Thesis Proposal:
Slice and Dice Algorithm Implementation in JPF

by

Eric S. Noonan

A thesis proposal, submitted to the faculty of Brigham Young University in partial fulfillment of the requirements for the degree of Master of Science.

Department of Computer Science

Brigham Young University

March 6, 2013

Abstract

The primary methods of finding errors in a parallel program are search prioritization strategies and partial order reduction. Neha Rungta and Eric Mercer developed a new search prioritization strategy called slice and dice. The proof of concept slice and dice algorithm used at least one ninth the number of states as JPF's native partial order reduction on any given test it was run on. This proposal outlines implementation details of the complete slice and dice algorithm and how to measure the algorithm's performance against other error finding methods on a few specific benchmarks in JPF.

1 Introduction

There are many parallel programs being developed or that have been developed that operate on critical systems whose failure could lead to disastrous real-life consequences. Therefore it is necessary to ensure these systems have the correct behavior before using them. A program can be checked for correct behavior using a model checker. A model checker takes a program as input and checks for the reachability of error states in the program. JPF is a widely used model checker for Java programs. JPF is used by NASA to verify their critical systems. Model checkers like JPF have to deal with a problem called the state explosion problem.

The state explosion problem refers to the fact that exhaustively generating all possible states due to non-determinism in the scheduling or non-determinism in the input of the program yields an exponential number of states to explore. This number of states can be so large that it is not practical in terms of execution time or required memory to do an exhaustive exploration of all possible states in order to discover error states. In order to solve the state explosion problem, there are numerous methods developed that could be used. One method of solving the state explosion problem is partial order reduction.

Partial order reductions (PORs) can use the notion of dependence in order to determine redundant execution paths that generate the same global state with respect to properties that are being checked [10]. A transition is an operation that changes the global program state. Two transitions are dependent if they enable or disable each other or yield a different global result when executed in a different order. In contrast, two transitions are independent if executing them in either order yields the same global program state. The basic idea behind a POR is to execute all the dependent transitions from a given state to generate a set of new states [21, 11, 12]. This exploration of all interleavings of dependent transitions ensures PORs are sound which means if there is an error, it will be found by a POR. The interleaving of dependent transitions is done recursively until the program terminates or hits a given search depth. As the POR is performed, details about the program state are available. These program states can be checked against constraints. If the model checker finds something that violates the specified constraints while performing a POR, then the program is provably wrong and needs to be changed.

PORs reduce the number of explored states by only exploring one schedule of execution between independent operations. A POR still explores the whole state space of the system, which is still an exponential state space based on the number of dependent transitions. Exploring the entire unique state space of a program is a costly endeavor in terms of execution time and memory requirements even with a POR. This means model-checking using POR is impractical for larger parallel programs. For this reason, under-approximation methods have been developed [16].

There are many different kinds of under-approximation methods that have been published [16, 23, 9, 22]. Under-approximation methods leave a piece of the state space unexplored in which an error could reside. This is a serious problem if a program is model-checked by an under-approximation method still has an undetected critical error. This undetected error may never be detectable if the program is too large to be model checked with a POR or other reduction method that explores the entire unique state space.

The slice and dice algorithm was designed to address this issue of finding errors too large to be found by a POR [24]. The assumption behind the slice and dice algorithm is this: If there is a given line in the program where an exception is thrown in response to a violated property, then it is possible to determine if a property is violated by detecting reachability to the target location that represents the violated property. slice and dice works by executing the thread that contains the target location in isolation. This thread is called the initial program slice. If the target location is not reachable by executing the initial slice, then another thread is added to the initial program slice that

could modify data that affects control flow to the target location in the thread where the property violation exists. This process of adding threads that could potentially help the program reach the error state is called refinement. After refinement, the slice is executed again multiple times, attempting all interleavings of the threads in the slice in order to detect the error. If the error is not found, then another refinement and round of execution is done. This process of slice construction and execution is repeated until the error is found or no additional refinements can be made.

Slice and dice has the advantage over POR in that it does not need to explore the whole relevant state space, just the parts that could lead to a property violation. The advantage slicing and dicing presents over under-approximation methods is that it does not attempt to under-approximate the parts of the state space that could be relevant in reaching an error. The advantages of the slice and dice algorithm over both POR and under-approximation methods mean that slice and dice could solve the problem of finding errors in larger parallel programs that under-approximation methods miss.

