Accelerating Reachability Analysis on Petri Net for Mutual Exclusion-Based Deadlock Detection

Yunkai DU\textsuperscript{(a)}, Naijie GU\textsuperscript{(b)}, Nonmembers, and Xin ZHOU\textsuperscript{(†c)}, Student Member

SUMMARY Petri Net (PN) is a frequently-used model for deadlock detection. Among various detection methods on PN, reachability analysis is the most accurate one since it never produces any false positive or false negative. Although suffering from the well-known state space explosion problem, reachability analysis is appropriate for small- and medium-scale programs. In order to mitigate the explosion problem several kinds of techniques have been proposed aiming at accelerating the reachability analysis, such as net reduction and abstraction. However, these techniques are for general PN and do not take the particularity of application into consideration, so their optimization potential is not adequately developed. In this paper, the feature of mutual exclusion-based program is considered, therefore several strategies are proposed to accelerate the reachability analysis. Among these strategies a customized net reduction rule aims at reducing the scale of PN, two marking compression methods and two pruning methods can reduce the volume of reachability graph. Reachability analysis on PN can only report one deadlock on each path. However, the reported deadlock may be a false alarm in which situation real deadlocks may be hidden. To improve the detection efficiency, we proposed a deadlock recovery algorithm so that more deadlocks can be detected in a shorter time. To validate the efficiency of these methods, a prototype is implemented and applied to SPLASH2 benchmarks. The experimental results show that these methods accelerate the reachability analysis for mutual exclusion-based deadlock detection significantly.

key words: deadlock detection, Petri-net, mutual exclusion, reachability

1. Introduction

Petri Net (PN) is a graphical mathematical tool that can be utilized to model discrete events. It is very apt to be applied to detect deadlocks in concurrent programs since programs are essentially discrete. The term deadlock means a system state that has no successor.

Decades of study have yielded several approaches to deadlock, but none is a panacea. All approaches have to balance the precision and the efficiency. Some researchers propose effective but imprecise methods \cite{1}–\cite{7} to conquer the explosion problem, so that large-scale real world systems can be detected efficiently. Engler and Ashcraft present RacerX \cite{1}, a static tool that conducts the lock-order graph and reports cycles in the graph as possible deadlocks. RacerX is fast, a 1.8 million line system can be detected in 15 minutes, but is highly imprecise. Naik et al. present Chord \cite{2}, another effective tool that expresses the deadlock freedom problem in terms of six necessary conditions of deadlock. Ding et al. \cite{3} present an algorithm that represents a given concurrent system by a set of ordinary differential equations, and reports deadlocks built on the solutions of these equations, but reporting numerous false alarms.

Some other researchers focus on how to make precise detection more scalable. Among these precise detection techniques, reachability analysis on PN is one of the most straightforward approaches. Although suffering from the state space explosion problem, reachability analysis is useful when applying to small- and medium-scale systems which need high reliability such as submodule of bank systems, since it produces no false positive or false negative, and reports the deadlock paths. Over the past few decades, a number of optimization techniques have been proposed to make it applicable for larger programs. Structure-level approaches attempt to reduce the size of the net by reduction \cite{8}–\cite{10}, abstraction \cite{11}, \cite{12} or transformation \cite{13}, \cite{14} techniques. The most famous efforts are six well-known PN reduction rules \cite{8}, which reduce the net while preserving concerned properties (boundedness, liveness and reversibility). \cite{15} proposed three new reduction rules for PN which have tokens. Tokens represents resources and the entry of programs, so this approach can be applied to reduce the beginning of programs. However, these rules are for general PN, and have no obvious effect for a particular language-based PN. Language-level approaches take language features into consideration, thus proposing language specific optimization methods. For Ada language, \cite{16} derived ten reduction rules among which four are for general PN and six are for Ada Net (PN for ada programs) specifically. And this approach reduced Ada Net drastically, thus accelerating the deadlock detection of Ada programs.

