Nark: Evolving Bug-Finding Compiler Extensions with Genetic Algorithms

Kenneth Ashcraft
Stanford Computer Science Department
Stanford University
Stanford, California 94305
kash@cs.stanford.edu

ABSTRACT

Static analysis techniques have successfully found thousands of bugs in software systems. With these techniques, it is possible to encode a wide variety of rules such as "don't use freed memory" and "don't call blocking functions with interrupts disabled. Unfortunately, writing these rules requires learning a new programming language or a set of interfaces, which has proven to be a barrier to entry.

This paper describes Nark, a tool that uses genetic algorithms (GA) to evolve the encoding of the rule to find a desired class of bugs. In addition to this novel application of GA, this work uses state machine canonicalization to drastically improve cache performance in the GA evaluation function.

1. Introduction

Static analysis techniques such as Metacompilation (Engler et al. 2000) have successfully found thousands of bugs in software systems. Metacompilation (MC) allows the user to encode system specific rules such as "don't use freed memory" and "don't call blocking functions with interrupts disabled". These rules are then enforced by the compiler at compile time. As opposed to dynamic analysis, MC does not require the execution of any code. This allows it to analyze the entire source base, catching bugs in code that may not be exercised during execution. However, it is possible for MC to make mistakes. It is possible for MC to miss bugs (false-negatives) and for it to report errors spuriously (false-positives).

To encode a rule, the user describes a state machine in Metal, a state machine description language (Chelf, Engler, and Hallem 2002). Commonly called checkers, these rules are automatically applied to the entire source base. Unfortunately, writing a checker requires learning Metal: the syntax, interfaces, and programming paradigm. Though it is a relatively small language, Metal does take time to learn.

This paper describes Nark, a system that aims to make learning Metal unnecessary. Rather than having the user write checkers in Metal, the user supplies a C source file that contains functions that have instances of the class of bugs that the user desires to find. Nark then uses genetic algorithms (GA) to evolve a checker to find that class of bugs.

Section 2 of this paper describes the Metacompilation system and the Metal language. Section 3 describes the Nark system. Section 4 describes using Nark to find a simple class of bugs. Section 5 discusses the results of the application of Nark. Section 6 concludes, and Section 7 discusses future work.

2. Metacompilation

Using MC involves three main steps: encode the rule in Metal as a checker, apply the checker to the source code base, and inspect the output of the checker. These three steps are described below.

2.1. Metal

For a complete description of Metal, see (Chelf, Engler, and Hallem 2002). This is a summary of the relevant points in that paper.

Checkers written in Metal leverage the fact that rules frequently map to source code actions such as function calls or pointer dereferences. Using the terminology of automata, these actions are the *alphabet* of the checker, and the checker as a whole defines the *language* of the rule. A finite state machine is then used to find illegal sequences of the alphabet. These illegal sequences are potentially bugs.

A checker is the list of states in the state machine mentioned above. The first state in the list is implicitly set as the start state. Each of the states in the list contains a set of transitions that can potentially execute while the state machine is in that state. Each transition is specified by a *pattern* and a destination state and/or a C code *action*. A pattern is a template written in the source language. The pattern identifies a source construct that, when encountered in the source base, will cause the transition to execute. An action is an arbitrary block of C code that most often will either execute a state transition or print an error message.

Figure 1 shows a simple checker that looks for violations of the rule "re-enable interrupts after disabling them" in the Linux Kernel. A call to `cli()` disables interrupts, and a call to `sti()` enables interrupts. These two functions are the alphabet of the checker. Conceptually, it checks that each call to disable interrupts has a corresponding call to enable interrupts along all paths stemming from the disable.