The research plan described here is to implement the slice and dice algorithm to work on a general set of accepted benchmarks in JPF in order to establish a stronger proof of concept beyond the highly experimental prototype. If the slice-and-dice algorithm proves efficient in time and space in a broader set of accepted benchmark programs, then future work, which is not part of this research plan, is to generalize slice-and-dice to other programming languages, prove the algorithm is complete and possibly sound, and establish asymptotic bounds on its complexity in time and space.

2 Thesis Statement

The slice and dice algorithm, in the context of the JPF model checker, will explore fewer states on average before error discovery than other state of the art algorithms to mitigate state explosion in scheduling non-determinism including POR, heuristic guided search, and K-bounded delay, over a set of commonly accepted benchmark programs.

3 Related Work

The primary focus of this proposal is program verification. More specifically, program verification using model checking. Model checking is a way of exploring possible program states in search of an error. When model checking, two forms of non-determinism are explored in order to generate unique program states: non-determinism with respect to the input to the program or non-determinism with respect to the way the program can be scheduled. In both cases, there are many possible program states that can result from non-determinism. The explosion of states caused by either form of non-determinism is referred to as the state explosion problem. The work in this thesis proposal is concerned with combating state explosion in model checking. This section briefly covers important technologies treating state explosion in data non-determinism (under/over approximations, bounded model checking, proving non-termination, symbolic/concolic execution) to give context to the extent of the state explosion problem, and then looks more particularly at scheduling non-determinism (POR and under-approximated heuristic schedulers) upon which this thesis proposal builds.

3.1 Over/under Approximations

When using counter-example guided abstraction refinement (CEGAR), a model of the program is constructed using an abstraction that represents an over-approximation of the system's actual state space [5]. Once that model is constructed, the state space of the model is explored in search for errors. If no errors are found in the over-approximation, then the program is error-free. This method can detect erroneous errors because it starts out with an over-approximation of the actual state space [5]. To alleviate this, the model is refined whenever an error state is detected in order to determine whether or not the error was a legitimate error [5]. This method is employed in two different model checkers, BLAST and SLAM [3, 1]. This method helps solve the state explosion problem because

the abstraction contains fewer states than the actual program. This method only refines and expands the state space as needed when errors are found. As a response to the issue of detection of non-existent errors under-approximation methods were developed. One of these methods is predicate abstraction with under-approximation refinement [23].

In predicate abstraction with under-approximation refinement, transitions play a critical role. This under-approximation method uses actual transitions derived from the program to generate new program states and under-approximates the state's representation using predicates [23]. It only explores unique states during model checking [23]. It trims states by not exploring the children of states that have identical abstractions to states previously seen.

3.2 Bounded Model Checking

Property violations can be detected using constraint solving. The model checking method that detects property violations using constraint solving takes a program and unwinds the program's execution down to a certain user specified depth of looping or recursive calls. It translates that unwinding into a verification condition that can be checked using an SMT solver [7, 4]. If the result shows an error state is satisfiable given the unwinding, then an error is reported [7].

3.3 Non-termination

Proving non-termination of a program is another form of verification. Recent advances have shown that proving termination or non-termination for some classes of programs is possible and also solvable in a reasonable amount of time [6]. This is important because proving termination or non-termination of a program can represent an error in a program. For example, reactive systems need to not terminate, so a termination condition indicates an error.

Except for non-termination proving, the aforementioned methods focused on model checking methods that followed a pattern. The first part of that pattern was to us the program to create an abstraction. The next part of that pattern was to prove properties about that abstraction. The final part of that pattern was to relate the properties proven back to the original program. In contrast, there are methods of model checking that focus on using the original program itself as the model to be checked.

3.4 Symbolic/Concolic Execution

There are methods of model checking that use the program itself as the model in order to explore the state space due to data non-determinism. One of the methods for doing this is concolic testing. In concolic testing, the program is given input data for one run on the program. As the program is executed, the model checker keeps track of the path conditions that guide execution of the program. Once that is finished, the model checker uses the path conditions in order to generate new input data that will guide it along a different branch in its internal structure [27]. The point of concolic testing is to automate test case generation as well as give the code full branch coverage in search for errors [27]. Another method that operates independently of input data is generalized symbolic execution (GSE). In GSE the program's internal variables are represented as symbolic variables, then as each branch is explored, it generates classes of input that cause errors based on path conditions [20]. The last method exploits heap symmetry in order to identify equivalent states. If a heap has the same structure in two different states, then the states are considered equivalent and exploring from only one reduces the state space explored [18].

