Explicit Storage and Analysis of Billions of States using Commodity Computers

Yin Wang * **Jason Stanley** ** **Stéphane Lafortune** **

* *Hewlett-Packard Laboratories, Palo Alto, CA, USA (e-mail: yin.wang@hp.com)*
** *Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, USA (e-mail: {jasonsta,stephane}@umich.edu)*

Abstract: The objective of this paper is to develop a framework and associated algorithms for explicit state space exploration of discrete event systems that can scale to very large state spaces. We consider classes of resource allocation systems (RAS), where a set of resources are shared by concurrent processes. In particular, we focus on Gadara RAS, whose Petri net representations have recently been used for liveness enforcement in multithreaded software. We present a framework where each reachable state of the RAS is represented by a single bit. We show how single-bit representations can lead to efficient implementations of supervisory control algorithms. In order to support single-bit state representations, we develop two indexing functions that map each state to a unique integer that serves as the corresponding index of the state in the large bit array. These functions exploit the invariants of the given RAS. Experimental results show that our techniques scale up to exploration and analysis of billions of states on commodity computers.

Keywords: Resource Allocation Systems, Petri nets, Supervisory Control, State Space Exploration

1. INTRODUCTION

A Resource Allocation System (RAS) consists of a set of resource types and a set of process types (Reveliotis, 2005). Each process type has multiple stages, organized according to the execution logic. Each stage requires a combination of resources to complete. A state of an RAS consists of one or multiple instances of different process types running at different stages. A taxonomy of various types of RAS is presented in Reveliotis (2005), based on the process execution logic and the resource allocation scheme. Petri nets are often used to represent RAS. There are various classes of Petri nets defined for different types of RAS; see (Ezpeleta and et al, 1995, 2002; Liao and et al, 2012a). The dynamics of an RAS can be captured by a finite state automaton (Nazeem and Reveliotis, 2011), which is equivalent to the reachability graph obtained from the Petri net representation. While the study of RAS originated in manufacturing systems, theoretical results on RAS have recently been applied to computer systems (Wang and et al, 2009).

The objective of this paper is to develop a framework and associated algorithms for efficient state space exploration of RAS that can scale to very large state spaces. The necessity of performing state space exploration of RAS occurs in several problem contexts, such as liveness enforcement. Liveness enforcement has been a central research theme in the RAS literature in the past two decades. Due to the state explosion problem, early solutions focused on structural properties using the Petri net representation in order to avoid state space exploration. These solutions typically sacrifice maximal permissiveness for computational efficiency. The survey paper Li et al. (2008) mentions that the only maximally permissive solution is based on the *Theory of Regions* (Ghaffari and et al, 2003), which requires exploring all reachable states, and then uses linear programming to synthesize a control place for each unsafe state. With the complete knowledge of the reachable state space, an alternative solution recently proposed in Nazeem and Reveliotis (2011) uses decision trees to classify safe and unsafe states and obtain a maximally permissive solution. For a special class of Petri nets called *Gadara nets*, structural analysis can be used to obtain a maximally permissive solution by bookkeeping unsafe states in the form of *coverings* (Liao and et al, 2012b), which can be proportional to the number of reachable states. Since determining whether a state is safe or not is an NP-complete problem even for one of the simplest classes of RAS (Reveliotis, 2005), an exponential complexity algorithm is unavoidable for provably achieving maximally permissive control (unless P≡NP).

Model checking methods have been applied to systems with astronomical numbers of states (Burch and et al, 1992). These methods do not store each state explicitly, but instead rely on state reduction and compression techniques such as *partial order reduction* and *symbolic state representation* to achieve scalability. These techniques have been applied to generating Petri net state

[1] This work was partially supported by NSF grant CCF-0819882 and an award from HP Labs Innovation Research Program.

spaces (Wolf, 2007). However, the effectiveness of state reduction and compression heavily depends on the redundancy and symmetry present in the system. When applied to "minimalist" RAS, symbolic model checking scaled up to only a few millions of states in our previous study (Wang and Wu, 2003). Coincidentally, other application domains report scalability limits in the million state range as well (Cimatti and Roveri, 2000; Jhala and Majumdar, 2009). The computation time is almost always proportional to the number of states.

It is well known that memory space rather than computation time is the bottleneck for explicit state enumeration (Wolf, 2007). The fundamental challenge for efficient storage is the data structure used to store and locate a state. The latter is needed in order to skip states already visited during the search process. One common technique is to use a conflict-free hash function to store each state in a compressed form (Wolf, 2007). Known hash functions are typically very inefficient for RAS since they are designed for generic Petri nets. In addition, these functions are designed for efficient compression, not uncompression. The latter is not needed in reachability-related analysis. But for control synthesis with *partial controllability*, restoring a state from its hash value is necessary to calculate the *supremal controllable nonblocking* sublanguage; see (Cassandras and Lafortune, 2008).

The alternative and novel state exploration solution presented in Nazeem and Reveliotis (2011) exploits features of RAS and avoids state hashing. Each resource type defines an *invariant*. States that satisfy all invariant constraints subsume all reachable states, and typically not by much. Based on this observation, Nazeem and Reveliotis (2011) propose to enumerate all solutions of invariant equations, sort them, and use binary search to locate a state during state space exploration. However, we have observed that this approach does not scale up to a billion states because it needs to explicitly store all states satisfying invariant constraints. For example, using the random RAS generator that we have constructed, a system of a billion reachable states typically has more than 200 stages. The number of states satisfying invariant constraints is typically twice of the reachable states, i.e., two billion. These systems have unit resource capacity so each stage has at most one process instance. Therefore we can use 200 bits to represent a state, or 25 bytes. The memory space for two billion sates is 50 GB. For general *Disjunctive/Conjunctive* (D/C)-RAS with larger resource capacities, terabytes of memory are needed if we use a 32-bit integer to record the number of instances at each stage. External storage is not a viable option either due to its high latency. For example, a NAND-based Solid State Disk has a read latency of at least $25\mu s$ due to its physical characteristics. Assuming zero read time, locating a state in a billion entry array requires around 30 accesses, or $750\mu s$. It would therefore take 208 hours to explore one billion states even if we query each state just once. Using mechanical disks would take years. For maximally permissive control solutions, experiments reported in the literature are typically limited to a few million states or less (Li et al., 2008; Nazeem and Reveliotis, 2011; Nazeem and et al, 2011; Liao and et al, 2011).

