Model Checking Tools for Software System Implementations

EPFL Technical Report

LPD-REPORT-2010-003

Maysam Yabandeh

School of Computer and Communication Sciences, EPFL, Switzerland

email: maysam.yabandeh@epfl.ch

Abstract

Systematic State Exploration or Model Checking techniques have been used for years to check the model of softwares against user-specified properties. Nevertheless, they never achieved a wide-spread usage because of the difficulties and problems in translating from the programming languages, which are used to develop the software, to the modeling language on which the model checker can work. Recently, there have been several efforts in direct state exploration of software system implementations. In this survey, we illustrate the challenges in this domain and explain the different solutions

adopted by the state-of-the-art developed tools for state exploration of software systems. The focus of this paper in on the developed model checking tools for software systems, and it does not include solutions for unit testing and selecting the optimal test scenarios.

keywords: software systems, distributed systems, reliability, testing, model checking, systematic state space exploration

1 Introduction

Systematic State Exploration, which is also known as *model checking*, has been used for years to systematically explore reachable states in a model and to verify them against user-specified properties. The model can be made based on a software system, a hardware system, or any general phenomena. The languages that are used to develop software systems are different from the modeling language, and the developer has to go through the tedious, time-consuming task of translating the original program into the modeling language, i.e., *abstraction*. These difficulties discourage the developers to use the model checker tools for large software systems.

To alleviate these problems, two general approaches have been used: i) automatically generating a model of the software, and ii) directly exploring the state space of the original software itself rather than a model of that. The former approach still suffers from the drawbacks of translation into a model; due to mismatches between the original software and the abstracted model, the found bugs

are not sound. In other words, the reported bugs can not necessarily manifest in the original application as well.

The second approach, which is the focus of this paper, systematically explores the state space of the software system by controlling the scheduler and the inputs to the software. In this way, the developers can check the state space of their software without going through the error-prone, expensive task of abstraction. Although appealing, there are certain challenges in this approach that affect both applicability and efficiency of such tools. In this paper, we demonstrate these challenges and the different solutions adopted by the related tools.

In this paper, we refer to the employed algorithm for state exploration, as exploration algorithm or search algorithm. Given a explored state, the search algorithm invokes a property checker module on that. The property checker could check for deadlocks or user-specified properties which are also known as invariants. The assert statements inside the source code can either be checked by the property checker or their violation can be trapped and be reported to the developer.

Similar to the tools for checking the models, the main challenge is still the state space explosion issue: the phenomena of exponentially increase in number of states by exploring deeper levels. However, here more number of non-determinisms as well as the large size of the real applications worsen this problem in a way that exhaustive exploration techniques become ineffective. Section 2 demonstrates this problem and its direct correlation with non-determinisms in the software systems. Section 3 discusses that how we can alleviate the state explosion problem by more precise emulation of the environment. Section 4, explains

```
1 bool x = read();
2 bool y = read();
3 if (x && y)
4     z = 5;
5 else z = 8;
6 if (x)
7     z = 2 * z;
8 else z = 0;
```

Figure 1: Snippet of code used for state exploration.

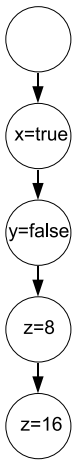

Figure 2: State space obtained by running the program in Figure 1. The changes in the state are shown inside the circle.

the changes in the exploration algorithms that can make efficient exploration of such large state spaces feasible. Finally, in Section 5 we compare the state-of-the-art tools and illustrate how each of them addresses the described challenges.

2 State Explosion Problem

The processing unit (CPU) that runs a software program computes only the single next state of the system by executing the next instruction in the program. Fig-

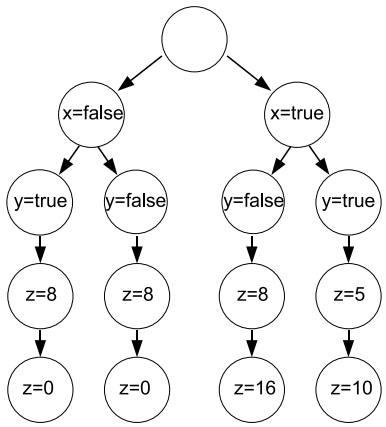

Figure 3: State space obtained by applying early binding in state exploration of the program in Figure 1. The changes in the state are shown inside the circle.

ure 2 depicts the state space explored by running the sample program presented in Figure 1. In contrary to that, more than one state can result from an instruction execution in systematic state space exploration. For example, when an if-then-else instruction checks for a variable that its value is received from an input device such as keyboard, the next state of the system depends on the concrete value of the variable. Although this value will be determined at run-time, during state space exploration it is a source of *non-determinism* for the search algorithm. Corresponding to each non-determinism point, a branch will be introduced to the state space to cover all the possible options. Taking each branch leads the system to a potentially different state. The number of reachable states exponentially increases with the number of branches, which is known as *state explosion* phenomenon.