There are two states in the checker. The `enabled` state is the starting state and contains a transition that triggers when a call to `cli()` is found in the source code. This transition switches the checker to the `disabled` state. There are two transitions in the `disabled` state. The first triggers on a call to `sti()` and switches the checker back to the `enabled` state. The second transition contains a special pattern that triggers when a program path ends. If a path ends with interrupts disabled, the checker prints an error message indicating the location and type of the error.

```
sm interrupt {
  enabled:
    { cli() }==> disabled;

  disabled:
    { sti() }==> enabled
    | $end_of_path ==>
        error("Did not enable"
              "interrupts!");
}
```

Figure 1 A simple interrupt checker

```
sm printer {
  decl any_pointer ptr;

  start:
    { *ptr } ==>
      print("Pointer deref");
}
```

Figure 2 A checker to that prints for every pointer dereference

In addition to the literal matching of function calls, Metal can also match on generic types. Figure 2 shows a simple checker that prints a message every time it sees a pointer dereference. The template variable `ptr` is declared as a pointer type, and the pattern `*ptr` matches every type of pointer dereference whether the pointer be a `char*`, `int*`, or `struct foo*`. Also, the pattern `${1}` will trigger on every token in the function and is useful for debugging or as a placeholder transition. This feature will be used in section 3.

The last pertinent feature of Metal is that if multiple transitions could trigger on a given input, the transition listed first will subsume the others. For example, if we have the two patterns (1) `{ malloc(n) }` and (2) `{ ptr = malloc(n) }`, the pattern (1) will trigger even though it is less specific than the (2). This paper refers to this feature as the *subsumption rule*.

2.2. Applying Checkers

After the user has encoded his rule in a Metal state machine, the MC system compiles that state machine as a loadable module for the gcc compiler. For each function, the checker is applied to the control flow graph (CFG), which is computed from the abstract syntax tree. After analyzing a single path through the CFG, MC backtracks to the last branch point, resets the state of the state machine to the state previously computed at that point, and continues analysis along one of the unexplored branches. The analysis of the function is not complete until all paths are checked.

2.3. Inspecting Reports

After the checker has analyzed the entire source base, the user must inspect the set of checker-generated reports. Since some of the reports may be false-positives, it is necessary to hand filter them. It is often possible to adjust the checker to prevent it from emitting false positives of a certain type. For example, in the checker described in section 2.1, it is possible that there are other functions besides `sti()` that enable interrupts. Since those functions are not part of the checker's alphabet, it will ignore them and mistakenly remain in the `disabled` state. Because interrupts

are not actually disabled, the checker will emit a series of false positives. In fact, the Linux kernel uses the `restore_flags()` routine in addition to `cli()` to enable interrupts. After noticing the pattern of false positives, the user can insert an additional transition to the `disabled` state so that the false positives do not appear in the future. Figure 3 shows the modified checker.

```
sm interrupt {
  decl any_expr flags;

  enabled:
   { cli() }==> disabled;

  disabled:
   { sti() }==> enabled
   | { restore_flags(flags) ==>
      enabled
   | $end_of_path ==> error("Did "
      "not enable interrupts!");
}
```

Figure 3 Modified interrupt checker to eliminate some false positives

Patterns
`{ cli() }`
`{ sti() }`
`$end_of_path$`

Table 1 Patterns present in the simple interrupt checker

3. Nark

Rather than having the user write a checker in Metal and be forced to learn a new programming language and paradigm, Nark is a system that allows the user to simply give examples of a class of bugs, and Nark will evolve the checker to find those example bugs. Assuming that the examples are representative of real bugs, the evolved checker will be effective on real systems. This system is analogous to dogs that are used to find smuggled drugs. After successfully finding drugs in training situations, the dogs are used in airports to find real drugs. Sometimes the dogs will make mistakes and bark when there are no drugs present, and sometimes they will not bark when there are drugs present. The better the training is, the fewer the mistakes the dogs make.

3.1. State Machine Representation

Nark uses GA to evolve the bug-finding state machine. As described in the Artificial Ant problem (Jefferson et al. 1991), a state machine can be concisely represented by a set of transitions. In the case of Nark, those transitions have three parts: the source state, the pattern necessary to trigger the transition, and the destination state. Since the state names are irrelevant to solving the problem Nark simply refers to them with an integer. The patterns can be stored in an array where they can be referenced by their index in that zero-indexed array. Table 1 is the list of patterns that are present in the interrupt checker of Figure 1. Nark can then represent a transition as a set of three integers: `(source, pattern, destination)`. For example, the transition `(1, 2, 5)` represents an edge in the state machine that starts at state one, is triggered by the third pattern in the array (end_of_path), and ends at state five.