Mutual exclusion is a common synchronization mechanism which refers to the requirement of ensuring that no two concurrent processes are in their critical section at the same time, such as pthread_mutex_lock in POSIX and synchronized in Java. Many techniques are for mutual exclusion-based deadlock detections, such as \cite{2}, \cite{17}, \cite{18}. Mutual exclusion-based deadlock detection does not need to consider the diversity of synchronous mechanisms (such as condition variable, barrier, etc.), so that we can customize spe-
cific accelerating methods for it.

Generally speaking, reachability analysis on PN consists of two steps: net conduction and reachability graph traversal. The present research involves both steps. Inspired by existing net reduction techniques, in this paper, a mutual exclusion-based reduction rule is proposed to reduce the PN, customizing for mutual exclusion-based concurrent programs. Further more, a pruning strategy, customized for unlock of mutual exclusion, is proposed to accelerate the traversing of the reachability graph.

Furthermore, according to the definition of PN, the reachability analysis on PN can only report one deadlock on each path. That is because successor markings which are blocked by the deadlock are unaccessible. However, this deadlock may be a false alarm which are introduced by other procedures (such as the information abstraction procedure). In these situations, real deadlocks may be hidden. To overcome this limitation, we proposed a deadlock recovery algorithm.

To verify the effectiveness of our methods, a prototype is implemented and applied to SPLASH2 benchmarks. The experimental results demonstrate that our strategies accelerate the reachability analysis on PN for mutual exclusion-based deadlock detection significantly.

The remainder of the paper is organized as follows. Section 2 presents a net reduction rule. Section 3 presents several strategies for accelerating the traversal of the reachability graph. A case study is discussed in Sect. 4, and experimental results are shown in Sect. 5. Finally, Sect. 6 concludes this paper.

2. Petri Net Reduction

This paper follows the basic notations of PN in [8], e.g., a PN is a 4-tuple $PN = (P, T, F, M_0)$ where $P$ is the set of places, $T$ is the set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is the set of arcs, $M_0$ is the initial marking. For $p \in P$, $\bullet p$ is the set of input transitions of $p$, and $p\circ$ is the set of output transitions of $p$. For $t \in T$, $\bullet t$ is the set of input places of $t$, and $\circ t$ is the set of output transitions of $t$.

2.1 Reduction Rule

In the situation of deadlock detection, places can be divided into two categories: Thread places (TP) represent control of threads and Mutual Exclusion places (MEP) represent shared resources. One should note that the division is for easy understanding so that we can summarize the common case and propose net reduction rule accordingly. When searching the reachability graph, all places are equivalent and all transitions are equivalent too. Similarly, although the scale of the net may be tremendous, the whole net is composed of the following four types of transitions.

- **Thread-create transition** creates a new thread. As illustrated in Fig. 1 (a), it takes a TP as input and creates two TPs as outputs. For specific languages, it can be $\text{pthread}_\text{create}$ in POSIX or $\text{run}$ function in Java, etc.
- **Thread-join transition** waits for the finish of a thread. As shown in Fig. 1 (b), it takes two TPs as inputs and creates only one TP as output.
- **Lock transition** locks a mutual exclusion variable. As shown in Fig. 1 (c), a MEP is taken as input. Default mutual exclusion variables are reentrant in Java which can be occupied repeatedly by one thread, while are non-reentrant in POSIX which can only be occupied once. To distinguish the two, some additional information is required and it is out of scope of this paper. So here we only consider reentrant mutual exclusion variables since they are more popular.
- **Unlock transition** unlocks a mutual exclusion variable. As shown in Fig. 1 (d), a TP and a MEP are taken as inputs, and a TP is created as output.

Before the declaration of the reduction rule we denote all transitions as $T$ and all places as $P$.