2.1 State Space

As mentioned before, the space of potential states that a software can visit is way larger than the set of visited states by a particular run. For example, the state space of the snippet of code shown in Figure 1 is illustrated in Figure 3. As you can observe, in contrary to the visited states by running the program, the state space is not sequential; there exist branches corresponding to each non-determinism point. Each set of joint branches bind some values to a particular non-determinism point. For example, corresponding to Variable x which be assigned at run-time, there is a branch at Figure 3 to cover both values *true* and *false*.

The policy that specified the position that a branch appears in the state graph is determined by the exploration algorithm. The policy defines when the possible values should be bound to the non-determinism points.

2.1.1 Early Binding

In early binding, a branch covering the possible values is added when the variable is assigned. For example, in the snippet of code shown in Figure 1, Boolean x is assigned by reading from keyboard. In Figure 3, which is the visited states following by the early-binding policy, the first branch is added immediately after assigning the value to Boolean x in the program.

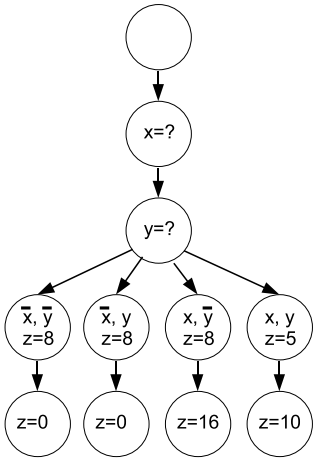

Figure 4: State space obtained by applying late binding in state exploration of the program in Figure 1. The changes in the state are shown inside the circle. The variable with bar are assigned to false and vice versa.

2.1.2 Late Binding, Before Usage

In late binding before usage, a set of branches covering the possible values is added right before the assigned variable is actually used [3]. For example, in Figure 4, which depicts the visited states following by the late-binding policy, the first branch is added right before Variable x is used at Line 3. The resulting state space by following the late binding policy is equivalent to the state space explored by early binding policy. However, it could potentially be more compact by merging equivalent states. The disadvantage is the slight added complexity for late binding.

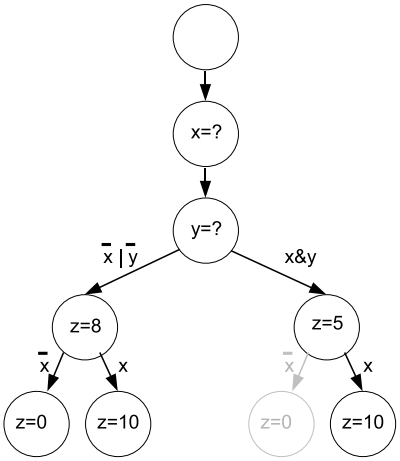

Figure 5: State space obtained by applying symbolic execution on program in Figure 1. The changes in the state are shown inside the circle. The conditions on the edges represent branch conditions inside the program. The bar symbol is used for negation.

2.1.3 Late Binding, During Property Evaluation

The binding can be postponed till the time the model checker evaluates the user-specified properties against the system state. Since the concrete value is not bound, a *symbolic* value is kept for the variable. The model checker then has to keep track of the operations performed on the symbolic value and pass them to the property checking module. This technique is called *Symbolic Execution*.

The major challenge in symbolic execution is to executing *conditional branch* instructions. Conditional branch in programming languages is the essential part for implementation of if-then-else and loop statements [1]. In the normal run of the program, the CPU jumps to the position specified by the instruction, if the corresponding condition is evaluated to true. In symbolic execution, the model checker

[1]Note that conditional branches are different from the branches in state space graph

does not have the concrete values to evaluate the condition. Therefore, a branch is added to the state space to cover both true and false values. In Figure 5, the first branch is added right before evaluating x && y in the first if-then-else statement of the code snippet shown in Figure 1. By taking each branch, an assumption is made about the evaluation of the condition. The set of assumptions made along the path will be passed to the property checking module.

Note that the state space graph obtained by symbolic execution is not equivalent to the state space graph obtained by early binding. Here, the branches in the graph correspond to branch points in the program structure. Corresponding to each if-then-else statement, a branch is added to the current position at state graph.

Not all the branches in the state graph that is obtained by symbolic execution are valid; some branches can be impossible to be traversed in a real run. The property checking module can help to prune some impossible branches. It can be used to evaluate the condition in the instruction; if the condition is evaluated as false, then the corresponding branch will be pruned, and vice versa. In Figure 5, the grayed branches are pruned from exploration. Consequently, the size of the space graph can be potentially much smaller than the case with early binding.

The disadvantage of symbolic execution is that the complexity is pushed to the property checking module. However, there are well-known off-the-shelf SAT solvers, which take the history of operations performed on the symbolic variable as well as the list of assumed conditions and return weather the condition is satisfiable or not. The problem is that computation time of SAT solvers is not bounded,

and it can take forever for a SAT solver to response. Besides, the size of the assumed condition set and the sequence of performed operations could become too large to be efficiently handled by state-of-the-art SAT solvers. For these reasons, Symbolic Execution can also quickly stick in the state explosion problem.

