Model Checking of Multi-Process Applications Using SBUML and GDB

Yoshihito Nakagawa
University of Tokyo

Richard Potter
Japan Science and Technology Agency

Mitsuharu Yamamoto
Chiba University

Masami Hagiya
University of Tokyo

Kazuhiko Kato
Tsukuba University

Abstract

Model checking, aka systematic exploration of the state space, is widely used as a method of verification of concurrent programs that may involve nondeterminism in their execution order. Especially, direct verification of implementations has been focused on in recent years. The crucial issues for such a model checker include how to explore state space systematically and how to control nondeterminism. Although existing model checkers can do this by running the target programs under a scheduler specialized for verification, this approach is subject to restrictions in its use of OS resources such as file systems.

In this work, we propose and implement a framework for the model checking of multi-process applications on an environment that is very close to the one for the actual execution. Our key idea is to combine ScrapBook for User-Mode Linux (SBUML) and the GNU Debugger (GDB) to implement systematic exploration and control of nondeterminism. SBUML provides not only a full-fledged execution environment but also the save-and-restore feature of the entire OS state including its file system. Nondeterminism of the target multi-process application is controlled via the breakpoint mechanism on GDB. Use of breakpoints also enables us to control the granularity of scheduling in a flexible way. We also give some verification examples using this framework.

1. Introduction

As a method of verifying a concurrent program with nondeterminism, model checkers search the state space of the program exhaustively. Early model checkers did verifications mainly at the specification and design levels. But recently, the direct verification of actual code implementations has been attracting the interest of researchers and software engineers.

In model checkers such as CMC [17] and VeriSoft [13, 12], it is possible to analyze nondeterminism by executing the program under a specialized scheduler. But the target programs are subject to major restrictions. For example, OS resources like files are not available.

In this study, we propose and implement a new framework that allows the model checking of a multi-process application in a real environment by using ScrapBook for User-Mode Linux (SBUML [18, 1]), an extension of the User-Mode Linux (UML [2]) virtual machine. SBUML provides a facility to save and restore complete Linux computation states as snapshots.

Besides SBUML, our model checker is comprised of the debugger GDB [3], an internal monitor, and an external monitor. The internal monitor on a virtual machine makes GDBs control each process, detects the nondeterminism, and tells the information about the computation state to the external monitor on the host OS. The external monitor manages the state space of the application and searches it exhaustively based on the information the internal monitor provides. Currently, the user has to prepare some additional code by hand. However, the target application can be executed without any modification in our framework. This shows the potential for model-checking existing off-the-shelf applications on Linux.

As a sample application, the remote agent experiment (RAX [14]), for which we have actually done the model checking, is shown. We also introduce a possible application for the purpose of discussing the verification of applications that use OS resources.

The rest of this paper is as follows. In Section 2, we describe a virtual OS called ScrapBook for User-Mode Linux (SBUML), that is used as an execution environment for the target application. In Section 3, we propose a new framework for model checking using SBUML and GDB. In Section 4, we verify a sample application, the remote agent experiment, using our model checker. Section 5 is a discussion about improvement and possible application of our framework. In Section 6, we give some related works and compare them to our system. In Section 7, we summarize our work in this paper and present the future work.

2. ScrapBook for User-Mode Linux

Recently, the virtualization of system software has been drawing a great deal of attention. With a virtual computer system, we can construct various virtualized hardware resources inside a real computer (host computer), and make multiple independent virtual computers inside a single host computer. Virtual machines enable us to run one operating system virtually on another operating system. The OS virtually run is called *guest OS*, and the base OS on which the guest OS is run is *host OS*. The utilization of virtual machines is becoming more and more practicable in accordance with the progress of CPU and growth of memory, and commercial software such as VMware [4] have become widely used.

User-mode Linux is an extended Linux kernel developed by Jeff Dike. It is a Linux virtual machine that was created by porting the Linux OS to Linux system calls. UML runs inside a real Linux machine as a collection of user-mode processes. This means that UML can be regarded as a virtual operating system. We can execute multiple instances of the guest OS inside the host OS. Since UML is open source software, it is more suitable for adding flexible extensions than commercial software like VMware.

ScrapBook for User-Mode Linux is an extension to User-Mode Linux. In addition to the original functions of virtual machines provided by UML, SBUML can save complete Linux runtime states in mid-execution as *snapshots*, including all hard disks, devices, processes, and kernel space. The same state can be restored at a later time on the same PC, or migrated to another PC. SBUML provides a script language to control machine via its API, which provides a convenient foundation for implementing our real-environment model checker.

