Counterexample Guided Abstraction Refinement via Program Execution

Daniel Kroening, Alex Groce and Edmund Clarke [1*]

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA, 15213

Abstract. Software model checking tools based on a Counterexample Guided Abstraction Refinement (CEGAR) framework have attained considerable success in limited domains. However, scaling these approaches to larger programs with more complex data structures and initialization behavior has proven difficult. Explicit-state model checkers making use of states and operational semantics closely related to actual program execution have dealt with complex data types and semantic issues successfully, but do not deal as well with very large state spaces. This paper presents an approach to software model checking that actually *executes* the program in order to drive abstraction-refinement. The inputs required for the execution are derived from the abstract model. Driving the abstraction-refinement loop with a combination of constant-sized (and thus scalable) SAT-based simulation and actual program execution extends abstraction-based software model checking to a much wider array of programs than current tools can handle, in the case of programs containing errors.

1 Introduction

1.1 Software Verification using Predicate Abstraction

Software model checking has, in recent years, been applied successfully to real software programs, within certain restricted domains. Many of the tools that have been instrumental in this success have been based on the Counterexample Guided Abstraction Refinement (CEGAR) paradigm [Kur95,CGJ$^+$00], first used to model check software programs by Ball and Rajamani [BR00]. Their SLAM tool [BR01] has demonstrated the effectiveness of software verification for device drivers. BLAST [HJMS02] and MAGIC [CCG$^+$03] have been applied to security protocols and real-time operating system kernels.

* This research was sponsored by the Gigascale Systems Research Center (GSRC), the National Science Foundation (NSF) under grant no. CCR-9803774, the Office of Naval Research (ONR), the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, and by the Defense Advanced Research Projects Agency, and the Army Research Office (ARO) under contract no. DAAD19-01-1-0485, and the General Motors Collaborative Research Lab at CMU. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of GSRC, NSF, ONR, NRL, DOD, ARO, or the U.S. government.

A common feature of the success of these tools is that the programs and properties examined did not depend on complex data structures. The properties that have been successfully checked or refuted have relied on control flow and relatively simple integer variable relationships. For device drivers, and at least certain properties of some protocols and embedded software systems, this may be sufficient. However, even the presence of a complex static data structure can often render these tools ineffective. SLAM, BLAST, and MAGIC rely on theorem provers to perform the critical refinement step, and the logics used do not lend themselves to handling complex data structures, and may generally face difficulties scaling to very large programs. Explicit-state model checkers that (in a sense) actually *execute* a program, such as JPF [VHB$^+$03], Bogor [RRDH04], and CMC [MPC$^+$02], on the other hand, can handle complex data structures and operational semantics effectively, but do not scale well to proving properties over large state spaces, unless abstractions are introduced. The approach described and implemented in the CRunner tool combines the advantages of these approaches: an abstract model is produced and refined based on information obtained from actually executing the program being verified. The abstract model is used to provide inputs to drive execution and the results of execution are used to refine the abstract model. Although this does not reduce the difficulty of proving a program correct (the model must eventually be refined to remove all spurious errors), this method can be used to find errors in large programs that were not previously amenable to abstraction-refinement based model checking.

1.2 Counterexample Guided Abstraction Refinement

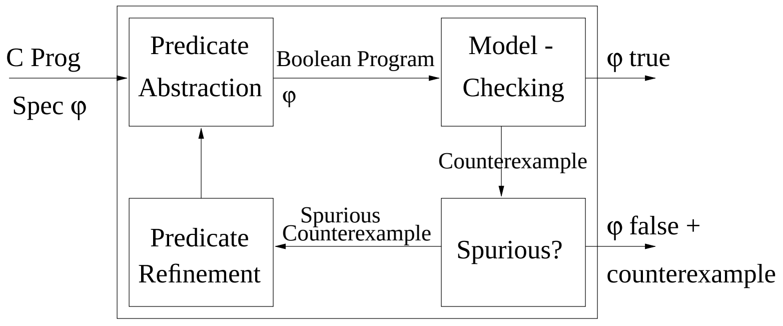

Fig. 1. The Counterexample Guided Abstraction Refinement Framework.