2.2 Modeling Environment

The only thing that the state space exploration algorithm can be certain about is the sequence of program instructions. The sequence of states that will be explored by a program run depends on the particular environment that the program is deployed in. The *environment* is everything except the sequence of instructions in the program, which includes the hardware, the operating system, the communication environment, the input devices, the time, and also the other programs that will interact with the software. Every uncertainty about the environment introduces a non-determinism point into the search algorithm and worsens the state explosion problem. On the other hand, every assumption about the environment eliminates the corresponding non-determinism point and alleviates the state explosion problem.

One major problem of model checking of software systems is the numerous unknown parameters about their potential environment. The number of states, therefore, grows very quickly in a way that exhaustive search algorithms become totally ineffective. However, by assuming a model for each part of the environment, we can reduce the uncertainty regarding that part and consequently reduce the corresponding branching factor, i.e., the number of immediate states after the

branch joint.

Another advantage of modeling the environment is eliminating the states which are impossible or improbable for the deployed software system to get into. For example, if we know that the input values to the system are always non-zero, we can ignore the branches that check for zero values. Beside the reduction in number of explored states, the search algorithm does not report the invariant violations that are impossible or improbable to occur in practice. Therefore the accuracy of the search algorithm increases and the number of false positive reports reduces.

The disadvantage of modeling the environment is that the state exploration software becomes i) more complicated and ii) more environment dependent. For each model, we have to add some new logics that implement the model to the state exploration system, which makes the state exploration system complicated. On the other hand, every model makes some assumptions about the environment in which the software will be deployed. These assumptions could change from system to system or from time to time. However, the expenses for updating the model might not be trivial. For example, using a model of TCP rather than its actual implementation is one major source of complexity in the state exploration tools. Moreover, the model has to be updated when new versions of TCP are deployed.

In the next section, we categorize the parts of the environment surrounding the software systems and discuss the employed techniques for modeling them or reducing uncertainty regarding them.

3 Environment

We define the environment as all the elements that will directly or indirectly affect the behavior of the deployed system. For each part of environment that we are uncertain about its behavior, we must explore all the possible actions and reactions in model checking. Uncertainty regarding each part of the environment can lead to more non-determinism points in the exploration process and consequently worsen the state explosion problem. By assuming a model for each part of the environment, we can further reduce the uncertainty and hence alleviate the exponential growth of the state space.

In this section, we categorize the different parts of the environment and present different approaches that have been taken to reduce the uncertainty regarding each category. For a software system, we can split the environment into three general categories: i) the upper layer applications, which uses the provided service by the software, ii) the lower layer services, which supply the software with some services, and iii) the peers, which are the identical replicas of the software with whom the software interacts. In the following, we explain each category in more detail.

3.1 Upper layer application

Every software supplies the users with some interfaces to use the provided service. The users can range from human operators, who interact with the software via GUI (Graphical User Interface), to other software systems, which interact via IPC

(Inter Process Communication) or via invoking the software's public functions. For the sake of simplicity in this section, we focus on general form of function calls that can accept some input parameters as well.

The order of the calls and the content of the input parameters can affect the next state of the system, and a systematic state exploration algorithm has to consider all their possible values and orders. However, considering all the possible values for input parameters is not always feasible. For example, for a 32-bit integer variable the number of possible values is 2^{32}. Having no model for the upper layer, the exploration algorithm has to check all these values to achieve a complete search [2]. The test driver plays the role of the model for the upper layer application by focusing on a limited set of application requests. For instance, a test driver for a database service sends a particular set of queries to the database software.

The test drivers are obviously not complete. There is a trade-off between the completeness and feasibility in systematic exploration. In the case of large software systems, it is inevitable to sacrifice the completeness for feasibility of the search. In Section 4 we will see that the exhaustive exploration algorithms have to be bounded to some depths anyway. More accurately selected test scenarios would lead to systematic exploration of more relevant and important states of the software system.

[2] Symbolic execution techniques can check for much less number of values by considering a symbolic value for the variables instead of concrete values.

3.2 Lower layer services

No software implements all the required functionalities from scratch. Each software is supplied with some functionalities by the lower layer services, such as libraries, operating system, and other software services. For example an application in C++ uses some library functions to obtain the current time or to communicate with other softwares through TCP. The implementation of this kind of functionalities can vary from deployment to deployment. Therefore their behavior, which is sometimes even dependent to the physical environment, is not fixed, and their concrete return value is not determined before deployment. For example, depending on the network traffic, a sent packet through TCP can be arbitrarily delayed.

In the following, we cover some important lower layer services which are the major issues for most of the applications: i) Time, ii) Random Values, and iii) Communication Objects.

3.2.1 Time

The current system time is usually provided by a special hardware on the motherboard. The operating system supplies the applications with some interfaces to inquire about the current time. The usage of current time varies from application to application; examples are triggering a scheduler, assigning time to items in the database, and using it as a seed value to pseudo random generators. There are too many possible values for the current system time which makes iterating over all of them infeasible. On the other hand, it is unrealistic to assume a model for the

time as the system can be deployed and run at any given time.

