ENFORCING SAFETY PROPERTIES IN WEB APPLICATIONS USING PETRI NETS

Liviu Grigore
Computer Science Department
University of Illinois at Chicago
Chicago, IL, 60607
lgrigore@cs.uic.edu

Ugo Buy
Computer Science Department
University of Illinois at Chicago
Chicago, IL, 60607
buy@cs.uic.edu

ABSTRACT
Web applications are often based on the client-server model which relies on concurrent execution of asynchronous processes. Enforcing correctness of concurrent software is notoriously difficult. In general, automatic verification checks if a given system has a certain property, while supervisory control enforces the same property by restricting system behavior. Supervisor control problems are often computationally more tractable to solve than verification problems. Most verification problems are NP-hard, while some supervisory control problems can be solved in polynomial time with the appropriate representation. Here we present two algorithms, one for enforcing mutual exclusion and the other for deadlock prevention for web applications written in Java. We combine these two methods in order to guarantee that a web application or web service comply with given safety properties including freedom from deadlock and system specific mutual exclusion properties.

KEY WORDS
Petri nets, formal methods, supervisory control.

1 Introduction

During development concurrent Java applications face mutual exclusion and deadlock problems because these applications contain asynchronous processes. Efficiently testing web applications represents an essential condition for modern day software. Finding and correcting deadlocks can be difficult. Testing for deadlock is not always a solution since testing all possible interleavings of computations performed by asynchronous processes might be infeasible because there is a huge number of such interleavings. In addition, replicating a deadlock situation is troublesome once it appeared, making the source of the deadlock difficult to determine [2].

Verification for concurrent system is a research area that has been covered for more than twenty years. Since most of these problems are NP-hard, they are difficult to tackle and the advances in this domain have been relatively slow. Supervisor control problems are sometimes more tractable than the corresponding verification problems. Thus, results can be achieved more swiftly than in case of verification. An example of a tractable supervisory control algorithm compared with its verification counterpart is mutual exclusion [6]. Supervisory control approach is well-suited to the situations when detecting a property is not sufficient; instead that property must be enforced as well.

The goal of this project is to enhance the reliability of web applications by automatically generating supervisory control code that will prevent the violation of safety properties such as freedom from deadlock and mutual exclusion properties. Past work in supervisory control of discrete event systems has led to the definition of various methods for enforcing freedom from deadlock and mutual exclusion constraints. The systems under consideration are typically modeled as finite state automata (FSAs) or Petri nets. Our systems are modeled as Petri nets. We prefer this model to finite state automata since Petri net methods for enforcing supervisory control properties are more tractable than methods that use finite state automata. In particular, our methods for enforcing safety properties (mutual exclusion and deadlock prevention) are more tractable when Java programs are represented as Petri nets.

Modern web applications have a three-tiered structure. The three tiers are: a web browser, a web server and a database. Our method to enforce safety properties refers to Java concurrent programs. Our work concerns Java applications that are included in a web server. Java is a modern programming language that provides various capabilities for concurrency. Java is by far the most popular tool for platform independent development, in particular for web and multi-tiered applications. It is simple, effective and easy to use; it offers a wide range of high-quality libraries and open network support. Our method consists of the following steps:

1. Concurrent Java code (e.g., Java RMI code) is translated into a Petri net model.

2. The Petri net model is further augmented by the addition of a supervisory controller enforcing programmer-defined mutual-exclusion properties.

3. The new Petri net model is further augmented by the addition of a supervisory controller enforcing freedom from deadlock.

4. The generated controller is translated back into Java statements that are inserted into the original code.

In this paper we focus on steps (2) and (3) since these steps are the most challenging. The method overview is presented in Figure 1. First, the Java source code is converted into a Petri net model. Various programming languages [3], [12] have been translated to Petri nets for different reasons (verification, testing, supervisory control). The level of granularity of the translation depends on the purpose of the translation. Capturing all the details of a Java program is synonymous with constructing a prohibitively large Petri net. For our Petri net models we chose to capture explicitly a subset of Java constructs for defining control flow (e.g., if and while statements) and the most popular interprocess communication primitives (e.g., threads and monitors).