The traditional Counterexample Guided Abstraction Refinement framework (Figure 1) consists of four basic steps:

1. **Abstract:** Construct a (finite-state) abstraction $A(P)$ which safely abstracts P by construction.
2. **Verify:** Use a model checker to verify that $A(P) \models \varphi$: i.e., determine whether the abstracted program satisfies the specification of P. If so, P must also satisfy the specification.

3. **Check Counterexample:** If $A(P)\neg \models \varphi$, a counterexample C is produced. C may be *spurious*: not a valid execution of the concrete program P. SLAM, BLAST, and MAGIC generally use theorem prover calls and forward or back-wards propagation of weakest preconditions or strongest postconditions to determine if C is an actual behavior of P. If C is not spurious, P does not satisfy its specification.
4. **Refine:** If C is spurious, refine $A(P)$ in order to eliminate C, which represents behavior that does not agree with the actual program P. Return to step 1 [1].

1.3 Counterexample Guided Abstraction Refinement via Program Execution

The *Abstract*, *Verify*, *Check*, and *Refine* steps are also present in the execution-based adaption of the framework. However, the *Check* stage relies on program execution to refine the model. The basic approach can be seen as a combination of the typical Counterexample Guided Abstraction Refinement steps with a depth-first exploration of the program's execution behavior.

Figure 2 presents a high level view of the execution-based refinement loop.

Consider the simple example program shown in Figure 3. The first step is to recompile the program. Wherever the original program uses library calls to obtain input (whether from a socket, a file, or a user), a call to CRunner is inserted. In this case, the call to `getchar` will pass through to the model checker, which will provide the program with an input value.

The next step is to run the program. Execution will proceed normally until an input is required. For `example.c`, the first call to the model checker will occur when the while loop is entered. CRunnerwill model check an (initially very coarse) abstraction, starting from an initial state determined by the program state of the running program. In this case, the coarse abstraction will indicate that the assertion may be violated if any value other than `EOF` is returned by `getchar`. The use of conservative abstraction ensures that if no error trace is found in the abstract model, the program cannot reach an error from the current state. If, as in this initial state there are no states on the stack, this completes verification and guarantees that the program cannot violate any assertions.

On the other hand, as with `example.c`, a counterexample may exist in the abstract model. Because the abstraction contains no predicates, the model checker will determine that from the initial abstract state, the assertion can be violated in one step (the abstract model, recall, contains no information about the value of `i`).

CRunner uses Bounded Model Checking and a SAT solver [KCL04] to concretize the abstract error trace. From an initial state in which `i` is 100, any input other than `EOF` will cause an assertion violation. An arbitrary non-`EOF` input generated by the SAT solver is provided to the program, and execution continues. It is important to note that if `example.c` had included a guard at the beginning of the while loop checking that `i < 100`, the SAT solver would have been unable to produce an input violating the assertion, and the abstraction would have been refined to exclude the spurious execution. CRunner does not assume that the entire abstract counterexample can be unwound and concretized: it only assumes that the current input value can be concretized.

[1] This process may not terminate, as the problem is in general undecidable.