A simple, common solution for the time issue is to use a monotonically increasing counter as a model for the physical timer. The counter increases by a constant each time that a thread reads the current time. Obviously, this model in not accurate and any exploration algorithm on top of that will not be complete. The time issue in general is still an open problem in systematic exploration of software state space. Nevertheless, some attempts have been made to tackle this problem in some specific domains. In the following, we discuss these approaches and their limitations.

Scheduling One major usage of the current time is to trigger the scheduled timers. Usually a dedicated scheduler thread in the system regularly checks for the current system time; if the current time is passed the scheduled time of a timer, the scheduler invokes the timer registered function. Specialized libraries often handle the details of scheduling, and the only thing that the developer should do is to call a specific function to schedule the timer. On the other extreme of the spectrum, the developer might have implemented its own scheduler, weaved into the source code, beside the other implemented functionalities.

In the case of specialized libraries for scheduling, one solution is to model the whole scheduler instead of dealing with the difficulties of the concrete time value. Having a model for the scheduler, the systematic exploration algorithm requires considering only the different order of triggering the timers and not the exact time for trigger.

Operating systems also do scheduling to share the processing units among

multiple threads and processes. The actual time of the system as well as the behavior of threads affect the thread preemption pattern. The preemption point is important because it affects the order in which the threads access the shared resources. Similar to the used models in the application layer schedulers, we can use a model for the operating system scheduler. In this way, the exploration algorithm only considers the different preemption points in threads and does not get into details of the concrete time value.

The scheduling functionalities weaved into the source code are still a challenge as the exploration algorithm cannot be sure about the precise usage of the requested time value. One approach is to use symbolic execution to analyze the way the time value is used inside the program. If the time value is used only in simple adding operations and comparison tests, the state exploration software could check only for a limited set of time values that are enough for covering the different branch decisions that are made based on the time value. Depending on the application, this approach can be effective if the time value is used immediately after its request and also in a simple way, i.e., only simple mathematic operations such as add and subtract. The limitations comes from the fact that symbolic execution can be feasible only till a certain depth and after that it also faces state explosion problem.

The current time can also be used as the seed parameter passed to pseudo random generators. Without the seed value, pseudo random generators will generate the same sequence in all runs. In the next section, we explain the problem of random values in more detail.

3.2.2 Random Values

Pseudo random generators are used to obtain a random number. Their usage in software programs raises a difficulty for state space exploration, as we have to consider all the possible return values for the random number. Like the issue of time, the range of possible values for random numbers is too large to be exhaustively checked. There are application-specific solutions though that we explain in the following.

Load balancing Some random values are used to pick an item among a few choices. In this case, the application of the randomness is to balance the load over several processes (or entities in general). The key point here is that even though the developer intends to select among a few items, she might use the general form of random functions, which return a float value between 0 and 1. The simple solution is to supply the developer with some library functions that let them to invoke the appropriate function for choosing a member among a set. We can register the mentioned functions to the state exploration software and replace them with a simple model during exploration. The model simply adds a branch corresponding to all the items in the set.

Although simple, the above solution might not be practical in all cases. The legacy applications still use the general form of the pseudo random functions. Even for the new applications, we cannot guaranty that the developer will always stick to the policy and will use the provided high-level random functions. Similar to time issue, a solution based on symbolic execution can be applied here; it can follow the usage of the returned random value in program instructions. In such

cases, the random value usually ends up in a switch-case command to pick an option. The exploration algorithm could then check only for a limited set of random values that are enough for covering the different branch decisions that are made based on the random value.

Scheduling Random values are also used to schedule some timers. For example, some transport protocols wait for a random duration before retransmitting the data. The benefit of random duration is to avoid network congestion that is caused by several transmissions at the same time. From the exploration algorithm perspective, the concrete value of the random duration does not matter. Nevertheless, the different order of triggering the events, which is resulted from the random duration, is important. The scheduler model, therefore, can handle this usage of random numbers. The challenge for the state exploration tool is to either provide dedicated interfaces for this kind of usage, or to detect the usage that is weaved into the source code and then apply the scheduler model on that.

3.2.3 Communication Objects

One major role of the operating system is to provide mechanisms for the processes to communicate with each other. The communication object can range from simple file system interfaces to specialized interfaces for Inter Process Communication (IPC). The most complex communication objects are the transport protocols for communication over asynchronous network such as TCP.

The access to communication objects must be through operating system. The operating system includes logics for accessing the communication objects and

the processes can access them by some proper system calls. Since this logic is not included in the software application, the exploration algorithm does not know how to execute the system calls invoked by the application. There are two different approaches in tackling this issue. One solution is to include the operating system into the state exploration process; after the application invokes the system call, the state exploration algorithm runs the corresponding logic inside the operating system and returns the results. The other approach is to model the communication object and simulate the effects of the system call on the model.

Including operating system into the state exploration process makes the state exploration to be operating system-dependent; the results might be different if the software system is deployed on another operating system or a different version of the same operating system. Furthermore the state of the operating system (which can be very large) has to be included in the state exploration process.