**Rule 1** (Chain-Fusion of Transitions): A subset $G$ of $T$ is chain-fusion with another subset of transitions iff there exists a place $m$ such that the following three conditions are satisfied:

1. $\forall f \in G, | \bullet f | : | f \circ | = 2$.
2. $\forall f_1, f_2 \in G, \exists t_1 \in G$ and $t_2 \in G$ and $1 < i < k, [t_i\bullet] \cap [\bullet t_{i+1}] \neq \emptyset$ and $[f_1\bullet] \cap [\bullet t_i] \neq \emptyset$ and $[t_i\circ] \cap [\bullet f_1] \neq \emptyset$.
3. $\exists m \in P, \forall f \in G, m \in [f\circ] \cup [f\bullet]$.

Expressing these formulas with understandable sentences, the first one indicates that all transitions in $G$ have one input place two output places or two input places one output place; for the second condition, all transitions in $G$ are linked in a chain; the third means that all transitions in $G$ take a shared place $m$ as input or output. Undirected edges
refer to a nondeterminacy that both directions of each edge are possible.

In order to facilitate understanding, the graphical representation of Rule 1 is shown in Fig. 2. The pivot of the rule is to fuse multiple continuous lock and unlock transitions. We can prove the correctness by demonstrating that the impact of the chain to other threads remains unchanged after reducing. Since the chain involves no other shared places, it can be replaced by a brief subnet which only impacts m. If lock transitions are more than unlock transitions, the terminal state of the chain is locking m for i\(\times j\) times. On the contrary, if lock transitions are less than unlock transitions, we cannot remove all lock transitions because the first lock transition may result in a deadlock. The input place (I) and output place (O) of G remain unchanged, thus the reduction will not impact the properties (i.e. reachability and liveness) of other parts of the net.

Note that transitions and places in Fig. 3 are not marked with the category name, this is because the rule is applicable to deal with the reachability analysis not only in mutual exclusion-based PN, but also in general PN.

2.2 Reduction Algorithm

After showing the rule, the next step is to discuss the concrete algorithm of applying the rule and how the rule operates with other reduction rules proposed by other researchers. PN models can be represented in at least two ways: graphical and matrix representation [8]. Here we design a PN as an incidence matrix, where each line represents a place (denoted by \(p\)), each column represents a transition (denoted by \(t\)), and the value of elements can be 1 when \((p, t) \in F\), or -1 when \((t, p) \in F\), or 0 otherwise.

Rule 1 can be represented as a pattern for the matrix representation of the net. The reduction algorithm looks for a line that has more than one nonzero values and then verify if each column where these values belong has just one other “1” value (denoted by \(x\)) and one other “-1” (denoted by \(y\)) value. If so, the algorithm puts all \(x\) in a set \(X\) and puts all \(y\) in a set \(Y\), and then calculates the difference set \(Z = X - Y\). If there are only two elements in \(Z\), the pattern of Rule 1 is matched and then applying corresponding reduction to the pattern.

We will encounter a combinatorial problem when multiple reduction rules are available, since different reducing order may lead to different results. Some techniques have been proposed to resolve this problem. [19] randomly selects one element in PNs and verifies if any characteristic at some point near the selected element enables some reduction rules. Inspired by evolutionary computing, [10] proposes an automated reduction scenario which is modelled as Genetic Algorithm (GA). In the GA algorithm, PNs are forced to receive some genetic variability using some reduction rules, achieving diverse possibilities of crossover and based on that improve the convergence rate of the algorithm.

3. Accelerating Reachability Analysis

After showing the net reduction rule that leads to a reduced net, the next step is to introduce some strategies to accelerate the reachability analysis on the reduced net and how these strategies work together. Notably, in PN, the nodes of a reachability graph are markings actually, so we will use the term marking instead of node in the following.

3.1 Marking Compression

In PN, a marking indicates how many tokens each place holds, and is often denoted by a place vector (Fig. 4). When traversing the reachability graph, it is necessary to define how to represent a marking efficiently. To solve this problem two compression methods are presented in this section separately, namely byte compression and sparse vector compression. They are applicable in different situations depending on the net structure and the initial marking.
3.1.1 Byte Compression

Each place in general PN can hold more than one token, therefore should be presented by a 8-bits character at least in practical programming language. However, in the context of mutual exclusion-based PN, each place can only hold one token at most, because of the semantics of mutual exclusion, thus rendering it could be presented in a 1-bit boolean. The compression procedure is illustrated in Fig. 4 where each character C corresponds to eight boolean Ps, and zeroes after $P_N$ is used for supplying when N is not a multiple of 8. Obviously, volume of compressed vector is $M = \lceil \frac{N}{8} \rceil$.