The second step in Figure 1 is based on the work of Yamalidou et al. [6]. The method is computationally attractive and amenable to large systems. It constructs a feedback controller for untimed Petri nets. The desired net behavior is enforced by making it an invariant of the controlled net. The main advantage of this method is that it computes the controlled net very efficiently by a single matrix multiplication. That method and a subsequent extension by Moody et at. [10] will be the basis for generating controllers that enforce mutual exclusion properties in Java code. A shortcoming of this method is that deadlock might be introduced when the technique is applied.

The third step in Figure 1 represents the core of our technique. Several directions have been investigated to achieve the deadlock-freeness goal. The siphons method [7] can be applied to a general class of Petri nets (bounded Petri nets), but the computation of minimal siphons is expensive. Shatz et al. [13] defined a different method for sidestepping the complexity of deadlock detection. The disadvantage of that method is that its applicability is restricted to a special class of Petri nets called Ada nets. Melzer and Romer [9] define an efficient algorithm for deadlock detection. It uses a technique called unfolding to explore all the possible behaviors of the net and to detect deadlock. Based on the detected deadlock states, we generate supervisory controllers that eliminate these states.

The last step in Figure 1 consists of translation back to Java. The observations made for the first translation are in general valid for this phase also. To date, the translation from Petri nets to Java has not been fully explored; however, we believe that this translation is relatively straightforward. Petri nets have been translated to programming languages before. Yao and He generate parallel program skeletons from Petri nets [14]. We plan to use a similar approach for Petri net translation into Java.

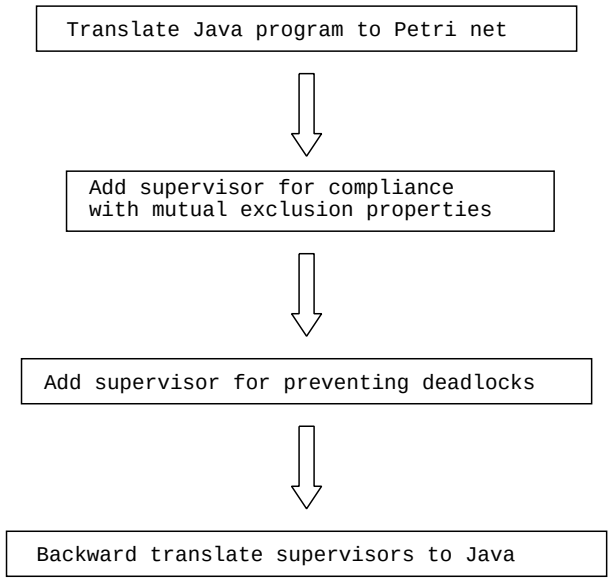

Figure 1. Diagram of supervisor generation for concurrent Java programming.

2 Framework

2.1 Definitions

Petri nets are a graphical and mathematical tool that can model many systems. They met considerable success in describing and studying information processing systems that are characterized as being concurrent, asynchronous, distributed, nondeterministic and/or stochastic. They can be used as a graphical aid tool similar to flow charts and block diagrams [11].

A *safe ordinary Petri net* is a directed graph with two distinct set of nodes, places and transitions. Using a mathematical notation a safe ordinary Petri net is an a 4-tuple $\mathcal{N} = (P, T, F, M_0)$ where:

1. $P = \{p_1, p_2, ..., p_n\}$ is a finite set of places.
2. $T = \{t_1, t_2, ..., t_m\}$ is a finite set of transitions.
3. $F \subset (P \times T) \cup (T \times P)$ is the set of arcs.
4. $M_0 : P \rightarrow \{0, 1\}$ is the initial marking.
5. $P \cap T = \oslash$ and $P \cup T \neq \oslash$.

The set of predecessors and successors of a node x are defined in the following way:

$\bullet x = \{\ y \mid (y, x) \in F\}$ – set of input nodes of x

$x\bullet = \{\ y \mid (x, y) \in F\}$ – set of output nodes of x

The input and the output sets of a place $p \in P$ will be (possibly empty) subsets of T. Likewise, the input and output sets of a transition $t \in T$ will be (possibly empty) place subsets.

To describe the dynamic behavior of the system, a marking in an ordinary Petri net can be changed according to specific rules. To simulate the dynamic behavior of a net, a marking can be changed according to the following rules [11]:

1. A transition is enabled if each input place is marked (contains one token).
2. An enabled transition may fire.
3. The firing of a transition removes a token from each input place and deposits a token in each output place of that transition.