Nark then represents the entire state machine as a vector of these transitions plus a single integer representing which state to use as the start state. Concatenated together, Nark has a vector of integers with which it can perform the standard genetic operations of crossover and mutation. Figure 4 shows an example mapping between a chromosome and the Metal state machine. For each run of the GA, Nark hardcodes the number of transitions present in the state machine.

While all of the intermediate states' names are irrelevant to the semantics of the state machine, the error state's name is important. Nark treats the highest possible state number as the error state. Also, states lacking outgoing transitions are given the no-op transition of `${ 1 } ==> { }`.

Nark's state machine representation differs from that of the Artificial Ant problem in that not all possible inputs are accounted for at each state. Whereas the Artificial Ant problem had an explicit transition for each of the two possible inputs at every state, Nark only has explicit transitions for those inputs that seem important. There is a default, self-transition for those inputs that are not explicitly handled. The subsumption rule from section 2.1 means that it is important not to have those spurious transitions in Metal because they could possibly subsume meaningful transitions.

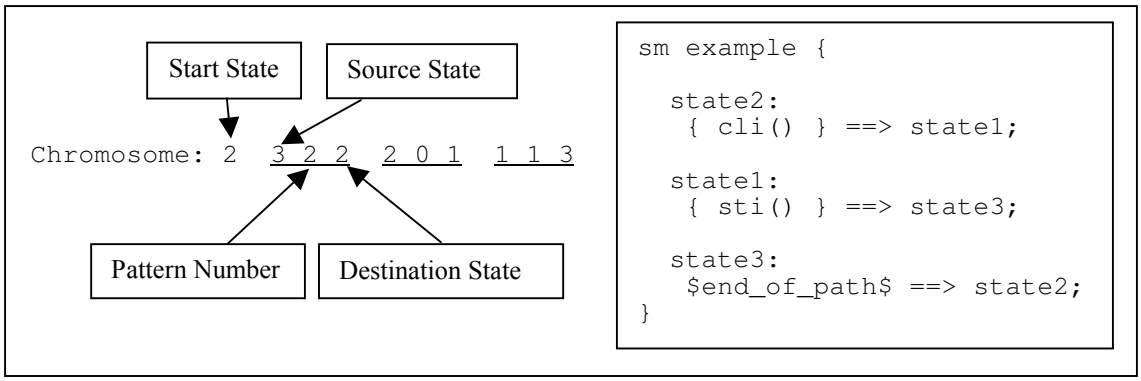

Figure 4 An example mapping between a chromosome and a Metal state machine. The chromosome's three transitions (each with source state, pattern number, and destination state) are underlined and can also be found in the state machine on the right.

3.2. State Machine Canonicalization

While the mapping between a GA chromosome and a Metal state machine is straightforward, it is less than satisfactory. It is possible to have unreachable states and untriggerable transitions. The unreachable state case is demonstrated in Figure 5. Furthermore, multiple chromosomes can map to semantically identical state machines. Figure 6 shows an example of two such chromosomes. These artifacts of the Nark representation result in inefficiency in the evaluation function. Unnecessary states and transitions slow down MC. In addition, Nark will be forced to re-evaluate semantically identical state machines.

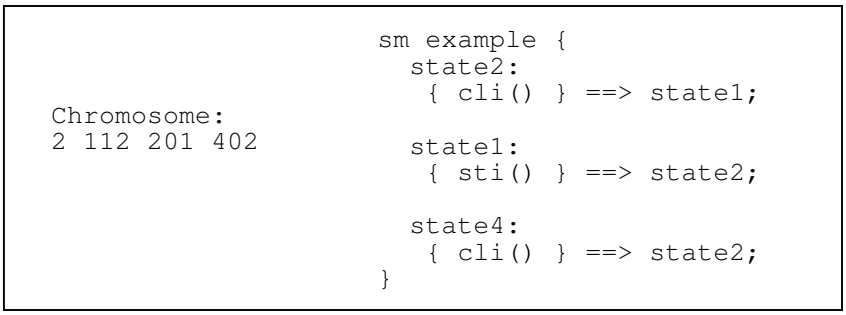

Figure 5 A chromosome with an unreachable state. state4 is unreachable because there are no paths from the start state (state2) to it.

