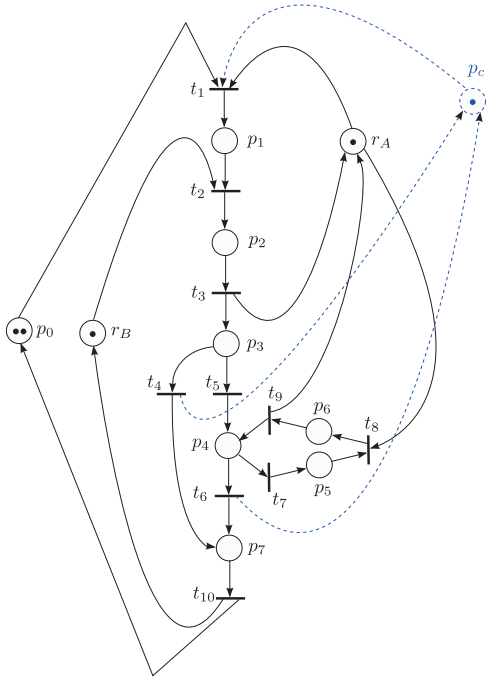



Figure 4: A deadlock example in the OpenLDAP Software

Recall that each operation place models a code segment where a thread can execute. So the random delay time associated with each operation place should reflect the execution time of the involved thread. Methodologies for determining the accurate execution time of code segments have been developed by researchers in the area of real-time embedded systems (Harmon, Baker, and Whalley 1994). One conventional approach is to assume that the execution time of each instruction is a constant. However, factors such as machine status can also add randomness to the execution time. For the purpose of our present study, we will assume that the delay associated with each operation place is exponentially distributed with mean equal to 1. At the beginning of each replication, we schedule each token in $p_0$ to fire $t_1$ after a random delay time that is exponentially distributed with mean $\mu$, which is chosen to be $0.5$ in this study. Simulations for other values of $\mu$ can be carried out in a similar manner.

We employ sensitivity analysis to study the effect of the program parameter, namely branch selection probability distribution, on the program's performance. There are two branch selections in this example: $t_4/t_5$ and $t_6/t_7$. If there is a thread in $p_3$, we assume this thread will choose branch $t_4$ with probability $\pi_4$ and choose branch $t_5$ with probability $1 - \pi_4$. Similarly, if there is a thread in $p_4$, we assume this thread will choose branch $t_6$ with probability $\pi_6$ and choose branch $t_7$ with probability $1 - \pi_6$. The deadlock probability $P_d$ of the uncontrolled model under various values of ($\pi_4$, $\pi_6$) is shown in Figure 5. We observe that the smaller the values of $\pi_4$ and $\pi_6$ are, the larger the value of $P_d$ is. This observation agrees with our intuition: when $\pi_4$ and $\pi_6$ decrease, the thread holding lock B is more likely to enter the loop ($p_4$, $p_5$, and $p_6$) to acquire lock A, and thus more likely to enter a CMW deadlock. The deadlock probability of the controlled model is always 0, which is verified by simulation

