# Bounded Model Checking and Feature Omission Diversity

Amin Alipour and Alex Groce
School of Electrical Engineering and Computer Science
Oregon State University, Corvallis, OR
Email: {alipour,alex}@eecs.oregonstate.edu

***Abstract*—In this paper we introduce a novel way to speed up the discovery of counterexamples in bounded model checking, based on parallel runs over versions of a system in which features have been randomly disabled. As shown in previous work, adding constraints to a bounded model checking problem can reduce the size of the verification problem and dramatically decrease the time required to find counterexample. Adapting a technique developed in software testing to this problem provides a simple way to produce useful partial verification problems, with a resulting decrease in average time until a counterexample is produced. If no counterexample is found, partial verification results can also be useful in practice.**

## I. Introduction

Model checking [5] is a technique for formal verification, which inspects all reachable states of a system for violations of a property. If a violation found, a model checker returns a counterexample—a trace that exhibits the property violation. Providing a counterexample is one of the most important features of model checking, crucial for debugging both systems and properties [3].

In programs with very large or infinite state spaces, the explicit enumeration of all of reachable states is impractical. Alternative techniques, such as symbolic model checking [11], model checking with abstraction [4], and *bounded model checking (BMC)* [2], avoid the need for explicit enumeration by representing multiple states at once via logical constructs. In bounded model checking, in particular, there is an additional emphasis on the discovery of counterexamples, and verification is only with respect to some upper bound on the length of allowed counterexample traces[1].

Even with the power of symbolic techniques, the state-space explosion is a fundamental problem for model checking: real-world systems have very large state spaces, and effective abstractions and symbolic representations are challenging to discover. With the growing prevalence of multi-core processors, effective methods for parallelizing model checking have become an increasingly important part of the effort to combat the state-space explosion. *Swarm verification* [9] is an explicit-state approach in which, rather than allocating all computational resources to a single model checking search strategy (e.g., a bitstate search with a very large hash table), a "swarm" of model checking runs with limited memory and time, and different search strategies (primarily transition orderings) are performed. These runs are completely independent, with no communication between processes; the cost of communication can outweigh the gains of avoiding duplicated work. The diversification of search strategies leads to exploration of different parts of the state space, however, decreasing the time required to find a counterexample if one exists, and increasing code and state-space coverage even if a counterexample is not found. For very large models, including an experimental network architecture, a real-time operating system kernel, and flash file systems for space missions, swarm verification has proven essential to finding subtle system flaws.

*Swarm testing* [8] generalizes the search diversification of swarm verification beyond the choice of a depth-first search strategy to the selection of *test features*. A test feature is a predicate over test cases, controlled by the test generation process. Features may be API calls or more general predicates; for example, in testing a file system a test feature might be whether the test case includes calls to `close`, and in C compiler testing a feature might be whether a test case includes pointer operations. Building off the discovery that disabling certain API calls (e.g., directory operations) in an explicit-state model checking harness for a flash file system could greatly increase path coverage for other operations (e.g., file read/write operations), swarm testing relies on *randomly omitting certain features from test cases*. In the context of random testing, this means that, rather than using the entire test computation budget to generate and execute test cases drawn from a single distribution in which every test feature is represented, we generate a fixed swarm of test configurations, each of which only includes a subset of all possible test features. The swarm configurations each receive an equal share of the computational budget. The *feature omission diversity* provided by swarm testing results in much better fault detection and code coverage than the conventional approach of a single, all-encompassing, default configuration. Even though swarm testing, unlike swarm verification, provides no additional parallelism, diversity is valuable due to the fact that *some features can suppress system behaviors*. E.g., when looking for a stack overflow bug, `pop` operations make it hard to reach error states, and when testing a file system, `close` operations prevent complex file access behaviors.

Swarm testing is clearly applicable in explicit-state model

[1]In practice, the bound can sometimes be shown to be at equal to or greater than that of the maximum length for the shortest counterexample to the system, in which case BMC verification implies system correctness.

checking of very large state spaces (in fact, the inspiration for swarm comes from such "testing via model checking"). A more interesting question, however, is whether feature omission diversity can be useful in symbolic approaches. In this paper, we propose the application of swarm configuration techniques to constraint-based bounded model checking.

Rather than simply using an *ad hoc* approach of removing certain choices from a model checking harness that defines system behaviors, we base swarm bounded model checking (swarm BMC) on a trace-based approach to bounded model checking [7]. This approach provides a generalized basis for feature omission in constraint-based approaches, and swarm provides the trace-based technique with a source of traces. We show very preliminary results on small systems, indicating that 1) feature omission diversity can speed up the discovery of counterexamples in BMC when multiple cores are available and 2) it is possible to use the results from swarm runs to aid verification, even if no counterexamples exist.