Beside the large size, the main difficulty with operating system is that it is not easily controllable. The exploration algorithm must be able to initialize the software state and reproduce a specific sequence of events. This is feasible in the case of a process which has a clear memory footprint. It is, however, a challenge to reproduce the sequence of events in a large operating system, full of non-determinism points.

A new challenge arise when the communication object itself is not under control of the operating system, and its behavior is, hence, not predictable. For example, when a packet is sent through network, there is no guaranty on its delay, loss, and unwanted duplication. Hence when the state exploration algorithm wants

to reproduce a sequence of events that involve a packet transmission, there is no way for the state exploration algorithm to force the network to apply the same delay, loss, and duplication pattern. These parameters are also another source of non-determinism that a complete exploration algorithm has to consider. They are, however, beyond the control of exploration tools.

Models can hide the complexities of the operating system services and increase state exploration performance. For example, a PIPE in Linux operating system can be modeled by a simple queue structure. The problem with models is that they are valid as long as conform to the implementation of the original service. In the case of complex services such as TCP, the model is not trivial and can be very complicated. This increases the risk of a mistake in modeling the service as well as expenses of updating the model according to the new changes in the service implementation. The other advantage of using models is that due to simplification in the model, the state of the model is much simpler and more controllable. Thus, it will be feasible to reproduce a series of events on them.

The expenses of modeling the operating system and its maintenance increase with the increase in the number of operating system services. For example, Windows offers more than 100 system calls [17]; modeling all these system call in an operating system which has more than one million lines of code is very expensive and unreliable.

3.3 Peers

In centralized systems, only one copy of the software exists. This is in contrast with distributed systems where several copies of the software (i.e., peers) are working concurrently. The state of the system is then distributed between peers, and a perfect state exploration algorithm has to take them into consideration.

An approach is to explore states of only one copy of the software and consider other peers simply as part of the environment. Although simple, this would increase non-determinism in the state exploration and thus makes the state exploration less efficient and less accurate.

The other approach is to start the exploration with N peers where N is a fixed number. If we take N small, the inconsistencies that would only manifest for larger N stay undetected. On the other hand, the number of states increases exponentially with the increase in peer count, and the state exploration, hence, is impractical for large values of N.

Some related works [15] take the middle ground: they start the state exploration for a large number of peers, N. However the exploration algorithm executes only the events that are related to a small set of nodes (with size of M), and ignores the events which are related to the other nodes. For $M \ll N$, the state exploration can be efficient although it only partly stresses the system.

4 Exploration Algorithm

In the simplest from, the exploration algorithm at each step computes the ready events, picks one event from the ready event list, and executes it. To make the search complete (till a certain depth), the exploration algorithm has to execute all the events in the list; this approach is also known as exhaustive search. It is possible to do this by forking the application process. However, that would be very inefficient and would quickly run out of memory.

Two well-known algorithms for exhaustive search in the state space are Breadth First Search (BFS) and Depth First Search (DFS). The BFS algorithm saves the current state of the exploration algorithm, which includes the ready event list, the picked event, application state, and the environment state, in a queue. At each step, it dequeues one item from the queue, executes all the ready events one by one (after each execution it rolls back to the dequeued state) and enqueues the resulting states. By that, it explores all the states at a breadth before going to the next breadth. The BFS algorithm is very memory consuming and thus impractical for non-toy software applications. Although it can be fast, since it has to execute each event only once, the costs of taking the snapshot of the system state might not be trivial. We will discuss this more in the next subsection.

The DFS algorithm keeps only the system state along the path from the root till the current position. When the algorithm reaches the maximum depth or runs out of ready events, it backtracks one depth upper, load the system state, and iterates over the next event in the ready event list. The DFS algorithm requires

much less space compared to BFS and hence are more suitable for exploring large state spaces. Since it is infeasible to exhaustively explore all the state space in the non-toy software applications, the maximum depth of DSF is usually bounded (BDFS) to guaranty complete exploration till a certain depth. After exploring all states till the bounded depth, the maximum depth is increases by a constant and the algorithm starts over from root. This is computationally less efficient compared to BFS because of duplicate execution of events in the next rounds.

Both BFS and BDFS suffer from the expensive operations of storing and loading the whole system state from memory. In the case of software systems, the system state includes the whole memory footprint of the application and the environment (files, operating system state, network state, and etc.). In the following, we present a version of BDFS that is more efficient where keeping the whole system state in memory is expensive.

In BDFS algorithm, instead of the full system state we can only keep the index of the picked items from the root till the bottom of the state space. We call this I-BDFS. When the algorithm reaches the maximum depth, all the states along the path from root to the current state are checked. To backtrack it needs to obtain the system state for the last step. It obtains the last state by starting over from the root and executing the same sequence of events (picking the same index from the ready event list). Since the maximum achievable depth is shallow anyway, the expenses of re-executing the events are often less than expensive operations of storing and loading states from memory.

The challenge in I-BDFS is reproducibility of the event sequence; after execut-

ing the same sequence of events we expect the system to reach the same state. As we discussed in Section 3, there are lots of non-determinism in the environment which are not necessarily under control of the exploration algorithm. For example, the implementation of a system call inside the operating system might use some random values which are different at each run. Using models can address this problem since the model are intentionally developed to be controllable.