In this paper, we present a framework and carefully crafted state manipulation algorithms that can explicitly store and analyze billions of states efficiently using a commodity computer. Our proposal is based on the idea of using a *single* bit to represent a state, which we show can lead to efficient implementations of state exploration and supervisory control algorithms. The key difficulty of this single-bit representation is the indexing function that maps each state to a unique integer serving as the *index* of the state in the large bit array. Exploiting the invariants of certain classes of RAS, we develop two indexing functions with different space-time tradeoffs. In addition to these novel indexing functions, our implementation applies numerous optimization techniques. Most notably, we implement multithreading to achieve almost linear speedup in a multicore computer. Under this bit-representation framework, we further implement the basic supervisory control algorithm that calculates the supremal controllable nonblocking subset of states with respect to partial controllability. When applied to randomly generated RAS with a billion states, our program finishes state exploration and control synthesis in a few hours using less than 5GB of memory on an iCore 7 desktop.

The ability to analyze and control RAS in the billion states range is a novel contribution of this paper, as we are not aware of other methods for synthesizing maximally permissive solutions that can handle such large state spaces in a reasonable amount of time using commodity computers. Our tool, along with the random Gadara RAS generator that we use in this paper, are available open source at Gadara (2012).

This paper is organized as follows. We start with a description of the RAS model and a presentation of the basic search algorithm in Section 2. We then present our main results regarding the single-bit state representation and associated mapping functions in Section 3. Section 4 gives a brief description of our implementation, while Section 5 presents experimental results that demonstrate the scalability properties of our approach.

2. SYSTEM MODEL AND SEARCH ALGORITHMS

2.1 Resource Allocation Systems

Definition 1. (Reveliotis, 2005) A Resource Allocation System is defined as a 4-tuple $\Phi = \langle R, C, \mathcal{P}, D\rangle$ where:

(1) $R = \{r_1, ..., r_m\}$ is the set of resource types.
(2) $C : R \to \mathbb{Z}^+$ defines the capacity of each resource type, which we abbreviate as $C(r_i) \equiv C_i$
(3) $\mathcal{P} = \{\Pi_1, ..., \Pi_n\}$ is the set of process types. Each type Π_j is a composite $\langle \mathcal{S}_j, \mathcal{G}_j\rangle$, where $\mathcal{S}_j = \{\Xi_{j1}, ..., \Xi_{j,l_j}\}$ is the set of processing stages, and $\mathcal{G}_j$ is the sequential logic that governs the execution of any process instance of type Π_j.
(4) $D : \bigcup_{j=1}^{n} \mathcal{S}_j \to \prod_{i=1}^{m}\{0, ..., C_i\}$ is the resource allocation function associating every processing stage Ξ_{jk} with an $m-$dimensional resource allocation vector $D(\Xi_{jk})$ required for its execution.

We aggregate all processing stages of all process types, and order them sequentially as $\Xi_1, ...\Xi_\xi$. A state of RAS Φ is then a ξ-dimensional vector s, where each component

$s[k], k = 1, ..., \xi$ represents the number of process instances at the corresponding processing stage. The event set of the RAS includes loading a new process instance to the corresponding first stage, advancing an existing instance to the next stage, and unloading a finished instance from its last stage. With the above state representation and event set, the dynamics of RAS Φ can be captured by a finite state automaton $G(\Phi)$. We refer the reader to Nazeem and Reveliotis (2011) for the detailed definition of $G(\Phi)$. The size of RAS Φ is $|\Phi| = |R| + \xi + \sum_{i=1}^{m} C_i$. The number of states of automaton $G(\Phi)$ grows exponentially with $|\Phi|$, the so-called state explosion problem. Our goal in this paper is to enumerate and analyze all the states of this automaton.

Each resource type defines an *invariant*, i.e., for each state of the automaton, the number of units occupied by all running process instances plus the vacant units must be equal to the capacity of that resource type. Therefore, every state must satisfy the following inequality:

$$\forall i = 1, \ldots, m, \sum_{k=1}^{\xi} s[k] \cdot D(\Xi_k)[i] \leq C_i \qquad (1)$$

Using the terminology of supervisory control theory for regular languages represented by automata (see, e.g., Cassandras and Lafortune (2008) for formal definitions), a state is *accessible* if it can be reached from the initial state $s_0 = [0, ..., 0]$, where there is no process instance running. A state is *co-accessible* if it can reach $s \in S_M$, where S_M is the set of *marked* states; in RAS as above, the initial state s_0 is usually the only state that is marked. A deadlock state is a state without a successor. An RAS is live if every accessible state is co-accessible. An event of an RAS can be either controllable or uncontrollable.

Numerous classes of RAS have been proposed in the literature (Reveliotis, 2005). There are also many classes of Petri nets that capture different types of RAS. We present our ideas and results using a special class of RAS that arises from the modeling of multithreaded software, called Gadara RAS. Its precise definition is given in Nazeem and et al (2011), and its definition in Petri net form, called Gadara net, is given in Liao and et al (2012a). Here, we summarize the features of Gadara RAS/nets relevant to the design of our algorithms.

