Counterexample Guided Abstraction Refinement Via Program Execution

Daniel Kroening, Alex Groce, and Edmund Clarke*

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) Boolean satisfiability-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. Experimental results from applying the CRunner tool, which implements execution-based refinement, to faulty and correct C programs demonstrate the practical utility of the idea.

1 Introduction

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 most notable as contributing to this success have been based on the Counterexample Guided Abstraction Refinement (CEGAR) paradigm [20, 11], first used to model check software programs by Ball and Rajamani [4]. Their SLAM tool [5] has demonstrated the effectiveness of software verification for device drivers. BLAST [18] and MAGIC [8], making use of similar techniques, 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.

J. Davies et al. (Eds.): ICFEM 2004, LNCS 3308, pp. 224–238, 2004.

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: 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 [24], Bogor [23], and CMC [21], 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.

This paper describes a variation on the traditional counterexample guided abstraction-refinement method and its implementation in the tool CRunner. Our approach combines the advantages of the abstraction-based and execution-based approaches: an abstract model is produced and refined based on information obtained from *actually running* 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. Section 1.3 describes more precisely where CRunner fits in the larger context of software model checkers.

1.1 Counterexample Guided Abstraction Refinement

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.

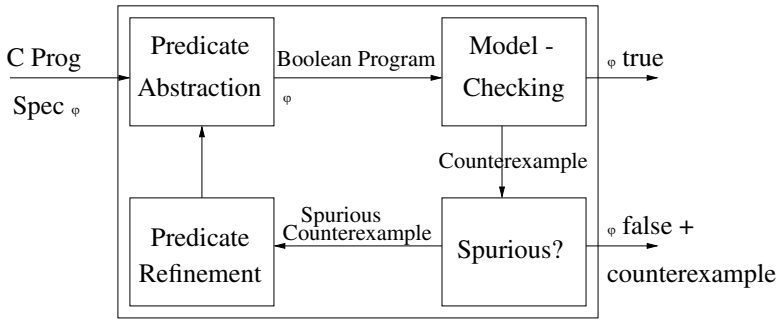

Fig. 1. The Counterexample Guided Abstraction Refinement Framework

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 $A(P) \not\models \varphi$, 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 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].

1.2 Counterexample Guided Abstraction Refinement Via Program Execution

The *Abstract*, *Verify*, *Check*, and *Refine* steps are also present in our execution-based adaptation of the framework. However, the *Check* stage relies on program execution to refine the model. The method presented in this paper can be seen as embedding these steps in a depth-first search algorithm more typical of explicit-state model checkers.

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

Consider the simple example program shown in Figure 3. In order to model check this program, the first step in the execution based approach is to *compile* the program — a marked difference from most abstraction-based tools, which rely only on the source text. The `compile program` step in the diagram refers to a modified compilation, in that all calls to library routines (here, `getchar`) are replaced by calls to the model checker (this process is described in detail in Section 2). Where the original program would receive input from standard I/O, the recompiled program will receive input from the CRunner tool. The only restrictions on programs to be checked are that (1) all potential non-determinism in the program has been replaced with calls to CRunner, (2) no execution path enters an infinite loop without obtaining input, and (3) the program does not make use of recursion. If these conditions are violated, it is possible to miss errors. For the class of programs we have investigated, these restrictions are easily met; in particular, utilities written in C seldom make use of recursion. It is always, as with any abstraction-based scheme, possible that a correct program cannot be verified due to undecidability.

After compilation, the next step is `execute program`. The CRunner verification process is initiated by running the new compiled executable of the program to be checked. Execution will proceed normally until the call to `getchar` on line 4 is reached. At this point, we proceed to `program does input` in Figure 2. Had an assertion been violated before the input call was reached, execution would have terminated, reporting the sequence of inputs that caused the error.

Flow proceeds to `refine if necessary`. Refinement is only required if execution has diverged from the behavior of an abstract counterexample. In this case, however, no model checking has yet been performed, so refinment is unnecessary. The details of when refinement is required are presented in Section 3.