3.5 POR

Other methods of model checking exist that use the program itself as the model in order to explore the state space due to scheduling non-determinism. One of these methods uses a stateless approach. It was implemented in the model checker Verisoft [13]. The reduction in the state space searched was generated by showing commutativity across sequences of transitions. If a sequence of transitions was commutative, then taking transitions in a different order yielded the same state. The explored state space was reduced by not model checking the result of an alternate ordering of commutative transitions [13]. This approach was essentially performing a POR without the use of

explicit states. POR is a method of model checking the entire state space of a program. Monotonic PORs amplify the power of PORs by restricting the interleavings explored [19].

There are two methods of POR: static and dynamic. In static partial order reductions (SPOR), the program analyzed at compile time to determine dependencies between operations [21]. In dynamic partial order reductions (DPOR), the program determines dependence based on changes in program state as it is run [11]. There are methods to do this. One of these methods uses persistent and sleep sets [11]. A persistent set is a set of transitions such that all transitions outside the set are independent of the transitions inside of it [12]. The idea behind sleep sets is that they are sets of transitions that are independent with each other (exploring a schedule with an interleaving of two transitions in a sleep set yields the same global state). Once one interleaving of the independent transitions has been explored, it is not explored again in the opposite order [12]. The idea behind a DPOR using the sleep set and persistent set technique is to schedule every transition in a persistent set for each state and execute those schedules without repeating any interleavings in a sleep set more than once. If this sort of exploration is done, then it is guaranteed that all possible global results for a given program have been explored [11]. Another method is the stubborn set technique. This method finds independence between transitions by showing commutativity over a sequence of actions [11].

In both SPOR and DPR instances the model checker keeps track of which schedules of the program it has already run and which ones it needs to try in the future. The model checker has the means to control execution of the program in a way that allows it to explore all relevant schedulings. The primary advantage of a DPOR is that it has more information about the program available to it during execution. The primary disadvantage of the DPOR is that it costs even more memory to actually keep track of all the extra information in the executed program in order to perform the reduction. The time and memory requirements on PORs have necessitated development of other methods to find errors due to scheduling non-determinism. As a result, methods of under-approximation have been developed.

3.6 Under-approximated Heuristic Schedulers

One method of under-approximation is K-bounded methods. K-bounded methods work by putting a bound on the state space explored. Another K-bounded method in addition to K-bounded delay scheduling discussed previously is iterative context deepening. In this K-bounded method, K preemptive context switches are added to the threads and then executed in hopes of detecting an error [22]. All combinations of K preemptive context switches within the threads are tried. All possible combinations of the next thread after the context switch are also tried. This ensures that all errors found within K context switches are guaranteed to be found. Another K-bounded method of under-approximating a system's state space to yield faster results for detecting errors is K-bounded delay scheduling. K-bounded delay scheduling takes a deterministic scheduler operating on threads [9]. In this case, deviations from the normal scheduler are taken and K is the bound on the number of times any given execution of the program can deviate from the deterministic scheduler [9]. The main problem with K-bounded methods is that no guarantee is given that all errors will be found within the given K. The heuristic guided search is a better search prioritization method when catching errors that occur with a high number of context switches than K-bounded methods [25].

Heuristic guided search uses heuristics to guide search of the state space when finding an error. States with a more favorable heuristic are chosen first in hopes of finding an error [8]. A simple heuristic is one that uses a boolean function, system state and valid transitions within the model. Using this information, two parts of the heuristic are computed. The first part is the number of transitions the model checker would have to take in order to violate the boolean function given the current state [8]. The second part of the heuristic is the number of transitions the model checker would have to take out of the current state in order to satisfy the boolean function. The first part of the heuristic is used in order to check invariants on program states that represent properties that should always hold during runtime [8]. If the model checker can use the first part of the heuristic to violate the property, then it knows an error state has been detected. The second part of the heuristic is used as a heuristic to guide the search towards assertion violations. If the model checker can guide the search in a way to violate an assertion, then an error state has been detected [8]. Additional heuristics can be used to improve the search when using the heuristics based on

boolean functions.

