Parameterized Verification of Multithreaded Software Libraries

Thomas Ball Sagar Chaki Sriram K. Rajamani
tball@microsoft.com, chaki+@cs.cmu.edu, sriram@microsoft.com

Software Productivity Tools
Microsoft Research
`http://research.microsoft.com/slam/`

Abstract. The growing popularity of multi-threading has led to a great number of software libraries that support access by multiple threads. We present *Local/Global Finite State Machines* (*LGFSMs*) as a model for a certain class of multithreaded libraries. We have developed a tool called `Beacon` that does parameterized model checking of *LGFSMs*. We demonstrate the expressiveness of *LGFSMs* as models, and the effectiveness of `Beacon` as a model checking tool by (1) modeling a multithreaded memory manager `Rockall` developed at Microsoft Research as an *LGFSM*, and (2) using `Beacon` to check a critical safety property of `Rockall`.

1 Introduction

Software libraries traditionally have been designed for use by single-threaded clients. Due to the increasing use of multi-threading both on servers and clients, most libraries designed today accommodate simultaneous access by a multitude of threads. A software library typically provides its interface through a set of functions that a thread can call. Furthermore, the library usually maintains internal state between calls from clients. Even though multiple threads can access a library simultaneously, the library provides a consistent sequential semantics to all threads.

We are interested in checking properties of multithreaded software libraries. In particular, we are interested in checking that a library is well-behaved with respect to sequences of calls made upon it by a *multitude* of client threads.

We recently proposed boolean programs [BR00b,BR00a] as a model for representing abstractions of imperative programs written in languages such as C. Boolean programs are imperative programs in which all variables have boolean type. Boolean programs contain procedures with call-by-value parameter passing and recursion. Questions such as invariant checking and termination (which are undecidable in general) are decidable for boolean programs.

In order to model multi-threaded programs, we have extended the boolean program model with threads. Threads in a multi-threaded boolean program execute asynchronously, and communicate with each other using shared global variables. If B_1 and B_2 are two threads of a boolean program, we denote their asynchronous composition by $B_1 || B_2$. Unfortunately, even for boolean programs with only two threads, invariant checking is undecidable (this can be proved along the lines of [Ram99]).

Nonetheless, in practice we believe that the interaction between threads (in boolean programs as well as programs in general) usually can be modeled by a finite state machine. Therefore, we further abstract each thread of a boolean program to a *LGFSM* (local/global finite state machine), which makes the distinction between local and (shared) global states explicit.

The relationship between a boolean program B and its *LGFSM* abstraction F is one of refinement: the boolean program refines the interaction behavior specified by its *LGFSM* abstraction. We write this as $B \Rightarrow F$.

Suppose B_1 and B_2 are two threads of a boolean program B, whose interactions are described by *LGFSMs* F_1 and F_2 respectively. Then the following proof rule can be used to check if the composition of B_1 and B_2 satisfies invariant φ:

$$\begin{array}{ll} (1)\ B_1 & \Rightarrow F_1 \\ (2)\ B_2 & \Rightarrow F_2 \\ (3)\ F_1 \| F_2 & \models \varphi \\ \hline (4)\ B_1 \| B_2 & \models \varphi \end{array}$$

Note that proof obligations (1) and (2) involve checking refinement between a boolean program, and an *LGFSM*, and proof obligation (3) involves checking if a composition of two *LGFSMs* satisfies an invariant. All these questions are decidable.

Now, suppose that we want to check if a boolean program with an arbitrary number of threads satisfies an invariant φ. Let B^* denote the composition of an arbitrary number of threads of a boolean program B. Then the following proof rule can be used to check if B^* satisfies invariant φ:

$$\begin{array}{ll} (5)\ B & \Rightarrow F \\ (6)\ F^* & \models \varphi \\ \hline (7)\ B^* & \models \varphi \end{array}$$

In this paper, we give an algorithm to automatically check proof obligation(6), which has been implemented in a tool called `Beacon`. We model each thread (F) of a multi-threaded library by a *local/global finite state machine*, or *LGFSM*. An arbitrary number of instances of an *LGFSM* (F^*) comprise a *parameterized library system*, or *PLS* for short. We consider the question of whether or not a particular global state (a particular valuation to the global variables) is reachable in a *PLS*. We show that this problem is decidable, even when there are an arbitrary number of *LGFSMs*.

The results of this paper are four-fold:

- We formally define the *LGFSM* and *PLS* models, which can be used to model a wide class of concurrent software systems, namely those in which multiple anonymous clients require the services of a centralized library.
- Given a *PLS* system with m global states and n local states, we show that: (1) a global state is reachable in a *PLS* comprised of an arbitrary number of threads iff it is reachable in a *PLS* comprised of $2m^{n!}$ threads; (2) the global state reachability problem for a *PLS* can be decided deterministically in space $\mathcal{O}(2^{2nlog(n)+2loglog(m)})$ and time $\mathcal{O}(2^{2^{2nlog(n)+2loglog(m)}})$. These complexity results are based directly on the work of Rackoff [Rac78].
- We present an *LGFSM* model of an industrial-strength multi-threaded memory manager called `Rockall`, developed in Microsoft Research. `Rockall` is written in C++. We manually wrote a boolean program abstraction of a single thread of `Rockall`, and (automatically) inlined the procedure calls to obtain a *LGFSM*. The *LGFSM* model has $m = 2048$ global

states and $n = 256$ local states. In the *LGFSM* for `Rockall`, the global states represent the internal data structures of the memory manager while the local states represent the states of the clients of the memory manager.
- We present an algorithm for checking the reachability of a global state in a *PLS* that is similar to the algorithm for computing the minimal coverability graph for Petri nets presented in [Fin93]. The algorithm has been implemented in a tool called `Beacon`. When applied to the `Rockall` model, `Beacon` was able to prove a critical safety property of the model in about 4 hours, despite the fact that the algorithm might have had to explore a system with $2 \times 2048^{256!}$ threads, in the worst case.