We will first present an example of Java code for translation [1]. The *transferMoney* method describes a transfer transaction from one account to another. Each account is forbidden from performing another operations during money transfer.

```
public void transferMoney(Account fromAccount,
Account toAccount, DollarAmount amountToTransfer) {
  synchronized (fromAccount) {
   synchronized (toAccount) {
     if (fromAccount.hasSufficientBalance(
       amountToTransfer) {
        fromAccount.debit(amountToTransfer);
        toAccount.credit(amountToTransfer);
      }
    }
  }
}
```

```
Thread 1:
transferMoney(accountOne, accountTwo, amount);

Thread 2:
transferMoney(accountTwo, accountOne, anotherAmount);
```

The Java method implements a money transfer from one account to another. It takes a specified amount of money *amountToTransfer* and transfers it from *fromAccount* to *toAccount*. It first checks whether there is enough money to transfer. In this case, it subtracts the amount from the first account and the sum is deposited into the second account.

The method is not thread safe. For example if two transactions are operated on two accounts at the same time in opposite directions, then a deadlock could occur. Suppose that a thread calls a transfer operation from the first account to the second account and another thread calls a transfer operation from the second account to the first account. For a deadlock to happen, the first thread must acquire the monitor for the first account and the second thread must acquire the monitor for the second account before the first thread does so. In this situation there cannot exist a normal program execution since both threads are waiting to acquire a monitor that is held by another thread.

The Petri net representation of this code is presented in Figure 2. This is the kind of Petri net that can be obtained by translation from Java code. Each transition is

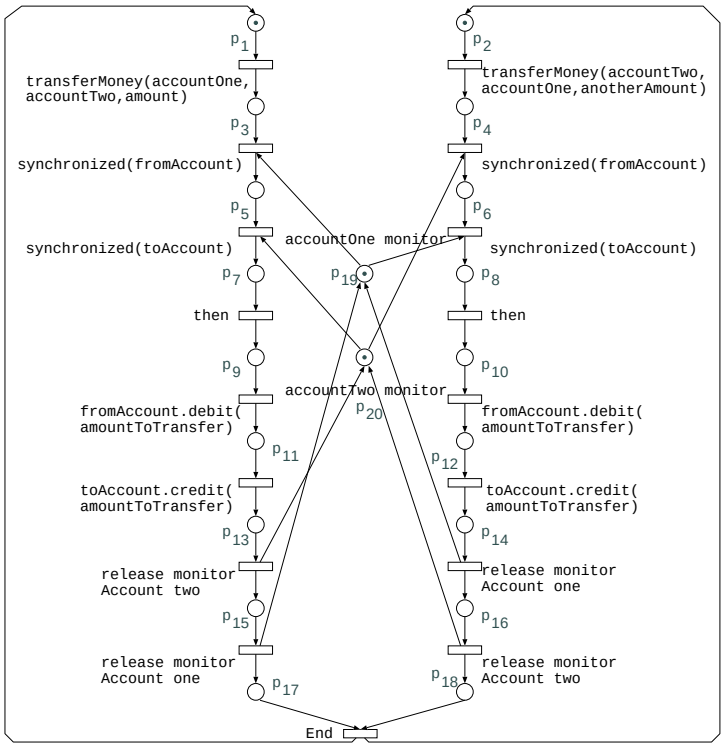

Figure 2. Petri net representation of Java code

associated with a statement, while a location in the flow of control of a thread is described by a place. Besides a flow of serial transitions for each thread, a place associated with each thread monitor was added to the Petri net. The transitions *transferMoney(accountOne,accountTwo,amount)*, *synchronized(fromAccount)*, *synchronized(toAccount)*, *fromAccount.debit(amountToTransfer)*, *toAccount.debit(amountToTransfer)* correspond to the homonymous statements. The *then* transition corresponds to body of the if statement. The last two transitions *release monitor Account two*, *release monitor Account two* will release the monitor for the two objects *toAccount*, *fromAccount*.

