Slicing and Dicing Bugs in Concurrent Programs

Neha Rungta
NASA Ames Research Center
Moffett Field, CA 94035-1000
neha.s.rungta@nasa.gov

Eric Mercer
Brigham Young University
Provo, UT 84660
eric.mercer@byu.edu

ABSTRACT

A lack of scalable verification tools for concurrent programs has not allowed concurrent software development to keep abreast with hardware trends of multi-core technologies. The growing complexity of modern concurrent systems necessitates the use of abstractions in order to verify all the expected behaviors of the system. Current abstraction refinement techniques are restricted to verifying mostly sequential and simpler concurrent programs. In this work, we present a novel abstraction refinement technique that uses program slicing based on reachability properties to generate an initial abstraction over a single thread. The initial single thread abstraction is coupled with a concrete execution to drive the single thread; generating an underapproximation of the program behavior space. If the target location is reached in the underapproximation, then we have an actual concrete trace. Otherwise, the initial abstraction is refined to include another thread that affects the reachability of the target location. In this case, the concrete execution only considers the two threads in the abstraction and preemption points between the threads only occur at locations in the abstraction. This refinement process is repeated until the target location is reached or is shown to be unreachable. Initial results indicate that the incremental technique can potentially allow the discovery of errors in larger systems using fewer resources and produce a better reduction in systems that are correct.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software Verification

Keywords

Abstraction-refinement, concurrency, underapproximation

General Terms

Reliability, Verification

1. INTRODUCTION

We expect to see an increase in the number of application programmers writing concurrent applications to take advantage of the multi-core processors. Access to efficient verification tools is an important aspect in making these systems reliable.

In this work, underapproximation of a system is explored by limiting the number of threads and preemption points using information generated from program slicing with respect to a reachability property. The underapproximations are refined incrementally by adding behaviors (interleavings) at each iteration. The goal is use the information from program slicing and concrete execution incrementally to create a reduction in the number of interleavings explored before finding errors or proving the correctness of the program.

Explicit model checking is a precise verification technique that exhaustively explores all possible behaviors of the program to check for errors or demonstrate a proof of correctness [9, 4]. The number of possible behaviors in the system often increases exponentially with the number of threads and program locations.

Partial order reduction techniques have been applied to combat the explosion in the number of behaviors [1, 2]. Partial order reduction only considers interleavings at operations on global variables that are relevant in verifying a particular property. These techniques are not sufficient to make model checking tractable. Partial order reduction techniques do not incrementally refine their dependence relation based on an underapproximation of a concrete execution as proposed in this work. We believe that such an incremental approach produces a more aggressive reduction than what is seen in other partial order reduction techniques.

A closely related work incrementally explores the underapproximations of a concurrent system using bounded model checking [3]. The technique uses a SAT solver to determine the satifisability of the property. The information from the SAT solution is used to refine the system if needed. The technique presented in this work does not rely on SAT and is designed for verifying object-oriented languages such as Java. Also we use dependence analyses and program slicing to refine the abstraction. We also believe the proposed technique in this work has the potential to scale to larger systems and can be applied to a varied domain of programs.

In our previous work, we generated an abstract system from standard static analysis techniques such as control dependence, data dependence, and backward slicing to generate program locations relevant in verifying a property [8]. A class of heuristics were developed to guide the program ex-

```
Global vars initial values : $x := 0,\ y := 1$

Thread $t_0$                          Thread $t_1$

1: if $x == 0$ then                   1: if $y \geq x$ then
2:     if $y \leq 1$ then             2:     $y := y + 1$
3:         $x := x + 12$              3: $x++$
4: else
5:     $y := y * 30$                  Thread $t_2$
6:     $x := x + 9$
7: if $x == 10$ then                  1: $y := y - 1$
8:     throw exception                2: if $y == 9$ then
9: return                             3:     $y := y * 10$
```

Figure 1: A program where threads t_0, t_1, and t_2 operate on shared global variables x and y.

ecution along a single sequence of program locations in the abstract system in the hope of finding an error. Additional program locations are added to the trace when the concrete execution cannot make progress. The main limitation of the work is that it cannot detect the absence of an error and it is restricted to following a single sequence of locations and other sequences have to be explored iteratively or in parallel.

Various other underapproximation exploration techniques have been researched using must-may abstractions [7]. While in the testing domain, the dynamic analysis tool Chess explores the underapproximation of the system by restricting the number of preemptions along a particular path [6]. It is limited because a lot of time is spent in exploring all paths and interleavings on variables that may not be related in determining the feasibility of a particular error.