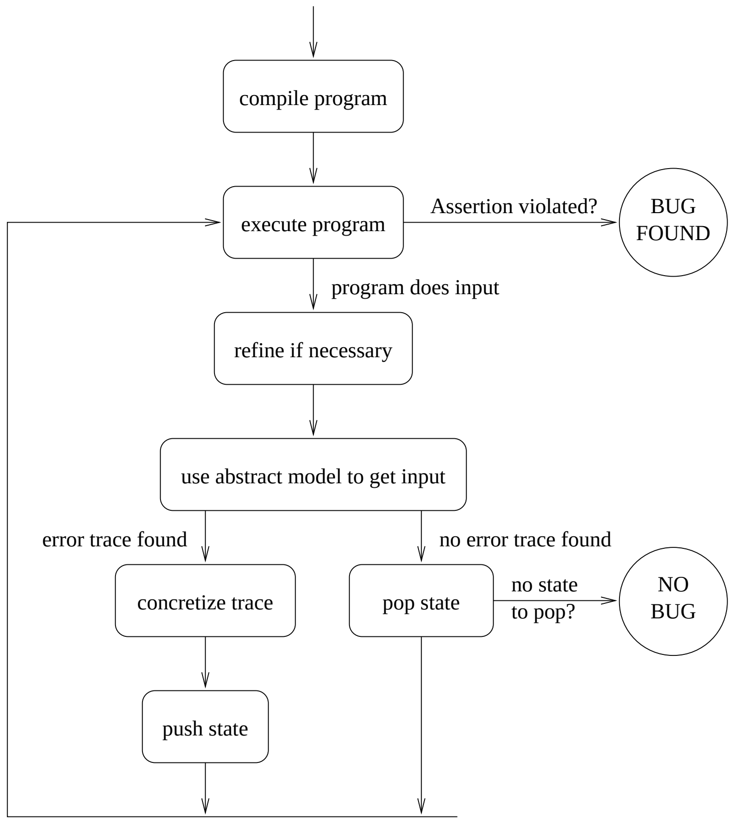

Fig. 2. Counterexample Guided Abstraction Refinement via Program Execution

```
int main() {
  char buffer[100];
  unsigned i=0;
  int ch;
  while((ch=getchar())!=EOF) {
    assert(i<100);
    buffer[i++]=ch;
  }
}
```

Fig. 3. example.c

If execution behavior deviates from the abstract counterexample, the model checker will backtrack.

The program again reaches a call for input. The initial abstract state remains unchanged, but the program state has changed, so there is no need to backtrack. The model checking and BMC results *can be reused*, as the abstraction and initial state remain unchanged. The loop will iterate 100 times, incrementing `i` until the assertion is violated. Previous Counterexample Guided Abstraction Refinement based model checkers would have been forced to generate predicates for the value of `i` in order to produce a non-spurious counterexample. CRunner is able to generate a counterexample without refining the model, and with only one call to the expensive model checking and concretization steps. The speed of model checking, in this case, should be roughly comparable to actual execution speed.

Outline In section 1.4, we discuss related work. In section 2, we show how the program that is to be verified is modified prior compilation. In section 3 we describe the algorithm and implementation details. In section 4, we provide experimental results. We conclude in section 5 and point out directions for future research.

1.4 Related Work

The verification approach presented in this paper is based on a Counterexample Guided Abstraction Refinement framework [CGJ$^+$00,Kur95] in which spurious counterexamples are detected and eliminated by a combination of Bounded Model Checking and program execution. Abstraction-refinement for software programs was introduced by Ball and Rajamani [BR00], and is a widely used approach for software model checking, represented by the well known SLAM, BLAST and MAGIC tools [BR01,HJMS02,CCG$^+$03]. These tools all rely on theorem provers to determine if an abstract counterexample *CE* represents an actual behavior of a program *P* [BR02,PR02,CCG$^+$04].

A second popular approach to software model checking is to rely on either actual execution of a program or on a model with similar operational semantics to actual execution. Tools following this method include VeriSoft [God97], JPF [VHB$^+$03], Bogor [RRDH04], and CMC [MPC$^+$02]. While some of these tools provide a degree of automated abstraction, reduction to Boolean Programs or LTS models is not an essential part of the model checking process.

This paper presents a meeting of these two approaches: a program is *executed* in order to drive the refinement of an abstract model of that program, and the inputs provided to the executing program are derived from the abstract model.

Predicate-complete test coverage [Bal03,Bal04] is conceptually related in that it combines predicate abstraction with a testing methodology. However, the final aims are fundamentally different: the coverage approach, as the name suggests, seeks to build a better test suite by using a measure of coverage. The method presented in this paper executes a program, but uses the information obtained to guide an abstraction refinement framework towards exhaustive verification or refutation of properties rather than to produce test cases.