Assumption 1: The execution logic of each process type P_j corresponds to a connected digraph where there is a one-to-one mapping between the node set and the stage set $\mathcal{S}_j$. Every execution of an instance follows a path on the graph and there are no "AND-fork/join" stages.

Assumption 2: $\forall i = 1, \ldots, m$, $C_i = 1$, and $\forall k = 1, \ldots, \xi, D(\Xi_k) \in \{0,1\}^m$.

Assumption 2 implies that each processing stage has at most one process instance. Therefore each state is a binary vector in $\{0,1\}^\xi$. This is because in multithreaded software the resources are mutual-exclusion (*mutex*) locks that can be acquired by one thread at a time. The *controlled* variant of Gadara RAS (cf. Liao and et al (2012a)) allows additional resource types that need not be unit capacity. It can allocate more than one unit to a processing stage too, much like a *semaphore* in multithreaded programs. However, each processing stage requires at least one resource type of unit capacity, so the state is still a binary vector. Finally, the class of *Disjunctive/Conjunctive* (D/C)-RAS removes assumption 2 completely. We extend our solution to Controlled Gadara RAS and D/C-RAS in Section 3.4.

Algorithm 1 Generic state search algorithm (I, F)

```
Input: starting state set $I$, forbidden state set $F$
Output: All states explored from $I$ are flagged
1: $queue = I$, flag all states in $I$
2: while $queue \neq \emptyset$ do
3:     $s = queue$.remove()
4:     for all $s' \in s$.neighbor(), $s' \notin F$, $s'$ not flagged do
5:         $queue$.add($s'$), flag $s'$
6:     end for
7: end while
```

2.2 Generic State Search Algorithm

Algorithm 1 is the generic state search algorithm. It iteratively explores unflagged neighbors of the head of the queue, adds them to the queue, and then flag them. If the queue is first-in-first-out, e.g., a double queue, the algorithm is breadth-first search. If it is first-in-last-out, e.g., a stack, the algorithm is depth-first search. This generic search algorithm is instantiated into the following three operators on automata.

Accessibility `acc`$(\{s_0\}, F)$ calculates the set of states reachable from the initial state s_0 through a path that does not intersect F, the set of forbidden states. In this case, s.`neighbor()` is instantiated to calculate all *successor* states of s, i.e., forward search. If we start with an empty set of F, the algorithm returns all reachable states.

Co-accessibility `co_acc`(S_M, F) calculates the set of states that can reach some state in the marked state set, S_M, through a path that does not intersect F. In this case, s.`neighbor()` is instantiated to calculate all *predecessor* states of s, i.e., backward search.

Uncontrollable Reach `uncon_reach`$(I, \emptyset)$ calculates the set of states that can reach some state in I through a sequence of uncontrollable transitions, including I. In this case, similar to the co-accessibility calculation, s.`neighbor()` is instantiated to calculate all predecessor states of s, but through uncontrollable transitions only.

2.3 Supremal Controllable Nonblocking Subset of States

We consider a simple form of supervisory control problem, where the control specification is avoiding a set of forbidden states rather than enforcing a regular language. Using the three operators described in the previous subsection, Algorithm 2 shows the iterative algorithm that calculates the *supremal controllable nonblocking* subset of states with respect to partial controllability (see, e.g., (Cassandras and Lafortune, 2008)). When the input is $S_M = \{s_0\}, F = \emptyset$, Algorithm 2 returns the maximally-permissive liveness-enforcing solution for the RAS (Reveliotis, 2005).

3. SOFTWARE DESIGN

We present our main ideas using the class of Gadara RAS described in Section 2.1. The extension to D/C-RAS is

Algorithm 2 Supremal Controllable Nonblocking Subset

```
Input: an RAS of state set $S$ and marked state set $S_M \subseteq S$, a forbidden state set $F$
Output: Allowed state set $A \subset S$, $A \cap F = \emptyset$
1: $i = 0$, $F_0 = F$
2: repeat
3:     $i = i + 1$
4:     $F_i = S \setminus (\texttt{acc}(F_{i-1}) \cap \texttt{co\_acc}(S_M, F_{i-1}))$
5:     $F_i = \texttt{uncon\_reach}(F_i)$
6: until $F_i = F_{i-1}$
7: $A = S \setminus F_i$
```

discussed in Section 3.4. Some of our techniques can be generalized to more complex classes of RAS too, but this is beyond the scope of this paper.

3.1 Representing States by Bits

Memory space instead of computation time is typically the bottleneck for state exploration. A memory-lean implementation of Algorithm 1 requires efficient data structures to store and locate all visited states. A popular idea is to store each visited state in a compressed form (Wolf, 2007). Exploiting features of RAS, we push the idea to the extreme by effectively storing each state as one bit. A one-to-one mapping between the state set and the indices of the bit array is needed for this efficient storage. Since we do not know how many reachable states an RAS has before state exploration, the mapping function has to be conservative, and allocate a bit array large enough to store all reachable states. We call the ratio between the size of the allocated array and the state space size as the *inflation ratio*. There is a tradeoff between the inflation ratio and the computational complexity of the mapping. The next two subsections present two alternative mapping functions that optimize time and space, respectively.