The paper is organized as follows. Section 2 defines the *LGFSM* and *PLS* models, defines the global state reachability problem and shows that it is decidable. Section 3 introduces the `Rockall` memory manager and describes our *LGFSM* model of `Rockall`. Section 4 gives our algorithm for determining the reachability of a global state in a *PLS*, proves that the algorithm terminates and is sound and complete, and describes our experiences applying `Beacon` to `Rockall`. Section 5 discusses related work and Section 6 concludes the paper.

2 Modeling Multi-threaded Libraries

This section formally defines the concepts of the local/global finite-state machine (*LGFSM*) model and a parameterized library system (*PLS*), presents the reachability problem for a *PLS*, and shows that this problem is decidable. Finally it highlights some relationships between *PLS* and Petri nets.

2.1 Model

An *LGFSM* P is a 4-tuple $\langle \Lambda_P, \Gamma_P, \hat{\sigma}_P, T_P \rangle$, where

- Λ_P is a finite set of *local states.*
- Γ_P is a finite set of *global states.*
- $\hat{\sigma}_P \in \Lambda_P \times \Gamma_P$ is the *initial state.*
- $T_P \subseteq \Lambda_P \times \Gamma_P \times \Lambda_P \times \Gamma_P$ is a *transition relation* that prescribes how a pair of a local and global states transitions to another pair of local and global states.

Given an *LGFSM* P, and $f \geq 1$, the parameterized library system P_f consists of an interleaving composition of f instances of P, where all the instances share the same global states. Formally, P_f is a finite state machine $\langle \Sigma_{P_f}, \hat{\sigma}_{P_f}, T_{P_f} \rangle$, where

- Σ_{P_f} are $(f+1)$-tuples in ${\Lambda_P}^f \times \Gamma_P$. For a state $\sigma = \langle l_1, l_2, \ldots, l_f, g \rangle$ in Σ_{P_f}, we define projection operators $\sigma(i)$, for $1 \leq i \leq f+1$ to extract the components of σ.
- $\hat{\sigma}_{P_f}$ is $\langle \hat{l}, \hat{l}, \ldots, \hat{l}, \hat{g} \rangle$, where $\langle \hat{l}, \hat{g} \rangle = \hat{\sigma}_P$ and the $|\hat{\sigma}_{P_f}| = f+1$.
- $T_{P_f} \subseteq \Sigma_{P_f} \times \Sigma_{P_f}$ is a set of transitions, such that $\langle \langle l_1, l_2, \ldots, l_f, g \rangle, \langle l'_1, l'_2, \ldots, l'_f, g' \rangle \rangle$ if for some $1 \leq i \leq f$, we have that $\tau = \langle \langle l_i, g \rangle, \langle l'_i, g' \rangle \rangle \in T_P$, and for all j, where $1 \leq j \leq f$ and $i \neq j$, we have that $l_j = l'_j$. We say that the second state of the transition is the *image* of the first state under the transition. Formally, $\langle l'_1, l'_2, \ldots, l'_f, g' \rangle = \mathit{Image}(\langle l_1, l_2, \ldots, l_f, g \rangle, \tau)$.

A sequence $\overline{\sigma} = \sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_j$ over Σ_{P_f} is a *trajectory* of P_f if (1) $\sigma_0 = \hat{\sigma}_{P_f}$, and (2) for all $0 \leq i < j$, we have $\langle \sigma_i, \sigma_{i+1} \rangle \in T_{P_f}$. A state σ is *reachable* in P_f if there exists a trajectory that ends in σ. A global state $g \in \Gamma_P$ is *reachable* in P_f if there exists a reachable state σ in P_f such that $\sigma(f+1) = g$.

2.2 Decidability of the Reachability Problem

An instance of the parameterized reachability problem for software libraries consists of an *LGFSM* P and a global state $g \in \Gamma_P$. The answer to the parameterized reachability problem is "yes" if there exists some $f \geq 1$ such that g is reachable in P_f, and "no" otherwise.

We exploit two characteristics of *LGFSM* models. First, in an *PLS*, each state transition can change the local state component of at most one *LGFSM*. Because of this restriction, it is not possible for an arbitrary number of clients to change their local states in a single instant in a *PLS*.[1] Second, because the size of the global state component is bounded and the number of clients unbounded, it is not possible for clients to communicate their identity to each other through the global state.

We give an upper bound to the number of threads we need to consider, in order to to decide the global state reachability problem for *LGFSMs*. In the sequel, we denote the number of global states in a *LGFSM* ($|\Gamma_P|$) by m and the number of local states ($|\Lambda_P|$) by n. The proofs of the following theorems are present in Appendix A.

Theorem 1. Let P be an *LGFSM* with m global states and n local states. Let $g \in \Gamma_P$. For all $f \geq 1$, global state g is reachable in P_f iff g is reachable by a trajectory of length at most $2m^{n!}$ in P_f.

Corollary. Let P be an *LGFSM* with m global states and n local states. A global state g is reachable in P_f for some $f \geq 1$ iff g is reachable in $P_{2m^{n!}}$.

Theorem 2. An instance of the parameterized reachability problem with a *LGFSM* that has m global states and n local states can be decided deterministically in space $\mathcal{O}(2^{2nlog(n)+2loglog(m)})$ and time $\mathcal{O}(2^{2^{2nlog(n)+2loglog(m)}})$.