4.1 Stateless or Stateful

To avoid loops and exploring duplicate states, it is necessary to keep track of the visited states. As explained above, this is an expensive operation for software systems with large state size. To alleviate the cost, one approach is to obtain a hash of the state and keep track of the hash codes instead of the whole state. Although it reduces the required memory space for keeping the state as well as the cost of comparison between two states, nevertheless, obtaining the state hash still requires touching the whole state once which can be nontrivial in the case of large states.

Another approach is stateless exploration as opposed to stateful search. Visiting duplicate states makes stateless approach to be very inefficient. Using Partial Order Reduction (POR) techniques, can remedy the performance if we assume that the state space graph is acyclic. We explain POR in the next section.

4.2 Partial Order Reduction

The POR technique can improve performance of any of exploration algorithms described above. However, its usage is inevitable in the inefficient stateless approach to avoid visiting duplicate states. Recall that the stateless approach is interesting in state exploration of software systems because taking a hash of the system and environment state is very expensive in large software systems.

The POR techniques prune the state space of a concurrent system to avoid unnecessary interleaving of events. For example, if executing $< s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} s_2 >$ and $< s_0 \xrightarrow{e_2} s'_1 \xrightarrow{e_1} s_2 >$ result in the same state, exploring only one of them is enough for checking the invariants of the software system. In this case, e_1 and e_2 are called independent. Independence is not enough to prune e_2 from the state space graph. It is because of the fact that there might be other events enabled at state s'_1 that following them gets the system into states which are not reachable from s_1. To be able to prune e_2, we must first prove that e_1 and e_2 are in a *persistent* set at s_0.

Obtaining independent events and persistent sets requires static analysis on the source code. Static analysis tools might not be available in all programming languages; specifically, if the environment (such as operating system) is included in the analysis. For example, independence of two operations which use different system calls is not easy to prove. Furthermore, it is shown that static analysis is not efficient in dealing with dynamic data such as pointers [1]. This is because the value referenced by the pointer is not available at the time of analysis.

Dynamic Partial Order Reduction (DPOR) [1] is designed to solve the limita-

tions of static analysis. The DPOR algorithm, computes the dependency during exploration, when the concrete values of the pointers are available. According to the observed dependencies, it adds appropriate branches to guaranty the completeness of the exploration. The limitation of DPOR is that it works only for multi-threaded programs and is not applicable to distributed systems. DPOR-DS inspired from the main insight of DPOR and design an algorithm for distributed systems [16].

There are other techniques based on static analysis named by *sleep sets* which are beyond the scope of this paper.

4.3 Big Steps

Where more than one process are being model checked, the model checker should consider different interleavings between the processes. This is because the process shared variables can change by other processes. Therefore, after executing each atomic instruction, the model checker should add a branch for the case that the thread is preempted and another thread carries on.

By taking big steps, the model checker assumes a sequence of instructions as a big atomic instruction and do not preempt the process in the middle of their execution. Obviously, it alleviates the state explosion problem by reducing the number of branches. For example, Chess [10] limit the number of preemptions per each thread in model checking of multi-threaded programs. In this case, big step is a trade-off between completeness of the exploration and its feasibility in a limited time.

Taking big steps does not necessarily make the exploration incomplete. For example, the sequence of instructions that do not touch shared variables can be assumed atomic, since preemption in the middle of them is equivalent with the preemption after them [1]. MaceMC [8] also takes the whole instructions inside a handler as atomic. The exploration is complete since the Mace system is event-based and the handler code will not be interrupted with another handler execution.

From a high-level point of view, models of the environment always take big steps. Each operation on the model can be equivalent to multiple steps in the real environment counterpart, which potentially can be interrupted.

4.4 Parallelizability

In the multi-core and cloud computing era, it is a must for software tools to be parallelizable over multiple cores. This is much more important in state exploration of software systems, since a single thread cannot go very deep into the large state space. However, the shared variables such as the history of the visited states make it difficult to efficiently parallelize the task.

4.5 Heuristics

In state exploration of small models, since the state space was small, it was a key feature for the exploration algorithm to be complete. It is a fact that due to the large state space of the real software systems, the completeness is not an objective anymore. Therefore, the heuristics are more welcome; the heuristics

which sacrifice the completeness of the search to explore more relevant states in the limited time of state exploration.

4.5.1 Random walk

The simplest form of exploration heuristic is random walk: to pick one event from the ready event list and expand only that particular branch of the state graph. The random walk can go very deep in the state space. Nevertheless, it also misses exploring some states that are accessible from the initial state by a few steps. One intuitive way to address this problem is combining the exhaustive search and random walks. The different possible combinations of these two are discussed in [12].

In the pure random walk, the likelihoods of exploring a very rare state and a very common state are the same. Depending on the objectives of the testing, it might be more desirable to explore the states that will be mostly visited after deployment of the system. One approach is to assign weights to the events and randomly pick an event from the ready event list according to their weights. For example, the chance of a packet drop is very low in the network and the assigned weight to that could be low. A more complicated approach can analyze the log files to obtain the probability of different sequence of events. *Bayesian networks* [6] sounds like a right match for this purpose.