3. Framework of Model Checker

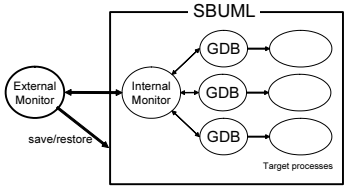
As we already mentioned, more and more focus is being put on model-checking program code directly. Model checking at the specification or design level is now called "classical model checking", while model checking at the program code level is called "modern model checking." Some tools, such as SLAM [10] and Blast [16], for modern model checking make abstraction of program code and verify the abstract program instead of the real one. On the other hand, some other tools directly execute application programs for modern model checking. This approach may be called "ultra-modern model checking".

In existing ultra-modern model checkers, such as CMC [17] and VeriSoft [13, 12], it is possible to analyze race conditions between processes and detect nondeterminism by executing the actual application program. However, since the program is executed under a scheduler specialized for verification, it is subject to many kinds of restrictions. For example, OS resources like files are usually not available.

In our framework, although some additional code should be added by hand, the existing code in the application program does not need to be modified. This means that already existing Linux applications, off the shelf, can be model-checked.

3.1. Outline of Our Model Checker

The figure below gives an overview of our model checking method.



A target multi-process application is run on a virtual machine. The *internal monitor* on the virtual machine has one GDB process for each process of the application. The breakpoint mechanism which GDB provides is used to control each process. Nondeterminism will occur when the processes race to get shared resources. As soon as this kind of nondeterminism is detected, the internal monitor computes the current state of the application and which processes can be resumed next and informs the *external monitor* on the host OS about such information.

Basically, each state has to be saved by SBUML during the search, since the number of virtual machines which can be run at a time is restricted. Then based on the information the internal monitor provides, the external monitor determines which process will be resumed, and requests the corresponding virtual machine snapshot to be restored.

As stated above, our model checker explores the state space exhaustively. In the next sections, we will give more detailed explanation about each component of our system and interaction between these components.

3.2. Internal Monitor and GDB

In order to control each process of the application, we have utilized the breakpoint mechanism that GDB provides. There are two types of breakpoints to be set.

- Where the computation result depends on the execution sequence of processes such as writing to and reading from shared variables.
- Where the execution of a process may block for synchronization such as access to a lock or a pipe.

Ideally, the portions should be extracted where nondeterminism occurs by statically analyzing the program. But currently, a user needs to determine where to set the breakpoints in the target code. At the same time, the user needs to specify a *blocking condition* for each breakpoint, which is used to determine if the process can be resumed or not. A blocking condition is a C statement that can be evaluated by GDB.

In the cases that blocking conditions invoke a system call or need complicated judgments, it is necessary to add functions called *inspection functions*. We are planning to realize a system for automatically generating such functions by statically analyzing the target program in the future.

If a blocking condition returns false, then the process can proceed to the next breakpoint without blocking the execution. If not, since the process would block the execution, the process is removed from the next candidates. If all blocking conditions return true, that is, when it turns out that no process can be resumed, it follows that the state is a deadlock.

In order to cover all the possible execution states, it is necessary to record visited states as a *hash value* and quit further search when the state is visited again. The internal monitor computes the hash value by the program counter (and the stack trace if needed) for each process, the values of inspection functions and the states of shared variables. It then sends the hash value to the external monitor.

3.3. External Monitor

The external monitor running on the host OS perform the depth-first search of the execution state space in accordance with the notification from the internal monitor on a virtual machine. The internal monitor provides the hash value, which expresses the current execution state, and the value of the blocking condition for each process.

The external monitor records the list of the states (hash values) already visited and quits the further search if the current state's hash has already been recorded. If the current state is unvisited, then the external monitor selects one of the candidate processes that can be resumed and requests the internal monitor to resume it.

Basically, every time an unvisited state is detected, the state has to be saved for the search at a later time. But if only one process can be resumed, the state will not be restored later. So it is possible to decrease the time for model checking by skipping such saving. And if no process can be resumed, it means that the state is a deadlock.

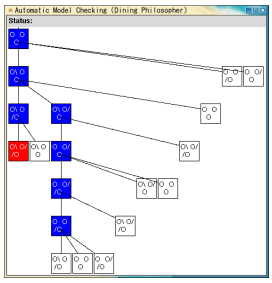
3.4. Implementation

We have implemented the internal and external monitor of our model checking system in Ruby 1.8.1 [5].

The communication between the host and guest OS is done by using socket connections. The external monitor on the host OS always open its port as a TCP server, and tries to receive a string from the internal monitor on the guest OS. The string represents the hash value of the current state and the candidate processes which can be resumed next.