2 Preparing the Program

The first step is to re-compile the program that is to be verified. Before compilation, we automatically make the following changes:

1. For each function, we add a variable. This variable is set to a constant prior to each call of the function. The constant is derived from the position of the call in program order. The information maintained in these variables rougly corresponds to the call stack, and allows us to distinguish the various instances of the functions at run-time without function inlining. As we do not permit recursion, one variable per function is sufficient.
2. Any calls to operating system input or output (I/O) functions are replaced by calls to CRunner. The CRunner code is linked together with the program that is to be verified. Examples of I/O functions are `printf`, `getc`, and `time`.

We replace the functions in the I/O library by prefixing the function name with `CRUNNER_`. As example, `printf` becomes `CRUNNER_printf`. For each I/O function within the I/O library, we have a replacement. If the function performs output, the output is simply discarded. If the function performs input, we call the model checker to obtain an input value. This is done using three functions declared as follows:

```
unsigned char crunner_get_byte();
int crunner_get_int();
_Bool crunner_get_bool();
```

We collectively refer to these functions as the `crunner_get_` functions.

As an example, consider the implementation that replaces `fgetc` in figure 4. It may either return an error, indicated by return value -1, or return a byte corresponding to data from an input stream. The first choice is made by a call to `crunner_get_bool`, while the data value is obtained from `crunner_get_byte`. This implementation can be extended in order to catch more program errors, e.g., it should assert that `stream` is actually a valid pointer to a `FILE` object.

Figure 5 shows our replacement for `fprintf`. The data that is to be written is simply ignored. The function `crunner_get_int` is used to obtain the return value. Again, more errors could be detected by checking the arguments passed to the functions using assertions.

```
int CRUNNER_fgetc(FILE *stream) {
  // EOF or not?
  if(crunner_get_bool()) return -1;
  return crunner_get_byte();
}
```

Fig. 4. Replacement for `fgetc`

```
int CRUNNER_fprintf(FILE *stream, const char *format, ...) {
  return crunner_get_int();
}
```

Fig. 5. Replacement for `fprintf`

3 Abstraction Refinement with Program Execution

3.1 Overview

CRunner is structurally similar to an explicit state model checker performing depth-first-search (DFS): it maintains a stack of states and performs backtracking after exhaustively exploring branches of the search tree. CRunner does not store all visited states on the stack – it is only necessary to store states just prior to input calls.

3.2 Execution with Trapping of Input

As described in the previous section, all calls to the I/O functions provided by the operating systems are replaced with versions that invoke the model checker.

The program is executed until I/O is performed or an assertion is violated. Note that we therefore are unable to detect infinite loops without input within the program.

Assertions include explicit assertions as given by the user within the code, and automatically generated assertions. Automatically generated assertions are used to detect errors such as dereferencing of `NULL` pointers, out of bounds indexing, or dereferencing of objects that have exceeded their lifetime.

When an assertion violation occurs, the process aborts with an error message. In contrast to existing tools that use a theorem prover or SAT solver to perform the simulation step, we are unable to produce a counterexample trace in this case. However, we output the sequence of inputs that results in the violated assertion.

If the program produces output, the output data is simply discarded and the program execution resumes normally. Note that file system function calls such as `remove` are also considered to be output functions.

It is when the program performs input that program execution is suspended, and the model checker is able to guide execution. CRunner uses the abstract model in order to produce an input value. This is described in the next section.

3.3 Generating the Abstract Model

Existential Abstraction We perform a predicate abstraction [GS97] of the ANSI-C program: i.e., the variables of the program are replaced by Boolean variables that correspond to a predicate over the original variables. As we are not using pushdown automata to verify the abstract program, we simply inline any function calls within the abstract program. Otherwise, the control flow structure of the program remains unchanged.

Formally, we assume that the algorithm maintains a set of n predicates $\pi_1, \ldots, \pi_n$. Let S denote the set of concrete states. The predicates are functions that map a concrete state $x \in S$ into a Boolean value. When applying all predicates to a specific concrete state, one obtains a vector of n Boolean values, which represents an abstract state $\hat{x}$. We denote this function by $\alpha(x)$. It maps a concrete state into an abstract state and is therefore called the *abstraction function*.