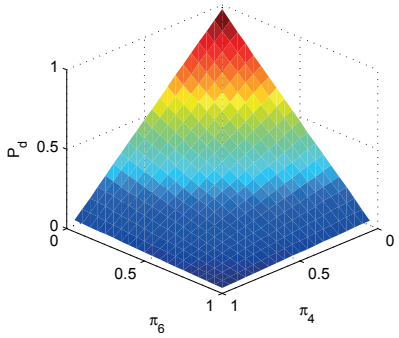
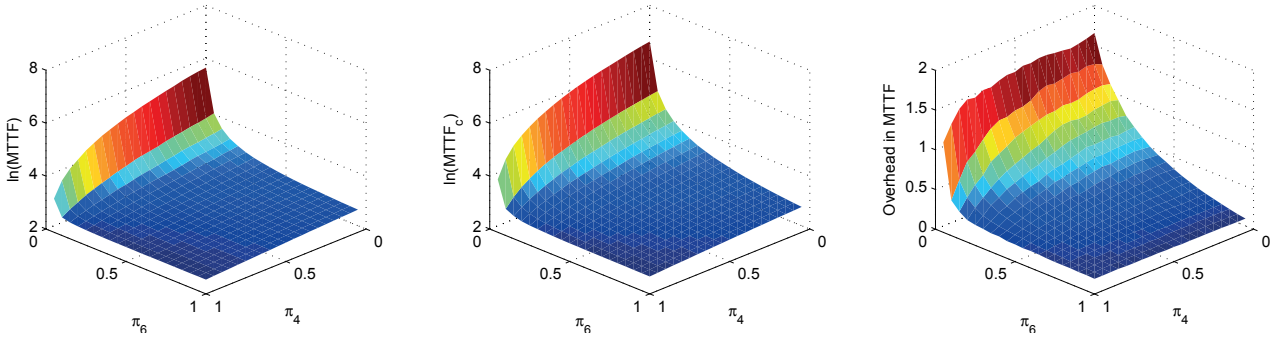



Figure 5: $P_d$ of uncontrolled program model under various values of ($\pi_4$, $\pi_6$)

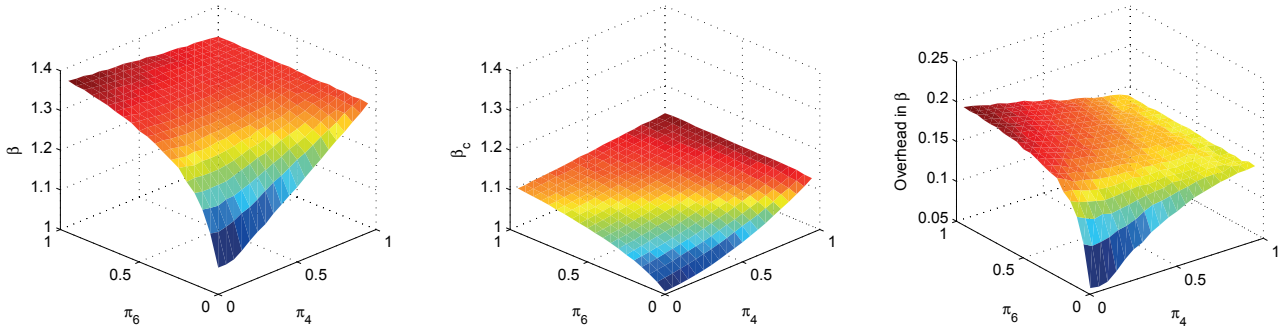
The MTTF of the uncontrolled and controlled models, under various values of ($\pi_4$, $\pi_6$), are shown in Figures 6(a) and (b), respectively, where the z-axis is on a log-scale. We observe that MTTF increases in controlled models. One reason leading to the increase is the imposition of the synthesized monitor place. Another important reason is that the calculation of the statistics of MTTF before control only takes into account those replications that did not deadlock, and ignores the ones that deadlocked. In other words, MTTF before control is "biased downwards" because it only considers deadlock-free replications. To quantify this comparison, we further compute the overhead in MTTF, which is defined as the ratio of the increase in MTTF after control and the original MTTF before control. The overhead in MTTF is shown in Figure 6(c). We see that $\pi_6$, which directly affects the probability of entering the loop ($p_4$, $p_5$, and $p_6$), is the key factor in the MTTF performance of program models. There exists a threshold value for $\pi_6$, denoted as $TH_{\pi_6}$. When $\pi_6$ is smaller than $TH_{\pi_6}$, MTTF in uncontrolled and controlled models as well as the overhead in MTTF dramatically increase.




