Ranged Model Checking

Diego Funes[1] Junaid Haroon Siddiqui[2] Sarfraz Khurshid[1]
[1]University of Texas at Austin, Austin TX 78712
[2]LUMS School of Science and Engineering, Lahore, Pakistan

ABSTRACT

We introduce *ranged model checking*, a novel technique for more effective checking of Java programs using the Java PathFinder (JPF) model checker. Our key insight is that the order in which JPF makes non-deterministic choices defines a total ordering of execution paths it explores in the program it checks. Thus, two *in-order* paths define a *range* for restricting the model checking run by defining a start point and an end point for JPF's exploration. Moreover, a given set of paths can be linearly ordered to define consecutive, (essentially) *non-overlapping* ranges that partition the exploration space and can be explored separately. While restricting the run of a model checker is a well-known technique in model checking, the key novelty of our work is conceptually to restrict the run using *vertical* boundaries rather than the traditional approach of using a *horizontal* boundary, i.e., the search depth bound. Initial results using our prototype implementation using the JPF libraries demonstrate the promise ranged model checking holds.

1. INTRODUCTION

Software model checkers, such as the Java PathFinder (JPF) [16], now provide the foundation of an increasingly effective tool-set for systematic checking of programs written in commonly used languages. Recent technological advances have well-supported the core model checking techniques [4] and software model checkers are being applied to larger and larger programs. However, state-space explosion remains a fundamental problem in scaling model checking to real-world applications and realizing its true potential in increasing our ability to deploy more reliable software systems.

We introduce *ranged model checking*, which provides a novel way to deal with state-space explosion by restricting the run of a model checker, specifically JPF, to check execution paths that lie within a given *range*, thereby allowing the model checking problem to be partitioned into several sub-problems of lesser complexity. Our key insight is that the order in which JPF makes non-deterministic choices defines a total ordering of execution paths it checks. Thus, two *in-order* paths define a range for restricting the model checking run by defining a start point and an end point for JPF's exploration (Figure 1). Moreover, a given set of paths can be linearly ordered to define consecutive, (essentially) *non-overlapping* ranges that partition the exploration space and can be explored separately.

Bounding the run of a model checker to restrict the state-space under exploration is a well-established technique to enable more effective software model checking [3, 16, 7, 12].

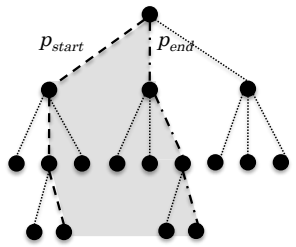

Figure 1: Exploration range (shaded region) defined by two execution paths p_{start} and p_{end}

For example, bounded depth-first search with iterative deepening and bounded input space are commonly used in the context of stateful model checking [16] as well as stateless model checking [7]. The key novelty of ranged model checking is conceptually to bound the run using *vertical* boundaries that are defined by execution paths of the program under checking, thereby enabling a technique that applies in synergy with other bounding techniques, including input space bounding (e.g., using non-deterministic initialization) and search-depth bounding.

We make the following contributions:

- **Ranged model checking**. We introduce the idea of restricting the run of a model checker to a range of execution paths in the program under test defined by an initial path and a final path.
- **Ranged analysis for distributing model checking**. We show how ranging the run of a model checker allows partitioning the problem of model checking into several sub-problems that can be solved separately.
- **Prototype implementation and initial experiments**. We describe our ranged model checking prototype that is based on the open-source JPF libraries. Initial experiments using small but representative Java programs show the promise ranged analysis holds for more effective software model checking.

2. RANGED MODEL CHECKING

In this section we first define execution path ordering and ranged model checking. We then discuss the generation of random execution paths as a way to partition the state space and distribute work among parallel processes. Finally, we consider ranged model checking in the presence of common optimization techniques, including state matching and partial order reduction.

Model checking in JPF is driven by the execution of Java bytecodes acting on the system state maintained by the JPF Java Virtual Machine (JVM^{JPF}). Bytecode instructions are executed until a non-deterministic choice is detected, at which point the search branches and a state transition is introduced. Sources of non-determinism can be explicitly defined by the program being checked, or implicitly introduced by the JVM^{JPF} when multiple threads are ready for scheduling. When an execution path terminates, the search algorithm backtracks to the previous state transition and moves forward on the next branch.

2.1 Path Ordering and Ranged Search

Our key insight into ranged model checking is that the order in which non-deterministic choices are made by the search algorithm defines a total order between all execution paths explored. The order between two paths is determined by inspecting the choices in each path made at the same depth, starting from the first choice. The order of the first *distinct* choice defines the order between the two paths.