Similarly, the internal monitor also always open its port as a TCP server and tries to receive the string from the external monitor. Since SBUML holds connection states of sockets just before saving a snapshot, the internal monitor can receive the message even after restoring.

We made it possible to grasp the overview of a state space by GUI on the host OS.



A red node (the leftmost bottom one) means that the state is a deadlock. A white (unfilled) node means that the state need not be extended any more because it has already been visited.

4. Application: Remote Agent Experiment

The Remote Agent (RA) is an autonomous spacecraft controller developed by NASA Ames conjointly with the Jet Propulsion Laboratory (JPL). The RA contained an error in the exclusive control among threads, which occurred during an actual flight. The cause was found by the data of the spacecraft. On the other hand, the RAX (RA Experiment) team accepted the challenge if the error can be located only by a formal method and succeeded in detecting the error by using the model checking tool, Path Finder. Here, we implement the portion which contains the error in C and verify it by using our model checker.

```
void signal_event(Event *e)
{
  mutex_lock(e); /* may block */ /*(1)*/
  e->count = (e->count + 1) % MOD_COUNT; /*(U1)*/
  cond_signal(e); /* doesn't block */
  mutex_unlock(e);
}

void wait_for_event(Event *e, int remote_count)
{
  mutex_lock(e); /* may block *//*(2)*/
#ifndef COND_OUTER
  if (remote_count == e->count) /*(U2)*/
#endif
    cond_wait(e); /* may block inside */
  mutex_unlock(e);
}
```

```
void first_task(Event *e1, Event *e2)
{                /* for the first process */
  int count = 0;
  while (1) {
#ifdef COND_OUTER
  if (count == e1->count)      /*(A)=(D1)*/
#endif
    wait_for_event(e1, count); /*(B)*/
    count = e1->count;         /*(D2)*/
    signal_event(e2);
  }
}

void second_task(Event *e1, Event *e2)
{                /* for the second process */
  int count = 0;
  while (1) {
    signal_event(e1);            /*(C)*/
#ifdef COND_OUTER
  if (count == e2->count)        /*(D3)*/
#endif
    wait_for_event(e2, count);
    count = e2->count;           /*(D4)*/
  }
}
```

Two functions are defined for event structures. One of them is `wait_for_event`, which makes a process that calls this function wait on the specified event structure. The other is `signal_event`, which wakes up any process that called `wait_for_event` and is waiting on the specified event structure. These two functions are supposed to be executed exclusively by using IPC semaphores.

On the other hand, an event structure has a counter variable `count` as its member variable. The counter variable is used to catch the arrival of a new event. Here, the counter variable is incremented (changed) in `signal_event`. In the remote agent experiment, each process alternately calls the two functions. The condition for `wait_for_event` to be called in each task is that the counter has not been changed.

Actually, the exclusive control mentioned above is not enough when the macro `COND_OUTER` is defined, and it finally turns out that the code contains an error. For example, if the counter `count` has not been changed at (A), it follows that the second process has not called `signal_event` while the first process has been active. Therefore, the first process tries to proceed to (B) and wait on the event structure `e1`. However, if the second process calls `signal_event` at (C) in `second_task` between (A) and (B), the first process will miss the event and go asleep. Then if the second process also calls `wait_for_event` and goes asleep, a deadlock will occur.

In order to avoid the deadlock, we should regard both (A) and (B) as one critical section and the exclusive control should be done around it. When `COND_OUTER` is undefined, the check of the counter variable and the execution of `cond_wait` are supposed to be done sequentially because the executive control is done around them in `wait_for_event`. Therefore, a deadlock would not occur.

4.1. Verification

First, we need to set breakpoints where the execution of the application may block for synchronization. Concretely, they are set where the semaphore values will be decremented. One of them is in `mutex_lock`, where the IPC routine `semop` is used. Since `mutex_lock` is called twice in the loop of each process (in `wait_for_event` and `signal_event`), we have set the breakpoints at (1) and (2) so that we can distinguish them only by the line numbers of the breakpoints.

The other call to decrement the semaphore values is in `cond_wait`. Since `cond_wait` is called just once in the loop for each process, we can set the breakpoint inside the body of the function.