(a) (b) (c)

Figure 6: MTTF under various values of ($\pi_4$, $\pi_6$): (a) Before control; (b) After control; (c) Overhead

The $\beta$ of the uncontrolled and controlled models, under various values of ($\pi_4$, $\pi_6$), are shown in Figures 7(a) and (b), respectively. We see that $\beta$ decreases in controlled models. Similarly, we compute the overhead in $\beta$, which is defined as the ratio of the decrease in $\beta$ after control and the original $\beta$ before control. The overhead in $\beta$ is shown in Figure 7(c). We also see that $\beta$ decreases when $\pi_4$ and $\pi_6$ decrease. This observation agrees with our analysis above. When $\pi_4$ and $\pi_6$ decrease, the uncontrolled model has a higher deadlock probability, and the effect of the monitor place in the controlled model is more prominent, thus the overall thread activity decreases. Note that in this situation, the overhead in $\beta$ also decreases.




(a) (b) (c)

Figure 7: $\beta$ under various values of ($\pi_4$, $\pi_6$): (a) Before control; (b) After control; (c) Overhead

**Remark 1** In practice, if the consequence of the potential deadlock is manageable, then we can operate the monitor place to balance $P_d$ against MTTF and $\beta$. When $\pi_4$ and $\pi_6$ are large, we know from Figure 5 that $P_d$ is small in the uncontrolled model, thus we can turn off the monitor place, and avoid the overhead in MTTF and $\beta$. As shown in Figure 7(c), the saved overhead in $\beta$ in this case is relatively large.

### 4.2 Case Study 2: Two Threads Sharing Three Common Resources

The second case we will study is the Gadara net model of a multithreaded simulator for a hypothetical concurrent healthcare system. As shown in Figure 2 in Section 2.3, the subnet $\mathcal{N}_1$ models the first patient flow, and the subnet $\mathcal{N}_2$ models the second patient flow. The resource places $r_A$, $r_B$, and $r_C$ model the nurse, physician, and medical equipment, respectively. The two patient flows represent two prototype procedures of medical treatment. Each flow consists of five treatment stages, as modeled by the operation places. The requirement of resources in each treatment stage is self-explanatory from the Gadara net. As discussed in Section 1, the multithreaded simulator can use mutexes to prevent the above three resources from being accessed concurrently. Thus, the potential deadlocks of the system can manifest themselves in the multithreaded simulator at run time. There are two potential deadlock scenarios in this example: one deadlock occurs when $M(p_{11}) = M(p_{23}) = 1$; another deadlock occurs when $M(p_{13}) = M(p_{21}) = 1$. (The unspecified operation places are empty by default; the marking of resource and idle places can be uniquely determined based on the marking of operation places.) There are also three *deadlock-free unsafe states*: (i) $M(p_{11}) = M(p_{21}) = 1$, (ii) $M(p_{11}) = M(p_{22}) = 1$, and (iii) $M(p_{12}) = M(p_{21}) = 1$. When the net is in any of these states, even if it is not in a deadlock, it will unavoidably enter a deadlock in the future. Thus, we need to synthesize control logic to prevent all of the aforementioned states. The control synthesis constructs three monitor places, as shown in dashed lines in Figure 8. The various combinations of ON/OFF of these monitor places lead to eight different control strategies, as defined in Table 1.