Given two execution paths p_{start} and p_{end}, where $p_{start} \leq p_{end}$, we define *Ranged Model Checking* to be the exploration of all states reachable by all execution paths p, such that $p_{start} \leq p \leq p_{end}$.

Consider the state graph shown in figure 1. Two execution paths are given as $p_{start} = [1, 2, 2]$ and $p_{end} = [2, 3, 2]$, where the integer values indicate the branch taken at each level. Since the first distinct choice is $1 < 2$, we define $p_{start} < p_{end}$. The gray area highlights all states and execution paths bounded by p_{start} and p_{end}.

State transitions in JPF are managed by *Choice Generators* instances, which determine the choices available at the transition point and keep track of choices already made as the search progresses. The type of choice generator is determined by the nature of the state transition, whether it is, for example, an explicit non-deterministic choice in the test program; or an implicit thread scheduling choice. The search moves forward by querying choice generators at each state transition for the next choice and applying the result to move to the next state. When an end state is reached, the search backtracks until it reaches a previously registered choice generator with more choices available.

Ranged model checking is implemented as a JPF extension that operates on choice generator objects as they are advanced by the search algorithm. All choice generators in JPF inherit from a common *ChoiceGenerator*<*T*> interface, which provides sufficient functionality for our ranged model checking extension to handle all transitions abstractly.

The abstract ChoiceGenerator interface allows us to treat all state transitions as an abstract list of choices, making the concrete choice type irrelevant. At each state transition, the only information required by ranged model checking is the number of choices available and the current choice. To bound the search within a range, we make use of the ability to advance choice generators to an arbitrary point and mark them as *done* to indicate that no more choices are available.

Given this abstract representation for choice generators we can encode any execution path as a list of integers, where each integer represents the choice made by the search algoritm. The position in the list indicates the depth in the search where the choice was made. JPF includes the class *ChoicePoint* that encodes an execution path as a list of choices. We implement a variant of the ChoicePoint class, which we call a *Path*, and extend it with the notion of ordering. As a practical matter, path ordering is exposed using the *java.lang.Comparable*<*T*> interface, which allows the use of Java containers to store and sort a set of paths.

Path comparison works by finding the first level where the choices made by each path differ. At this point, the two paths encode two distinct choices on *the same choice generator*. This fact is what makes the comparison between these two choices meaningful. The path with the lower choice value is considered to be the lesser path between the two.

Bounding a search within a range given by two paths p_{start}, p_{end} is done on-the-fly using a VM listener to monitor when a choice generator is advanced. The start bound of a ranged search is obtained by forcing the JVM^{JPF} to replay the execution path encoded by p_{start}. The VM listener stops the search when it detects that the current execution path matches p_{end}. Listing 1 shows the implementation of the ranged execution VM Listener.

```
d_start = 0;
d_end = 0;
void choiceGeneratorAdvanced() {
 if (d_start < start.length &&
     depth == d_start) {
  cg.advance(start[d_start].choice);
  d_start += 1;
 }
 if (d_end < end.length &&
     depth == d_end) {
  if (cg.processed() == end[d_end].choice){
   cg.setDone();
   d_end += 1;
}}}
```

Listing 1: Range Execution VMListener

Variables maintained by the listener include *start* and *end*, which store the path bounds p_{start} and p_{end}, respectively; *d_start* and *d_end* store the depth of the next item in p_{start} and p_{end} to be processed by the listener. The search depth (*depth*) and the choice generator at each transition (*cg*) are provided by the JPF execution environment.

Given p_{start} the listener will replay the execution path by advancing choice generators to the choice number encoded in *start*. Each choice in p_{start} is applied only once when the search depth matches the depth of the next unprocessed choice tracked by *d_start*. Note that JPF includes a *ChoiceSelector* VM Listener that works in much the same way, with one important difference: choice generators are advanced and set as done, which prevents the search from moving beyond the replayed path. In contrast, the ranged execution listener only advances the choice generator allowing it to generate more choices and move the search forward.

The current execution path is matched progressively against p_{end} to avoid a full path comparison after each state transition. The variable *d_end* holds the position of the next choice in p_{end} to be processed next. When *d_end* is equal to the current search depth, we can compare the current choice generator with the choice in $p_{end}[d\_end]$. If the choices are equal we have reached the last choice for that transition and mark the generator done. We can reason about this algorithm inductively to see how it works. When *depth* is 0 the search is advancing the top choice generator, which must be the same choice generator as $p_{end}[d\_end]$, thus $d\_end = 0$. Once $p_{end}[0]$ is processed, *d_end* is incremented. For any given value of *d_end*, all previous values must have been processed and the current execution path must match p_{end} up until *d_end*, not inclusive.