Places p_1 and p_2 represent the initial start places for the two processes. Triggering the *transferMoney(accountOne, accountTwo,amount)* transition means that a Java program entered the method with the same name. Firing transitions *synchronized(fromAccount)*, *synchronized(toAccount)* means taking a token from places p_{19} and p_{20} that represent the monitors locks associated with the Java objects *fromAccount* and *toAccount*. The two accounts are debited and credited by firing the two corresponding transitions. At the end, tokens are placed back to the monitor places (by firing transitions *release monitor Account two* and *release monitor Account one*). This net contains a deadlock state if the two transitions labeled with *synchronized(fromAccount)* are fired subsequently.

2.2 Unfoldings

Petri nets can be "unfolded" into a particular type of Petri net named an occurrence net. The nodes of the occurrence net are labeled with the places and the transitions of the original net. The unfolding process can be stopped as in the graph theory case at different stages and new nets are obtained. By unfolding "as much as possible" a net a unique, usually infinite occurrence net is obtained.

McMillan introduced the concept of truncated unfoldings (i.e., finite prefixes) [8] and proved that the truncated net preserves all the markings existent in the original net modulo a homomorphism mapping nodes in the unfolding to nodes in the original net. Subsequently, Esparza et al. [4] refined the initial definition of the truncating condition, in such a way that the new prefix constructed is significantly smaller than the McMillan prefix and never larger (up to a small constant) than the state space of the Petri net.

When we unfold a Petri net, we build a structure called a labeled occurrence net. This is a Petri net in which every place and transition is labeled by a corresponding place or transition in the original net. Formally, we establish a homomorphic mapping from nodes in the occurrence net to nodes in the original net. The occurrence net is a specialized form of net which must satisfy certain restrictions. First, it must be well founded, meaning that any path starting from any node to the initial places must have a finite number of elements. Second, two arcs cannot converge on the same place. Third, no transition may be in conflict with itself. Fourth, the occurrence net is acyclic.

The unfolding algorithm associates with each transition added to the occurrence net a reachable marking of the initial Petri net. For this we will need to compute the local configuration of a transition t. The local configuration of a transition t represents all the transitions preceding t in the unfolding. We say that x precedes y, denoted by $x < y$, if a directed path from x to y exists in $\mathcal{N}$.

A mapping exists between the occurrence net and the original net. It preserves the node types, the original marking and the restriction to the preset and the postset of any transition is a bijection. The set of markings of the occurrence net is equivalent to the set of states of the original net.

DEFINITION [Local configuration] The local configuration of a transition t, denoted $[t]$, is the set $\{t' \in T | t' < t \text{ or } t' = t\}$.

A local configuration has a final state, resulted after firing all transitions belonging to the local configuration.

DEFINITION [Cut of a configuration] The cut of a configuration C belonging to an occurrence net O is defined as $cut(C) = (M_0 \cup C\bullet) \setminus \bullet C$.

The cut of a configuration represents the set of places marked after the firing of all configuration transitions. The main idea of the unfolding algorithm is to determine two transitions (t_1 and t_2) such that two conditions are respected: (1) The cuts of local configurations of both transitions are the same (modulo homomorphism). (2) The local configuration of t_1 is smaller (based on a special partial order) than the local configuration of t_2. Transition t_2 represents *a cutoff transition*.

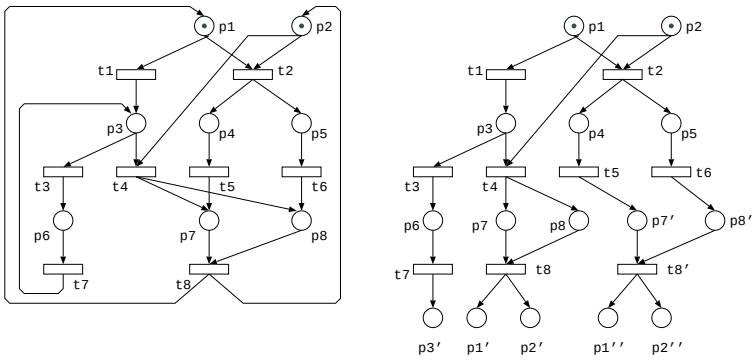

Figure 3. Example of unfolding for a Petri net.

In Figure 3 we present an example of a Petri net and its unfolding. Intuitively a configuration is a set of transitions fireable from the initial marking. The following set of transitions $\{t_1, t_2, t_5, t_6\}$ represents a configuration. The local configuration of transition t'_8 is $\{t_2, t_5, t_6\}$. The cut of local configuration of transition t'_8 is $\{p''_1, p''_2\}$.