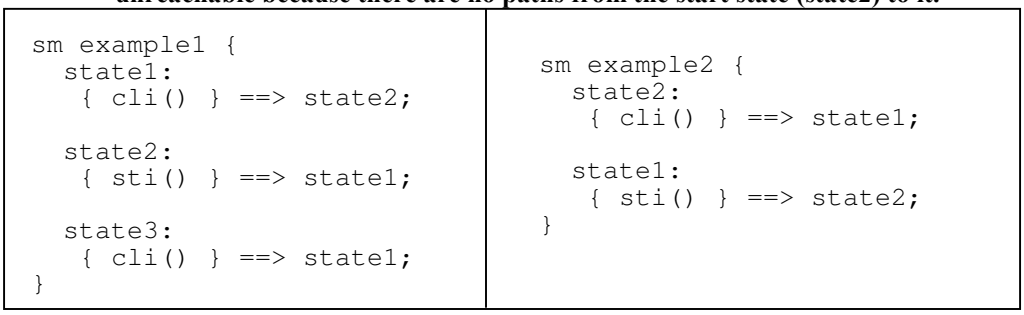

Figure 6 Two semantically identical state machines. state3 of the state machine on the left is unreachable.

To avoid these two problems, Nark uses state machine canonicalization (SMC). Similar to the canonicalization techniques used in the model checking community (Musuvathi et al. 2002**)**, SMC converts state machines to a common format that throws away superfluous states and transitions. Since the state names are irrelevant to the functionality of the state machine, SMC makes all state machines start in state 1 and does something like a topological sort (Cormen, Leiserson, Rivest 1990) so that the transitions of semantically identical state machines will

appear in the same order. Figure 7 gives the canonicalized representation of the two semantically identical state machines of Figure 6.

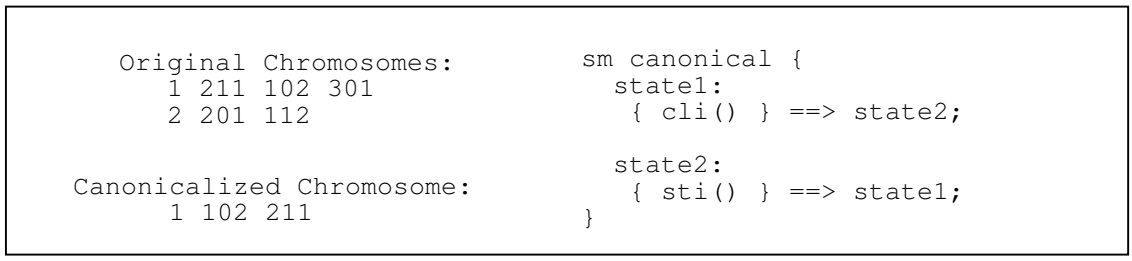

Figure 7 Canonicalization of two semantically identical chromosomes. The state machine on the right is canonicalized because there are no spurious states and the start state is state1.

```
1 Vector Canonicalize(int startState, Vector transitions) {
2   Vector buckets[NUM_STATES];
3   Vector result;
4
5   /* Do a stable bucket sort of transitions */
6   foreach Transition t in transitions {
7     buckets[t.source].append(t);
8   }
9
10  int nextState = 1;
11  int translations[NUM_STATES];  /* Zero initialized */
12  ChangeSourceAndAppend(buckets[startState], result, nextState);
13  translations[startState] = nextState;
14  nextState++;
15  for (int i = 0; i < result.length; i++) {
16    Transition t = result.get(i);
17    int oldDest = t.dest;
18    if (oldDest != ERROR_STATE) { /* Don't translate the error state */
19      if (translations[oldDest] == 0) {
20        ChangeSourceAndAppend(buckets[oldDest], result,
21                              nextState);
22        translations[oldDest] = nextState;
23        nextState++;
24      }
25      t.dest = translations[oldDest];
26    }
27  }
28  return result;
29}
30
31void ChangeSourceAndAppend(Vector toChange,
32                           Vector canonical, int state) {
33  foreach Transition t in toChange {
34    t.source = state;
35    canonical.append(t);
36  }
37}
```

Figure 8 State machine canonicalization algorithm.