2.3 Relationship between *PLS* and Petri nets

The *PLS* and Petri net (PN) models of computation are quite intimately related, as underscored by the following two claims. Given a Petri net P (respectively *PLS* P_f), we denote by $\mathcal{R}(P)$ (respectively $\mathcal{R}(P_f)$) the set of its reachable states.

Claim 1. *PLS* and Petri nets are computationally equivalent in the following sense. Given a *PLS* P_f, there exists a Petri net P and a mapping γ from the states of P_f to the states of P such that for each $s \in \mathcal{R}(P_f)$, $\gamma(s) \in \mathcal{R}(P)$. Also given a Petri net P, there exists a *PLS* P_f and a mapping δ from the states of P to the states of P_f such that for each $s \in \mathcal{R}(P)$, $\delta(s) \in \mathcal{R}(P_f)$.

Claim 2. An instance of the parameterized reachability problem for software libraries can be reduced to an instance of the coverability prblem for Petri nets and vice-versa.

---

[1] This is consistent with the interleaving semantics usually given to threads.

The justifications for these claims are quite simple and are left as an exercise for the reader. The decidability of the coverability problem for PNs has been known since [KM69]. Combined with claim 2, this result gives another proof for the decidability of the parameterized reachability problem for software libraries.

3 The `Rockall` Memory Manager

In this section, we describe the `Rockall` memory manager and our boolean program and *LGFSM* models of it.

3.1 A Quick Tour of `Rockall`

`Rockall` is a parameterized thread-safe object-oriented memory manager. The basic data structure that `Rockall` uses for managing memory is the “bucket”. Each bucket is responsible for allocating chunks of memory of a particular size. Buckets are arranged in a tree-like hierarchy. When a bucket runs out of memory, it requests a larger chunk of memory from its parent and then breaks up this big chunk into smaller chunks (corresponding to its own size), which it can then allocate as needed. The bucket at the root of this hierarchy gets its memory directly from the operating system. The number of buckets, their allocation sizes, and the tree hierarchy can be configured by the user at startup.

`Rockall` has a number of other features that are pertinent to our modeling. First, unlike most memory managers, `Rockall` maintains all information regarding the allocated memory chunks (two bits per chunk) separately in its own data structure (a hash table) rather than padding the memory chunk given to the user process with these bits. This prevents the user process from accidentally (or intentionally) trampling on the manager’s data. This information is require for `Rockall` to determine which bucket a memory chunk was allocated from when memory is deallocated. Several locks are used in `Rockall` to ensure that each thread sees a consistent view of memory and also to achieve high performance.

The critical safety property of `Rockall` that we want to ascertain is the following : no memory location should be allocated or deallocated by `Rockall` twice or more in succession. In other words, allocation and deallocation of every memory location should occur alternately. Since the actual addresses of the memory chunks are not important for the verification of this property, we abstract away the address values completely. Also, we consider a scenario where `Rockall` has only two buckets, $B0$ and $B1$, where $B1$ is $B0$’s parent, and there is only one memory chunk present. Even with these restrictions, the abstract model for `Rockall` is of non-trivial complexity.

3.2 Boolean program model

We first describe an abstract Boolean Program model for `Rockall`. There are nine global boolean variables in this model:

- *B0_lock* : this variable is used to lock bucket $B0$. It protects the variable *B0_allocated.* The lock must be acquired before $B0$ can allocate or deallocate a chunk. The lock is initially free (the variable has the value **false**).

- *B1_lock* : this variable is used to lock bucket $B1$ and protects the variables *B1_allocated* and *B1_subdivided*. The must be acquired before bucket $B1$ can allocate or deallocate a chunk. The lock is initially free.
- *newpage_lock* : this lock must be acquired before the ownership of a chunk is transferred from one bucket to another. The lock protects the variables *available* and *find*. It is initially free.
- *find_lock* : this lock must be acquired before the hash table is searched to find the bucket that owns a chunk. This lock must also be acquired before the ownership of a chunk is transferred from one bucket to another, as the hashtable will be updated as a result (*newpage_lock* comes before *find_lock* in the lock order).
- *B0_allocated* : this variable is **true** if bucket $B0$ has allocated its chunk to the user process, otherwise it is **false**. The variable is initially **false**.
- *B1_allocated* : this variable is **true** if bucket $B1$ has allocated its chunk to bucket $B0$ or to the user process, otherwise it is **false**. The variable is initially **false**.
- *B1_subdivided* : this variable is **true** if bucket $B1$ has allocated its chunk to bucket $B0$, otherwise it is **false**. The variable is initially **false**.
- *available* : this variable is **true** if bucket $B1$ has the right to allocate the chunk and **false** if bucket $B0$ has the right to allocate it. It is initially **true**.
- *find* : this variable is **true** if bucket $B1$ holds the chunk and **false** if bucket $B0$ holds it. This variable models the hash table. It is initially **true**.

The boolean program abstraction of `Rockall` contains seven procedures, whose behavior we summarize below:

- *B0_New()*: this procedure models the allocation of a chunk to the user by bucket $B0$. It returns **true** if a successful allocation occurs and **false** otherwise. It calls the procedure *FetchFromB1()* in the case that $B0$ has no available memory and needs to get memory from $B1$ before completing the allocation request.
- *FetchFromB1()* : this procedure models the allocation of $B1$'s chunk to $B0$. It returns **true** if a successful allocation occurs and **false** otherwise.
- *B1_New()* : this procedure models the allocation of $B1$'s chunk to the user. It returns **true** if a successful allocation occurs and **false** otherwise.
- *B0_Delete()* : this procedure models the deallocation of $B0$'s memory chunk. It returns **true** if a successful deallocation occurs and **false** otherwise, and calls the procedure *GiveToB1()* in case $B0$ needs to return the chunk to $B1$ after the deallocation.
- *GiveToB1()* : this procedure models the return of the chunk by $B0$ to $B1$.
- *B1_Delete()* : this procedure models the deallocation of $B1$'s chunk. It returns **true** if a successful deallocation occurs and **false** otherwise.