3 Enforcing mutual exclusion properties

The method for enforcing mutual exclusion properties is described by Moody et al. [6]. A controller is generated that consists only of places connected to the original net and no transitions. The controller size depends on the number of the properties enforced. The central notion in this method is the place invariant for a Petri net. The controller ensures that the net will not enter a forbidden state. The resulting net (the supervisor and the supervised net) relies on place invariants to enforce the constraints. The controller is maximally permissive; it restricts any behavior that violates the constraint but it allows any behavior that conforms to the mutual exclusion constraints. The constraints enforced are linear based on place markings ($M(p_{i1})+M(p_{i2})+...+M(p_{ik}) \leq c$), where c is a positive integer and $M(p)$ denotes marking for place p.

To enforce mutual exclusion, a matrix D_c is introduced to describe the controller. This matrix will describe how the controller places are connected to the controlled net. The incidence matrix D_p is associated with the original Petri net. D_p is an $n \times m$ incidence matrix, where n is the number of places and m is the number of transitions; the controller matrix D_c is an $n_c \times m$ matrix, where n_c is the number of constraints. The constraint matrix L is a $n_c \times n$ matrix. The formula to compute D_c is:

$$D_c = -LD_p$$

The initial marking of the controller places can be calculated based on the fact that the place invariants equations are valid for the initial marking too.

$$M_{0c} = b - L\ M_{0p}$$

Here M_{0c} represents the initial marking of controller places, M_{0p} represents the initial marking of the original net and b is a $n_c \times 1$ vector.

We will exemplify this method using the Petri net associated with the Java code in Figure 2. We want places p_5 and p_6 to be mutual exclusive. This state is particularly meaningful since it is a deadlock state. If two transactions occur at the same time in opposite direction, involving the same two accounts, then a deadlock might be reached.

The constraint to be introduced is $M(p_5) + M(p_6) \leq 1$, meaning that p_5 and p_6 can contain at most one token combined. Enforcing this constraint is equivalent with the removal of the deadlock state represented by places $\{p_5, p_6\}$. One additional place p_s will be added to enforce this constraint: $M(p_5) + M(p_6) + M(p_s) = 1$. For example, the first row of the incidence matrix corresponding to first place is $D_{1,} = [-1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1]$ and the constraint is specified by the following matrix with one row: $L = [0\ 0\ 0\ 0\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0]$.

The supervisor controller generated is described by the following data:
$D_c = [0\ 0\ 1\ 1\ -1\ -1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0]$ and $b = 1$. One place (p_s) and four new arcs will be added to enforce the mutual exclusion: two arcs from the two transitions labeled with $synchronized(fromAccount)$ to the controller place, and two arcs from the controller place to the two transitions labeled with $synchronized(toAccount)$. The initial marking of the controller place is 1. The mutual exclusion constraint in this example will eliminate also the deadlock. It will be impossible for each thread to acquire just one lock associated with an account and to wait for the second account lock to release. Enforcing a mutual exclusion constraint in this example implies the elimination of a deadlock state.

4 Enforcing deadlock avoidance

One of the main goals of our method is to allow future extensions of Java features implemented. This objective can be accomplished only by considering a class of Petri nets that is sufficiently general to capture the behavior of our Java subset.

We defined a new supervisor generation algorithm based on Melzer and Romer's detection algorithm [9]. The output of the deadlock detection method is a state where no transition can be fired or an indication that such state does not exist. Our deadlock avoidance strategy is based on the idea that by preventing that specific state from being reached, then deadlock is avoided.

Melzer and Romer's method for detecting deadlocks utilizes the token conservation equation. For a given firing sequence σ, the number of tokens for a place p is equal to the number of tokens at M_0 plus the tokens added by the firing of input transitions of p, minus the tokens removed by the firing of its output transitions. In mathematical notation the equation can be written (for each place p):

$$M(p) = M_0(p) + \sum_{t \in \bullet p} \nu(\sigma, t) - \sum_{t \in p \bullet} \nu(\sigma, t)$$

where $\nu(\sigma, t)$ represents the number of occurrences of transition t in the firing vector σ, $M(p)$ represents the current marking of place p and $M_0(p)$ represents the initial marking of place p. This equation shows how many tokens are present in a place for current marking.