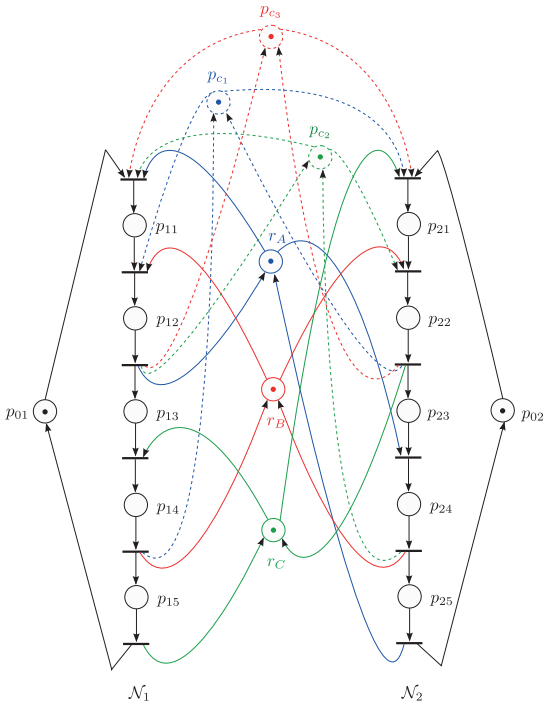



Figure 8: A Gadara net model of two threads sharing three resources: After control

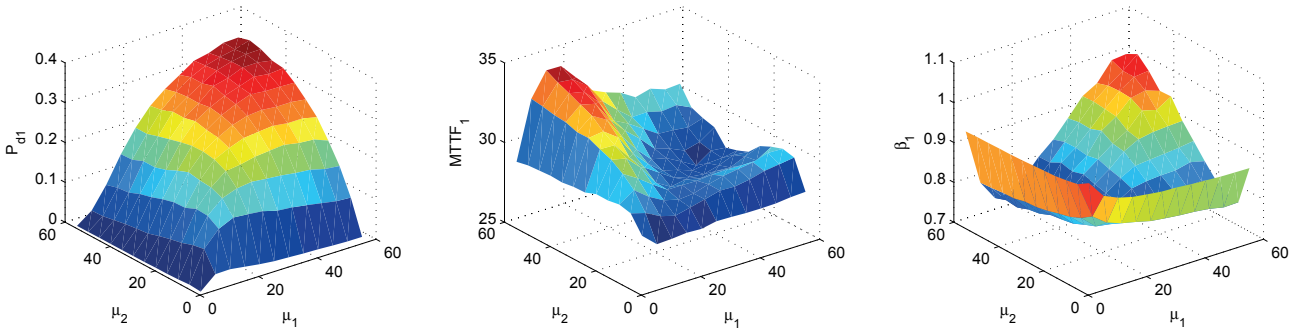
In the simulation, the delays associated with operation places are exponentially distributed. For the five operation places $p_{11}$ to $p_{15}$ in $\mathcal{N}_1$, the mean parameters are 1, 2, 1, 5, and 12, respectively; for the five operation places $p_{21}$ to $p_{25}$ in $\mathcal{N}_2$, the mean parameters are 3, 5, 1, 2, and 1, respectively. In addition, the delays associated with $p_{01}$ and $p_{02}$ are exponentially distributed with means $\mu_1$ and $\mu_2$, respectively.

For the eight control strategies shown in Table 1, one can think of Strategy 1 as the most "aggressive" control, since it turns off all three monitor places for better performance in terms of MTTF and $\beta$; on the other hand, Strategy 8 can be considered as the most "conservative" control, since it turns on all three monitor places, which *guarantees* deadlock-freeness in the controlled model at the price of degraded performance for MTTF and $\beta$. We have conducted sensitivity analysis to study the effect of $\mu_1$ and $\mu_2$ on the program's performance under these eight control strategies. *Our goal is:* given a tradeoff criterion between $P_d$ and MTTF (or $\beta$), find the best control strategy for any pair of $(\mu_1, \mu_2)$.