3.3 Instrumented program

Recall that we want to check if no memory location should be allocated or deallocated by `Rockall` twice or more in succession. We add the following instrumentation to our `Rockall` model, in order to reduce the problem of checking this safety property to a problem of checking an invariant.

We add two variables *safe0* and *safe1* to the boolean program. These variables summarize the allocation/deallocation behavior seen so far:

- if both variables are **false** then there have been an equal number of alternating allocations and deallocations;
- if *safe1* is **false** and *safe0* is **true** then there has been an additional allocation;
- if *safe1* is **true** and *safe0* is **false** then there has been an additional deallocation;
- finally, if both variables are **true** then there have been two or more successive allocations or deallocations (this is the error state)

Part of the instrumentation is a new procedure *UpdateState()* that updates the two shared variables *safe1* and *safe0* in accordance with the allocation/deallocation that has occurred and the above-mentioned protocol for updating these two variables. It is called every time a successful allocation/deallocation occurs.

3.4 Translation to *LGFSM*

Since the boolean model does not have any recursion, it can easily be transformed to a finite state model by inlining all procedure calls. An *LGFSM* abstraction of `Rockall` was obtained by automatically inlining the procedures of the boolean program. Local variables are used to explicitly track important control locations in the boolean program (which are implicit in the boolean program representation). The abstract *LGFSM* for `Rockall` has eleven global variables and eight local variables. Let us denote the set of global variables by γ_P and the set of local variables by λ_P. We then have $m = |\mathit{\Gamma}_P| = 2^{|\gamma_P|} = 2048$ and $n = |\mathit{\Lambda}_P| = 2^{|\lambda_P|} = 256$.

4 The `Beacon` Tool

The decidability result from Section 2 is of theoretic interest only, as it is infeasible to explicitly check all trajectories of length $2m^{n!}$ even for small values of m and n. We have implemented an algorithm which has the effect of exploring all such trajectories but employs certain key optimizations to reduce the amount of exploration required. In this section, we present the algorithm and prove that the optimizations are sound and complete. Although the algorithm could, in the worst case, still explore all trajectories of length at most $2m^{n!}$, the optimizations seem to be extremely effective in practice.

The `Beacon` tool was able to verify the desired safety property of `Rockall` for an arbitrary number of threads. It ran on a 800 MHz Pentium III machine with 512 MB of RAM and took about 240 minutes to complete. In the process it explored roughly 2 million states. The complexity result of section 2 implies that (in the worst case) the algorithm might check all trajectories of length at most $2 \times 2048^{256!}$ which is of the order of $10^{10^{600}}$. The fact that `Beacon` managed to verify the property indicates that the optimization techniques we employ might be quite effective in practice.[2]

---

[2] We had initially attempted to verify the safety property for a fixed number of threads of the LGFSM using SMV [McM]. We wrote descriptions of the composition of a fixed number of threads of the LGFSM in the SMV language and tried to model check the safety property using Cadence's SMV tool. However the tool was unable to verify the property for more than 4 threads when run on the above mentioned machine.

4.1 The Algorithm

We start by defining an alternate representation for the states of a *PLS* P_f. As before, let $m = |\Gamma_P|$ and let $n = |\Lambda_P|$. A state σ of P_f, for any $f \geq 1$, can be represented as $(n+1)$-tuple $\theta \in \mathbb{N}^n \times \Gamma_P$. where the global states of σ and θ are the same, and for $1 \leq i \leq n$, the i-th component of θ is equal to the number of times l_i occurs in σ. Formally, we have (1) $\theta(n+1) = \sigma(f+1)$, and (2) for $1 \leq i \leq n$, $\theta(i)$ is equal to the number of occurrences of Λ_i in σ. The advantage of this alternate representation is that it provides a uniform way to represent the states of P_f for all f.

Representing Infinite Sets of States With Configurations. The number of reachable states of P_f for all f, is potentially infinite. We use the following trick to represent certain infinite sets of states. We allow a special symbol $*$ in our state representation to implicitly represent the set of all natural numbers. Formally, a *configuration* is an element of the set $\{\mathbb{N} \cup \{*\}\}^n \times \Gamma_P$. Note that every state is a configuration. A configuration θ which contains one or more occurrences of $*$, is interpreted to represent the infinite set of states obtained by replacing each occurrence of $*$ by some natural number. For example, if $n = 4$, then the configuration $\langle 3, *, 0, *, g\rangle$ represents the set of states $\{\langle 3, i, 0, j, g\rangle | i \in \mathbb{N}, j \in \mathbb{N}\}$. Note that we cannot use this trick to represent any infinite set of states compactly. For example, we cannot represent the set of states $\{\langle 3, 2i, 0, 5, g\rangle | i \in \mathbb{N}\}$ using a configuration.