Another advantage of the bit representation is that set operations required for supervisory control in Algorithm 2 can be implemented by logic bit operators that are extremely fast. For example, calculating "AND" of two bit arrays of 10 billion entries takes less than half second using a single thread on a 64-bit 2.26GHz iCore 7 CPU. Figure 1 shows the Java implementation of Algorithm 2 using bit operators, where `BitSet` is the standard JDK class `java.util.BitSet`. Marked bits in `S` represent reachable states, and marked bits in `F` represent forbidden states. After the calculation, marked bits in `S` are safe, while those in `F` are unsafe. We optimize the algorithm at lines 7-9 such that the uncontrollable reach is calculated for newly generated unsafe states only, at each iteration. This greatly reduces the computation time in our experiments where a significant portion of the reachable state space is unsafe. Finally, counting the number of "1"s of a bit array, or *Hamming weight*, is extremely efficient. The function `cardinality()` invoked at line 8 currently takes less than a quarter second to count "1"s in 10 billion bits. The latest SSE instruction set supports hardware Hamming weight calculation, which can further reduce the calculation time.

3.2 Cartesian Product Mapping

For Gadara RAS where each state is a binary vector, one of the simplest mapping functions is to take the decimal value

```
 1: public void supcon(BitSet M, BitSet S, BitSet F) {
 2:   while (true) {
 3:     BitSet acc = acc(F);             //accessible states
 4:     BitSet coacc = co_acc(M, F);        //co-accessible
 5:     acc.and(coacc);      //accessible and co-accessible
 6:     acc.xor(S);                          //unsafe states
 7:     acc.xor(F);              //newly found unsafe states
 8:     if (acc.cardinality() == 0) break;  //no new unsafe
 9:     F.or(uncon_reach(acc));      //uncontrollable reach
10:   }
11:   S.andNot(F);  //supremal controllable nonblocking set
12: }
```

Fig. 1. Algorithm 2 implemented using bit representation

of the binary vector as the index. However, this mapping is impractical for RAS because of its huge inflation ratio. For example, a Gadara RAS of 10 stages needs an array of $2^{10} = 1024$ bits. But if these stages all require one resource type, i.e., they are the support of the resource invariant, there are only 10 states. Based on this observation, we consider the state space defined by the Cartesian product:

$$\|r_1\| \times \|r_2\| \times ... \times \|r_m\| \qquad (2)$$

where $\|r_i\|$ denotes the *support* of the invariant defined by r_i, i.e., set $\{r_i\}$ union the set of stages that require resource type r_i, $\|r_i\| = \{r_i\} \cup \{\Xi_k \mid D(\Xi_k)[i] > 0\}$, or $D(\Xi_k)[i] = 1$ for Gadara RAS. A state $(\Xi_{s_1}, ..., \Xi_{s_m})$ in the Cartesian product space indicates that each resource type r_i is occupied by a process instance at stage Ξ_{s_i}. We include r_i in the support so (2) contains states that have unallocated resources. We slightly abuse the notation here and in the remainder of this section by using Ξ for both stages and resources; note that both are *places* in the Petri net representation of an RAS.

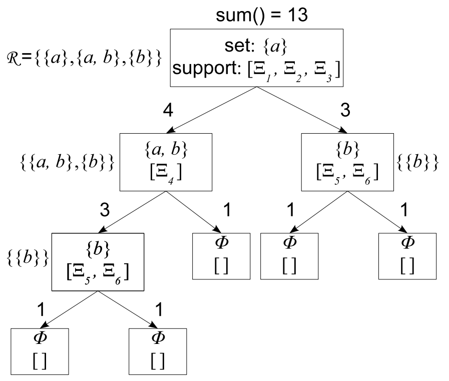
The state space defined by (2) subsumes all states that satisfy the invariant constraint (1), and therefore it includes all reachable states. Our *Cartesian product mapping* scheme works as follows. We allocate one bit for each state defined by (2). Assuming $\|r_i\|$ is an ordered set where r_i is the first element, the Cartesian product naturally defines a total order for the elements in its space, which we employ for the mapping. More specifically, a state $(\Xi_{s_1}, ..., \Xi_{s_m})$ in Cartesian space (2) is mapped to an index integer:

$$I_1 + I_2 \cdot \text{ sizeOf}(\|r_1\|) + I_3 \cdot \text{ sizeOf}(\|r_1\|) \cdot \text{ sizeOf}(\|r_2\|) + ...$$

where I_i is the index of Ξ_{s_i} in $\|r_i\|$, $I_i = 0$ if $\Xi_{s_i} = r_i$, and sizeOf($\|r_i\|$) is the cardinality of $\|r_i\|$. We reverse the calculation to map an index integer back to a state; such a reverse calculation is straightforward to implement.

Example 2. Consider a Gadara RAS with two resource types, denoted by a and b, and six stages, where $\|a\| = \{a, \Xi_1, \Xi_2, \Xi_3, \Xi_4\}$, $\|b\| = \{b, \Xi_4, \Xi_5, \Xi_6\}$.

Applying our Cartesian product mapping to Example 2, we need an array of sizeOf($\|a\|$)·sizeOf($\|b\|$)=20 bits. Each state in $\|a\| \times \|b\|$ is mapped to an index as follows: $(a, b) : 0, (\Xi_1, b) : 1, \ldots, (\Xi_4, b) : 4, (a, \Xi_4) : 5, (\Xi_1, \Xi_4) : 6, \ldots, (\Xi_4, \Xi_6) : 19$. Obviously all pairs $(\Xi_4, ?)$ and $(?, \Xi_4)$, 8 in total, correspond to only one state that satisfies the invariant constraint (1), which is the state containing only one instance at Ξ_4. In general, a state in (2) may not satisfy the invariant constraint (1) if there are stages requiring multiple resource types, i.e., $\|r_i\| \cap \|r_j\| \neq \emptyset$. A variant of the Cartesian product mapping considers the following state space:



state	index
[ ]	0
$[\Xi_5]$	1
$[\Xi_6]$	2
$[\Xi_4]$	3
$[\Xi_1]$	4
$[\Xi_1, \Xi_5]$	5
$[\Xi_1, \Xi_6]$	6
$[\Xi_2]$	7
$[\Xi_2, \Xi_5]$	8
$[\Xi_2, \Xi_6]$	9
$[\Xi_3]$	10
$[\Xi_3, \Xi_5]$	11
$[\Xi_3, \Xi_6]$	12