The bounding paths used by the ranged execution listener are optional. If no start path is given, the search starts from the initial state. Likewise, if no end path is given the search will continue until there are no more states available to explore. The start path condition is evaluated before the end path condition to handle the case where p_{start} and p_{end} encode the same execution path. Finally, we assume a depth-first search strategy. Although conceptually the algorithm would work for breadth-first search, alternative search strategies have not been validated for this implementation.

2.2 Random Execution Path Generation

Random path generation can be use as a simple mechanism to statically partition the state space and distribute search ranges among distributed processes. Consider N processes with unique IDs from 1 to N. Given a shared seed value, each process can independently generate $N-1$ execution paths, sort them to define N search ranges, and explore the range p_{id}, p_{id+1}.

To generate random paths we use a depth-first search strategy that, instead of backtracking, restarts the search when it reaches an end state. A VM listener detects choice generator is advanced and sets its choice to a random value within the range of the choice generator. When the search reaches an end state the listener captures the execution path.

Random paths will almost certainly result in unbalanced partitions. At this stage, we have not designed an efficient dynamic load balancing scheme. Nevertheless, we describe the generation of random paths within a range as a potential primitive operation in the design of such a scheme, which could be used repartition any given range and distribute it among idle processes.

```
void choiceGeneratorAdvanced(JVM vm) {
 cg = vm.getLastChoiceGenerator();
 int r; // random value
 // bound the random value
 if (d_start == depth &&
     r < start[depth].choice)
  r = start[depth].choice;
 if (d_end == depth &&
     r > end[depth].choice)
  r = end[depth].choice;
 if (r == start[d_start].choice)
  d_start += 1;
 if (r == end[d_end].choice)
  d_end += 1;
 cg.select(r);
}
```

Listing 2: Random execution path generation bounded by range

Listing 2 extends the random path generation to take into account bounding execution paths (some details, such as array bound checks have been omitted for clarity). The random choice is clipped by the bounding paths when the search depth matches the depth of either bound. The *d_start* and *d_end* are incremented after bounding the random choice to handle the case when paths match.

2.3 State Matching and Partial Order Reduction

State matching in model checking eliminates redundant search paths by maintaining a set of visited states and backtracking when the search reaches a state in the set. The state set is built as the search progresses and it cannot be reconstructed from a single execution path.

Random path generation is stateless and can't know apriori which execution paths will be pruned by state matching. It is possible to generate bounding paths that would not be executed in a full search. The state set cannot be recreated from a single execution path and, as a consequence, a ranged search will likely explore states that a full search would have detected as already explored. While we believe it is possible to take advantage of the visited set within a range, here, we consider simply whether a search can be bounded given an arbitrary range in the presence of state matching.

If p_{start} corresponds to a pruned path in the full search, the ranged search will simply replay the complete path and start the search from that point. For p_{end} we note a partial match is enough to stop the search and bound the search within p_{end}. The search within the range is exhaustive even if p_{end} is not executed in its entirety. The states in p_{end} not visited would have already been explored by a lesser execution path.

Partial Order Reduction (POR) is another important technique used to reduce the number of states that need to be explored. Only instructions that are determined at runtime to be *scheduling relevant* will be treated as state transition boundaries and introduce a non-deterministic choice. As it relates to ranged model checking, the important feature of POR as implemented in JPF is that the determination of whether an instruction is scheduling relevant only depends on the JVM^{JPF} kernel state (instruction type, object reachability, runnable threads, locks, etc), which can be recreated by an execution path. This suggests that POR optimizations are compatible with ranged analysis. We believe ranged model checking can further optimize JPF in the presence of partial order reduction.

3. INITIAL EXPERIMENTS

For this initial implementation of ranged model checking we are interested in comparing the performance a full sequential search (both stateful and stateless) against a set of ranged searches that are assumed to execute in parallel.

We consider five test programs that introduce state transitions implicitly by the use of multiple threads or explicitly using JPF's data choice generators:

- *Integer Choice.* Single-thread program instrumented with 3 non-deterministic integer choices from 1 .. 50.
- *Threads (n).* Main thread starts n number of threads and terminates. Each thread updates a non-shared field variable and terminates. This test exercises ranged execution over thread scheduling choice generators. Two configurations with 3 and 6 threads are tested.
- *Linked List Tester.* A linked list test input generator as described in [17] that generates all possible configurations of a linked list with up to four nodes. Valid node configurations are selected using the linked list class invariant method included in the program.
- *Dining Phil (n).* Dining philosophers example program from the JPF distribution modified to eliminate deadlocks using Dijkstra's solution of ordering resource acquisition. Two configurations with 3 and 6 philosopher threads are tested.
- *Dining Phil (n) + choice.* Dining philosopher program where each philosopher updates a non-shared field variable using a non-deterministic integer choice generator from 1 to 5.