We define two unary operators *Inc* and *Dec* over the domain $\mathbb{N} \cup \{*\}$. If $k \in \mathbb{N}$ then $Inc(k) = k+1$, and $Dec(k) = k-1$. For $k = *$, we have $Inc(*) = Dec(*) = *$. Let $\theta_1 = \langle k_1, k_2, \ldots, k_i, \ldots, k_j \ldots, k_n, g\rangle$ be a configuration. Consider i, j such that $k_i > 0$ and $\tau = \langle\langle g, l_i\rangle, \langle g', l_j\rangle\rangle \in T_P$. Then, the image of θ_1 under τ is defined as

$$Image(\theta_1, \tau) \;=\; \langle k_1, k_2, \ldots, Dec(k_i), \ldots, Inc(k_j), \ldots, k_n, g'\rangle$$

We note that the image operator is distributive with respect to the states in a configuration. That is, $Image(\theta_1, \tau)$ exactly represents the set $\{\sigma_2 \mid \exists \sigma_1 \in \theta_1 . \sigma_2 = Image(\sigma_1, \tau)\}$.

We extend the comparison operators $\leq$ and $<$ to operate over the natural numbers extended with $*$. Let $\leq^{\mathbb{N}}$ and $<^{\mathbb{N}}$ be the usual comparison operators in $\mathbb{N}$. Let i, j be in $\mathbb{N} \cup \{*\}$. We say that $i \leq j$ if (1) $j = *$, or (2) $i, j \in \mathbb{N}$ and $i \leq^{\mathbb{N}} j$. We say that $i < j$ if (1) $j = *$ and $i \in \mathbb{N}$, or (2) $i, j \in \mathbb{N}$ and $i <^{\mathbb{N}} j$.

Given two configurations Ω_1, and Ω_2 , we say that Ω_2 *covers* Ω_1 , written $\Omega_1 \leq \Omega_2$ if (1) $\Omega_1(n+1) = \Omega_2(n+1)$, and (2) for every $1 \leq i \leq n$, we have that $\Omega_1(i) \leq \Omega_2(i)$. We say that Ω_2 *dominates* Ω_1, written $\Omega_1 < \Omega_2$, if (1) $\Omega_1 \leq \Omega_2$, and (2) for some $1 \leq i \leq n$, we have that $\Omega_1(i) < \Omega_2(i)$. Note that if $\Omega_1 \leq \Omega_2$, then all the global states reachable from Ω_1 are also reachable from Ω_2.

Let Ω_1 and Ω_2 be two configurations such that $\Omega_1 < \Omega_2$. Then, we define $Closure(\Omega_1, \Omega_2)$ to be the configuration Ω_3 obtained in the following way:

- $\Omega_3(n+1) = \Omega_1(n+1) = \Omega_2(n+1)$, and
- for every $1 \leq i \leq n$, if $\Omega_1(i) = \Omega_2(i)$, then $\Omega_3(i) = \Omega_1(i)$, otherwise $\Omega_3(i) = *$.

```
          $WorkList := \{\theta\}$, where let $\hat{\sigma}_P = \langle l_i, g \rangle$ in
                    $\theta(n+1) = g$,
                    $\theta(i) = *$, and
                    $\theta(j) = 0$ for $1 \leq j \leq n$, $j \neq i$
          $Reach_v := WorkList$
          $Reach_e := \{\}$
          while ($Nonempty$ ($WorkList$)) do
             $c := Remove$ ($WorkList$)
             foreach transition $\tau$ enabled in $c$
[1]             $d := Image(c, \tau)$
[2]             if there exists a vertex $a \in Reach_v$ such that $d \leq a$ then
[3]                drop $d$ and do nothing
[4]             elsif there exists a vertex $a \in Reach_v$ such that $a < d$ and
                   there is a path from $a$ to $d$ through edges in $Reach_e$ then
[5]                   $e := Closure(a, d)$
                      let $V$ be the set of vertices reachable so far from $a$ (excluding $a$) in
                         delete vertices from $V$ from $WorkList$ and $Reach_v$
[6]                      delete edges connecting to/from vertices in $V$ from $Reach_e$
[7]                      replace $a$ with $e$ in $Reach_v$ and $Reach_e$
[8]                      add $e$ to $WorkList$
                else
[9]                $Reach_v := Reach_v \cup \{d\}$
[10]               $Reach_e := Reach_e \cup \langle c, d \rangle$
[11]               add $d$ to $WorkList$
                if;
             endfor
          endwhile
```

Fig. 1. Algorithm for global state reachability in a *PLS*.

The Algorithm and Its Properties. Figure 1 presents our algorithm for the parameterized reachability problem. The algorithm constructs a reachability graph $\langle Reach_v, Reach_e \rangle$, where $Reach_v$ is a set of vertices, and $Reach_e$ is a set of directed edges. Each vertex in $Reach_v$ is a configuration (we use the terms "vertex", and "configuration" interchangeably in the ensuing description). We maintain a worklist of unexplored configurations. The worklist is initialized with the initial configuration. The algorithm proceeds by picking a configuration c from the worklist and investigating every transition τ enabled in c (which leads to a configuration d). If d is covered by an existing reachable configuration a then no new global states can be reached from d that could not be reached from a, so d is "dropped". Instead, if d dominates a configuration a from which d is reachable then a compression step is possible (lines [5-8]). Otherwise, d is added to the set of reachable configurations and is added to the worklist.

Three properties remain to be proved about this algorithm:

- **Completeness:** Every reachable state in P_f for all f is contained in some configuration reached by the algorithm.
- **Soundness:** Every state contained in configurations reached by the algorithm is reachable in P_f for some f.
- **Termination:** The algorithm terminates.

The proofs of these properties are similar to proofs of the minimal coverability graph algorithm for Petri Nets presented in [Fin93] (the interested reader is referred to Appendix B for details).