## II. Related Work

This work builds on the ideas of search diversity explored in the work of Holzmann et al. [9] and Dwyer et al. [6], and continued in swarm testing [8]. Feature diversity is implemented by equating features to log elements in a program trace, based on the approach to trace-based BMC reduction introduced by Groce and Joshi [7]. Arguably, swarm's model checking via feature omission, where model checking results indicate correctness (or a counterexample) for a subset of program behaviors is a dual of *conditional model checking*, proposed by Beyer et al. [1], where conventional model checking can return partial results.

## III. Bounded Model Checking with Feature Omission

Bounded model checking (BMC) reduces the problem of model checking to a satisfiability problem [2]. For a system $C$ and a property $p$, BMC unrolls the control structure of the system for $k$ steps and derives a propositional formula $f$ such that $f$ is satisfiable if and only if there is a violation of $p$ in computations of length ($\leq$) $k$ of $C$. The solution to $f$ defines a counterexample for the property $p$ over $C$. A SAT solver (or SMT solver) is used to solve the satisfiability of $f$. If a sufficiently large $k$ is selected and the SAT solver can handle the resulting constraints, BMC is highly effective for finding faults in both hardware and complex software systems, including operating system kernel code.

Groce and Joshi [7] proposed an extension to BMC in which the only counterexamples considered are those that would produce a given log tracing system behavior. Consider a program containing `log(s)` statements that append a string `s` to a log that partially indicates program behavior. Such logs are ubiquitous; "`printf`" debugging is applied at some point to almost all software systems. An obvious variation of bounded model checking is to consider only counterexamples that produce a given log of system behavior (perhaps hypothesized as impossible by a systems engineer, or perhaps derived from a failed test case). Restricting the executions considered in a static analysis or symbolic execution by a trace can be very expensive, if the restriction requires additional "history" variables to record the log produced by an execution. In order to avoid this overhead, the trace-restriction algorithm, after adding the desired log state as a post-condition at program termination, propagates the final assumption to arrive at a new program annotated with assumptions that force the desired log output, and slices the constraint problem according to these assumptions. Experimental results show that restricting behavior to a trace can greatly reduce the size of SAT constraints and the time required to produce a counterexample.

We exploit this approach as a basis for feature omission diversity. In particular, we define features, in our swarm BMC approach, as logging events. To omit a feature, therefore, means to consider only execution traces that do not contain a given log event. This definition of features is quite general: because logging statements can be guarded with conditionals in a program, features can include not only particular functions, statements, or input values, but also any variable valuations at particular program locations. Given a program containing `log` statements and a set of features to be omitted (the swarm configuration), performing BMC under the configuration is simple: 1) First, all logging statements that do not involve the omitted features are removed from the program. 2) The assumption that the program log is empty at termination is added as a post-condition. Restriction of execution to an empty log is most elegantly achieved by simply replacing all calls to `log` with the statement `assume(false)`. These `assumes` can be propagated via the techniques described by Groce and Joshi, or by any slicing technique implemented in a bounded model checker.

Algorithm 1 illustrates the feature diversity omission approach. Given a program and a set of features, some features of the program are randomly selected (Line 3). These features are omitted from the program by adding new assumptions to the program (Lines 4 and 5) to derive a set of new programs with the same behavior as the original program, except that none of the omitted features will be allowed in counterexamples. We run BMC in parallel on the set of new programs (Line 6).

What is the value of this "parallel" BMC? Obviously, there is no traditional compositional verification in this case: if no $sp$ has a counterexample, it does not follow that the program $p$ has no counterexample (unless one $F_i$ is the null set). However, if any $sp$ has a counterexample, it is also a counterexample for the original program $p$. If the time to produce a counterexample for any $sp$ is shorter than the time required to produce a counterexample for $p$ then we have obtained a counterexample more quickly than what is possible with the traditional approach. Moreover, it may be the case that BMC for reasonable $k$ on $p$ itself produces a SAT instance that is too complex, and the SAT solver exhausts memory or time available for verification. It many such cases, at least one $sp$ will produce a counterexample, due to its smaller (due to slicing) or at least more constrained SAT problem. In these cases, swarm BMC makes effective BMC possible when it

was previously not feasible.