The state equation for a Petri net is written as a matrix equation:

$$M = M_0 + D * X\ ;\ X \geq 0$$

where X denotes transition firing vector.

It M is reachable, then there is an integer solution for this equation. However, in general it is possible for the equation to have an integer solution, even though the marking M is not reachable. This scenario is possible since a transition might "borrow" tokens that do not exist in one of its input places, even though subsequently those tokens are deposited back.

A special case for this equation is an acyclic Petri net. In an acyclic net M is reachable from the initial marking if and only if the state equation has a nonnegative integer solution. Evidently, unfoldings are acyclic, which allows us to represent all the reachable markings as an integer linear system. The central theorem of this paper relates the deadlock in an unfolding to the original net [9].

Theorem Let $\mathcal{U} = (P, T, F, h)$ be the unfolding for a safe Petri net $\mathcal{N}$. $\mathcal{N}$ is deadlock-free if and only if the following system of inequalities has no solution:

$$Variables : M, X : integer$$

$$M = M_0 + D * X$$

$$\sum_{p \in \bullet t} M(p) \leq |\bullet t| - 1, \forall t \in T$$

$$X(t) = 0, \forall t\ \textit{cutoff transition}$$

In the above, the first equation (state equation) captures all reachable markings. This affirmation is valid since the unfolding is an acyclic net. The second set of inequalities ensures that no transition is fireable for a specific marking. For each unfolding transition, the transition is disabled by the absence of at least one token in an input place. The last set of inequalities guarantees that no cutoff transition was fired. The idea behind this last subsystem is that if a cutoff transition was fired, then it would only replicate previous states that were investigated in case of deadlock.

The key idea behind this theorem relies on the completeness of the unfolding. The unfolding has the same state space as the original net modulo homomorphic mapping of unfolding nodes to the net nodes, thus any deadlock in the original net can be detected via unfolding using the

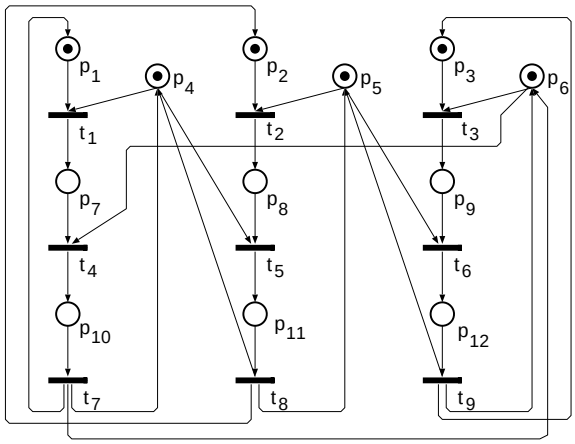

Figure 4. Dining philosophers net.

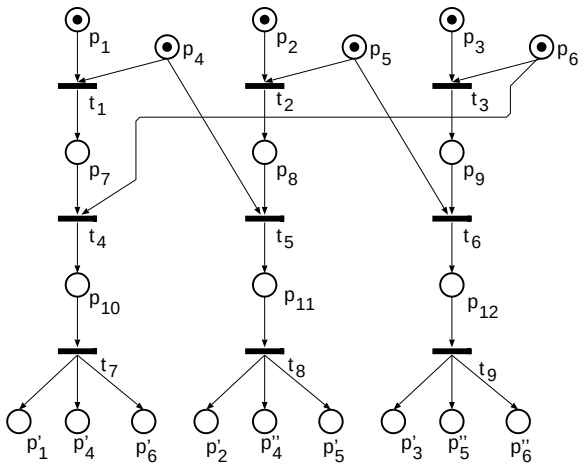

Figure 5. Unfolding of dinning philosophers net.

above system. Some dead markings in the unfolding based on its finiteness and acyclicity do not correspond to actual deadlocks in the original net. But these markings are not detected by the previous system of inequalities because of the last set of equations stipulating that unfolding transitions are never fired.

For any deadlock we must make sure that this state is never reached. Thus if the deadlock state is triggered by the marking $p_{i1}, p_{i2}, ..., p_{ik}$, then we must enforce the constraint: $M(p_{i1}) + M(p_{i2}) + ... + M(p_{ik}) < k$.