4.2 Implementation Details

Below we summarize some key features of the implementation of the `Beacon` tool:

- `Beacon` constructs a reachability tree instead of a graph by ensuring that the same state is not explored more than once. Maintaining a tree makes it much easier to perform the check in step [4] since there can be at most one trajectory between two vertices in a directed tree.
- The reachability tree is constructed in a depth-first manner. We are currently experimenting with a breadth-first implementation.
- We represent $*$ by the largest unsigned integer. While computing the image in step [1] we check for overflows. In our experiments we have found that the non-zero local state counts are either $*$ or small integers.
- The representation of $*$ as a finite integer coupled with the overflow check automatically puts a bound on the length of any explored trajectory, and hence on the running time of `Beacon`. The bound on the length of the trajectory is much smaller than what is required by the result of section 2 but we have found it to be more than sufficient for `Rockall`. This bound can be increased to an arbitrary level simply by using a larger value for $*$.
- A configuration could be represented as an array of n unsigned integers. However we discovered that most of these counts are actually zero in the explored states. To reduce space requirements, we use a sparse representation where we only maintain the non-zero local state counts along with the corresponding local states.

5 Related Work

Petri nets (PNs) [Pet62] were introduced in 1962 by C. A. Petri in his doctoral dissertation. A few years later, Karp and Miller [KM69] independently proposed Vector Addition Systems (VASs) for analyzing the properties of *parallel program schemata*. Ultimately it was realised that they are mathematically equivalent. An excellent survey of PNs, VASs, and various decidability issues relating to them can be found in Esparza [EN94]. Over the years several other models were proposed for representing infinite state systems. Many of them, like *timed PNs* were extensions to PNs, and some, like VASSs, were shown to be mathematically equivalent to VASs. There has been a lot of interesting work on decidability of problems like reachability and coverability for infinite-state systems [ACJYK96,AJ97]. Very recently, there has been a remarkable attempt at trying to unify a diverse set of infinite-state systems having similar decidability properties under a single framework of *well-structured transition systems* [FS00].

The coverability problem for VASs has been known to be decidable since [KM69]. But the algorithm proposed there is notorious for its complexity. It involves the construction of a *coverability tree*, and might require non-primitive recursive space in the worst case. Lipton [Lip76] proved that deciding the coverability problem for VASs requires at least exponential space in the size of the VAS. More specifically, Lipton showed that for some constant $d > 0$, the problem cannot be decided in space $2^{d\sqrt{n}}$. His lower bounds are valid even if one only considers input whose vectors have components of value -1, 0, or 1. Nobody has been able to propose an algorithm that matches Lipton's lower bound. Rackoff [Rac78] gave a near-optimal algorithm that requires space bounded by an exponential of $nlog(n)$, where n is the size of the VAS. Unfortunately, Rackoff's algorithm is impractical for even VASs of moderate size. According to [FS00], all implemented algorithms for the coverability problem [Fin90,Fin93] use Karp and Miller's coverability tree, or the coverability graph, or some complex forward-based method. The work most related to ours is the construction of the minimal coverability graph for PNs given by Finkel [Fin93]. To the best of our knowledge, this approach has not been applied to the parameterized verification of multi-threaded software libraries, and has not succeeded on a design as large as `Rockall`. The Petri net for the PNCSA communication protocol used in [Fin93], for example, has only 31 places and 36 transitions.

The link between PNs and parameterized networks has also been known for a long time. German and Sistla investigated temporal logic model checking of parameterized networks [GS92]. Out of the two models presented by them, one is comparable to *PLS*. The algorithm they present for this model is based on Rackoff's algorithm and has double-exponential time complexity. There has also been significant research on model checking of programs written in languages like Java which support multi-threading [CDH^{+}00,HP00]. These approaches however concentrate on general Java programs and do not consider arbitrary numbers of threads. They impose an apriori bound on the number of threads in order to do model checking.

6 Conclusion and Future Work

In this paper, we have presented a model called *LGFSM* for representing multi-threaded libraries. Using the model, we have been able to extend well-known complexity results and algorithms from the domain of PNs and VASs to multi-threaded software libraries. We have implemented our algorithm in a tool called `Beacon` and use it to verify critical safety properties of an industrial-strength memory manager called `Rockall`. Below we summarize some interesting and challenging research directions:

- The current implementation of `Beacon` could be optimized further. In particular, it would be interesting to see if data structures employed in similar algorithms for verification of cache coherence protocols [EN96,Del00] can be used in the domain of *LGFSMs*.
- As mentioned before, we believe that in most concurrent programs the interaction between threads is regular can be captured using finite state machines. One of the major challenges in software model checking is extracting this finite state behavior (sometimes called a *synchronization skeleton*) from concurrent program descriptions. Often the actual program description is too large to be verified, and the *synchronization skeleton* is sufficient to decide the property of interest. We are interested in extracting such finite state models automatically and efficiently.
- Another challenging problem is to efficiently check refinement between a *LGFSM* and a boolean program. The motive behind doing this is that if we prove a safety property about a *LGFSM* and then prove that the *LGFSM* is refined by a C program, we could conclude that the safety property holds for the C program also.

- Finally we would also like to develop parameterized verification techniques for other, slightly more relaxed models. For example we would like to model *PLS* where the threads have a sense of identity of themselves and others, say through a thread identifier.

Acknowledgement

We thank Michael Parkes for giving us access to Rockall, and for laboriously explaining its internal details. We also thank Giorgio Delzanno for useful comments and suggestions.

References

[ACJYK96] P. A. Abdulla, K. Cerans, B. Jonsson, and T. Yih-Kuen. General decidability theorems for infinite-state systems. In *Proc. 11th IEEE Symp. Logic in Computer Science (LICS '96)*, pages 313–321, July 1996.

[AJ97] P. A. Abdulla and B. Jonsson. Ensuring completeness of symbolic verificatiom methods for infinite-state systems. *Theoretical Computer Science*, 1997. To appear.