**Algorithm 1** Algorithm For Swarm Bounded Model Checking

**Input:** program $p$, set $F$ of features

1: **while** budget allows **do**
2: **for** all processors available **do**
3: Pick a random set $F_i \subseteq F$
4: Build $sp$ by replacing log statements in $p$ for $F_i$ with `assume(false)` and removing other `log` statements from $p$
5: Propagate assumptions and slice $sp$
6: Perform BMC on $sp$
7: **end for**
8: **end while**

We illustrate the potential value of the swarm approach with a simple example, a stack overflow bug. Figure 1 depicts an implementation of a simple stack API where `push` adds an integer to the top of stack, `pop` removes an item from top of stack and `top` returns the value of the top of the stack. The stack library is flawed in that 1) `top` actually returns the value one above the top of the stack and 2) the `push` function does not check for stack overflow. The lack of a specification prevents us from detecting the first fault, but the second fault will produce an array bounds violation when a `push` call is made on a stack already containing 64 items. The API calls are the natural features of this system.

The effectiveness of swarm testing and swarm BMC relies on the insight that, whether features are API calls or predicates over inputs, most faults can be exhibited by counterexamples that only exhibit a small portion of program's behavior. For example, in our stack example, only `push` operations are required to produce a stack overflow — `pop` operations actually delay failure, and `top` operations are irrelevant.

```
#define SIZE 64
#define TLEN 100
int s = 0;
int stack[SIZE];
int top() {
  log("top");
  return stack[s];
}
void push(int i) {
  log("push");
  stack[s++] = i;
}
void pop() {
  log("pop");
  if (s > 0) {
    s--;
  }
}

int main () {
  int v, action;
  for (int i = 0; i < TLEN; i++) {
    action = nondet_int();
    __CPROVER_assume ((action >= 0) && (action <= 2));
    switch (action) {
    case 0:
      v = top();
      break;
    case 1:
      v = nondet_int();
      push(v);
      break;
    case 2:
      pop();
      break;
    }
  }
}
```

Fig. 1. Stack code.

```
int top() {
  __CPROVER_assume(false);
  return stack[s];
}
void push(int i) {
  stack[s++] = i;
}
void pop() {
  if (s > 0) {
    s--;
  }
}
```

Fig. 2. Snippet of input to CBMC to omit *top* function calls.

We ran the C bounded model checker CBMC [10] on this program and on three variations, each of which omitted one feature (Figure 2 shows the version where `top` is omitted). Figure 3 summarizes the results.

The results show that swarm BMC can returns a counterexample up to 71% more quickly than BMC on the program with all features enabled. Omitting `top` provides little benefit, but omitting `pop` allows us to produce a counterexample very quickly. If all BMC runs are performed in parallel on a multi-core machine with sufficient memory for each run, or in a cloud computing setting, the time-to-first counterexample is greatly reduced. Moreover, we argue that counterexamples from swarm BMC are likely to be much more useful for debugging systems, as they will contain fewer features that are not relevant to the fault. In this trivial case, a counterexample consisting exclusively of `push` operations is clearly ideal.

What if we correct the program by fixing `top` and `push`? The swarm verifications are still (slightly) quicker than the full verification, but do not provide a conclusive proof of correctness for the program. However, we can theorize some benefit even in the case of correct systems for swarm verification: verifying the original program takes three seconds longer than *verifying a version of p which requires that every trace include at least one call to each of* `top`, `pop`, *and* `push`.

## IV. Experimental Result

In this section we present preliminary results for swarm BMC. We have experimented with feature omission diversity on a number of data structures in C. In these experiments, we used only the basic CBMC slicer, rather than any more sophisticated reduction approach.

| Omitted Feature | Time without Slicing (Seconds) | Time with Slicing (Seconds) | Verification Status |
|---|---|---|---|
| none | 325 | 49 | Counterexample |
| push | 40 | 4 | Verified |
| pop | 101 | 14 | Counterexample |
| top | 291 | 48 | Counterexample |

Fig. 3. Swarm BMC for a simple stack.

We used Weiss's [12] algorithms text source [2] as a source for simple examples. We modified the source code for programs to introduce array bounds and null pointer dereference faults.

We chose two data structures of this set: Array-Queue and Stack List. The main program for each experiments invokes the functions of the libraries in a harness similar to the main function in Figure 1.

We used CBMC version 4.0 [10] for bounded model checking on a four-processor Intel 2.8GHz system with 8 GB RAM. Depth of bounded model checking in the following experiments is the maximum number of steps allowed in a trace. In each experiment, we tried CBMC with and without slicing, to show that simply adding additional constraints based on feature omission is valuable, even if no slicing is performed to reduce the SAT problem size.