4.5.2 Initial state

The root state in the explored state graph is normally the initial state of the software system and the environment. Due to state explosion problem, the depth of a complete exploration would be limited to few steps. This does not allow the system to be stressed against complicated configurations. Starting a complete exploration after a long random walk would alleviate this problem. In [8], it is proposed to disable the faulty events such as packet drop and connection break during the random walk. This allows the system to get into a stable state before starting the complete search.

Another approach is to obtain a system state from an actual live run and use this state as the initial state in the exploration [15]. The advantage of this approach is that the exploration starts from a state in which the system has gone through complicated interleaving of events. Moreover, the explored states would be more relevant as they are accessible from a state taken from the live run. After a few steps, the exploration can be restarted from another state also taken from the live run.

4.5.3 Event Interleaving

The non-determinism in event interleaving is a major contributor to exponential growth of the state space. POR techniques, which are sound and complete, alleviate this problem slightly. Nevertheless, the state explosion still manifests after a few steps. Because of that, most of the developed tools for model checking of

software systems were forced to eventually rely on random walks [8, 17].

Realistically speaking, in large software systems the completeness property of POR techniques is not appealing as much. Therefore, the heuristics which are not complete but leads the search to more relevant states are more interesting. One example is *Consequence Prediction*, which is proposed in CrystalBall [15]. It filters a non-network handler, if the handler is already run on the same node local state, i.e., the process state.

5 Related Tools

In this section, we explain the design of the developed tools for model checking of software systems implementations.

Verisoft To avoid challenges of large state size in software systems, Verisoft [2] takes the stateless approach. POR techniques are then applied to alleviate the drawbacks of the stateless approach. Since efficient persistent sets require information about the static program structure, POR techniques used in Verisoft are mostly successful in reducing the number of transitions (because of using sleep sets) rather than number of visited states.

It uses test drivers as a model for the application layer. The test driver should use Verisoft specialized functions: VS_toss and VS_assert. VS_toss(n) is offered to pick a random number between 0 and n. Calls to communication objects is intercepted and handled by models of the communication objects, although the paper does not discuss the methods for intercepting the calls on communication

objects. In an operating system with more than hundreds of different system calls, intercepting all of them and replacing them with a model is very challenging.

The safety properties checked by Verisoft are deadlocks and user-specified assert statements. Verisoft does not discuss the random numbers that are used inside the program as well as the time issue.

Verisoft takes big steps in model checking by dividing the instructions in two visible and invisible groups. A visible instruction executes an operation on a shared object. The set of invisible instructions between two visible ones are considered atomic with the last visible instruction.

A free download is available at [13].

Java Pathfinder Java Pathfiner (JPF) [4], is an explicit state model checker for Java bytecode. It checks for deadlocks and user-specified assert statements. JPF follows the stateful approach for state exploration. The Java Virtual Machine (JVM) is instrumented to store/load the application and exploration algorithm states.

JPF offers specialized methods for picking a random value, Verify.randomInt(n). It also offers a random function for double values, Verify.randomDouble(). However, the returned values do not systematically cover the whole range, and a user-defined heuristic model is used to choose only one single returned value.

To model the environment, the user has to write some model classes which emulate the environment. For each method in the model classes, JPF automatically intercepts the corresponding calls from the application and return the control

to the model class (instead of original class in JVM).

It applies on-the-fly POR, which is similar in spirit to DPOR, to alleviate the state explosion problem. To identify dependent operations at run-time, they monitor read/write operations on the shared objects. To make it less expensive, they suggest that the monitoring piggybacks on garbage collection.

In the original version of JPF, the upper layer application must be modeled by a test driver. The recent version of JPF is instrumented with symbolic execution to address this limitation [14]. The developed techniques tackle the aliasing and dynamically generated objects. They use late binding (called lazy initialization) to make the state space graph smaller. Note that the environment must still be modeled.

The source code is available at [5].

CMC In contrast to Verisoft [2], CMC [11] takes the stateful approach for model checking of C programs. The global variables and the heap content is stored and loaded for switching the state. The user also specifies some memory locations that she thinks are not necessary for the model checking purpose, to be eliminated from the saved states.

CMC applies on event-driven applications, and the whole implementation of a handler is taken as a big atomic step. However, the user has to manually specify the handler boundaries. The user also has to define some functions for initialization. CMC checks for user-specified asserts as well as memory leaks.

The upper layer application is modeled by test programs. The operating system calls and specially the network calls are also replaced by some models. To

model time, CMC offers a specialized function to obtain the current time, gettimeofday(), which will be replaced with an autonomic counter during model checking. The random values also can be obtained by invoking CMCChoose() function. During model checking, the returned values will cover the whole range of options. The paper, however, does not specify that how it motivates the users to use only this particular offered functions.

The exploration algorithm is BFS. Since the elements in queue are referenced in order, the queue of states in BFS can be kept mostly in hard disk rather than memory. This alleviates the problem of keeping track of large states. Nevertheless, loading and storing of large states still is a time-consuming task.