2. INCREMENTAL APPROACH

Initially, an abstract system is constructed to contain program locations from a single thread such that the locations are relevant in verifying the reachability property. On the first iteration of the refinement loop, the lone thread in the abstract system is executed in the actual concrete system to create an underapproximation of the actual behavior space. If the target is reached the analysis stops. When the target, however, cannot be reached the information about non-executed locations in the abstract system is used in the refinement process. Refinement incrementally adds additional threads, one at a time in each iteration, along with relevant locations in the threads to the abstract system. The new abstraction is again coupled with the concrete execution of the real system to restrict the real system to only execute threads in the abstraction and only consider preemption at locations in the abstraction. The new underapproximation of the actual behavior space contains additional interleavings caused by the additional thread and preemption points in the abstraction. The process is repeated until the target can be reached or is shown to be unreachable.

The input to our technique is a concurrent program and a reachability property. The threads in the concurrent program communicate with each through shared variables and use synchronization primitives to allow exclusive access to threads while performing operations on shared variables. The programs do not have any data input. The non-determinism in the program behavior, hence, arises from different thread execution order and not from data input. The other input is a reachability property consisting of target location. In order to simplify the presentation we describe the technique for a single target location, but the technique extends to multiple target locations, for instance, lock acquisition locations in a deadlock or pair of unprotected accesses in a race-condition. An example of a concurrent program meeting our input requirements is shown in Fig. 1. It has three unique thread entities: t_0, t_1, and t_2 that operate on two shared global variables x and y. The property being checked in this case is whether the exception on line 8 of thread t_0 can be raised.

2.1 Abstract System

An abstract system is created for the thread that contains the target location. The abstract system contains the program locations relevant in verifying the reachability of the target location. Interprocedural backward slicing is used to generate the set of relevant program locations [5]. Backward slicing uses a combination of control and data dependence analyses to generate a set of program locations relevant to the reachability of the target. The relevant synchronization operations on the global variables of interest are also added to the abstract system.

The abstract system for thread t_0, Fig. 1, is shown in Fig. 2(a). The reachability of the target location on line 8 is control dependent on the predicate, $x == 10$, at line 7 evaluating to true. There are two definitions of the global variable x in the system and their reachability is also determined by other conditional branches. In Fig. 2(a), the locations represent a backward slice and are relevant in verifying the reachability of the target location. This is the initial abstraction. It is used to restrict the concrete execution of the real system to the single thread in the abstraction. For the example in Fig. 1, the single thread is t_0.

2.2 Explore initial underapproximation

A runtime environment implements an interleaving semantics over the restricted threads in the program. The runtime environment operates on a program state, s, that contains (1) the values of the global shared variables, (2) the local variable values and stack contents for each thread, and (3) information about the locks held by each thread. An underapproximation of the actual system is explored by limiting the locations where preemptions occurs and also restricting the threads that are allowed to execute in the system.

The initial underapproximation of the concrete system explored for thread t_0 is shown in Fig. 2(b). The initial abstract system in Fig. 2(a) only contains a single thread t_0, hence, in the restricted concrete execution of the total system, there is no thread non-determinism; only thread t_0 runs. At the *init* state the values of x and y are zero and one respectively. The initial values allow the assignment, $x := x + 12$, to be executed. The value of x is not equal to ten and concrete execution of t_0 in the actual system ends without reaching the target.

When the target location cannot be reached there are two possibilities, (1) the underapproximation explored does not contain the required behaviors to find the target location and needs to be refined, or (2) the target location is not reachable. We discuss the conditions for both cases.

2.3 Refinement

In the refinement phase, an additional thread that *may* affect the reachability of the target location is added to the abstract system. The locations in the abstract system that