This method for supervisor generation ensures that in any state there are at most $k-1$ tokens in the set of places that are marked in the deadlock state. This constraint effectively prevents the original Petri net from entering its deadlock state. Enforcing the above property will ensure that the system never reaches the deadlocked state detected before.

A classic example that involves deadlock is the *dining philosophers problem*. The net portraying the problem of dining philosophers is depicted in Figure 4. The unfolding for this net is shown in Figure 5.

To detect the deadlock we write the system of equalities and inequalities:

1. State equation:

$$\begin{array}{lll} p_1 = 1 - t_1 & p_2 = 1 - t_2 & p_3 = 1 - t_3 \\ p_4 = 1 - t_1 & p_5 = 1 - t_2 & p_6 = 1 - t_3 \\ p_7 = t_1 - t_4 & p_8 = t_2 - t_5 & p_9 = t_3 - t_6 \\ p_{10} = t_4 - t_7 & p_{11} = t_5 - t_8 & p_{12} = t_6 - t_9 \\ p'_1 = t_7 & p'_4 = t_7 & p'_6 = t_7 \\ p'_2 = t_8 & p''_4 = t_8 & p'_5 = t_8 \\ p'_3 = t_9 & p''_5 = t_9 & p''_6 = t_9 \end{array}$$

2. Transitions inequalities:

$$\begin{array}{lll} p_1 + p_4 \leq 1 & p_2 + p_5 \leq 1 & p_3 + p_6 \leq 1 \\ p_6 + p_7 \leq 1 & p_4 + p_8 \leq 1 & p_5 + p_9 \leq 1 \\ p_{10} \leq 0 & p_{11} \leq 0 & p_{12} \leq 0 \end{array}$$

3. Cut off transitions equalities:

$$t_7 = 0 \quad t_8 = 0 \quad t_9 = 0$$

This system admits a solution: $p_1 = p_2 = p_3 = p_4 = p_5 = p_6 = p_{10} = p_{11} = p_{12} = p'_1 = p'_2 = p'_3 = p'_4 = p'_5 = p'_6 = p''_4 = p''_5 = p''_6 = 0$, $p_7 = p_8 = p_9 = 1$ and $t_1 = t_2 = t_3 = 1$, $t_4 = t_5 = t_6 = t_7 = t_8 = t_9 = 0$. This solution corresponds to the situation when each philosopher acquires only one resource (chopstick for example) and waits for the other one to be released. This might never happen if these resources are already held by the other philosophers.

To avoid deadlock the state that generated the deadlock is avoided by enforcing a new linear constraint: $p_7 + p_8 + p_9 < 3$. The result will be a supervised net with one extra place (p_{13}). The net is shown in Figure 6. To continue applying the technique we have to compute the unfolding for the supervised net for the dining philosophers problem. The unfolding is shown in Figure 7. Next, we check if there is any deadlock state for the supervised net. The system of equalities and inequalities is very similar to the unsupervised net, but it does not admit any integer solution because of the new constraint above. This step concludes our process for deadlock avoidance for this net.

5 Research directions

We presented a new framework for Java programming that focuses on deadlock prevention. We first determine whether deadlock states exist. A deadlock state is discovered using a set of equalities and inequalities. For deadlock to be possible, a state must be reachable in which no transition can be fired. This search can be performed efficiently using efficiently using Melzer and Romer's method [9]. Our supervisors avoid any deadlock state using a linear inequality forbidding the Petri net to enter that state. The linear inequality states that the number of tokens in that deadlock state must be less than the number of places composing that state. The method is successfully integrated with an

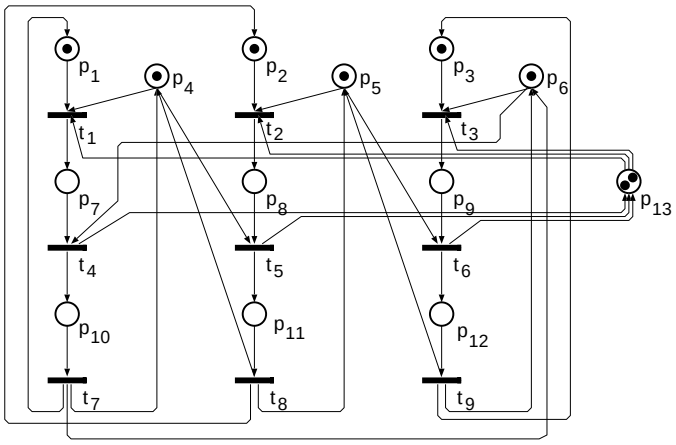

Figure 6. The supervised net for dining philosophers example.