Subject	Full Search		Ranged Search (26 ranges)				
			Min.		Max.		Total
	states	time	states	time	states	time	states
Integer Choice	127551	10385	178	68	15954	1471	127651
Threads (3)	400	342	9	47	356	222	3913
Linked List Tester	416656	53242	1738	276	52853	6842	412512
Dining Phil (3)	407	353	5	56	253	155	2496
Threads (6)	12844	2511	1176	162	11851	1939	157648
Dining Phil (6)	66716	6919	69	118	54722	5757	784816
Dining Phil (6) + choice	256340	21550	136	103	201970	16461	3087411

Table 1: Ranged Execution compared with Stateful Search (times in milliseconds)

Subject	Full Search		Ranged Search (26 ranges)				
			Min.		Max.		Total
	states	time	states	time	states	time	states
Integer Choice	127551	7347	178	62	15954	1208	127651
Threads (3)	72804	5185	9	45	21277	1698	73071
Linked List Tester	416656	39737	1738	239	52853	5811	416931
Dining Phil (3)	8938	1213	5	51	2775	354	14213

Table 2: Ranged Execution compared with Stateless Search (times in milliseconds)

Test Name	states	time (ms)
Integer Choice	100	845
Threads (3)	267	900
Threads (6)	481	1093
Linked List Tester	275	923
Dining Phil (3)	192	943
Dining Phil (6)	332	1177
Dining Phil (6) + choice	512	1084

Table 3: Random Path Generation

Performance is measured in terms of execution time and number of states visited during the search. When state matching is used, the number of states shown is the sum of new and visited states reported by JPF. In a stateless search all states are considered new and the number of states shown is the number of 'new' states reported by JPF. All times shown are in milliseconds.

For each test we present results of a full search and a ranged search over 26 partitions created by 25 randomly generated execution paths. Results for stateful search mode are shown in table 1. Results for stateless search mode are shown in table 2. Not all configurations were tested in stateless mode due to state space explosion.

Table 3 shows the execution times required to generate the 25 random execution paths used for the ranged search test. The reported time includes executing the paths and saving them to disk.

For every subject in the experiment we observe a speed up ranging from 1.2x to 7.8x, using the worst performing range compared against the full search. The results suggest that ranged execution is particularly effective when used with programs instrumented with data choice generators. On the other hand, the effectiveness of state matching with multi-threaded programs is evident. For the Dining Philosopher program with 6 threads there is a slight speed up (1.2x) when compared to the slowest range, but the total amount of work done by all the ranged searches combined is much larger than the work done by the full search. This indicates that there is considerable state overlap between the ranges. We also observe that load balancing is uneven with some ranges doing almost no work at all. This is primarily a consequence of using random paths to partition the search and suggests that better partition heuristics are needed.

4. RELATED WORK

The second author's doctoral dissertation introduced the idea of ranged analysis in the context of symbolic execution [13, 14] – a well-known program analysis technique that was first presented over three decades ago [5, 9] but has seen a lot of progress in the last decade [2]. Ranged symbolic execution uses two test inputs to define a range for performing symbolic execution. Experimental results using a prototype based on KLEE [1] showed that ranged analysis can provide more effective symbolic execution both in a sequential setting with a single worker node as well as a parallel setting using 10 workers for the chosen subjects from the GNU Coreutils set of Unix programs. This paper on ranged model checking follows the spirit of ranged analysis for symbolic execution but defines ranged analysis in the context of execution-driven model checking as implemented in an explicit-state model checker, where thread interleaving and non-deterministic choice are foundational elements, which do not feature in symbolic execution of sequential imperative programs. We expect ranged analysis to offer similar benefits in more effective model checking as those observed for symbolic execution. In addition, our work on ranged model checking introduces ranged symbolic execution to the JPF symbolic execution component, Symbolic PathFinder.

There is extensive work on parallel model checking applied to various model checkers, including Murϕ [15], JPF [11] and SPIN [10], where the focus is on partitioning and distributing the set of visited states. In ranged model checking partitioning is defined by execution paths not the state set. Based on the experiment results, it is clear that ranged model checking will have to take into account state set partitioning to reduce the total amount of work.