These heuristics are based on ideas about the structure of the state space being explored [14]. One of these heuristics is based on attempting to achieve full code coverage [14]. This heuristic gives states that are more likely to cover new branches a higher heuristic value [14]. This helps guide the search to cover states it has not seen before, allowing errors in new branches of code to be detected [14]. Another structural heuristic deals with thread interleaving. This heuristic gives a higher score to states that cause context switches where there is no explicit yield. This helps discover errors due to scheduling non-determinism quickly [14]. An additional structural heuristic helps the model checker ignore state space that is not valid in the program [15]. Many heuristic-guided searches deal with abstractions that can be over-approximations of the actual state space of the program. These over-approximations introduce non-determinism that is actually not present in the native program. Therefore, structural heuristics have been developed [15]. Structural heuristics are used to give favor to states that are choose-free. A state is choose-free if it is not generated due to non-determinism in the abstraction [15]. Heuristic-guided searches have been improved by allowing the user specify error locations in addition to the normal static analysis to detect error locations. Such a method method uses the heuristics discussed to try to force the model checker to any error location chosen by either the user or static analysis [26]. Heuristic guided seartch is faster than POR because it does not search the entire global state space like a POR, instead it tries to control the search in order to reach problem areas. Heuristic guided search is different from slice and dice in that it still considers the whole program when trying to guide the search to the error location, whereas slice and dice only considers the portion of the program that is in the current slice. Slice and dice also does not use heuristics to guide its search of the state space, it only interleaves relevant dependent operations when trying to reach the error state.

4 Project Description

4.1 High-Level Slice and Dice Description

The slice and dice algorithm is an algorithm that underapproximates the state space of the program by only focusing on the parts of the program that can be used to reach an error. The slice and dice algorithm takes two inputs: a program and a reachability property such as a specific program location that represents an error state being reached. This program must be a closed program that receives no input. The output of the slice and dice algorithm is whether or not the error state is reached. There are four processes in the slice and dice algorithm that are performed in the algorithm in the following order: refinement, slice construction, slice abstraction construction and slice execution. These four processes are repeated until the the search space of the algorithm can no longer be expanded.

In refinement, variables involved in control flow or data dependence are isolated with respect to the location representing the error. This process is called control flow and data dependence analysis. The variables isolated during this analysis are called variables of interest. A variable of interest is a variable that influences reachability to the target location, directly or indirectly. Variables involved in control flow are variables that affect reachability to the property. One class of variables involved in control flow are variables that directly guard the if statement that protects the reachability property or that guard statements to other variables involved in control flow. Other variables involved in control flow include variables in the loop invariant of loops that modify variables already isolated, but only if the variables in the loop invariant are modified by other threads. Another type of variable involved in control flow are locks. A variable is part of the data dependence if it affects the value of a variable involved in control flow or data dependence. Variables of interest are variables found during control flow or data dependence analysis. The variables found during control flow and data dependence analysis are transitively related. This transitive relation between variables means that control flow and data dependence analysis needs to be done over and over again until no new variables of interest are found. Once no new variables are found, refinement is finished and the slice abstraction is constructed.

In slice construction, threads are isolated that are useful for finding the given property and added to the slice. The slice starts out empty. On the first iteration of the algorithm, the first thread is added to the slice. That thread is the thread that contains the reachability property. After the first iteration of the algorithm, any thread that manipulates

variables of interest that is not already in the slice is added to the slice. If there are multiple threads that could be added to the slice, one is chosen at random. If there are no more threads that can be added, then the algorithm terminates. In order to aid slice construction, methods of correctly adding threads and exploring the under-approximation in the presence of procedure calls and arbitrary branching exist [2, 17].

In slice abstraction construction, scheduling relevant locations in the program are isolated. Locations that involve operations on variables of interest are known as preemption points. A preemption point is a point where a scheduling decision has to be made. Preemption points are used to determine places where threads in the slice should be interleaved. This ensures relevant interleavings of the threads in the slice are explored.

In slice execution, all interleavings of the threads in the slice on all preemption points are tried. If the error state is reached, then an error is reported and the algorithm terminates.

4.2 Slice and Dice JPF implementation details

JPF simulates the Java Virtual Machine and explores the state space of a program. The JPF framework allows users to write components to control the state space exploration. Some of these components are choice generators, search strategies and schedulers.

Choice generators enumerate different pieces of non-determinism of interest. For example, a choice generator that enumerates non-determinism in the scheduling exists. This choice generator keeps track of all possible threads that can be scheduled out of a given state. A modified version of this choice generator will be used at preemption points in order to save all possible threads in the slice that can be executed from the current state.