The algorithm for SMC given in Figure 8 assumes the representation given in section 3.1 and the subsumption rule of section 2.1. The function `Canonicalize` takes in the starting state and the vector of transitions for the state machine and returns the vector of transitions for the canonicalized state machine.

Lines 6 – 8 do a stable bucket sort on the incoming transitions. This step allows easy lookup of the set of transitions that come from a given state. It is necessary to do a stable sort because of Metal's subsumption rule. The algorithm then appends the set of transitions that stem from the start state (line 12). Since the start state may not be

state 1, the algorithm changes all of the start state transitions to have a source state of 1. The algorithm then records this translation for later use (line 13).

Line 15 begins iterating through the transitions in `result`. For each transition in `result`, the algorithm must take the transitions from the bucket specified by the transition's destination and append them to `result` (line 20) like it did for the start state. It will not append the transitions that stem from the error state, nor will it translate the destination if it is the error state. This is because Nark needs some way of signifying to MC that a transition should emit an error when triggered. Finally, the transition's destination is translated to the canonicalized state (line 25). Note that `result` will be growing during the iteration, so it is necessary that the loop exit condition be evaluated dynamically. Also note that only reachable states will be included in the canonicalization. If a state is unreachable, its transitions will never be appended to `result` on line 20.

Once all of the transitions have been canonicalized, they are returned to the caller who can use a hashtable to determine if the state machine has already been evaluated. As a whole, this cache lookup is linear in the number of transitions used in the GA representation. The bucket sort takes linear time, appending transitions to `result` is linear since it can only append each transition at most one time, and the hashtable lookup takes constant time. As shown in section 4, SMC as a caching implementation is extremely useful and could have also been used in the Artificial Ant problem.

3.3. Fitness Evaluation

To evaluate the fitness of a given state machine, Nark writes the Metal representation to a file, invokes the Metal compiler to build the checker, and runs the checker over a piece of test code supplied by the user. The test code is a C file that contains a set of functions of which some contain bugs and some do not. Figure 9 is an example test file for the simple interrupt checker.

```
int bug1() {
   cli();
   return 0;
}

int bug2() {
   int x;
   cli();

   if (x)
     return -1;

   sti();
}

int bug3() {
   cli();
   sti();
   cli();
   return 0;
}
```

```
int ok1() {
   cli();
   sti();
   return 0;
}

int ok2() {
   int x;

   cli();

   if (x) {
     sti();
     return -1;
   }
   sti();
   return 0;
}

int ok3() {
   int x;
   if (x)
     return 1;
   return 0;
}
```

Figure 9 An example test file for the interrupt checker. It contains buggy functions as well as non-buggy functions

```
bug1:2
bug2:10
bug3:19
```

Figure 10 The list of function:line tuples for the bugs in the example test file from Figure 9

```
ok1
ok2
ok3
```

Figure 11 The list of functions in the example test file from Figure 9 that do not contain errors

Points	Reason
5	An exactly correct report
2	No reports in non-buggy functions
1	A report in a buggy function but on the wrong line

Table 2 Point breakdown for fitness evaluation

In addition to the test file, the user also supplies a list of the (function, line number) pairs that are the locations of the bugs in the test file (Figure 10). Finally, the user supplies the list of functions that do not contain bugs at all (Figure 11).

Nark parses the output from running the checker over the test file and compares it to the two lists described above. It awards points for emitting an error on exactly the right line, emitting an error in the right function, and for not emitting an error in functions that don't contain errors. The point breakdown is in Table 2. The rationale for the scoring is that we would like to evolve a checker that emits valid errors but has relatively few false positives. The

"close" point award prevents the scoring system from being "all or nothing". Instead, it allows less than perfect checkers to score some points and encourages checkers to emit errors.

The fitness of the individual is then the sum of its points for the test file supplied by the user. The more points awarded, the more fit the individual. Obviously, if the test file is not representative of bugs in real code, the evolved checker may not perform as well as hoped. In that case, the user can insert additional functions in the test file, re-run Nark, and evolve a new checker.