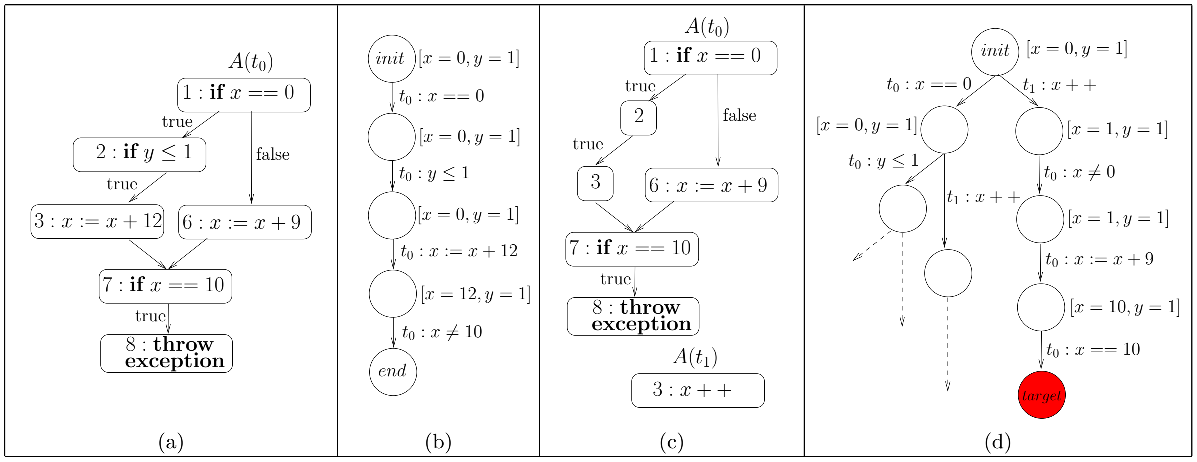

Figure 2: Incremental underapproximations of the example in Fig. 1 (a) Initial abstract system (b) First underapproximation exploration (c) Refined abstract state (d) Second underapproximation exploration.

do not get executed in the underapproximation of the concrete system are used to determine the thread to be added to the abstract system.

In the underapproximation from the concrete execution of t_0 shown in Fig. 2(b), lines 6 and 8 do not get executed. The reachability of line 6 is control dependent on the predicate at line 1, $x == 0$ evaluating to false. Similarly the reachability of the target location at line 8 is control dependent on $x == 10$ evaluating to true. The predicates in the conditional branches of interest are over the global variable x. In an attempt to reach the target location, another thread that modifies the global variable x is added to the abstract system. When there are multiple global variables hindering the reachability of locations in the abstract system, then all the variables are considered when adding a new thread.

Thread t_1 in Fig. 1 modifies the global variable x on line 3. An abstraction from a program slice on thread t_1 is added to the abstract system as shown in Fig. 2(c). The abstract t_1 only contains locations in the thread that affect the reachability of the program location at line 3 as given by the slice. These locations define the preemption points considered in the concrete execution of the actual system; furthermore, only threads t_0 and t_1 are allowed to execute in the concrete system. Control and data flow analyses are used to identify other program locations that affect the reachability of the assignments to the global variables of interest. In the example in Fig. 2(c), however, no other locations need to be added. In the case there are multiple assignments to the global variables of interest in the same thread then all the assignments and their dependencies are added to the refined abstract system. Also, in the case that more than one thread contains assignments to global variables of interest, we incrementally add one thread to the abstract system in each refinement cycle.

It is important to note that even though the global variable y is part of the original system, we do not need to add a thread that modifies its values. Line 3 in thread t_0 is control dependent on $y \leq 1$ and the initial value of y is sufficient to execute the program location at line 3. Since there exists a value of y that allows line 3 in thread t_0 to be executed in the first iteration we are guaranteed that line 3 will be executed in all subsequent iterations.

Part of the underapproximation of the concrete system with additional interleavings after the refinement process is shown in Fig. 2(d). The abstract system contains two threads t_0 and t_1. In the underapproximation from the abstraction restricted concrete execution, the interleavings between thread t_0 and t_1 are explored preempting at locations that are in the abstract system. At the *init* state in Fig. 2(d) when thread t_1 is executed, the value of x is set to one. Next, when thread t_0 is executed, the predicate $x == 0$ evaluates to false. After executing the instruction, $x := x + 9$, the predicate $x == 10$ evaluates to true and the target location is reached.

2.4 Unreachable Target

After exploring an underapproximation based on an abstract system when the refinement process does not add any new threads to the system, then the target location is termed unreachable. Suppose, for example, at line 7 in thread t_0, the predicate controlling the reachability of the target location is **if** $(x == 45)$ and the rest of the program is unchanged. Even after refinement, the target location cannot be reached. The global variable affecting the reachability of the target still remains x. All the interleavings in the program relevant to x have already been explored, hence, the target is unreachable.

This is an improvement over other partial order reduction techniques because the partial order reduction techniques construct the dependence relation over visible behaviors only once. For the example in Fig. 1, all the operations on x and y in threads t_0, t_1 and t_2 are considered as preemption points in the partial order reduction. Intuitively, the operations on x and y are visible for verifying the reachability of the target location. The incremental refinement in our technique produces a better partial order reduction based on the property being explored than a more typical dynamic reduction.