We have added two inspection functions to observe the states of the IPC semaphores. One of them is defined as follows, and the blocking conditions for (1) and (2) are expressed as `remc_can_lock(e) == 0`.

```
int remc_can_lock(Event *e)
{ return semctl(e->semid, SEM_MUTEX, GETVAL); }
```

We should also define another set of breakpoints where the result of execution depends on the execution order of the processes. Concretely, they are set where the shared variables are accessed. Therefore, we need to set the breakpoints where `e1->count` or `e2->count` is read from or written to. In this case, the blocking condition always returns false and the process can resume. We have set the breakpoints at (U1) and (U2) when `COND_OUTER` is undefined and at (D1), (D2), (D3) and (D4) when `COND_OUTER` is defined.

4.2. Result

As above, we have model-checked RAX and the results are as expected. The deadlock is detected when `COND_OUTER` is defined and not when `COND_OUTER` is undefined. The operation environment is RedHat Linux 8.0 on Dual 2.8GHz Pentium Xeon, 2GB RAM.

The detail about the performance when `COND_OUTER` is defined is as follows.

Total time	15min 13sec
Total number of possible states	116(4 Deadlocks)
Save times	66
Restore times	66
Save time	Total: 160sec Average: 2.42sec
Restore time	Total: 59.3sec Average: 0.90sec
Snapshot size	Total: 2.4Gb Average: 37Mb

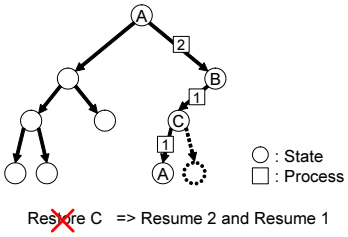
We have succeeded in completing the model-checking of the remote agent experiment within a practicable time. The total size of the saved snapshots for the search is about 2.4GB. However, it does not mean that we actually need about 2.4GB in the search. The actual disk space needed for the verification depends on the depth of the state space.

5. Discussion

5.1. Re-execution and Save-and-Restore

In the previous examples, we used save-and-restore in SBUML for implementing backtracking. Of course, this is not the only way to do that: it is possible to start from the restoration of the initial state, and then re-execute until the state that is backtracked. In general, it is a trade-off between the time for saving a snapshot and that for multiple transitions in re-execution. We can combine re-execution and save-and-restore so as to improve the total checking time by saving a snapshot only after a long transition. The previous examples apparently take much shorter time for transitions compared with saving a snapshot. Thus our result in the previous section shows that even in the extreme case that we do not use re-execution at all, the total checking time is still practicable.

There can be considered another optimization for reducing the total number of restores in some cases. Suppose that we have arrived at state A in the search of a state space as in the figure below. Since A has already been visited, no further search will be done from the state A. The state C is supposed to be restored for the search. However, we can check if there is a path from state A to state C. If it is possible to transit from A to C in a short time, we can arrive at the state C from the state A by re-executing processes 2 and 1 without restoring C, and decrease the whole search time.



5.2. Possible Application: Mbox Locking Problem

One of the features of our framework is that we do not have restriction on the use of OS resources such as the file system in the execution environment. This is because we use a fully-fledged OS (SBUML) as an execution environment on which the target application run. However, the examples we mentioned did not use this feature. Here, we discuss the possibility of a verification target that makes use of such OS resources.

The example we consider here is the “mbox locking problem.” The term “mbox” stands for a particular file format for mail boxes in general, but for simplicity, we mainly focus on the mail spool to which incoming mails are added. The mail spool is a kind of shared resource: MDAs (Mail Delivery Agents) may append an incoming mail to it, POP servers or IMAP servers may discard mails that are already read from it, and some MUAs (Mail User Agents) may directly access it to get new mails.

Of course there are lock mechanisms for mbox, but unfortunately, there are many kinds of them for historical reasons, and some programs support them only partly. Commonly used ones are (1) the `fcntl` system call, (2) the `flock` system call, and (3) the “dot-locking” that tries to create a special temporary hard link. Implementors or system administrators should care about the combination of applications so that it may not lead to bad situations such as deadlock or livelock.

Our framework enables us to run such combination of concurrent programs and verify all possible orders of executions between them without preparing a special environment that emulates the behavior of the file system including locks.

As an example, we consider how to check a combination of an MDA and a POP server. First, we set a breakpoint to each call of a lock primitive such as `fcntl`, `flock`, and `link`. We also have to set blocking conditions if these primitives are blocking, but they are not too difficult (one can test with the non-blocking version and release the obtained lock afterwords). Then, these programs are executed together with a dummy mail generator that continuously generates finite kinds of dummy mails, which are passed to the MDA, in turn. The hash values are generated from the program counters and the kind of the most recently delivered mail. Deadlock can be detected as in the previous examples. Livelock can be detected if the execution contains a loop that does not change the kind of the latest delivered mail.