### A. Array-Queue

Array-Queue is an implementation of a queue in C using arrays. Array-Queue includes `Enqueue`, `isEmpty`, `Dequeue`, `Front` and `DisposeQueue` functions. We introduced two bugs into the program: (1) violation of array boundary caused by Enqueue-ing more data than the size of the queue, and (2) null pointer dereferencing when calling an operation on a disposed queue.

Figure 4 shows the results of bounded model checking for Array-Queue. It shows the results for five different depths: 1000, 2000, 3000, 4000, and 5000. In all experimental results, the fastest time to produce a failure is shown in bold. In every case, with or without slicing, all swarm configurations help CBMC find a counterexample faster than with the default configuration. Additionally, if the swarm configurations can be executed in parallel (not unlikely; none of these runs required more than about 1GB of RAM), swarm will produce two counterexamples showing different faults before the default configuration can produce a single counterexample.

### B. Stack List

Stack List is a stack with dynamic memory allocation. It implements `Push`, `Top`, `Pop`, and `DisposeStack` functions. We added a null pointer dereferencing bug to the program. Again, swarm BMC outperforms standard BMC, whether we choose a configuration at random and run on a single processor or we perform runs in parallel.

[2]Available at `http://users.cis.fiu.edu/~weiss/dsaa_c2e/files.html`

## V. Discussion

In this paper, we proposed that feature omission diversity, known to be useful in software testing, may also be valuable in bounded model checking. By omitting features in a program's execution, we can produce smaller and more easily checked SAT instances, while often preserving at least one counterexample trace. This allows us to introduce parallelism into BMC without a parallel decision procedure for constraints. The counterexamples produced, moreover, are potentially simpler to understand and debug than typical BMC counterexamples.

As future work, we plan to apply swarm BMC to larger, more realistic examples that challenge the abilities of current software BMC. Further investigation of the ability of specialized slicing to improve runtime and the practicality of swarm parallel BMC are also needed. Finally, we speculate that even when no counterexamples exist, performing a number of swarm runs and then using the verified configurations to restrict traces to consider in a run on the full program may let us model check programs too large for verification without additional constraints.

## References


[1] Beyer, D., Henzinger, T. A., Keremoglu, M. E., and Wendler, P. Conditional model checking. In *Technical Report MIP-1107, University of Passau, Germany* (2011).

[2] Biere, A., Cimatti, A., Clarke, E., Fujita, M., and Zhu, Y. Symbolic model checking using sat procedures instead of bdds. In *Design Automation Conference, 1999. Proceedings. 36th* (1999), pp. 317 –320.

[3] Clarke, E., Grumberg, O., McMillan, K., and Zhao, X. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Design Automation Conference* (1995), pp. 427–432.

[4] Clarke, E. M., Grumberg, O., and Long, D. E. Model checking and abstraction. *ACM Trans. Program. Lang. Syst. 16* (September 1994), 1512–1542.

[5] Clarke, E. M., Grumberg, O., and Peled, D. *Model Checking*. MIT Press, 2000.

[6] Dwyer, M. B., Elbaum, S. G., Person, S., and Purandare, R. Parallel randomized state-space search. In *International Conference on Software Engineering* (2007), pp. 3–12.

[7] Groce, A., and Joshi, R. Exploiting traces in static program analysis: better model checking through printf's. *STTT 10*, 2 (2008), 131–144.

[8] Groce, A., Zhang, C., Eide, E., Chen, Y., and Regehr, J. Swarm testing, 2011. (In submission).

[9] Holzmann, G., Joshi, R., and Groce, A. Swarm verification techniques. *IEEE Transactions on Software Engineering 99*, PrePrints (2010).

[10] Kroening, D., Clarke, E. M., and Lerda, F. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems* (2004), pp. 168–176.

[11] McMillan, K. L. *Symbolic model checking; An approach to the state explosion problem*. PhD thesis, Department of Computer Science, Carnegie Mellon University, 1992.