3. RESULTS

We present some preliminary results from an initial implementation. The technique is implemented in the Java Pathfinder (JPF) model checker, [9], that uses a modified

SharedVar-v0, no error		
Search	States	Time (sec)
DFS	18199	7.0
Under(Iter-1)	2	0.1
Under(Iter-2)	385	0.4
Total(Iter)	387	0.5

(a)

SharedVar-v1, 1-thread error		
Search	States	Time (sec)
DFS	18	0.1
Under(Iter-1)	2	0.1
Under(Iter-2)	-	-
Total(Iter)	2	0.1

(b)

SharedVar-v2, 2-thread error		
Search	States	Time (sec)
DFS	5880	3.0
Under(Iter-1)	2	0.1
Under(Iter-2)	355	0.2
Total(Iter)	357	0.3

(c)

Table 1: Comparing depth-first with exploring underapproximations results for: (a) model containing no error (b) model with error found with one thread (c) model with error needs interactions between 2 threads

JVM to execute Java bytecode. It is implemented as an extension to JPF and can be obtained along with the examples described here from a mercurial repository:
http://babelfish.arc.nasa.gov/hg/jpf/jpf-guided-test

We compare three different versions of a model named `SharedVar`: v0, v1, and v2. The `SharedVar` example has three threads and operates on two shared variables. Among the variants of the program, v0 does not contain error, v1 requires simply the execution of one thread to raise an exception (error) in the program, and v2 requires interactions between two threads to raise an exception (error) in the program. The results for v0, v1 and v2 are shown in Table 1(a), Table 1(b), and Table 1(c) respectively.

The results in Table 1 compare a dynamic partial order reduced depth-first search implemented in JPF with our incrementally refined underapproximations of the concrete system. The Under(Iter-1) indicates that the values in the first iteration of exploring the underapproximation, while Iter-2 represents the second iteration. The total time reported is in seconds. When reporting the times for exploring the underapproximation the Total time includes generating the abstraction and refining the abstract system. States indicate the total number of states explored before reaching the target or states explored before showing it is unreachable.

In general, in all three versions of the example we see that in exploring the underapproximation of the system even with a refinement process, we explore fewer total states and take less total time in verifying the system. The trend is seen for versions that contain the error and the version that does not contain the error. In version v0, a depth-first search with dynamic partial order reduction takes seven seconds and explores 18,199 states before completing while even after refining once, searching the underapproximation takes only 0.5 seconds to show the absence of the error and generates 387 states in the process. This dramatic reduction we believe arises from the criteria used to detect when the refinement has a reached a fixpoint and adding further behaviors to the system will not change the output of the program.

4. IMPACT AND FUTURE WORK

An initial implementation and testing of the incremental underapproximation shows that (i) target locations are often reached with only a few refinements; and (ii) the total number of states necessary to show a target cannot be reached is dramatically less that what is obtained from a dynamic partial order reduction on the same program. These two observations suggest that it might be possible to create an abstraction refinement loop based on underapproximation that (i) can lead to error discovery in large systems where formal verification is normally intractable; and (ii) the incremental refinement produces a better partial order reduction based on the property being explored that a more typical dynamic reduction that could potentially allow us show the absence of errors in larger systems. Such contributions have the potential to move forward the current state-of-the-art in concurrent program verification based on exhaustive search techniques. Immediate future work will focus on extensive evaluation with other abstraction techniques and constructing formal proofs for the claims of refinement reaching a fixpoint.

5. REFERENCES

[1] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. POPL*, pages 110–121, New York, NY, USA, 2005. ACM.

[2] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. Foreword By-Pierre Wolper.

[3] O. Grumberg, F. Lerda, O. Strichman, and M. Theobald. Proof-guided underapproximation-widening for multi-process systems. *SIGPLAN Not.*, 40(1):122–131, 2005.

[4] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, 2003.

[5] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Not.*, 39(4):229–243, 2004.

[6] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *SIGPLAN Not.*, 42(6):446–455, 2007.

[7] C. Pasareanu, R. Pelánek, and W. Visser. Predicate abstraction with under-approximation refinement. *Logical Methods in Computer Science*, 3(1), 2007.

[8] N. Rungta, E. G. Mercer, and W. Visser. Efficient testing of concurrent programs with abstraction-guided symbolic execution. In *Proc. SPIN Workshop*, pages 174–191, Grenoble, France, June 2009. Springer–Verlag.

[9] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. ASE*, Grenoble, France, September 2000.