3.4. Pattern Generation

In order for the checkers generated by Nark to be useful, they must be able to recognize source code constructs that are related to the bugs that the user desires to find. In the case of the interrupt checker, the patterns in Table 1 would be a good start. Nark must not have too many extraneous patterns or else the search space will explode, and Nark may not be able to find a suitable checker for the problem.

The obvious way to inform Nark of the patterns that relate to the problem at hand is to have the user supply them. The user provides a list of strings that are the Metal patterns themselves. When Nark needs a certain pattern, it can index into that list to get it.

While this method is straightforward, it means that the user must know about patterns—something Nark tries to keep from the user. Instead, Nark could look at the parse trees that come from the user's test file. It could then use the most common or useful (as defined by some heuristic) to build the list of patterns.

Unfortunately, Nark uses the first method to get the list of patterns. The second, automatic method is left for future work.

4. An Application of Nark

We used Nark to evolve a checker to find the class of interrupt bugs described above. Figures 9, 10, and 11 are the user-supplied files necessary for fitness evaluation. In addition to the list of patterns in Table 1 we included the spurious pattern `{ *ptr }` in the list of patterns available to Nark to show that GA can ignore unnecessary patterns. Table 3 summarizes the preparatory steps for the GA part of Nark. Nark used the Genesis (Grefenstette 1994) GA software package to generate the populations of chromosomes and perform the genetic operations on those chromosomes. Nark used the MC software from the Metacompilation research group at Stanford University as part of the fitness evaluation.

Objective:	Evolve a Metal state machine that is capable of finding real bugs of the type "re-enable interrupts after disabling them".
Representation scheme:	Structure = 25 integer genes The first gene represents the start state and has 8 possible values. The remaining 24 genes represent 8 transitions. Of the three genes in each transition, the first is one of 8 source states, the second is one of 4 patterns, and the third is one of 8 destination states. Section 3.1 thoroughly describes the mapping between chromosomes and state machines. The underlying representation used by the GA software is a binary string that maps to the 25 integers described above. L = 67 K = 2 { 0, 1 }
Fitness cases:	The single test file supplied by the user in Figure 9
Fitness:	The scoring scheme of Table 2 applied to the user's test file
Parameters:	Population size M = 500, Number of generations G = 20 Crossover rate: 60%, Mutation rate: 3%
Termination criteria:	The GA has run for G generations
Result designation:	The best-of-run individual using randomness to break ties

Table 3 Tableau for the interrupt checker

4.1. The Search Space

With eight possible source states, four possible patterns, and eight possible destination states, the number of possible transitions is (8 * 4 * 8) = 256. This run of Nark allows for eight transitions and eight possible start states, so the number of possible state machines is 8 * (8 * 4 * 8)^8 = 1.475E20.

Since Nark uses the standard GA operators of reproduction, crossover, and mutation, Nark could possibly generate each of the 1.475E20 chromosomes counted above.

Performing measurements on the schemata involved in the run of GA is straightforward when looking at the underlying binary representation used by Nark. The chromosome of 25 integers (17 with 8 possible values, and 8 with 4 possible values) is actually represented by a binary string of length 67. The integers with 8 possible values require 3 bits, and the integers with 4 possible values require 2 bits. L = 17 * 3 + 8 * 2 = 67. Therefore, there are (K+1)^L = 3^67 schemata involved in processing. These schemata are the standard one used in GA problems that use a binary string representation. With a population of M = 500, there are M * 2^L = 500 * 2^67 schema represented in each generation of the run.

4.2. Running Time

This run of Nark was performed on a Dual 1.1GHz Pentium III Linux computer. Evaluation of a single individual takes approximately six seconds. The linker takes the majority of this time as it statically links the checker into the MC system. For the purposes of this experiment, the static linking is unnecessary, but modifying the MC system to do dynamic linking proved to require too much re-engineering.

At six seconds an evaluation, the 10,000 individuals evaluated in this run should take a total of 16.66 hours. However, the SMC cache prevented evaluation of 7,318 individuals, resulting in a running time of only 4.5 hours.