[BR00a] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN 00: SPIN Workshop*, Lecture Notes in Computer Science 1885, pages 113–130. Springer-Verlag, 2000.

[BR00b] T. Ball and S. K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Research, February 2000.

[CDH$^+$00] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera : Extracting finite-state models from Java source code. In *Proceedings of ICSE 2000 : International Conference on Software Engineering*, 2000.

[Del00] G. Delzanno. Automatic Verification of Parameterized Cache Coherence Protocols. In *CAV 00: Computer Aided Verification*, Lecture Notes in Computer Science 1855, pages 53–68. Springer-Verlag, 2000.

[EN94] J. Esparza and M. Nielsen. Decibility issues for petri nets - a survey. *Journal of Informatik Processing and Cybernetics*, 30(3):143–160, 1994.

[EN96] E. A. Emerson and K. S. Namjoshi. Automatic Verification of Parameterized Synchronous Systems. In *CAV 96: Computer Aided Verification*, Lecture Notes in Computer Science 1102, pages 87–98. Springer-Verlag, 1996.

[Fin90] A. Finkel. Reduction and covering of infinite reachability trees. *Information and Computation*, 89:144–179, 1990.

[Fin93] A. Finkel. The minimal coverability graph for petri nets. *Advances in Petri Nets, Lecture Notes in Computer Sceince*, 674:210–243, 1993.

[FS00] A. Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere ! *Theoretical Computer Science*, 2000. To appear.

[GS92] S. M. German and A. P. Sistla. Reasoning about systems with many processes. *JACM*, 39(3), July 1992.

[HP00] K. Havelund and T. Pressburger. Model checking Java programs using JavaPathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4), April 2000.

[KM69] R. M. Karp and R. E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3:147–195, 1969.

[Lip76] R. J. Lipton. The reachability problem requires exponential space. Technical report, Department of Computer Science, Yale University, 1976.

[McM] K.L. McMillan. `http://www-cad.eecs.berkeley.edu/~kenmcmil`.

[Pet62] C. Petri. Fundamentals of a theory of asynchronous information flow. In *Information Processing 62, Proceedings of the 1962 IFIP Congress*, pages 386–390, 1962.

[Rac78] C. Rackoff. The covering and boundedness problem for vector addition systems. *Theoretical Computer Science*, 6:223–231, 1978.

[Ram99] G. Ramalingam. Context sensitive synchronization sensitive analysis is undecidable. Technical Report RC21493, IBM T.J.Watson Research, May 1999.

Appendix A: Proofs of Theorems 1 and 2

Our proof of Theorem 1 runs along the same lines as that given by Rackoff [Rac78]. But the property that each state transition can change the local state component of at most one *LGFSM* enables us to obtain a bound with smaller constants.

The proof uses the alternative representation of states given in Section 4. We denote the number of global states ($|\Gamma_P|$) by m and the number of local states ($|\Lambda_P|$) by n. A vector of dimension k is any member of $(\mathbb{Z} \cup \{*\})^k$. All vectors will be implicitly assumed to be of dimension n, unless otherwise mentioned. A configuration consists of two components: a global state and a vector of dimension n. Note that this is a more general defintion of configuration than the one in Section 4, in that the components of the vector can be negative integers as well. The generalization to include negative integers is necessary for the proofs below. For any state S, we denote the global state component as $G(S)$ and the vector component as $L(S)$. Without loss of generality, let us assume that the first component of the vector corresponds to the initial local state. The operators *Inc* and *Dec* from Section 4 are extended to operate over $(\mathbb{Z} \cup \{*\})$. The image operator *Image* is also extended to operate over the more general configurations.

We require the other following definitions and two lemmas to prove Theorem 1.

Definition 1. *A vector A is said to be i-bounded if, for* $1 \leq k \leq i$, $A(k) \geq 0$ *or* $A(k) = *$

Definition 2. *A vector A is said to be i–j-bounded if, for* $1 \leq k \leq i$, $0 \leq A(k) \leq j$ *or* $A(k) = *$.

Definition 3. *Let us denote by* ϕ_1 *the set of all vectors* X *such that* $X(1) = *$.

Definition 4. *Given an LGFSM* M, *let* G_M *be the graph whose nodes are the global states of* M *such that there exists an edge between global states* G_1 *and* G_2 *if there exists two local states* L_1 *and* L_2 *such that* (G_1, L_1, G_2, L_2) *is a valid transition of* M.

Consider any *LGFSM* M. Let A and B be two arbitrary global states of M.

Definition 5. *An i-path between A and B is a sequence of configurations* $\overline{\sigma} = \sigma_0, \sigma_1 ... \sigma_s$ *with the following properties: (1) For all* k, $0 \leq k \leq s$, $L(\sigma_k)$ *is i-bounded; (2)* $G(\sigma_0) = A$ *and* $L(\sigma_0) \in \phi_1$*; (3)* $G(\sigma_s) = B$*; (4) for each* $0 \leq i < s$, *there exists a transtion* τ_i *such that* $\sigma_{i+1} = Image(\sigma_i, \tau_i)$.

Definition 6. *An i–j-path between A and B is any sequence of configurations with the following properties: (1) For all* k, $0 \leq k \leq s$, $L(\sigma_k)$ *is i–j-bounded; (2)* $G(\sigma_0) = A$ *and* $L(\sigma_0) \in \phi_1$*; (3)* $G(\sigma_s) = B$*; (4) for each* $0 \leq i < s$, *there exists a transtion* τ_i *such that* $\sigma_{i+1} = Image(\sigma_i, \tau_i)$.