Schedulers are a custom component that can be used to determine how choice generators for an input program are built. For the purposes of this project a custom scheduler will be written. This scheduler will keep track of threads are in the slice and where the preemption points are. The scheduler will work by creating a choice generator for all the threads in the slice. This will ensure it will try all combinations of start threads in the slice. If an instruction is executed that includes a preemption point boundary, then the scheduler will mark that preemption point as executed in the abstract slice representation and create a choice generator that contains all threads in the current slice that have not completed execution. If an instruction is executed that is not a preemption point, then the scheduler will create a choice generator with only the current thread being executed. This sort of choice generator construction will ensure that all interleavings of threads at preemption points are exhaustively explored and this method of thread interleaving will also combat state explosion by not interleaving on operations that do not affect the reachability property. Once all combinations in the slice have been tried that could affect the reachability, the program must finish execution in order for JPF to exit normally. Therefore the rest of the program outside the slice is executed once.

int x = 0, y = 0, z = 0			
	t_0	t_1	t_2
0:	`x++`	`int h = 5`	`int i`
1:	`if(x > 25){`	`y = h + 1`	`for(i=0; i < 5; i++){`
2:	`throw exception`	`x += y`	`z += x`
3:	`}`	`y *= 2`	`}`
4:		`int m = x * y`	
5:		`if(m >= 84){`	
6:		`x *= 4`	
7:		`}`	

Table 1: Example program

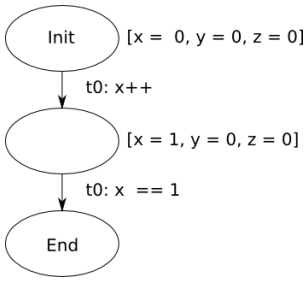

Figure 1: First slice execution

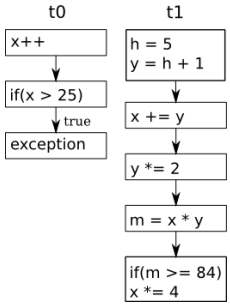

Figure 2: Two threads to interleave

4.3 Example Slice and Dice Execution

The algorithm will be applied to the program in Table 1 in order to illustrate how it works. The program contains three threads, t_0, t_1 and t_2. Thread t_0 contains the reachability property on line 2 and a modification to the global variable on line 0 that affects reachability to the property because of the conditional statement on line 1. Thread t_1 contains interactions of local and global variables that work together to affect the variable that can cause reachability of the target location. Thread t_2 contains a modification to a global variable that will not affect reachability of the target location. First, line 2 in t_0 is chosen as the target location to reach.

Because thread t_0 contains the property, it is chosen as the initial slice during slice construction. Next, variables of interest are found during refinement. In this example, reaching the target location is control flow dependent on the variable x on line 1 in t_0. This analysis yields x as a variable of interest. The algorithm performs another pass of control flow and data dependence to ensure no new variables are added. Next, an abstraction of the thread in the initial slice is constructed using the variables of interest and the target location to generate preemption points. All control flow instructions involving global variables of interest and writes to global variables of interest are instructions added to the abstraction as preemption points. The thread executes as demonstrated in Fig. 1. The target is not reached, so the algorithm continues.

Once the algorithm has finished one round of execution, it needs to choose another thread that potentially affects reachability to the target location in line 2 of t_0. Based on our chosen variables of interest, none of the operations in t_2 affect reachability of line 2 in t_0. In contrast thread t_1 is chosen because line 6 in t_1 could affect the outcome of the condition in line 1 of t_0 (control flow dependence). The algorithm performs a phase of refinement to add new variables of interest in the new thread. On the first iteration of control flow and data dependence analysis, the variable m in t_1 is added. On the next iteration of control flow and data dependence, y is added as a variable of interest, and then the last iteration adds h. The abstraction representing preemption points for the scheduler to keep track of is constructed in the same manner it was for the initial slice. The result of the abstraction construction is given in Fig. 2.

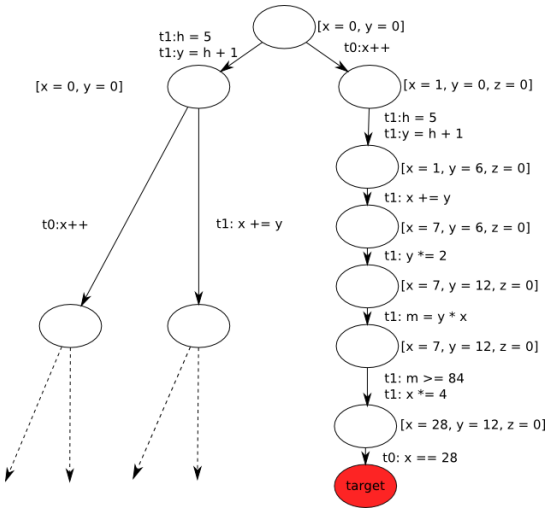

Figure 3: Error state reached, potential search space

During next step of the algorithm, slice execution, the algorithm works by interleaving all possible executions of each thread with each other on preemption points, respecting program order. At each preemption point, there are up to two possible choices of operations to execute, one from each thread. Once one thread is complete, then the choice of which operation to choose becomes deterministic. Fig. 3 shows an interleaving of t0 and t1 that demonstrates reachability of the target program location. The right half of the graph shows the target being reached. The left half shows some examples of other transitions the algorithm could have chosen. Fig. 3 shows the state space being searched in a Depth First Search (DFS) fashion, with each node on the tree representing a possible decision point using a choice generator as in Fig 3.