To my knowledge, the source code of CMC is not available.

MaceMC CMC [11] requires user involvement in various phases such as specifying initialization functions, the handler boundaries, and the important parts of state. MaceMC [8] takes advantage of the fact that these steps are mostly done in structured programs written in Mace framework [7]. In Mace language, the initialization function and the handler boundaries are part of the language. The framework also offers some utility functions such as serialization of state into a stream.

Similar to CMC, the big steps are specified by handler boundaries. The upper layer application is modeled by a test program. In Mace language, the used services by the program are explicitly specified in the source code. During model checking, the user can make an instance of such services in the test driver and pass them to the Mace program. The alternative solution is to make a correspond-

ing model class and pass it instead. The native operating system services are not modeled and hence are beyond the control of MaceMc. It, however, offers some wrappers for services such as UDP and TCP, and the developer is encouraged to use them. These wrappers must be replaced with corresponding models offered by Mace, in the test drivers.

The exploration algorithm is stateful and consists from a combination of I-BDFS followed by random walks. The state is loaded only for creating the initial state and the intermediate states are obtained by rerunning the event handlers.

The offered randint() function for random values is instrumented to cover the whole range during model checking. Moreover, if the developer uses the specialized time function offered by Mace, it will be replaced by a monotonic counter during model checking.

The source code is available at [9].

Modist Modist [17] is a stateless model checker for unmodified distributed systems in Windows. The big steps for model checking are the code between two system calls. It instruments the application binary to replace the system calls with some API wrappers. The wrapper mostly contacts the exploration back-end with RPC and then invokes the original system call or returns failure, depending on the back-end response. In the case of networking APIs, the wrapper redirects the call from the original networking API to a network model. The model is implemented by an asynchronous IO channel as well as a proxy thread for intercepting the packets. Therefore, the other processes can be located on remote machines.

The time issue is addressed by using symbolic execution starting from the

invocation position of the time function. The back-end tries different values of time that cover all the branches in the code till a certain depth. The paper does not propose a solution to random functions and their appearance in the application or the operating system can cause difficulties in deterministically replaying the error path.

The exploration algorithms are a heuristic inspired from DPOR, as well as random walks. To alleviate the state explosion problem, the number of injected faults into the system calls is limited.

The source code is not publicly available.

6 Conclusions

State Explosion phenomenon is still the major hurdle in model checking of large system implementations. Therefore complete search techniques such as partial order reduction are not compelling anymore. Instead, heuristics for moving the search towards more relevant states are totally welcome. The heuristic can be applied in different parts of the model checking process, such as modeling the environment, exploration algorithm, and initial state.

Storing/Loading of states in large software systems is very expensive, both in terms of time and memory. The stateful approach used to be necessary to achieve complete search. Having feasibility been prioritized over completeness, stateless approach sounds more realistic for quick exploration in state space.

Currently, modeling the environment is achieved mostly by heuristics. Precise

definition of the process as well as tools for automation of this process would make the effective model checking of software systems one step closer to reality.

Time and random values are still two big problems in model checking. Although application-specific solutions have been proposed, there is still no approach to force the developers to use them. Perhaps, the future programming languages can be *model checking-aware* in the sense that they force the developers to use only the mechanisms that are already instrumented to be used in model checkers.

References

[1] Cormac Flanagan and Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *POPL*, 2005.

[2] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186. ACM New York, NY, USA, 1997.

[3] T. Gvero, S. Khurshid, V. Kuncak, and D. Marinov. On Delayed Choice Execution for Falsification. Technical Report TR-2008-08, EPFL, 2008.

[4] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, 2000.

[5] Java pathfinder download page. http://javapathfinder.sourceforge.net.

[6] P. Judea. Bayesian networks: a model of self-activated memory for evidential reasoning. *Cognitive Science Society, UC Irvine*, pages 329–334, 1985.

[7] Charles E. Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: Language Support for Building Distributed Systems. In *PLDI*, 2007.

[8] Charles E. Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI*, 2007.

[9] Macemc download page. http://www.macesystems.org.

[10] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P.A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI.08)*, pages 267–280, 2008.

[11] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, 2002.

[12] H. Sivaraj and G. Gopalakrishnan. Random walk based heuristic algorithms for distributed memory model checking. *Electronic Notes in Theoretical Computer Science*, 89(1):51–67, 2003.

[13] Verisoft download page. http://cm.bell-labs.com/who/god/verisoft.

[14] W. Visser, C.S. P.s.reanu, and S. Khurshid. Test input generation with Java PathFinder. *ACM SIGSOFT Software Engineering Notes*, 29(4):97–107, 2004.

[15] Maysam Yabandeh, Nikola Knežević, Dejan Kostić, and Viktor Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *NSDI*, 2009.

[16] Maysam Yabandeh and Dejan Kostic. DPOR-DS: Dynamic Partial Order Reduction in Distributed Systems. Technical Report TR-2009-05, EPFL, 2009.

[17] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI*, April 2009.