Table 1: Definition of control strategies

| Strategy | Monitor $p_{c1}$ | Monitor $p_{c2}$ | Monitor $p_{c3}$ |
|---|---|---|---|
| 1 | OFF | OFF | OFF |
| 2 | OFF | OFF | ON |
| 3 | OFF | ON | OFF |
| 4 | ON | OFF | OFF |
| 5 | OFF | ON | ON |
| 6 | ON | OFF | ON |
| 7 | ON | ON | OFF |
| 8 | ON | ON | ON |

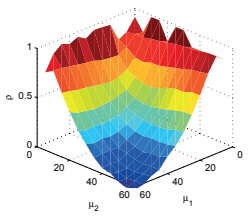
To illustrate, let us consider the sensitivity analysis results for Strategy 2 (i.e., $p_{c3}$ only), which are shown in Figure 9. To measure the effectiveness of deadlock reduction of a certain control strategy, we introduce the *deadlock probability reduction rate*, $\rho$, which is defined as the ratio between the decrease in deadlock probability due to this strategy in the controlled model and $P_d$ of the uncontrolled model. The metric $\rho$ of Strategy 2, derived from the results in Figure 9(a) and the counterpart of the uncontrolled model, is shown in Figure 10. It is interesting to see that when $\mu_1$ and $\mu_2$ are small, by using Strategy 2, the monitor place $p_{c3}$ alone can prevent most of the potential deadlocks. Therefore, similar in spirit to the discussion in Remark 1, if the consequence of the potential deadlock is manageable (e.g., one of the patients in deadlock can be rescheduled without jeopardizing his/her health), then we can employ Strategy 2 (instead of the most conservative Strategy 8) to gain better MTTF and $\beta$ performance when $\mu_1$ and $\mu_2$ are small. Similar analysis can be carried out for all the other strategies; we omit the details due to space limitations.



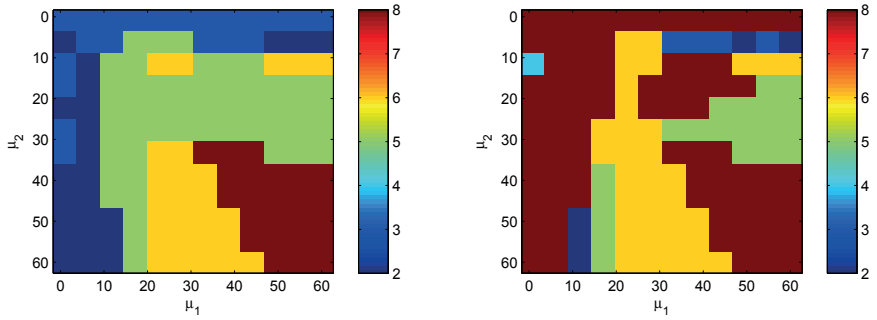

(a) (b) (c)

Figure 9: Sensitivity analysis results for Strategy 2: (a) $P_d$; (b) MTTF; (c) $\beta$

If one can specify a minimum allowed value for $\rho$, say $\rho_0$, based on the empirical analysis of system tolerance, then a possible tradeoff criterion (between $P_d$ and MTTF) for selecting a control strategy is: given $\rho_0$ and $(\mu_1, \mu_2)$, find the best strategy such that (i) its $\rho$ is greater than $\rho_0$, and (ii) its MTTF is as small as possible. Based



on our sensitivity analysis results obtained above, the best strategy at $(\mu_1, \mu_2)$ can be found by first searching for the set of strategies whose $\rho$ is greater than $\rho_0$, then selecting the one whose MTTF is the smallest in this set. After conducting this search process for all pairs of $(\mu_1, \mu_2)$ of interest, we can construct a *Control Strategy Map*, under the tradeoff between $P_d$ and MTTF, on the $\mu_1$-$\mu_2$ plane as shown in Figure 11(a). Using the above sensitivity analysis results, we can construct various forms of Control Strategy Maps according to specific needs. For instance, if we substitute Condition (ii) above by "(ii') its $\beta$ is as large as possible", then we can obtain a Control Strategy Map under the tradeoff between $P_d$ and $\beta$, as shown in Figure 11(b). For both maps in Figure 11, we chose $\rho_0 = 0.75$. Further extensions are possible by using a requirement in terms of a maximum allowed value for $P_d$, instead of Condition (i) above.