The strength of slice and dice is illustrated using the following example. What if the target location was still found to be unreachable using the current threads in the slice, t_0 and t_1? The slice construction phase of the algorithm would reveal that t_2 does not affect the value of any of the variables of interest (x, y, m or h) that affect program branching to the target location. Therefore, the algorithm would detect it was done and would exit letting the user know no error was found.

5 Validation

In order to test the effectiveness of slice and dice, it will be compared against other methods of discovering errors in a program. The tools that slice and dice will be compared to are a heuristic guided search, A K-bounded method and JPF's native POR. The average time to find an error state, the minimum, maximum and average states explored to find an error state, and the amount of memory used will all be recorded. JPF automatically outputs this information from a given run. It is hoped that the chosen metrics will demonstrate slice and dice is effective in terms of runtime and that it is also efficient in terms of memory and state space explored when compared to other algorithms.

5.1 Experimental Set-Up

In a paper by Neha and Mercer a few benchmarks for JPF are introduced with the ability for the user to make the errors in them more difficult to find [25]. From that paper, the following benchmarks will be used: Accountsubtype, Reorder and Airplane. Five experiments on each benchmark will be used. Each successive experiment will be changed to make the errors more difficult to find. JPF's native POR, Slice and Dice, the K-bounded delay scheduler (using the DFS scheduler written by Emmi et al. [9]) and a heuristic guided search algorithm will each be run twenty

times on each experiment. The run time of the twenty runs in each experiment will be averaged in order to report the run time of the algorithm the experiment was performed on. The experiments involving slice and dice and heuristic guided search will use the same error location for a given run and if there are multiple errors in the program, then another error will be selected for each successive run of both algorithms. The experiments involving the K-bounded scheduler will use a reasonable guess for the K-bound based how many context switches the other algorithms needed in order to reach the error.

6 Thesis Schedule

The following schedule is an outline of submission dates for my Master's Thesis:
Submission to Advisor: February 8, 2014
Submission to Committee Members: February 22, 2014
Master's Thesis Defense: April 4, 2014

References

[1] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *IFM*, volume 2999 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2004.

This paper discusses the model checker BLAST. BLAST uses an over-apporximation of a system to try to guarantee the system is error-free. If an error is detected, it uses path reachability algorithms to determine if the error was a real error. This method is called Counterexample guided abstraction refinement (CEGAR).

[2] Thomas Ball and Susan Horwitz. Slicing Programs with Arbitrary Control-Flow. In Peter Fritszon, editor, *AADEBUG*, volume 749 of *Lecture Notes in Computer Science*, pages 206–222. Springer, 1993.

This paper discusses the issues that can occur when using program slicing methods with arbitrary control flow. The information in this paper is relevant for writing the slice and dice algorithm so that no problems occur with arbitrary branching.

[3] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The Software Model Checker blast. *STTT*, 9(5-6):505–525, 2007.

This paper discusses the model checker BLAST. BLAST uses an over-apporximation of a system to try to guarantee the system is error-free. If an error is detected, it uses path reachability algorithms to determine if the error was a real error. This method is called Counterexample guided abstraction refinement (CEGAR).

[4] Edmund M. Clarke, Daniel Kroening, and Karen Yorav. Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking. In *DAC*, pages 368–371. ACM, 2003.

This paper describes a method for verifying a program called bounding. In this technique, a system and its specification are unwound into a boolean formula and then checked using a SAT procedure.

[5] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. *Journal of the ACM*, 50(5):752–794, September 2003.

This paper discusses the use of abstractions to perform model checking. It also describes a method for refining abstractions when they fail using counterexamples that caused the previous abstraction to fail.

[6] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving Program Termination. *Commun. ACM*, 54(5):88–98, 2011.