4.3. Generated Individuals

Figure 12 is the canonicalized best-of-generation individual from generation 0. Its superiority over other members of the generation comes from the inclusion of the error state. Few of the other individuals of generation 0 were capable of emitting an error. This individual emits an error at the end of a path if it does not see a `sti()` along that path. This behavior is correct for most of the functions in the test file, but it does not handle `bug3()` which disables interrupts twice in the same function. It also incorrectly finds errors in functions that never disable interrupts (`ok3()`).

```
                                sm interrupt {
                                  state1:
                                   $end_of_path$ ==>
                                     error("Did not restore interrupts");
                                   | { sti() } ==> state2;
Canonicalized Chromosome:
  1 137 112 203 314 304           state2:
                                   { sti() } ==> state3;

                                  state3:
                                   $end_of_path$ ==> state1;
                                }
```

Figure 12 The best-of-generation individual from generation 0. Notice how state1 contains a transition to the error state.

There were a number of individuals that scored perfectly on the test file given. While all semantically identical, some included spurious transitions. These transitions were spurious because the subsumption rule of Metal prevented them from ever triggering. For example, if a state has two transitions that trigger a state change on the same input, only one can ever be triggered. Figure 13 is an example of a perfectly performing checker with a spurious transition.

5. Evaluation

Multiple runs of the GA in Nark produced similar results to those found in section 4.3. Some runs found an optimal checker as early as the seventh generation, while others did not evolve an optimal checker until the final generation. The values for G and M were chosen after several efforts with smaller values failed to consistently evolve perfectly fit checkers.

```
                                sm interrupt {
                                  state1:
                                   { cli() } ==> state2;
                                   | { sti() } ==> state2;

Canonicalized Chromosome:         state2:
  1 102 237 211 217                $end_of_path$ ==>
                                     error("Did not restore interrupts");
                                   | { sti() } ==> state1;
                                   | { sti() } ==>
                                     error("Did not restore interrupts");
                                }
```

Figure 13 A best-of-run individual that performs perfectly on the supplied fitness case. The third transition in state2 is spurious because the subsumption rule of Metal prevents it from triggering.

5.1. The (Almost) Perfect Checker

To validate the fitness of the checker in Figure 13, we used it to find bugs in the 2.4.23 Linux kernel. Running this checker on that version of the kernel resulted in 1,463 error reports! After a small amount of manual inspection of these reports, we noticed that the vast majority were false positives that arose because this checker does not recognize `restore_flags()` as being capable of enabling interrupts. This oversight is not the fault of Nark. In fact, it was the user who failed to provide proper training in the test file.

To remedy this spew of false positives (1,446 were caused by the `restore_flags()` problem), we added the two functions of Figure 14 to the test file in Figure 9 and updated the lists in Figures 10 and 11 appropriately. We replaced the useless `{ *ptr }` pattern from the list of patterns with `{ restore_flags(flags) }` and re-ran Nark with the same parameters as before.

```
int ok4() {
  int flags;
  cli();
  restore_flags(flags);
}

int bug4() {
  int flags;
  cli();
  restore_flags(flags);
  cli();
  return 0;
}
```

Figure 14 Extra test file functions to inform Nark of restore_flags()

```
/* 2.4.23/net/irda/iriap.c */
int irias_proc_read() {
 Start→
  cli();
  switch(attrib->value->type) {
    ...
  default:
   IRDA_DEBUG("Unknown value type!\n");
 Error→
   return -1;
  }
  sti();
}
```

Figure 15 A real bug found with the second evolved checker. Interrupts are not re-enabled along the error path in the switch statement.

This second run of Nark generated the desired checker after 18 generations. Running it on the same kernel resulted in 17 error reports. Manual inspection of these results yielded four real bugs and 13 false positives. Figure 16 is one of those bugs. The false positives fall into two groups: false paths caused by data dependencies and wrapper functions for `cli()`. Eliminating those two groups of false positives is not possible with the Metal facilities provided to Nark and is outside the scope of this paper.

We constructed a checker by hand to compare it to that generated by the second run of Nark. The two checkers were semantically identical, so the results of running them on the Linux kernel were the same.

5.2. Problems Encountered

During the process of building and testing Nark, we ran into a few major problems. The first problem was that the lists of bugs and ok functions in Figures 10 and 11 are not automatically linked to the test file in Figure 9. As a user, it is easy to omit an entry from either of those lists. In fact, a mistyped entry on the bug list led to dozens of wasted runs of Nark since it was impossible to achieve the maximum score. A better system would have the user annotate the lines of the test file that contain the bug. Nark could then infer which functions are bug free.