3.1.2 Sparse Vector Compression

Mutual exclusion variables are used to synchronize multiple threads. To do so, all concerned thread must operate (i.e., lock or unlock) the same mutual exclusion variable for more than one time. Therefore MEPs only take a small part of all places, thus rendering the status vector is sparse. Token vector is defined to meet this requirement. Elements in the vector represent places which hold a token. Since there are totally N places, an element $s_i$ in T-vector can be stored in $\lceil \log_2 N \rceil$ bits.

**Definition 1** (token vector): A token vector T-vector comes from a place vector $(P_1, P_2, \ldots, P_N)$. T-vector = $(s_1, s_2, \ldots, s_k)$, where $P_s = 1 (k \geq 1, 1 \leq i \leq k)$

To compare the space occupation of these two compression methods, Eq. (1) is considered. The variable avg represents the average count of tokens.

$$N = \text{avg} \cdot \lceil \log_2 N \rceil \Rightarrow \text{avg} = \frac{N}{\lceil \log_2 N \rceil}$$

As a conclusion, when the average count of tokens in the net is larger than avg, byte compression method has a better compression ratio, otherwise, sparse vector compression method is better.

3.2 Pruning

Reachability graph contains all possible markings some of which are related to deadlocks and some are not. In traditional reachability analysis, after visiting each marking, all its enabled successors will be developed following. However, many of these successors are unnecessary, especially when the current transition is safe. The term safe transition means a transition, such as unlock transition, which is still enabled even if another transition is chosen. For easy understanding, an equivalent fact is that unlock operation can never be blocked by other threads in concurrent programs. Accordingly, we denote this pruning strategy as unlock pruning.

To prove the correctness of unlock pruning, supposing that the current transition t is an unlock transition and the consequent marking of current marking ($M_0$) is $M_1$ (by firing t). As the definition of PN, $M_0[\alpha]$ denotes all child markings of $M_0$ which can be divided into two categories: markings for which t has been fired and markings for which t has not been fired. For markings in the former category, there must be equivalents in $M_1[\beta]$ where $\alpha - \beta = t$. Alternatively, markings in the latter category must not be a terminal marking because the definition of safe transition determines t is enabled for them, and when firing t they are also equivalent with those in $M_0[\alpha]$. Child markings of $M_0$ are equivalent with child markings of $M_1$. Therefore all other successors of $M_0$ can be pruned and firing t is enough.

The pruning algorithm is shown in Algorithm 1. Generally speaking, it contains two pruning strategies (repetition pruning and unlock pruning) and a call to fire the chosen transition. As a prerequisite, StatusSet collects all before seen markings in a compressed format, and all current enabled transitions are kept in EnabledTransSet. Marking $M$ is initially compressed into status using the compressing technique mentioned above (LINE 1). And then the algorithm prunes the search space by excluding markings which have reached previously (LINES 2-6). The pruning for unlock transitions corresponds to a while loop (LINES 7-9). After these two pruning stages, t is either empty or an unsafe transition. Finally, t is fired and M is updated simultaneously (LINE 10).