6. Related Work

6.1. Virtual Machines

With the recent improvement of PC environments, there have appeared a number of virtual machines, and many of them have been put into commercial use. In particular, VMware [4] and Virtual PC [6] emulates a PC/AT compatible machine on Windows/Linux and Windows/MacOS, respectively. A computer that these products virtually realize can execute most operating systems which can be run on a PC/AT compatible machine. Bochs [7] is an open source IA-32 PC emulator that runs on most popular platforms. Another example is Cooperative Linux [8], which is free and open source software designed for running Linux directly on top of Microsoft Windows.

Compared with such virtual machines, SBUML allows flexible control of the virtual machine by providing a script language, which opens the possibility of new applications of virtual machines. In fact, our model checker makes use of the script language.

6.2. Modern Model Checking

In this study, we proposed a new framework for model checking by associating SBUML with GDB. There are many "ultra-modern" software model checkers developed so far.

The SBUML model checker [18], developed in Sato's work, is similar to our system in that it realizes model checking on SBUML. However, the SBUML model checker explores state spaces by using a modified UML scheduler which does model checking at a low granularity. In contrast, the model checking in our system is realized by associating SBUML with GDB and allows scheduling of application processes at any granularity.

CMC [17] is a software model checker for C programs written in an event-driven fashion. It compiles C source code together with the specialized scheduler program and directly model-checks the resulting binary code. CMC implements backtracking by saving and restoring the global variables and the heap.

VeriSoft [13, 12] is a model checker using the partial-order methods based on persistent sets and sleep sets. It verifies programs with stateless search, which is a method for exploring the state space without saving any internal states. It records execution paths of the target processes, and implements backtracking by re-execution of the processes.

Java PathFinder verifies Java source code. Java PathFinder 1 [15] is implemented as a translator from Java to PROMERA, the input language of the SPIN model checker [9]. And Java PathFinder 2 [11] is a tool for directly checking Java programs by itself.

Note that no such model checkers allow actual applications to be model-checked in a real environment like Linux.

7. Conclusion and Future Work

We have succeeded in completing the model-checking of the sample multi-process application within practicable time in the real environment. At the same time, we have succeeded in developing a flexible and practical framework which allows model checking at an appropriate granularity for target code by using the breakpoint mechanism GDB provides. We think that our system can be applied to various applications running under Linux.

Needless to say, much more speed-up is expected by making the exploration of the state space distributed on nodes of a cluster machine. SBUML already has a feature for migrating a snapshot to another PC.

As one of the current restrictions of our system, targets of verification are confined to multi-process applications, because GDB cannot handle selective execution of threads reliably. Another restriction is that a user must manually provide the places to set breakpoints, the blocking condition for each breakpoint, and the necessary inspection functions. Therefore, the automatic generation of those items by static analysis of the target program is future work.

Finally, as a more realistic application, we are targeting the "mbox locking problem" example mentioned in Section 5.2.

References

[1] `http://sbuml.sourceforge.net/`.

[2] `http://user-mode-linux.sourceforge.net/`.

[3] `http://www.gnu.org/software/gdb/gdb.html`.

[4] `http://www.vmware.com/`.

[5] `http://www.ruby-lang.org/`.

[6] `http://www.microsoft.com/windows/virtualpc/`.

[7] `http://bochs.sourceforge.net/`.

[8] `http://www.colinux.org/`.

[9] Spin – formal verification. `http://spinroot.com/spin/whatispin.html`.

[10] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. of SIGPLAN '01 Conference on Programming Language Design and Implementation*, 2001.

[11] G. Brat, K. Havelund, S. Park, and W.Visser. Java PathFinder – a second generation of a java model checker. In *Workshop on Advances in Verification*, 2000.

[12] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, pages 174–186, 1997.

[13] P. Godefroid. VeriSoft: A tool for the automatic analysis of concurrent reactive software. In *Proceedings of CAV'97*, 1997.

[14] K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, and J. L. White. Formal analysis of the remote agent before and after flight. In *Proc. of the 5th NASA Langley Formal Methods Workshop*, 2000. Available at `http://ase.arc.nasa.gov/havelund/Publications/rax.ps`.

[15] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transer*, 1999.

[16] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In *Proc. of the Tenth International Workshop on Model Checking of Software (SPIN)*, 2003.

[17] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. A pragmatic approach to model checking real code. In *OSDI*. Usenix Association, 2002.

[18] O. Sato, R. Potter, M. Yamamoto, and M. Hagiya. UML scrapbook and realization of snapshot programming environment. In *International Symposium on Software Security*, volume 3233 of *LNCS*, pages 281–295, 2003.