Fig. 2. Example decision tree and its index mapping

$$\|r_1\| \times \|r_2\| \setminus \|r_1\| \times ... \times \|r_m\| \setminus \left(\bigcup_{i=1}^{m-1} \|r_i\| \right) \tag{3}$$

This space is smaller than (2) but still larger than (1), because it still contains states that have instances at both a stage in $\|r_1\| \cap \|r_2\|$ and a stage in $\|r_2\| \setminus \|r_1\|$. Recalling the above example, we no longer generate pairs of the form $(\bullet, \Xi_4)$, but the state with one instance at Ξ_4 still corresponds to three pairs: (Ξ_4, b), (Ξ_4, Ξ_5), (Ξ_4, Ξ_6). The inflation ratios of space defined by (2) and (3) both grow exponentially in the size of the RAS in the worst case. The computational complexity of state mapping in both directions, on the other hand, is polynomial. Our implementation for the experiments discussed later in the paper adopts (3), and the experiments show an inflation ratio as large as $10^7\times$ when there are billions of states.

3.3 Decision Tree Mapping

The Cartesian product spaces defined by (2) and (3) are both larger than the solution space defined by the invariant constraint (1). The latter is very close to the reachable state space (Nazeem and Reveliotis, 2011). In our experiments so far, the largest inflation ratio of the solution space defined by the invariant constraint (1) is only 2.3× for Gadara RAS. Our *decision tree mapping* scheme enumerates the solution space of (1) exactly using a recursive formula.

First we introduce the concept *exact support*: $[\![R']\!]$ is the set of stages that require exactly set R' of resource types. Formally, $\forall R' \subseteq R, [\![R']\!] = \{\Xi_k \mid D(\Xi_k)[i] > 0 \text{ if } r_i \in R', \text{ otherwise } D(\Xi_k)[i] = 0\}$. For Gadara RAS, $D(\Xi_k)[i] \in \{0, 1\}$. We note that $[\![\{r_i\}]\!] \subset \|r_i\|$. The former is the set of stages that require only r_i, nothing more, and it does not include r_i. The exact supports of different resource subsets do not overlap. Therefore, all stages of a given RAS can be partitioned into exact supports of 2^R, the power set of R. For Example 2, we have $[\![\{a\}]\!] = \{\Xi_1, \Xi_2, \Xi_3\}$, $[\![\{b\}]\!] = \{\Xi_5, \Xi_6\}$, and $[\![\{a, b\}]\!] = \{\Xi_4\}$.

Theorem 3. Given a Gadara RAS, we define function $N : 2^{2^R} \to \mathbb{N}$ recursively as follows.

$$N(\emptyset) = 1 \tag{4}$$

$$N(\mathcal{R}) = N(\mathcal{R} \setminus \{R'\}) + \text{ sizeOf}([\![R']\!]) \cdot N(\mathcal{R} \setminus \{R'' \mid R'' \in \mathcal{R}, R'' \cap R' \neq \emptyset\}) \tag{5}$$

where $\mathcal{R} \subseteq 2^R$, and $R' \subseteq R$ is an arbitrary element of $\mathcal{R}$. Then $N(2^R)$ calculates the total number of states that satisfy (1).

Proof. (sketch) We prove the theorem by showing that $N(\mathcal{R})$ calculates the number of states that: i) satisfy (1), and ii) have process instances only on the exact supports of elements in $\mathcal{R}$, i.e., $s[k] \neq 0$ only if $\exists R' \in \mathcal{R}, \Xi_k \in [\![R']\!]$. When $\mathcal{R} = \emptyset$, the only state that satisfies these two conditions is the initial state, therefore $N(\emptyset) = 1$. When $\mathcal{R} \neq \emptyset$, $\forall R' \in \mathcal{R}$, the set of states satisfying i) and ii) for $\mathcal{R}$ consists of states without any process instance on $[\![R']\!]$, and states with process instances on $[\![R']\!]$. In the latter case, there is exactly one process instance on $[\![R']\!]$ because of Assumption 2. Since this instance takes all resources in R', there is no more process instance on any stage that requires resources in R'. Hence we derive Equation (5).

Based on Equations (4-5), we construct a binary decision tree recursively for state mapping. We begin with $\mathcal{R} = 2^R$ (all of the power set) and the root node picks a set R_{root} randomly from $\mathcal{R}$. At the left child, R_{root} is removed from $\mathcal{R}$, which corresponds to the left term of (5). At the right child, any set that intersect with R_{root} is removed from $\mathcal{R}$, which corresponds to the right term of (5). The recursive process continues until $\mathcal{R}$ is empty. Every left child removes exactly one element from $\mathcal{R}$, while the right child may remove multiple elements. Therefore the depth of the tree is the length of the leftmost branch, which is always $|2^R| = 2^{|R|}$. The element we pick to split each node, including R_{root}, does not affect the tree depth. For a given Gadara RAS, we begin with only resource sets whose exact supports are not empty, instead of the entire power set 2^R.

Figure 2 shows the decision tree for Example 2. The elements `set` and `support` inside each node correspond to R' and $[\![R']\!]$ in equation (5). The detailed data structure is shown in Figure 3a. At the root node, we begin with $\mathcal{R} = \{\{a\}, \{a, b\}, \{b\}\}$. Picking $\{a\}$ for decomposition, both $\{a, b\}$ and $\{b\}$ remain in the left node, while only $\{b\}$ remains in the right node. The process continues until $\mathcal{R}$ is empty, and we add a dummy empty node at each leaf for the convenience of the recursive calculation.