The second major difficulty was coming up with the points system in the evaluation function. At first we gave only three points for emitting an error on exactly the right line. Rewarding silence with two points resulted in the lack of incentive to emit errors, and Nark frequently found checkers with no transitions to the error state to be the most fit. A checker that never emits an error is useless, so we bumped up exact emissions to receive five points.

6. Conclusion

This paper has demonstrated how GA can be used to evolve Metal state machines that can find bugs in real systems software. For simple classes of bugs, it eliminates the need to learn Metal by providing an interface that is more familiar to the user. In comparison to other problems solved with GA, the evolution of these state machines takes an average population size and a small number of generations. Furthermore, state machine canonicalization resulted in a 75% cache hit rate for the interrupt checker evolved in section 4. This canonicalization concept could easily be used in a number of other GA applications to improve their cache performance.

The application of GA to static analysis is novel since only limited amounts of machine learning techniques have been used in static analysis to date. While the GA representation used by Nark is only applicable to static analysis techniques that use state machines, the idea of applying evolutionary programming to static analysis is worth pursuing further.

7. Future Work

The representation used by Nark to evolve state machines significantly limits the complexity of the classes of bugs that Nark is capable of evolving. First of all, very few checkers written by the Metacompilation research group use global states such as `enabled`. Instead, the rules often demand per-variable states where the state of the state machine is specific to a given variable. For example, a checker that looks for instances of the "don't use pointers after freeing their memory" bug would put each variable that is passed to `free()` into the `ptr.freed` state. Since multiple pointers can be freed within a single function, it is necessary to spawn a state machine for each of these variables. Evolving a checker with this type of capability with Nark would require mapping specific states to per-variable states. This would break SMC in its current form.

The second shortcoming of Nark is that it only uses the simple pattern matching facilities of Metal and ignores the powerful parse tree-querying interface. This interface is frequently used by the Metacompilation group to overcome inadequacies in Metal. However, the size of this interface would be overwhelming in the representation currently used by Nark. To use this interface, it might be better to try a genetic programming representation or to try to grow the state machine in a manner similar to growing hardware.

Finally, the running time of Nark needs to be reduced to be useful. With a turnaround time of 4.5 hours for a small checker, it seems that it would be much faster to write a more complex checker by hand. Even though the search space for the interrupt checker is very large, the set of checkers that are close to correct is relatively small. Perhaps growing state machines, as mentioned above, would be require the evaluation of fewer individuals and result in a smaller turnaround time.

Acknowledgements

I would like to thank Prof. John Koza for his advice in picking this project, the Metacompilation research group for letting me bounce ideas off of them, and Anthony Hui for proofreading this paper.

Bibliography

Chelf, Ben; Engler, Dawson; Hallem, Seth. 2002. How to Write System-specific, Static Checkers in Metal. In *Proceedings of Program Analysis for Software Tools and Engineering (PASTE)*.

Corman, Thomas; Leiserson, Charles; Rivest, Ronald. 1990. *Introduction to Algorithms*. Cambridge, MA: The MIT Press. Pages 485-487.

Engler, Dawson; Chelf, Ben; Chou, Andy; Hallem, Seth. 2000. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of Operating Systems Design and Implementation (OSDI)*.

Grefenstette, John J. 1994. *Genesis 5.0* ftp://ftp.aic.nrl.navy.mil/pub/galist/src/genesis.tar.Z.

Jefferson, David; Collins, Robert; Cooper, Claus; Dyer, Michael; Flowers, Margot; Korf, Richard; Taylor, Charles; Wang, Alan. 1991. *Evolution as a theme in artificial life: The genesys/tracker system*. In Langton, Christopher, et al. (editors), *Artificial Life II*. Addison-Wesley.

Musuvathi, Madanlal; Park, David Y.W; Chou, Andy; Engler, Dawson; Dill, David L. 2002. CMC: A pragmatic approach to model checking real code. In *Proceedings of Operating System Design and Implementation (OSDI)*.