We perform an existential abstraction [CGL92], i.e., the abstract machine can make a transition from an abstract state $\hat{x}$ to $\hat{x}'$ iff there is a transition from x to x' in the concrete machine and x is abstracted to $\hat{x}$ and x' is abstracted to $\hat{x}'$. Let R denote the transition relation of the concrete program. We call the abstract machine $\hat{T}$, and we denote the transition relation of $\hat{T}$ by $\hat{R}$.

$$\begin{aligned} \hat{R} := \{\hat{x}, \hat{x}' \mid\ & \exists x, x' \in S : xRx' \wedge \\ & \alpha(x) = \hat{x} \wedge \alpha(x') = \hat{x}'\} \end{aligned} \tag{1}$$

Abstraction using SAT A SAT solver can be used to compute $\hat{R}$ [CKSY04]. The main idea is to form a SAT equation containing all the predicates, a basic block, and two symbolic variables for each predicate: one variable for the state before the execution of the basic block, and one variable for the state after its execution. The SAT solver is then used to obtain all satisfying assignments in terms of the the symbolic variables. The technique has also been applied to SpecC [JKC04], which is a concurrent version of ANSI-C.

One advantage of this technique is that it models all ANSI-C bit-vector operators precisely. In contrast, tools using theorem provers such as Simplify [DNS03] model the program variables as unbounded integers, and do not model the proper semantics for overflow on arithmetic operators. These tools typically treat bit-vector operators such as shifting and the bit-wise operators as uninterpreted functions.

However, the runtime of the SAT-based existential abstraction typically grows exponentially in the number of predicates. Most tools therefore do not compute a precise existential abstraction, but compute an over-approximation of the existential abstraction. One approach to over-approximation is to partition the set of predicates into subsets of limited size. Abstraction is then carried out for each of the subsets separately. The resulting transition relations are then conjuncted. Note that this over-approximation results in additional spurious behavior.

We use the set of variables mentioned in the predicates in order to group together related predicates. However, techniques for computing over-approximations of the existential abstraction are beyond the scope of this paper.

Prior to abstraction, any calls to one of the `crunner_get_` functions are simply replaced by unique free variables $\nu_1, \ldots, \nu_q$. The SAT solver performs an existential quantification for these variables, as done for any other program variable.

Verification of the Abstract Model using NuSMV The result of the SAT-based abstraction is a symbolic transition relation for each basic block. We build a NuSMV [CCG+02] model for the abstraction. Prior to building the abstract model, we transform the program into a guarded `goto` program, i.e., all control statements such as `if`, `while`, `for` are replaced by guarded gotos. The control flow is then encoded by means of a simple program counter (PC) construction. The transition relations for the basic blocks is directly given to NuSMV by means of `TRANS` statements.

Note that the initial program counter is not the location of the first instruction in the concrete program. Instead, we use the *location of the first input call as the initial PC*. This location can be determined at run-time using the values that are set for each function prior to the calls of the functions, as described above.

As CRunner over-approximates program behaviors, if the property holds on all states reachable from the input location within the abstract program, it also holds on all states reachable from the input location within the original, concrete program.

If NuSMV returns that no error trace was found, we proceed as follows: as the PC is not the initial program location, CRunner cannot conclude that there are no bugs in the program. We can only conclude that there are no bugs reachable from the input location. In this situation, we examine the DFS stack:

– If the stack contains a state, let s denote the state on top of the stack. Furthermore, let s' denote the current state of the program. We have exhaustively searched all paths originating from the state s'. Thus, we can remove the path from s to s' from the abstract model. After that, we backtrack to the state s, i.e., we restore the program state to the state s and then pop the state s off the stack. The algorithm proceeds with generating an input in state s.
– If the stack contains no more states, we are at the very first input location within the program, and we can conclude that the program has no bugs. The algorithm terminates.

If, however, NuSMV finds an error trace in the abstract model originating from the input location, we aim at concretizing parts of the abstract trace in order to extract an input value. This is described in the next section.

3.4 Concretizing the Abstract Trace

If the model checker finds an error trace in the abstract model, this does not imply that such a trace also exists in the concrete model. This is due to the fact that the abstract model is an over-approximation of the original program. An abstract trace without any corresponding trace in the concrete model is called a spurious trace.