<table>
<thead>
<tr>
<th>Algorithm 1 Pruning for reachability analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> current transition t, current marking M, a hash set StatusSet which contains all status ever reached, a set EnabledTransSet which contains all current enabled transitions</td>
</tr>
<tr>
<td><strong>Output:</strong> the marking M after firing an unpruned transition</td>
</tr>
<tr>
<td>1: status $\leftarrow$ compress(M)</td>
</tr>
<tr>
<td>2: while status $\in$ StatusSet do</td>
</tr>
<tr>
<td>3: t $\leftarrow$ next transition from EnabledTransSet</td>
</tr>
<tr>
<td>4: M[t] $\leftarrow$ M</td>
</tr>
<tr>
<td>5: status $\leftarrow$ compress(M)</td>
</tr>
<tr>
<td>6: end while</td>
</tr>
<tr>
<td>7: while t is an unlock transition or t is a lock transition and next transition is paired unlock do</td>
</tr>
<tr>
<td>8: t $\leftarrow$ next transition in the same thread with t from EnabledTransSet</td>
</tr>
<tr>
<td>9: EnabledTransSet $\leftarrow$ EnabledTransSet $\cup$ {t}</td>
</tr>
<tr>
<td>10: M[t] $\leftarrow$ M</td>
</tr>
</tbody>
</table>

3.3 Enabled Transitions

In reachability analysis, a very time-consuming operation is to obtain all subsequent transitions whenever a transition is fired. The most straightforward solution is to traverse all transitions in the net, to find out who are enabled. But it only applicable to small nets.

Inspired by the fact that there will be no leap between two unrelated nodes when traversing the graph, an improved strategy is to update markings of related transitions after firing a transition. Figure 5 illustrates this algorithm. Initially, $t_1$ and $t_2$ are the sole two enabled transitions since $p_1$ and $p_2$ hold a token respectively. As one choice, $t_1$ is chosen to be fired. After firing $t_1$ our algorithm will check all transitions
which take $p_0$ (input place of $t_1$) as input. Since no transitions other than $t_1$ is qualified, $EnabledT ransS et$ will remain unchanged. Then it will check transitions which take $p_1$ (output place of $t_1$) as input. Since $t_1$ is enabled right now, $t_1$ will be added into $EnabledT ransS et$. Using this algorithm, a total of two transitions are checked other than five (all) transitions.

3.4 Deadlock Recovery

The purpose of deadlock detection is removing all real deadlocks. However, the fact is that each round of reachability analysis reports only one deadlock if multiple real deadlocks exist. The total detection time is $T \times R$ where $T$ refers to the time consuming per round and $R$ refers to the count of rounds. The preceding sections introduce some techniques to reduce $T$, and this section focuses on how to report as many as possible deadlocks in one round of reachability analysis, i.e., reducing $R$. Furthermore, in the context of static detection, the reported deadlock may be an unresolvable false alarm caused by the imprecise preprocessing, thus preventing real deadlocks being reported. To solve these problems, we proposed a deadlock recovery mechanism, by which the deadlock would be removed, so that other deadlocks in this reachability path can be detected.

The deadlock recovery algorithm is shown in Algorithm 2. A blocked transition is recovered by force in two steps: occupying all its input places (LINE 3-5), and then releasing all its output places (LINES 6-8). Then, the recovered transition is marked as “forced”. The marking is utilized to distinguish normal transitions and recovered transitions. In the final report, recovered transitions should be highlighted to users because the validity of other deadlocks in the same path depends on whether the recovered one is a false alarm. After that, the recovered transition, which is accessibly already, should be deleted from the waiting transition set (LINE 11). Finally, the recovered transition is inserted into the tail the current reachability path.

The situation without reporting deadlock refers to a legal exit of the program (in simulation), at which point other paths should be analyzed. Re-analysis from the beginning is not an ideal solution, as it will imply much repetitive work. For the sake of efficiency, the controller would revoke recent invoked transitions in reverse order to the nearest bifurcation point where undeveloped branches refer to different paths.

4. Case Study

In this section, a case study is presented to demonstrate our optimization methods. One aim of this case study is to illustrate how the net reduction rule, described in Sect. 3, works and how is the reduction efficiency. Furthermore, the case study also highlights the expressive performance of our pruning strategies for reachability analysis.