In order to calculate the index for each state, we need to first find how many states each subtree represents, calculated by the `sum()` function in Figure 3a, which implements equation (5). The `sum()` value of the root node is the total number of solutions to the invariant constraint (1). For Example 2, the `sum()` values are shown on top of each node in Figure 2. We cache the value in memory for each node since it is heavily used by the mapping function, discussed next.

The index of a state is calculated by walking down the decision tree, shown as function `indexOf()` in Figure 3b. The function `node.getSupportIndex(s)` at line 7 returns the index in `node.support` if state `s` has a process instance on the corresponding stage, or -1 if `node.support` is not occupied. For Example 2, all 13 stages that satisfy its resource invariant constraints are shown in the table of Figure 2 together with their mapped indices. Here a state is represented by an array of process stages that are occupied by process instances.

Our decision tree is not balanced since every left child removes exactly one element from $\mathcal{R}$, while the right child may remove multiple elements. The length of the left-most branch is the height of the tree, which in the

```
public class TreeNode {
  Collection<Resource> set;
  Stage[] support;
  TreeNode left, right;
  ...
  public long sum() {
    if (set.isEmpty()) return 1;
    else return left.sum()+ support.length*right.sum();
  }
}
```

(a) Data structure of a decision tree node

```
public class Tree {
  TreeNode root;
  ...
  public long indexOf(State s) {
    long result = 0;
    TreeNode node = root;
    while (!node.set.isEmpty()) {
      int i = node.getSupportIndex(s);
      if (i <= 0) {
        node = node.left;
      } else {
        result += node.left.sum()+ i*node.right.sum());
        node = node.right;
      }
    }
    return result;
  }
}
```

(b) Data structure of a decision tree and its index calculation

Fig. 3. Code snippet of decision tree mapping

worst case equals to $2^{|R|}$. Therefore the computational complexity of mapping is exponential in the number of resource types (polynomial in the number of stages). The average complexity depends on the number of resource sets with nonempty exact support. Our experiments expose worst case scenarios because randomly generated RAS tend to have more resource combinations than manually created ones. It remains an open problem whether the number of solutions to the invariant constraint (1) can be calculated in polynomial time. Such a solution would lead to a polynomial mapping function that would also be space efficient.

3.4 Extensions

The extension from Gadara RAS to controlled Gadara RAS (i.e., those with additional non-unit capacity resource types and corresponding non-unity resource allocation weights) does not change any algorithm or data structure discussed so far in this section. The reason is that if a state is given as a set of stages occupied by one process instance each, the remaining units of each added non-unit capacity resource type can be fully recovered from this state representation. The only adjustments needed are the data structure representing a state, and the `neighbor()` function in Algorithm 1 that must consider the availability of these added resources types. Since the full state data structure is only used temporarily during the search process, the extra space needed to store the additional resource types is negligible. *Disjunctive/Conjunctive (D/C)*-RAS have the same process structure as Gadara RAS, but do not require each stage to be associated with a unit capacity resource type. Therefore, a stage can be occupied by more than one process instance, and a state is not always a binary vector. We can extend the Cartesian product mapping to D/C-RAS. In this case, the i-th element in the Cartesion product will still correspond to resource type r_i , but its set will include all possible ways to allocate r_i to its support stages. Extension of the decision tree mapping technique to D/C-RAS is an open issue at this time.

4. IMPLEMENTATION DETAILS

As explained in Section 3, while we store each state as one bit, the full data structure must be used during the search phase in order to calculate predecessor and successor states. Algorithm 1 indicates that these unexplored states are stored in a queue. We discovered that depth-first search (using a FILO queue) results in a queue depth as much as half the size of the reachable state space. Storing half states in their full data structure form in memory would negate all the benefits of bit representation. Interestingly, breadth-first search (using a FIFO queue) reduces the depth to roughly one tenth of the state space size, but it is still too much to store in memory. Alternatively we can store only the neighbor that is going to be explored next in memory. In this case, the set of neighbors has to be recalculated each time we backtrack the search tree, which is a substantial amount of computation. We adopt an interesting idea reported in Wolf (2007), namely, we store only *events* in the queue instead of states. The only state in memory is the head of the queue. Moving up and down the search tree is realized by firing events backward and forward, respectively. Both can be calculated in linear time. Using depth-first search, we still store the search tree in a linear queue. A dummy event is inserted to indicate the boundary of each tree level. Currently, we store each event by a pointer to its data structure, which is 64 bits or 8 bytes in a 64-bit program. We plan to further optimize the computation space by sorting all events in an array and representing each event by its index using a minimum number of bits, e.g., 8 bits is sufficient for a RAS with no more than 256 events.

As we scale to the billion state range using the bit representation, computation time becomes more noticeable than computation space. For example, a billion bits occupy only 119MB of memory but their sequential exploration takes more than a day in our experiments. To mitigate this, we have implemented a parallel version of Algorithm 1, which is a non-trivial engineering task for depth-first search (Reif, 1985). Our idea is to let each thread maintain its own search tree, i.e., running Algorithm 1 separately. The starting state for exploration is stored in a shared work queue, which is initialized to the neighbors of the initial start state. A thread repeatedly grabs a state from the work queue and explores the state space on its own. Synchronization is done by the hardware-assisted command `CompareAndSwap`, which checks and updates a bit atomically. We also implemented a load balancing scheme, where each thread monitors the work queue and replenishes unexplored states from the top of its search tree whenever the queue is empty.