[12] Weiss, M. A. *Data Structures and Algorithm Analysis in C*. Addison Wesley, 1996.

| Omitted feature | Depth | Time(S) |
|---|---|---|
| | 1000 | 22.979 |
| Enqueue() | 1000 | 14.019 |
| IsEmpty | 1000 | 17.930 |
| **Dequeue** | **1000** | **12.791** |
| Front | 1000 | 20.739 |
| | 2000 | 38.030 |
| Enqueue() | 2000 | 24.125 |
| IsEmpty | 2000 | 25.260 |
| Dequeue | 2000 | 26.165 |
| Front | 2000 | 24.425 |
| | 3000 | 38.067 |
| **Enqueue()** | **3000** | **23.875** |
| IsEmpty | 3000 | 25.470 |
| Dequeue | 3000 | 26.245 |
| Front | 3000 | 24.316 |
| | 4000 | 38.023 |
| **Enqueue()** | **4000** | **23.765** |
| IsEmpty | 4000 | 25.298 |
| Dequeue | 4000 | 26.373 |
| Front | 4000 | 24.282 |
| | 5000 | 36.966 |
| **Enqueue()** | **5000** | **23.382** |
| IsEmpty | 5000 | 24.918 |
| Dequeue | 5000 | 26.193 |
| Front | 5000 | 24.414 |

(a) Without Slicing

| Omitted feature | Depth | Time(S) |
|---|---|---|
| | 1000 | 17.530 |
| Enqueue() | 1000 | 13.352 |
| **IsEmpty** | **1000** | **12.639** |
| Dequeue | 1000 | 13.928 |
| Front | 1000 | 13.048 |
| | 2000 | 27.292 |
| Enqueue() | 2000 | 25.549 |
| IsEmpty | 2000 | 25.120 |
| **Dequeue** | **2000** | **23.170** |
| Front | 2000 | 27.826 |
| | 3000 | 27.342 |
| Enqueue() | 3000 | 25.616 |
| IsEmpty | 3000 | 25.526 |
| **Dequeue** | **3000** | **22.594** |
| Front | 3000 | 27.733 |
| | 4000 | 27.196 |
| Enqueue() | 4000 | 25.821 |
| IsEmpty | 4000 | 25.252 |
| **Dequeue** | **4000** | **22.428** |
| Front | 4000 | 27.433 |
| | 5000 | 26.615 |
| Enqueue() | 5000 | 24.591 |
| IsEmpty | 5000 | 24.735 |
| **Dequeue** | **5000** | **21.678** |
| Front | 5000 | 27.157 |

(b) With Slicing

Fig. 4. Swarm BMC for Array Queue.

| Omitted Feature | Depth | Time (S) |
|---|---|---|
| | 100 | 0.291 |
| Push | 100 | 0.283 |
| **Top** | **100** | **0.281** |
| Pop | 100 | 0.283 |
| DisposeStack | 100 | 0.282 |
| | 200 | 1.592 |
| **Push** | **200** | **1.462** |
| Top | 200 | 1.498 |
| Pop | 200 | 1.465 |
| DisposeStack | 200 | 1.469 |
| | 300 | 7.151 |
| **Push** | **300** | **6.068** |
| Top | 300 | 6.472 |
| Pop | 300 | 6.281 |
| DisposeStack | 300 | 6.189 |
| | 400 | 28.509 |
| Push | 400 | 25.646 |
| Top | 400 | 26.468 |
| **Pop** | **400** | **19.392** |
| DisposeStack | 400 | 20.178 |
| | 500 | 69.376 |
| Push | 500 | 70.351 |
| **Top** | **500** | **50.165** |
| Pop | 500 | 64.351 |
| DisposeStack | 500 | 57.508 |

(a) Without Slicing

| Omitted Feature | Depth | Time (S) |
|---|---|---|
| | 100 | 0.297 |
| Push | 100 | 0.288 |
| Top | 100 | 0.289 |
| **Pop** | **100** | **0.285** |
| DisposeStack | 100 | 0.287 |
| | 200 | 1.798 |
| Push | 200 | 1.575 |
| **Top** | **200** | **1.593** |
| Pop | 200 | 1.595 |
| DisposeStack | 200 | 1.659 |
| | 300 | 7.312 |
| Push | 300 | 7.278 |
| Top | 300 | 6.829 |
| **Pop** | **300** | **6.441** |
| DisposeStack | 300 | 6.825 |
| | 400 | 27.276 |
| Push | 400 | 20.937 |
| **Top** | **400** | **20.092** |
| Pop | 400 | 21.188 |
| DisposeStack | 400 | 20.484 |
| | 500 | 74.217 |
| Push | 500 | 61.661 |
| Top | 500 | 59.349 |
| Pop | 500 | 60.722 |
| **DisposeStack** | **500** | **56.211** |

(b) With Slicing

Fig. 5. Swarm BMC for Stack List.