Existing tools for predicate abstraction of C programs build a query for a theorem prover by following the control flow given by the abstract error trace. If the query is satisfiable, a concrete error trace exists. The data values assigned along the trace and input values read along the trace can be extracted from the satisfying assignment. This is usually called simulation of the abstract trace on the concrete program, and is implemented as described above by SLAM, BLAST, and MAGIC. Incremental SAT has also been used to perform the simulation step [CKSY04], but the principle remains the same.

In large programs, in particular in the presence of dynamic data structures, error traces may easily have a thousand or more steps. The simulation of these long abstract traces quickly becomes infeasible as the program size and complexity increases. Thus, in contrast to the existing tools, we do not aim at simulating the abstract trace by means of a theorem prover or SAT solver.

Instead, we want to continue the execution of the program after the input location. We aim at returning an input value to the program that will guide the program along the abstract trace to the error location found within the abstract model.

We obtain this input value as follows: we build a simulation query similar to existing tools. However, we *limit the depth of the query* to a few steps. This should curb the computational effort required for the query, but may still provide sufficient information to compute the next input value. This is motivated by the fact that programs often perform control flow decisions based on the input values they read within a few instructions after the input is performed.

Partial Simulation using SAT Let the counterexample trace have k steps, and let $k' \leq k$ be the depth (number of steps) we use to obtain the input value. The simulation requires a total of k' SAT instances. Each instance adds constraints for one more step of the abstract counterexample trace. We denote the value of the (concrete) variable v after step i by v_i. All the variables v inside an arbitrary expression e are renamed to v_i using the function $\rho_i(e)$.

The SAT instance number i is denoted by Σ_i and is built inductively as follows: Σ_0 (for the empty trace) is defined to be true. For $i \geq 1$, Σ_i depends on the type of statement of state i in the counterexample trace. Let p_i denote the statement executed in the step i. As described above, we use guarded goto statements to encode the control flow.

Thus, if step i is a guarded goto statement, then the (concrete) guard g of the goto statement is renamed and used as conjunct in Σ_i. Furthermore, Σ_{i-1} is added as a conjunct in order to constrain the values of the variables to be equal to the previous values:

$$p_i = (\mathtt{goto}, g, l) \longrightarrow \Sigma_i := \Sigma_{i-1} \wedge \rho_i(g) \wedge \bigwedge_{u \in V} u_i = u_{i-1}$$

If step i is an assignment statement, the equality for the assignment statement is renamed and used as conjunct:

$$\begin{aligned} p_i = (\mathtt{v\!:=\!exp}) \longrightarrow \Sigma_i :=\ & \Sigma_{i-1} \wedge \\ & \rho_i(v) = \rho_{i-1}(exp) \wedge \\ & \bigwedge_{u \in V \setminus \{v\}} u_i = u_{i-1} \end{aligned}$$

As done during abstraction, any calls to one of the `crunner_get_`functions within the right hand side are simply replaced by unique free variables $\nu_1, \ldots, \nu_u$.

Note that in case of assignment statements, Σ_i is satisfiable if the previous instance Σ_{i-1} is satisfiable. Thus, the check only has to be performed if the last statement is a guarded goto statement. If the last instance $\Sigma_{k'}$ is satisfiable, the partial simulation is successful.

In this case, the SAT solver provides us with a satisfying assignment, which contains values for all variables in $\Sigma_{k'}$. This includes, in particular, a value for the first input ν_1. We simply use this value as return value of the `crunner_get_`function call, and return control to the program that is to be verified. Prior to returning control, we save the input value, the state, and the abstract trace we expect to see on the concrete program on the DFS stack.

If the partial simulation fails, the abstract counterexample is spurious, and the abstract model must be refined, as described in the next section. After refinement, we attempt to find another abstract error trace starting from the same concrete state.

3.5 Refining the Abstract Model

We have two ways to detect spurious behavior in the abstact model: first, as in the traditional refinement loop, we perform a simulation. While this simulation is bounded, it still allows us to detect some spurious behavior.