5. EXPERIMENTAL RESULTS

Our experiments are based on Gadara RAS randomly generated using our tool first reported in Wang and et al

(2008), and subsequently used in Nazeem and et al (2011) and Liao and et al (2011). The generator uses a random walk algorithm to decide at each step whether to acquire a new lock or release an existing lock. We have enhanced our tool to generate branches, with a probability configurable as a command line parameter. These branching transitions are not controllable since they are determined by program logic and input in multithreaded software. Because of these uncontrollable transitions, calculating the maximally-permissive and liveness-enforcing subset of states requires the iterative procedure of Figure 1; the typical number of iterations in our experiments was 1-3.

In addition to the two mapping functions discussed in Section 3, we built a more "conventional" state exploration algorithm that does not utilize bit representations for the purpose of performance comparison; it is referred to as method "non-bit" hereafter. This algorithm stores each state using an array of pointers that point to stages with one process instance. We use this representation instead of a binary vector to model each state because the latter is very sparse in Gadara RAS. Storing only pointers to nonzero entries is more efficient. This pointer-based state representation also benefits the calculation of successor and predecessor states. We store all visited states in a hash table in order to locate them efficiently. Hash tables sacrifice computation space for speed.

Cases 1-10 in Table 1 are tested using a single thread on an iCore 7 3.07 GHz computer with 24GB of memory. The left side shows the complexity of the RAS and the right side shows the space and time needed by the three methods. The parameters we used to generate these random examples are available at Gadara (2012) so interested readers can repeat our experiments. Our examples have an unusually high percentage of unsafe states because they acquire resources in a random fashion. In contrast, a real-world RAS often follows certain rules for resource allocation, e.g., global ordering, so it is easier to understand and is less prone to deadlock. Our randomly generated examples represent worst-case inflation ratios for our algorithms. For Cartesian mapping, there are more stages that allocate multiple resources. For tree mapping, there are more states that satisfy the invariant constraints but are not reachable since there are too many deadlocks.

Overall, we observe that the decision tree mapping is most space-efficient among all three algorithms. Its inflation ratio is less than 3× in all cases. The inflation ratio of Cartesian mapping varies. Higher percentages of unsafe states often result in larger inflation ratios. Since our goal is to understand the performance tradeoffs of different methods, our implementation currently uses only one 64-bit integer array to store all bits. This creates an artificial upper bound of 137 billion bits because the maximum array index in Java is 2^{31}. Therefore, there is no result for examples requiring more than 137 billion bits using Cartesian mapping. The non-bit method requires memory space linear in the number of states, which runs out of memory on Case 7 and beyond. The memory usage reported in the table is not fully accurate because of garbage collection in Java.

The three columns of computation time represent the calculation of reachable states, supremal controllable non-blocking subset of states, and the reachable states in the controlled Gadara RAS. We use the control synthesis algorithm described in Liao and et al (2011) to calculate the controlled Gadara RAS for each case. This algorithm uses an MIP formulation to find siphons iteratively, to which we apply a timeout threshold of 30 minutes. Only the first six cases finish successfully. The number of additional resource types added in the controlled Gadara RAS is shown in the parentheses under the "resources" column. We are unaware of any published solution that can scale to billions of states for Gadara RAS. We note, however, that once all unsafe states are found, there is a "trivial" solution that avoids each (boundary) unsafe state by exactly one *linear inequality* constraint (Nazeem and et al, 2011). This could result in too many linear inequalities that require special handling. We do not evaluate our program against this type of controlled Gadara RAS. The computation time of all three algorithms is roughly proportional to the number of states. The non-bit algorithm is fast, especially with smaller examples, because it does not require state mapping. Decision tree mapping is the slowest because of its complex mapping algorithm. For Case 8, Cartesian mapping calculates reachable states but not unsafe states because it runs out of memory on the latter (needs three more bit arrays to store intermediate results).

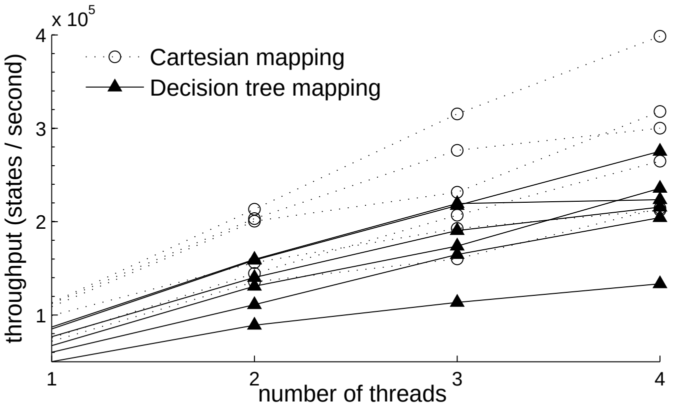


Fig. 4. Scalability with multithreading

Figure 4 shows the benefit of multithreading. There are six test cases. We run both tree mapping and Cartesian product mapping for all these cases using one to four threads on a four-core CPU, and each data point is the average of four runs. We can see that our multithreading is very effective. Finally, Cases 11-13 in Table 1 show three examples with more than a billion states using 16 threads on a dual 4-core (with hyperthreading) Xeon E5520 2.26Gz workstation with 96GB of memory. Exploring 2 billion states requires only a little over 5 hours on this workstation.