Definition 7. *Let us denote by* $\theta(A, B, i)$ *the length of a shortest i-path between A and B. If no i-path exists between A and B then* $\theta(A, B, i) = 0$. *Now let us define* $\theta(i) = max$ *over all A and B of* $\theta(A, B, i)$.

Lemma 1. $\theta(1) \leq m$ where m is the number of global states.

Proof. If two distinct global states A and B are connected in G_M, there must be a path P of length less than m between them in G_M. From this path it is possible to construct a 1-path of the same length as P between A and B. Thus, there is a 1-path of length less than m between A and B. If the global states are disconnected in G_M then (by definition) $\theta(A, B, 1) = 0$.

Lemma 2. $\theta(i+1) \leq m(\theta(i))^i + \theta(i)$.

Proof. Consider any two global states A and B. Let P be a shortest i+1-path from A to B. There are two cases to consider:

Case 1. P also is a i+1–$\theta(i)$-path. Since P is a shortest i+1-path from A to B it must be the case that no two states of P have vector components that are identical in their global states as well as the first i+1 components of their local state vectors. Since the first component of the local state vectors is $*$ (by assumption), the length of path Q must be less that $m(\theta(i))^i$.

Case 2. P is not a i+1–$\theta(i)$-path. In this case, P must be of the form $P'P''$ such that P' is a i+1–$\theta(i)$-path and the first state of P'' has a vector component which is not i+1–$\theta(i)$-bounded, say without loss of generality the (i+1)-th component is greater than $\theta(i)$. By an argument similar to *Case 1* we can choose P' to be of length less than $m(\theta(i))^i$. Now let the first state of P'' be α and the global state component of α be C. Since P'' is an i+1-path (and thus an i-path), there must be a shortest i-path Q from C to B that shares the same first state as P''. The length of Q is less than $\theta(i)$. Since the (i+1)-th component of Q is greater than $\theta(i)$, and since at any step, any component can decrease by at most 1, Q is also a shortest i+1-path between C and B. Hence $P'Q$ is a shortest i+1-path between A and B with length less than $m(\theta(i))^i + \theta(i)$.

Theorem 1. If any global state is reachable, then it is reachable by a path of length atmost $2m^{n!}$.

Proof. By **Lemma 1** and **Lemma 2**, it is clear that $\theta(i) \leq 2m^{i!}$, and there must be a path to a reachable global state which is of length less than $\theta(n)$.

Theorem 2. An instance of the parameterized reachability problem with a *LGFSM* that has m global states and n local states can be decided deterministically in space $\mathcal{O}(2^{2nlog(n)+2loglog(m)})$ and time $\mathcal{O}(2^{2^{2nlog(n)+2loglog(m)}})$.

Proof. By **Theorem 1**, to decide reachability requires examining trajectories of length at most $2m^{n!}$. This can be done non-deterministically using space $\mathcal{O}(2^{nlog(n)+loglog(m)})$. Hence by Savitch's Theorem, it can be done deterministically using space $\mathcal{O}(2^{2nlog(n)+2loglog(m)})$ and so in time $\mathcal{O}(2^{2^{2nlog(n)+2loglog(m)}})$.

Appendix B: Soundness, Completeness and Termination of the `Beacon` Algorithm

Completeness is easy to prove. If the algorithm drops a configuration d at line [2]-[3], then there is a configuration a such that $d \leq a$. Thus, every state reachable from d is also reachable from a. In lines [5]-[8], the algorithm deletes all vertices reachable from a and replaces a by e. Since $a \leq e$, every configuration reachable from a is covered by some configuration reachable from e. Thus, the algorithm is complete.

The only place in the algorithm that needs to be proved sound is at lines [5]-[8]. When the algorithm replaces a with e, we must show that the algorithm is adding no more states than those reachable in P_f, for some f. We know there exists a trajectory p from a to d. Let σ_e be any state in e. By starting from a, and repeatedly executing transitions along p sufficiently many times, it is possible to reach a state σ'_e such that $\sigma_e \leq \sigma'_e$. Thus, replacing a by e does not introduce any unsoundness.

Termination is not obvious since the number of configurations is infinite. Suppose the algorithm does not terminate. Since the number of transitions in P is finite, each vertex in $Reach_v$ has a finite out-degree. Then there exists a trajectory p in $Reach_v$ that passes through an infinite number of vertices. Since there are a finite number of global states, there exists an infinite number of vertices in this trajectory with the same global state, say g. Let q be the restriction of p to configurations with global state g. Note that q is an infinite trajectory as well.

Suppose a and d are any two vertices such that a precedes d in q. It cannot be the case that $d \leq a$, because d would have to be dropped by lines [2]-[3] of the algorithm. Thus vertices in trajectory q are non-decreasing. The only question is: Can they be non-increasing as well?. We prove that the answer is "No".

If there exist two vertices a and b in q such that a precedes b in q and $a < b$, then lines [5]-[8] of the algorithm will result in the deletion of a and we get a contradiction.

We prove by induction on n that in any infinite non-decreasing trajectory q produced by the algorithm, there exist vertices a and b, such that a precedes b in q and $a < b$. Suppose $n = 1$, then the second vertex of q has to dominate the first, and we are done. Suppose $n = k$, then consider the projection s of q to the first component of each configuration. Two cases are possible:

- s contains an infinite increasing subsequence s', or
- s contains an infinite subsequence s'' in which all elements are equal.

In either case, by restricting q to the corresponding elements in the subsequence s' or s'', we obtain an infinite sequence q' of configurations such that either the first components are monotonically increasing or the first components are all identical. By induction on the remaining $k - 1$ components of configurations in q', we prove the existence of a and b with the desired properties.