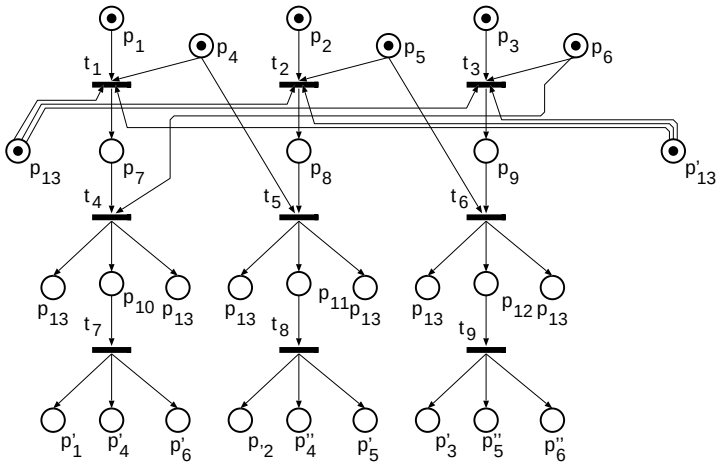

Figure 7. Unfolding of dining philosophers net.

existing method for enforcing mutual exclusion constraints.

There are several aspects in which this method is open to further research. The first optimization is related to the number of iterations of the supervisor generation algorithm. We will seek to reduce this number using heuristics. One solution is to reduce the Petri net using reduction rules. Transitions and places in a serial flow, for example, can be merged. Another solution is to identify the decision points that lead to deadlock and to avoid those points. Another direction is represented by the investigation of alternative deadlock prevention strategies. One such example is He and Lemmon's method [5]. In their article they described a method to prevent deadlock, but some of their results were not correct. Xie and Giua proposed a correction, but they did not define a fully functional algorithm.

References

[1] Avoid synchronization deadlocks. http://www.javaworld.com/javaworld/jw-10-2001/jw-1012-deadlock.html.

[2] W. T. Amy Williams and M. D. Ernst. Static deadlock detection for Java libraries. pages 602–629, July 27–29, 2005.

[3] A. Bourjij and P. Nus. A new methodology for hardware/software codesign using Petri nets. *IEEE Pacific Rim Conference On Communications, Computers and signal Processing*, 1:341–344, 2001.

[4] J. Esparza, S. Romer, and W. Vogler. An improvement of McMillan's unfolding algorithm. *Tools and Algorithms for Construction and Analysis of Systems*, pages 87–106, 1996.

[5] K. X. He and M. D. Lemmon. Liveness-enforcing supervision of bounded ordinary Petri nets using partial order methods. *IEEE Transactions on Automatic Control*, 47(7):1042–1055, 2002.

[6] M. L. Katerina Yamalidou, John Moody and P. Antsaklis. Feedback control of Petri nets based on place invariants. *Proceedings of the 33rd IEEE Conference on Decision and Control*, 3:3104–3109, 1996.

[7] J. O. M. Marian V. Iordache and P. Antzaklis. Synthesis of deadlock prevention supervisors using Petri nets. *IEEE Transactions on Robotics and Automatics*, 18:59–68, 2002.

[8] K. L. McMillan. A technique of state space search based on unfolding. *Formal Methods in System Design*, 6(1):45–65, 1995.

[9] S. Melzer and S. Romer. Deadlock checking using net unfoldings. *Computer Aided Verification*, pages 352–363, 1997.

[10] J. Moody and P. Antsaklis. Supervisory control of Petri nets with uncontrollable /unobservable transitions. *In Proceedings of the 35th IEEE Conference on Decision and Control*, pages 4433–4438, 1996.

[11] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77:541–580, 1989.

[12] T. Murata, B. Shenker, and S. M. Shatz. Detection of Ada static deadlocks using Petri net invariants. *IEEE Trans. Software Eng.*, 15(3):314–326, 1989.

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

[14] W. Yao and X. He. Mapping Petri nets to parallel programs in cc++. *IEEE Transactions on Robotics and Automatics*, pages 70–75, 1996.