**Thread 1 (t1):**

```plaintext
1 entry () {
2 lock (A) ;
3 lock (A) ;
4 unlock (A) ;
5 unlock (A) ;
6 lock (A) ;
7 lock (B) ;
8 unlock (B) ;
9 unlock (A) ;
10 }
```

**Thread 2 (t2):**

```plaintext
1 entry () {
2 lock (B) ;
3 lock (A) ;
4 unlock (A) ;
5 unlock (B) ;
6 lock (A) ;
7 lock (B) ;
8 unlock (B) ;
9 unlock (A) ;
10 }
```

The case includes two threads (denoted by $t1$ and $t2$). Variable A is a recursive mutual exclusion variable, thus rendering it can be occupied for multiple times in $t1$.

The first step of reachability analysis is modeling the programs using PN. The constructed net is shown in Fig. 6, including twelve transitions and sixteen places. As highlighted in the net, first five transitions of $t1$ are all related to
A (lock or unlock). It fits the prerequisite of Rule 1. Then the net reduction rule is applied to the net resulting in a reduced net in Fig. 7 which contains only eight transitions and twelve places.

If we store markings without compression and traverse the reachability graph without pruning the graph would be tremendous. The graph is shown in Fig. 8 where each marking is represented by twelve numbers corresponding to twelve places and there are totally more than ten markings. For reasons of space, the figure omits many markings.

Applying our accelerating strategies, the optimized reachability graph is shown in Fig. 9. Each node is represented by two numbers (using Sparse vector compression). As highlighted in the figure, the unlock pruning strategy is activated in node (1,4) and (0,5). Their successors are therefore pruned. Furthermore, if net reduction rule is not applied the reachability graph would contains more than one hundred nodes even if the case is so small.

5. Experimental Results

To evaluate the impact of our methods on the procedure of reachability analysis, a prototype is implemented and applied to SPLASH2 benchmarks [20]. SPLASH2 suite is the second release of the Stanford Parallel Applications for SHared-memory, a set of parallel applications for use in the design and evaluation of shared-memory multiprocessor systems, all written in C language. The SPLASH2 suite contains benchmarks from several domains including high-performance computing, signal processing and graphics. Since the number of threads in SPLASH2 is variable, we modify the number of threads in these benchmarks for a fixed number of 3 to conduct the evaluation. The case study was conducted on a Linux CentOS machine with eight 2.93GHz Intel(R) Xeon(R) X5670 processors and 16 GB RAM.

5.1 The Performance of Net Reduction Rule

In order to measure the reduction quantitatively, many evaluations have been proposed to represent the scale of a PN [10]. Here we measure the scale by the count of places and transitions respectively.

The effect of net reduction rule is shown in Fig. 10. As can be seen, the optimization ratio of place and transition in these benchmarks are almost synchronous. This is explained by the fact that a mutual exclusion-based PN consists of only four kinds of transitions, and the deletion of each of them reduces the same number of places and transitions. The ratio is therefore similar for place and transition.

As shown in the figure, the reduction ratio varies due to the benchmarks. The max ratio is the ratio of radiosity while the minimum is of three benchmarks. The reduction ratio of radiosity reaches 2.8 which means that more than half net has been reduced. On the other hand, the reduction ratio of barnes, ocean_cp and ocean_ncp is 1, that is because these three benchmarks do not conform to the reduction rule, and therefore no reduction is applied. The reduction ratio of

---

Fig. 7 Reduced net of the case study.
Fig. 8 The reachability graph without compression and pruning.
Fig. 9 The reachability graph after compression and pruning.
Fig. 10 Effect of PN reduction using Rule 1. The quantities of both places and transitions are measured. The optimization ratio is the quantity of elements in the original net dividing the quantity in reduced net.
the other four benchmarks is greater than 1 and less than 2. They are reduced more or less and the reduction will reflect the reachability analysis significantly.

5.2 The Performance of Reachability Pruning

This section will show the impact of our strategies to speed up reachability analysis. Because all strategies put forward in this paper are for accelerating the analysis. So we should construct a performance comparison between the reachability graph of the original net and the graph after applying all proposed strategies. However, these benchmarks are all middle-scale programs each of which contains at least thousands of lines of codes. Their nets consist of hundreds of places and transitions, thus rendering the analysis on the original net is insoluble. Therefore, we construct the comparison experiment on the reduced net. Notably, the actual ratio is higher.