(a) (b)

Figure 11: Control Strategy Maps: (a) Tradeoff between $P_d$ and MTTF; (b) Tradeoff between $P_d$ and $\beta$

## 5 CONCLUSION

We have used discrete event simulation to study the performance of deadlock-avoidance control in deadlock-prone concurrent programs that use mutexes for access to shared data. The given multithreaded program, which could be a concurrent simulator itself, is modeled by stochastic timed-place Gadara Petri nets, a special class of Petri nets introduced in our recent work. We proposed a formal methodology and data structure for the simulation of stochastic timed-place Gadara Petri nets. We conducted simulation analysis on two case studies and measured the performance metrics in terms of safety, efficiency, and activity level. We further carried out a comparison analysis on the models before and after deadlock-avoidance control logic is applied to the program. We reported results on the impact of key parameters and different control strategies on the performance metrics. We also discussed the implications for the practical implementation of control strategies to prevent deadlocks in multithreaded software.

## ACKNOWLEDGMENTS

This work was partially supported by NSF grant CCF-0819882 and an award from HP Labs Innovation Research Program. We thank Dr. Mark Van Oyen, Dr. Ricardo Lüders, and the anonymous reviewers for many helpful comments.

## REFERENCES

Bagrodia, R. L., and M. Takai. 2000, April. "Performance evaluation of conservative algorithms in parallel simulation languages". *IEEE Transactions on Parallel and Distributed Systems* 11 (4): 395–411.

Cassandras, C. G., and S. Lafortune. 2008. *Introduction to Discrete Event Systems*. 2nd ed. Springer.

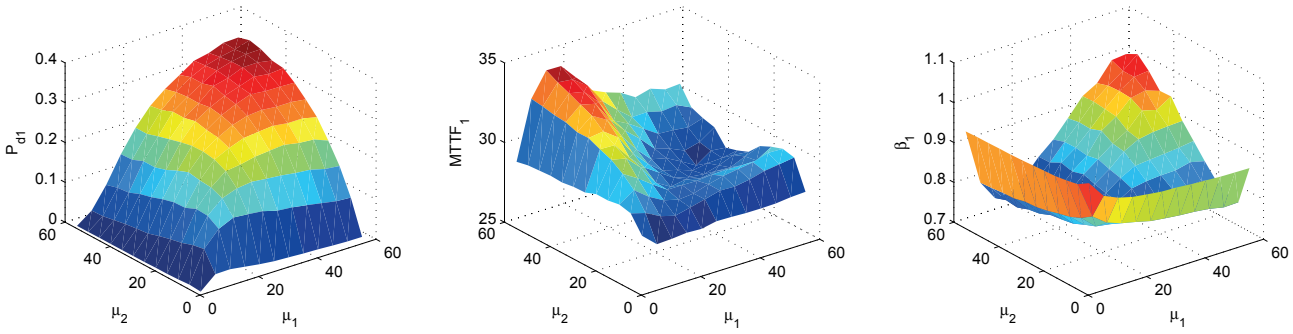
In the simulation, the delays associated with operation places are exponentially distributed. For the five operation places $p_{11}$ to $p_{15}$ in $\mathcal{N}_1$, the mean parameters are 1, 2, 1, 5, and 12, respectively; for the five operation places $p_{21}$ to $p_{25}$ in $\mathcal{N}_2$, the mean parameters are 3, 5, 1, 2, and 1, respectively. In addition, the delays associated with $p_{01}$ and $p_{02}$ are exponentially distributed with means $\mu_1$ and $\mu_2$, respectively.

For the eight control strategies shown in Table 1, one can think of Strategy 1 as the most "aggressive" control, since it turns off all three monitor places for better performance in terms of MTTF and $\beta$; on the other hand, Strategy 8 can be considered as the most "conservative" control, since it turns on all three monitor places, which *guarantees* deadlock-freeness in the controlled model at the price of degraded performance for MTTF and $\beta$. We have conducted sensitivity analysis to study the effect of $\mu_1$ and $\mu_2$ on the program's performance under these eight control strategies. *Our goal is:* given a tradeoff criterion between $P_d$ and MTTF (or $\beta$), find the best control strategy for any pair of $(\mu_1, \mu_2)$.