Parallel Randomized State-space Search [6] (PRSS) and Swarm [8] randomize the order in which choices are processed in order to diversify the search across multiple processes, but at least one process must complete the full search to indicate that the model does not contain errors. Ranged model checking as presented uses random execution paths

as a naïve partitioning algorithm, but other heuristics could be used to define ranges. Furthermore, PRSS techniques are compatible with ranged model checking since it is conceivable to apply them within a range, that is, randomize the order of the choices within a range.

Context-bounded model checking [12] introduced the idea of placing an arbitrary upper bound on the number of thread preemptions (context switches) in each transition. The maximum number of context switches is provided as an user configurable parameter. Ranged model checking bounds the number of choices on any given transition with an lower and upper bound given by two specific execution paths.

5. FUTURE WORK AND CONCLUSIONS

We introduced the idea of *ranged model checking*, and presented a technique that embodies the idea using the Java PathFinder model checker to distribute the model checking problem into sub-problems of lesser complexity. An initial experimental demonstration shows the potential the technique holds in addressing the state-space explosion problem.

To optimize the technique, a research problem we plan to study next is how to (statically or dynamically) compute effective ranges for distributing the model checking problem, especially in the context of state matching and partial order reduction, which may prune a significant part of the state space, thereby making the exploration of parts of certain ranges redundant.

We believe ranged analysis holds much promise for more effective model checking, not just in the context of one run of the model checker to check one program, but also in the context of multiple runs of the model checker, say to check the same program using iterative deepening, e.g., by focusing checking on ranges that include some feasible execution path that does not terminate in the previous depth bound, or to check a new program version when the program undergoes evolution, e.g., by focusing checking on ranges that capture the program modifications. Moreover, we believe *memoized* analysis—in the spirit of memoized symbolic execution [18]—where an encoding of sequences of nondeterministic choices and thread interleavings, e.g., using a trie data structure, which summarizes the previous run(s) of the model checker will provide a key enabling technology to leverage the effectiveness of state matching and partial order reduction in ranged model checking, in particular, and incremental model checking, in general.

6. ACKNOWLEDGMENTS

This work was funded in part by the NSF under Grant No. CCF-0845628. We thank Guowei Yang and the anonymous reviewers for insightful comments and feedback on this work.

7. REFERENCES

[1] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. 8th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, 2008.

[2] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: Preliminary assessment. In *Proc. 33rd International Conference on Software Engineering (ICSE)*, pages 1066–1071, 2011.

[3] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 168–176, 2004.

[4] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2001.

[5] L. A. Clarke. *Test Data Generation and Symbolic Execution of Programs as an aid to Program Validation*. PhD thesis, University of Colorado at Boulder, 1976.

[6] M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel Randomized State-Space Search. In *Proc. 2007 International Conference on Software Engineering (ICSE)*, pages 3–12, 2007.

[7] P. Godefroid. Model checking for programming languages using verisoft. In *POPL*, pages 174–186, 1997.

[8] G. J. Holzmann, R. Joshi, and A. Groce. Swarm Verification Techniques. *IEEE Transactions on Software Engineering* , 37(6):845–857, Nov. 2011.

[9] J. C. King. Symbolic Execution and Program Testing. *Communications ACM*, 19(7):385–394, July 1976.

[10] F. Lerda and R. Sisto. Distributed-Memory Model Checking with SPIN. In *Proc. 5th International SPIN Workshop on Model Checking of Software* , pages 22–39, 1999.

[11] F. Lerda and W. Visser. Addressing Dynamic Issues of Program Model Checking. In *Proc. 8th International SPIN Workshop on Model Checking of Software* , pages 80–102, 2001.

[12] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, pages 93–107, 2005.

[13] J. H. Siddiqui. *Improving Systematic Constraint-driven Analysis using Incremental and Parallel Techniques*. PhD thesis, University of Texas at Austin, 2012.

[14] J. H. Siddiqui and S. Khurshid. Scaling symbolic execution using ranged analysis. In *Proc. 27th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2012. (To appear).

[15] U. Stern and D. L. Dill. Parallelizing the Murphi Verifier. In *Proc. 9th International Conference on Computer Aided Verification*, pages 256–278, 1997.

[16] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proc. 15th International Conference on Automated Software Engineering (ASE)*, pages 3–12, 2000.

[17] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test Input Generation with Java PathFinder. In *Proc. 2004 International Symposium on Software Testing and Analysis (ISSTA)*, pages 97–107, 2004.

[18] G. Yang, C. S. Păsăreanu, and S. Khurshid. Memoized symbolic execution. In *Proc. 2012 International Symposium on Software Testing and Analysis (ISSTA)*, pages 144–154, 2012.