The comparison result is shown on Table 1. Numbers in column no prune represents the quantity of nodes in the reachability graph where no pruning strategy is applied. For comparison, repeat status pruning is applied and the result is listed in column R prune. On the base of repeat status pruning, unlock pruning is applied and the result is listed in column RU prune. Column RU/no means the number in column RU prune divides the number in column no prune, so it represents the acceleration of our strategies to the original method. Column RU/R is the number in column RU prune divides the number in column R prune, to represent the acceleration ratio of our strategies to repeat status pruning method.

From the table, it obvious that our strategies obtained a great acceleration ratio when establishing a comparison with the no pruning method. The max ratio is as high as 15111, this result means pruning is essential to reachability analysis.

In reachability analysis, multiple pruning strategies work collaboratively. In order to get the effect of unlock pruning, we compared the scale of reachability graph (quantity of markings) when only repeat status pruning is applied with the scale when both repeat status pruning and unlock pruning are applied. From the column RU/R we can see that unlock pruning strategy obtained a prominent acceleration ratio. The acceleration ratio is at least 1.33 and the highest ratio is prominently 13.28. This results seem very promising when consider the necessity of great acceleration on mid- and large-scale nets in order to analyze their reachability graphs. The radiosity benchmark, whose RU/R is 13.28, shows a different trend in Table 1 while the RU/R is under 4 for other benchmarks. To account for this exception, we recorded the hit rate of the unlock pruning strategy, finding that the rate of radiosity arrived at 99.1%. As Algorithm 1 presented, unpruned transitions are neither unlock nor lock whose successor is paired unlock. It means unpruned transitions are non-nested locks. The structure of radiosity confirmed the high hit rate that contains only seven nested lock patterns.

Note that, our methods are optional for the reachability analysis since the input and output are the same individuals with the same properties. Therefore, they are compatible with all other optimization methods, and the usage of them will not lead to an additional increase of the volume of the reachability graph.

6. Conclusion

In the area of mutual exclusion-based programming, PN is simplified into a combination of four kinds of transitions. Accordingly, a net reduction rule for this area is proposed, maintaining all deadlock-related properties. Experiments on a case study and SPLASH2 benchmarks demonstrate that the reduction rule obtains good results. After reducing the net, the subsequent procedure of deadlock detection is traversing the reachability graph. Two compression methods are devised aiming at decreasing the space occupied by each marking in the graph, two pruning strategies are proposed to accelerate the traversal, and a deadlock recovery algorithm is proposed to improve the efficiency of single reachability path. The experimental results show that these strategies accelerated the reachability analysis notably. In particular, the acceleration ratio of unlock pruning is at least 1.33 individually. Furthermore, using all proposed methods
simultaneously, the analysis can be accelerated significantly.

References


Yunkai Du received his B.E. from University of Science and Technology of China in 2011. Currently, he is a Ph.D. student at the School of Computer Science and Technology of China, University of Science and Technology of China. His research fields focus on the vulnerability of concurrent programs, especially on deadlock and data race detection issues.

Naijie Gu received his B.E. degree in Mathematics from University of Science and Technology of China in 1983, and his M.E. and Ph.D. degrees in Computer Science and Technology from the University of Science and Technology of China, in 1989 and 2005 respectively. He is currently a professor in the School of Computer Science and Technology, University of Science and Technology of China. His research interests includes parallel computing, parallel architecture and software vulnerability.

Xin Zhou received the B.E. from the Department of Communication and Electronics, Jiangxi Science and Technology Normal University in 2006. In 2006-2008, he was an software engineer at Tung Thih Electronic Co., Ltd. He received the M.E. from the Department of Information Engineering, Hiroshima University in 2013. Currently, he is a Ph.D. student at the Department of Information Engineering, Hiroshima University.