Table 1: Definition of control strategies

| Strategy | Monitor $p_{c1}$ | Monitor $p_{c2}$ | Monitor $p_{c3}$ |
|---|---|---|---|
| 1 | OFF | OFF | OFF |
| 2 | OFF | OFF | ON |
| 3 | OFF | ON | OFF |
| 4 | ON | OFF | OFF |
| 5 | OFF | ON | ON |
| 6 | ON | OFF | ON |
| 7 | ON | ON | OFF |
| 8 | ON | ON | ON |

To illustrate, let us consider the sensitivity analysis results for Strategy 2 (i.e., $p_{c3}$ only), which are shown in Figure 9. To measure the effectiveness of deadlock reduction of a certain control strategy, we introduce the *deadlock probability reduction rate*, $\rho$, which is defined as the ratio between the decrease in deadlock probability due to this strategy in the controlled model and $P_d$ of the uncontrolled model. The metric $\rho$ of Strategy 2, derived from the results in Figure 9(a) and the counterpart of the uncontrolled model, is shown in Figure 10. It is interesting to see that when $\mu_1$ and $\mu_2$ are small, by using Strategy 2, the monitor place $p_{c3}$ alone can prevent most of the potential deadlocks. Therefore, similar in spirit to the discussion in Remark 1, if the consequence of the potential deadlock is manageable (e.g., one of the patients in deadlock can be rescheduled without jeopardizing his/her health), then we can employ Strategy 2 (instead of the most conservative Strategy 8) to gain better MTTF and $\beta$ performance when $\mu_1$ and $\mu_2$ are small. Similar analysis can be carried out for all the other strategies; we omit the details due to space limitations.




(a) (b) (c)

Figure 9: Sensitivity analysis results for Strategy 2: (a) $P_d$; (b) MTTF; (c) $\beta$

If one can specify a minimum allowed value for $\rho$, say $\rho_0$, based on the empirical analysis of system tolerance, then a possible tradeoff criterion (between $P_d$ and MTTF) for selecting a control strategy is: given $\rho_0$ and $(\mu_1, \mu_2)$, find the best strategy such that (i) its $\rho$ is greater than $\rho_0$, and (ii) its MTTF is as small as possible. Based

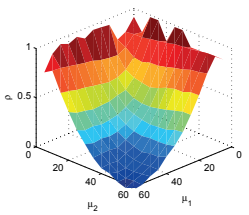


Figure 10: Deadlock probability

## AUTHOR BIOGRAPHIES

**HONGWEI LIAO** is a Ph.D. Candidate in Electrical Engineering-Systems (EE-S) at the University of Michigan, where he received a M.Sc. degree in EE-S in 2009, and a M.S.E. degree in Industrial and Operations Engineering in 2011. He is a recipient of the Rackham Predoctoral Fellowship Award and the College of Engineering Distinguished Achievement Award from the University of Michigan. His email address is hwliao@umich.edu.

**HAO ZHOU** is a Ph.D. student in Industrial and Operations Engineering at the University of Michigan. His research focuses on simulation and optimization of various aspects of Intelligent Transportation Systems, such as collision avoidance and car-following controls, and simulation analysis of concurrent software. His email address is haozhou@umich.edu.

**STÉPHANE LAFORTUNE** is a professor of electrical engineering and computer science at the University of Michigan. He joined UM in 1986 after obtaining his Ph.D. at the University of California at Berkeley. He is a Fellow of IEEE (1999) and twice recipient of the Axelby Outstanding Paper Award from the IEEE Control Systems Society. His email address is stephane@umich.edu.