The second way to detect spurious behavior is during execution: If the executed trace diverges from the expected abstract trace, we check if the abstract trace is spurious.

Following a distinction introduce in the context of hardware verification [CTW03], we distinguish two potential sources of spurious behavior in the abstract model:

1. The abstract counterexample may be found to be spurious because we are not performing a precise existential abstraction. Instead, we partition the predicates, which may result in *spurious transitions* in the abstract model.
2. The abstract counterexample may be spurious because of the abstraction is based on too few predicates. This is referred to as a *spurious prefix* [CTW03].

SLAM uses the following approach to distinguish these two cases [BCDR04]: first, SLAM assumes that the spurious counterexample is caused by a lack of predicates and attempts to compute new predicates using weakest preconditions of the last guard in the query. If new predicates are added, the refinement loop continues as usual. However, if the refinement process fails to add new predicates, a separate refinement procedure, called *Constrain* is invoked.

Following Wang, et al., [CTW03], we first check whether any transition in the abstract trace is spurious. If so, we refine the abstract model. The conflict graph is analyzed in order to eliminate multiple spurious transitions with one SAT solver call [CTW03]. This technique is also applicable to software. However, the refinement of spurious transitions is beyond the scope of this paper.

If no transitions are spurious, the spurious counterexample must be caused by a lack of predicates. In this case, we compute new predicates by means of weakest preconditions in a similar fashion to the various existing predicate abstraction tools.

4 Experimental Results

We implemented a prototype implementation of the algorithm described in the previous section and report experimental results on a number of ANSI-C programs. The experiments are performed on a 1.5 GHZ AMD machine with 3 GB of memory running Linux.

We first investigate a scalable, artificial example with a buffer overflow after *n* bytes of input from a file. Existing tools usually require an abstract trace of at least *n* steps in order to find such a bug. Furthermore, they use a theorem prover or SAT solver to concretize such an abstract trace. The run-time of these tools is typically exponential in *n*.

Table 1 contains the run-times for the artificial benchmark for various increasing values of *n* and the run-time of a conventional implementation using SMV and SAT. Note that for this example, most of the results in the loop can be cached, and thus, are only performed once. Even for very large *n*, the run-time is completely dominated by the compilation.

On the other hand, the conventional implementation degrades very quickly, as it has to refine *n* times, adding a single new predicate for the array index every time. However, it provides a full counterexample trace.

We also report the time to prove a correct version, which guards agains the buffer overflow. The conventional refinement loop is faster as no compilation is needed. However, the run-time (and almost all of the behavior) is the same after the execution starts. It does not depend on *n*.

	n						
Method	10	50	100	1000	10,000	100,000	no bug
Execution	1.5s	1.5s	1.5s	1.5s	1.5s	1.5s	1.5s
Conventional	41.4s	700s	*	*	*	*	0.01s

Table 1. Comparison of proposed algorithm and conventional implementation on artificial example with a buffer overflow after *n* bytes of input from a file. The run-times include the compilation time for the execution method. A star * denotes a time-out.

We also experimented with software obtainable from the internet. Spamassassin is a tool for filtering email messages. Most of it is written in Perl, but it has a frontend written in ANSI-C for efficiency reasons. Version 2.43 contains a (previously known) off-by-one error in the BSMTP interface. Figure 6 shows the relevant parts of the code.

The buffer overflow is triggered due to the special treatment of the dot in BSMTPD. Due to the large size of the buffer (1024), a long input stream is required to trigger the bug. Our conventional predicate refinement loop could not detect this overflow error within reasonable time. The execution-based implementation only required 3 seconds (most of which are spent in compilation) to detect the error and to produce an input stream that triggers it.