This work deals with proving program termination. In the past, program termination was decided as an NP-hard problem and therefore not worth trying to solve. This work proposes allowing a program to output an additional classification to solving the problem: an "unknown" one. With the usage of three possible outputs of the redefined problem, many different programs can be proven to terminate in a reasonable amount of analysis time. This is relevant to the work in this paper because it is a form of model checking, but it is concerned with solving a different kind of problem than the problems solved by the algorithm in this paper.

[7] Lucas Cordeiro, Bernd Fischer 0002, and Joo P. Marques Silva. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. *CoRR*, abs/0907.2072, 2009.

This work deals with SMT-based bounded model checking. This method involves translating a program into a transition system and unrolling the system up to a bound of k times. After that unrolling is done, the program is translated into a formula that can be used to verify a system property. This paper deals with performing this sort of verification using C programs.

[8] Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue. Directed Explicit Model Checking with HSF-SPIN. In Matthew B. Dwyer, editor, *SPIN*, volume 2057 of *Lecture Notes in Computer Science*, pages 57–79. Springer, 2001.

This work deals with deriving search heuristics using boolean functions in order to search the state space more effectively when searching for errors. This work is directly related to the guided search method of finding errors in a program. The guided search algorithm will be tested against the algorithm proposed in this paper.

[9] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. Delay-Bounded Scheduling. In Thomas Ball and Mooly Sagiv, editors, *POPL*, pages 411–422. ACM, 2011.

This paper outlines the theory behind K-bounded delay scheduling. The work in this paper is the basis used for writing the K-bounded delay scheduler that will be used for benchmark testing against slice and dice.

[10] Javier Esparza and Keijo Heljanko. Implementing LTL Model Checking with Net Unfoldings. In Matthew B. Dwyer, editor, *SPIN*, volume 2057 of *Lecture Notes in Computer Science*, pages 37–56. Springer, 2001.

This paper outlines the discovery of errors in a parallel program using net-unfoldings. Net-unfoldings are a form of partial order reduction.

[11] Cormac Flanagan and Patrice Godefroid. Dynamic Partial-order Reduction for Model Checking Software. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 110–121, New York, NY, USA, 2005. ACM.

This paper is the basic theory behind dynamic partial order reduction. Partial order reduction is another method of discovering bugs in a program and can be used as another means of comparison against slice and dice.

[12] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

This book outlines methods of partial order reduction. It is one of the founding works in the field of partial order reduction. This is relevant to my work because partial order reduction serves as a good comparison to under-approximation methods such as the one outlined in this proposal.

[13] Patrice Godefroid. Model Checking for Programming Languages Using Verisoft. In *POPL*, pages 174–186, 1997.

This paper discusses various methods suitable for executing a dynamic verification of a system using Verisoft. It proposes using a bound on the depth when a cycle is reached in a chain of passed messages. It also discusses the use of partial order reduction when dynamically verifying a concurrent system. Their results yielded a found bug in a program.

[14] Alex Groce and Willem Visser. Heuristic Model Checking for Java Programs. In Dragan Bosnacki and Stefan Leue, editors, *SPIN*, volume 2318 of *Lecture Notes in Computer Science*, pages 242–245. Springer, 2002.

This paper deals with the usage of structural heuristics in order to control state space exploration when searching for errors in a program. Structural heuristics can be used in addition to other heuristics in order to find error states faster. This work directly relates to the guided search error detection method.

[15] Alex Groce and Willem Visser. Model Checking Java Programs Using Structural Heuristics. In *ISSTA*, pages 12–21, 2002.

This paper focuses on finding errors by guiding path execution using search heuristics. The difference in this approach is that the whole system is still considered at once, whereas in slice and dice you start out with a subset of the concurrent system. These sort of heuristics could be applied to the way in which interleaved paths in slice and dice are explored in future work.

[16] Orna Grumberg, Flavio Lerda, Ofer Strichman, and Michael Theobald. Proof-Guided Underapproximation-Widening for Multi-Process Systems. In Jens Palsberg and Martn Abadi, editors, *POPL*, pages 122–131. ACM, 2005.

This paper shows how finding an error in an underapproximation can be extended to finding an error in the actual system. This paper is important because the algorithm in this work relies on underapproximation methods.

[17] Susan Horwitz, Thomas W. Reps, and David Binkley. Interprocedural Slicing Using Dependence Graphs. In Richard L. Wexelblat, editor, *PLDI*, pages 35–46. ACM, 1988.

This program shows how to abstract slices from programs that include procedures calling each other, as opposed to abstracting slices from huge, monolithic threads. This is important work because knowledge of how to do this will be very important for implementing the slice and dice algorithm.