6. CONCLUSION

We have presented a framework and specialized algorithms that make it possible to explicitly generate and explore state spaces of RAS that have as many as billions of states, on a commodity computer. Under this framework, we have exploited structural properties of certain classes of RAS, known as Gadara nets and designed efficient mapping functions that enable the representation of each state by a single bit. Our algorithms not only perform

Table 1: Sample experimental results of state exploration using three different algorithms

	RAS		states			space		time (s)		
Case	resources	stages	reachable	unsafe	method	bits	memory(MB)	reachable	unsafe	controlled
1	18	47	608,768	4,096	Tree	653,312	50	11	31	14
	(+8)				Cartesian	25,436,160	25	10	28	11
					non-bit	-	1,916	10	23	11
2	24	113	1,233,346	445,404	Tree	2,242,112	106	35	63	50
	(+86)				Cartesian	5,038,387,200	6,249	28	58	45
					non-bit	-	1,375	17	37	30
3	12	148	2,342,805	2,055,995	Tree	5,070,019	160	37	21	70
	(+636)				Cartesian	16,547,328,000	12,404	28	25	71
					non-bit	-	1,398	22	36	39
4	20	127	3,631,081	3,113,562	Tree	5,705,565	1,244	78	86	57
	(+226)				Cartesian	2,682,408,960	3,212	64	71	50
					non-bit	-	1,725	52	88	36
5	10	150	10,241,714	5,392,441	Tree	12,310,910	65	248	657	1,283
	(+511)				Cartesian	7,287,084,000	4,190	164	433	1,238
					non-bit	-	3,107	235	589	962
6	24	86	15,567,136	10,212,724	Tree	15,568,720	649	463	476	544
	(+175)				Cartesian	2,786,918,400	2,500	370	408	521
					non-bit	-	5,748	367	887	458
7	20	208	45,469,437	44,685,596	Tree	67,267,687	390	1,326	314	-
					Cartesian	1.65E+12	-	-	-	-
8	19	129	109,308,048	43,773,264	Tree	126,001,256	557	3,351	6,102	-
					Cartesian	55,738,368,000	15,268	2,917	-	-
9	25	251	431,175,853	427,807,294	Tree	795,837,265	7,528	23,318	2,635	-
					Cartesian	1.54E+14	-	-	-	-
10	25	388	945,575,159	938,579,554	Tree	1,581,800,715	4,491	61,885	11,762	-
					Cartesian	7.42E+16	-	-	-	-
11	20	220	2,310,010,077	1,849,976,998	Tree	2,685,593,093	16GB	323m	262m	
12	25	298	6,090,459,834	5,005,028,660	Tree	13,937,653,999	38GB	18h	22h	
13	25	495	46,551,271,496	45,800,121,633	Tree	89,269,794,166	70GB	142h	21h	

the generation of the reachable state space, but they also permit efficient analysis and control synthesis, such as the calculation of the supremal controllable sublanguage, a core operation in control synthesis problems. Our software tools are available open source at Gadara (2012).

REFERENCES

Burch, J.R. and et al (1992). Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2), 142–170.

Cassandras, C.G. and Lafortune, S. (2008). *Introduction to Discrete Event Systems.* Springer, second edition.

Cimatti, A. and Roveri, M. (2000). Conformant planning via symbolic model checking. *J. Artif. Intell. Res.*, 13, 305–338.

Ezpeleta, J. and et al (1995). A petri net based deadlock prevention policy for flexible manufacturing systems. *IEEE Trans. on Robot. and Automat.*, 11(2), 173–184.

Ezpeleta, J. and et al (2002). A banker's solution for deadlock avoidance in FMS with flexible routing and multiresource states. *IEEE Trans. on Robot. and Automat.*, 18(4), 621–625.

Gadara (2012). `http://gadara.eecs.umich.edu/`.

Ghaffari, A. and et al (2003). Design of a live and maximally permissive petri net controller using the theory of regions. *IEEE Trans. on Robot. and Automat.*, 19(1), 137–141.

Jhala, R. and Majumdar, R. (2009). Software model checking. *ACM Comput. Surv.*, 41(4), 21:1–21:54.

Li, Z., Zhou, M., and Wu, N. (2008). A survey and comparison of Petri net-based deadlock prevention policies for flexible manufacturing systems. *IEEE Trans. Systems, Man, Cybernetics C*, 38(2), 173–188.

Liao, H. and et al (2011). Deadlock-avoidance control of multithreaded software: An efficient siphon-based algorithm for Gadara Petri nets. In *IEEE CDC.*

Liao, H. and et al (2012a). Concurrency bugs in multithreaded software: Modeling and analysis using Petri nets. *Journal of DEDS.* To appear.

Liao, H. and et al (2012b). Optimal liveness-enforcing control of a class of petri nets arising in multithreaded software. *IEEE Trans. on Automat. Ctrl.* To appear.

Nazeem, A. and et al (2011). Designing compact and maximally permissive deadlock avoidance policies for complex resource allocation systems. *IEEE Trans. on Automat. Ctrl.*, 56(8), 1818 –1833.

Nazeem, A. and Reveliotis, S. (2011). A practical approach for maximally permissive liveness-enforcing supervision. *IEEE Trans. Automat. Sci. and Eng.*, 8(4), 766–779.

Reif, J.H. (1985). Depth-first search is inherently sequential. *Inf. Process. Lett.*, 20(5), 229–234.

Reveliotis, S.A. (2005). *Real-Time Management of Resource Allocation Systems: A Discrete-Event Systems Approach.* Springer.

Wang, Y. and et al (2008). The application of supervisory control to deadlock avoidance in concurrent software. In *WODES.*

Wang, Y. and et al (2009). The theory of deadlock avoidance via discrete control. In *Proc. of Symposium on Principles of Programming Languages.*

Wang, Y. and Wu, Z. (2003). Deadlock avoidance control synthesis in manufacturing systems using model checking. In *American Control Conference*, 1702 – 1703.

Wolf, K. (2007). Generating Petri net state spaces. In *28th Conf. on Applications and Theory of Petri Nets.*