We also experimented with `sendmail`, a commonly used mail gateway for Unix machines. We were able to reproduce previously known errors that are triggered by specially crafted email-messages. As an example, due to a faulty type conversion, the ASCII character 255 was used to exploit older versions of `sendmail`. The execution-based approach generates the necessary input sequence to trigger the bug, while the conventional implementation was unable to find it due to the required length of the traces.

```
char buffer[1024];
[...]
switch(m->type){
  [...]
  case MESSAGE_BSMTP:
    total = full_write(fd, m->pre, m->pre_len);
    for(i = 0; i < m->out_len; ) {
        jlimit = (off_t) (sizeof(buffer) /
                 sizeof(*buffer) - 4);
        for(j = 0; i < (off_t) m->out_len && j < jlimit; ) {
            if(i + 1 < m->out_len && m->out[i] == '\n' &&
               m->out[i+1] == '.') {
                if(j > jlimit - 4)
                    break;  /* avoid overflow */
                buffer[j++] = m->out[i++];
                buffer[j++] = m->out[i++];
                buffer[j++] = '.';
            } else {
                buffer[j++] = m->out[i++];
[...]
```

Fig. 6. Code from `spamc`

5 Conclusions and Future Work

We present a variant of the counterexample guided predicate abstraction framework introduced by Ball and Rajamani [BR00]. Explicit-state model checkers making use of states and operational semantics closely related to actual program execution have dealt with complex data types and semantic issues successfully, but do not deal as well with very large state spaces. We therefore combine techniques from abstraction refinement and explicit state modelchecking: In order to prove the property correct, we use the abstraction refinement. In order to disprove the property, we actually execute the program.

Experimental results indicate that no advantage over the existing approaches are obtained if proving correctness is the goal. However, the execution-based simulation approach allows to find bugs that have very deep error traces. The existing tools typically use theorem provers or SAT for the simulation of the error traces, which have exponential run-time in the length of the error trace.

References

[Bal03] T. Ball. Abstraction-guided test generation: A case study. Technical Report 2003-86, Microsoft Research, November 2003.

[Bal04] T. Ball. A theory of predicate-complete test coverage and generation. Technical Report 2004-28, Microsoft Research, April 2004.

[BCDR04] Thomas Ball, Byron Cook, Satyaki Das, and Sriram Rajamani. Refining approximations in software predicate abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 388–403. Springer-Verlag, 2004.

[BR00] T. Ball and S.K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report 2000-14, Microsoft Research, February 2000.

[BR01] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop on Model Checking of Software*, pages 103–122, 2001.

[BR02] T. Ball and S.K. Rajamani. Generating abstract explanations of spurious counterexamples in C programs analysis. Technical Report 2002-09, Microsoft Research, January 2002.

[CCG$^+$02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource tool for symbolic model checking. In *Computer Aided Verification*, pages 359–364, 2002.

[CCG$^+$03] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *International Conference on Software Engineering*, pages 385–395, 2003.

[CCG$^+$04] S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient verification of sequential and concurrent C programs. *Formal Methods in System Design*, 2004. To appear.

[CGJ$^+$00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[CGL92] E. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. In *POPL*, January 1992.

[CKSY04] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design*, 2004. To appear.

[CTW03] Edmund Clarke, Muralidhar Talupur, and Dong Wang. SAT based predicate abstraction for hardware verification. In *Proceedings of SAT'03*, May 2003.

[DNS03] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, July 2003.

[God97] P. Godefroid. VeriSoft: a tool for the automatic analysis of concurrent reactive software. In *Computer Aided Verification*, pages 172–186, 1997.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th INternational Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.

[HJMS02] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Principles of Programming Languages*, pages 58–70, 2002.

[JKC04] H. Jain, D. Kroening, and E.M. Clarke. Verification of SpecC using predicate abstraction. In *MEMOCODE 2004*. IEEE, 2004.

[KCL04] D. Kroening, E. Clarke, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004.

[Kur95] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata- Theoretic Approach*. Princeton University Press, 1995.

[MPC$^+$02] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: a pragmatic approach to model checking real code. In *Symposium on Operating System Design and Implementation*, 2002.

[PR02] T. Ball A. Podelski and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 158–172, 2002.

[RRDH04] Robby, E. Rodriguez, M. Dwyer, and J. Hatcliff. Checking strong specifications using an extensible software model checking framework. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 404–420, 2004.

[VHB$^+$03] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.