[18] Radu Iosif. Exploiting Heap Symmetries in Explicit-State Model Checking of Software. In *ASE*, pages 254–261. IEEE Computer Society, 2001.

This paper deals with a different kind of state space symmetry. This paper describes how to identify symmetric heap structures. This is important because in modern program languages many aliases can point to the same structure, determining that an alias is merely pointing to a different place in the heap. If the heap is the same in multiple states then a reduction in the state space of the system is possible because that means that there was no actual change in the underlying program state.

[19] Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 398–413. Springer, 2009.

This discusses an extension to partial order reduction methods. The idea is to find quasi-monotonic sequences and not repeat them when verifying software. This reduces the state space explored when perfroming a partial order reduction.

[20] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized Symbolic Execution for Model Checking and Testing. In Hubert Garavel and John Hatcliff, editors, *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer, 2003.

This work deals with how to represent program inputs as symbolic values and then represent outputs in terms of those values. This method is called generalized symbolic execution (GSE). When using GSE it is possible to show classes of input values that cause errors to be thrown or undesirable program behaviour. It can also detect good program behaviour if no undesirable states were reached. GSE can also be used for automatic test case generation. This specific paper deals with how to use this method when dealing with more complicated heap structures.

[21] Robert P. Kurshan, Vladimir Levin, Marius Minea, Doron Peled, and Hsn Yenign. Static Partial Order Reduction. In Bernhard Steffen, editor, *TACAS*, volume 1384 of *Lecture Notes in Computer Science*, pages 345–357. Springer, 1998.

This paper outlines how to perform a static partial order reduction. This is relevant to my work because static partial order reduction is another method of finding errors in a program.

[22] Madanlal Musuvathi and Shaz Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 446–455. ACM, 2007.

This paper discusses iterative context bounding. It is a K-bounded method related to K-bounded scheduling, the method the slice and dice algorithm will be compared against. This is why this work is important, it is one of the verification methods already in place for verifying parallel programs and can be used as a means of comparison in determining the effectiveness of slice and dice.

[23] Corina S. Pasareanu, Radek Pelnek, and Willem Visser. Predicate Abstraction with Under-approximation Refinement. *CoRR*, abs/cs/0701140, 2007.

This work deals with creating an abstraction of the system and exploring it using concrete transitions. This work is significant because it does not raise false errors like the over-approximation methods of model checking. The representation of the abstract system is used to explore the state space of the system. Redundant states are not explored.

[24] Neha Rungta and Eric Mercer. Slicing and Dicing Bugs in Concurrent Programs. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastin Uchitel, editors, *ICSE (2)*, pages 195–198. ACM, 2010.

This paper outlines the slice and dice algorithm and compares it to partial order reduction. This paper proves that the slice and dice algorithm already runs better than partial order methods. The algorithm in the paper was implemented, but did not run for more than a few example programs. The purpose of this project is to make this algorithm run on more examples.

[25] Neha Rungta and Eric G. Mercer. Hardness for Explicit State Software Model Checking Benchmarks. In *SEFM*, pages 247–256. IEEE Computer Society, 2007.

This paper is important because this is where the benchmarks that are going to be used for validation of the algorithm are described. Benchmarks with target error locations are benchmarks we plan on using to test the algorithm.

[26] Neha Rungta, Eric G. Mercer, and Willem Visser. Efficient Testing of Concurrent Programs with Abstraction-Guided Symbolic Execution. In Corina S. Pasareanu, editor, *SPIN*, volume 5578 of *Lecture Notes in Computer Science*, pages 174–191. Springer, 2009.

This paper deals with creating an abstraction of a system and then using the abstraction to guide a model checker. This is relevant because in the slice and dice algorithm, we will be constructing an abstraction of the system and trying to guide JPF along a path that mirrors the abstracted path in order to detect an error. The methods in this paper can be used to perform that operation.

[27] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C. In *ESEC/FSE'05*. ACM Press, 2005.

This paper deals with concolic testing. Concolic testing is a way to use symbolic model checking in order to maximize code coverage or detect errors. The user provides a sample input, then the model checker records path conditions as it executes the program with the given input. The model checker then generates input that will cause different branches to be executed using the path condition information it has and then re-executes a program. This is work is important because it deals with a practical way of doing basic unit testing which is a form of verification.

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis proposal submitted by

Eric S. Noonan

This thesis proposal has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Eric Mercer, Chair	Date
Jay McCarthy	Date
Mark Clement	Date
Quinn Snell, Graduate Coordinator	Date