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

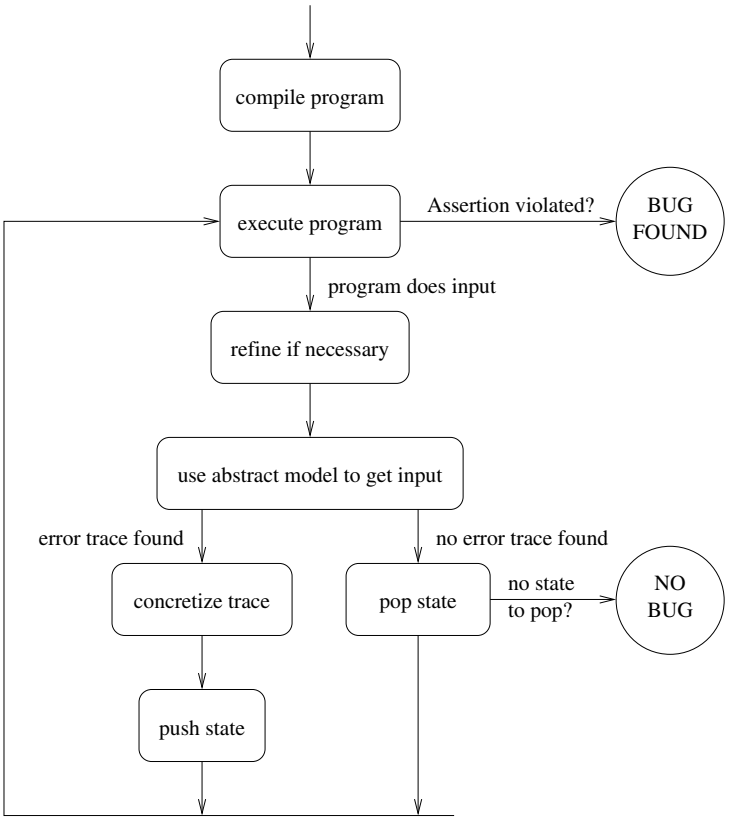

Fig. 2. Counterexample Guided Abstraction Refinement via Program Execution

The next step is `use abstract model to get input`. An initially coarse abstract model of the program is generated and model checked. The initial state for the abstract model is not the first line of the `main`, but the point at which the library call to obtain input appears. Note that in the case of programs with pointers or complex data structures, this can be a significant advantage for initial alias analysis.

Because the initial abstraction contains no information about the value of `i`, the model checker will report a counterexample in which the while condition is satisfied and the assertion on line 6 is violated.

Had no counterexample been produced, the algorithm would have proceeded to the `pop state` step. No states would have been on the stack, so verification would have terminated, proving the program error free. Changing line 5 to read:

```
5    while((ch != EOF) && (i < 100)) {
```

would produce this result.

Concretization of the abstract counterexample (`concretize trace`) shows that the assertion can be violated from the initial state of the abstract model if any input other than EOF is produced by the `getchar` call (and `i` is 100 or greater). Recall that the abstract state contains no information about the value of `i`.

```
1  int main() {
2    char buffer[100];
3    unsigned i = 0;
4    int ch = getchar();
5    while(ch != EOF) {
6      assert(i<100);
7      buffer[i]=ch;
8      i++;
9      ch = getchar();
10   }
11 }
```

Fig. 3. example.c

The current (initial) concrete program state is pushed on the stack, and control returns to `example.c`, with an arbitrary non-EOF value assigned to `ch`.

Execution of the program continues to line 9, with the assertion not violated. When execution reaches line 9, another input call is made. The program state does not match the initial state of the abstraction (the program counter is different), so a new abstraction (with a new initial state) is generated. Again, a counterexample is produced, and concretization indicates that any non-EOF input can violate the assertion. The algorithm passes through the same steps as before after the input call.

Execution reaches line 9 again, with another call for input. This time, when the `use abstract model to get input` step is reached, CRunner will make use of the fact that the program state matches the initial state of the current abstract model, and has not deviated from the model's behavior. The cached model checking results can be *reused without building a new abstraction or running the model checker again*. A non-EOF input is provided for the `getchar` call again, and the same loop is traversed. This will occur until `i` has reached a value of 100, and an error is detected. CRunner will then provide the sequence of inputs that produces the error, in lieu of a counterexample.

CRunner detects a buffer overflow in this program with only two calls to the expensive abstraction, model checking, and simulation algorithms, and *without adding any predicates to the abstract model*. Other abstraction based model checkers would require a series of predicates describing the value of `i` in order to produce a non-spurious counterexample. The speed of CRunner in this case is comparable to actual execution speed. Model checkers requiring the addition of new predicates on `i` would require many expensive refinement steps. It is likely that either memory or user patience would be exhausted before termination. Section 4 compares CRunner with an abstraction-refinement scheme without execution, supporting this claim, and presents experimental results for real-world examples.

1.3 Related Work

The verification approach presented in this paper is based on a Counterexample Guided Abstraction Refinement framework [11, 20] 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 [4], and is used by the well known SLAM, BLAST and MAGIC tools [5, 18, 8]. These tools all rely on theorem provers to determine if an abstract counterexample CE represents an actual behavior of a program P [6, 22, 9].

A second popular approach to software model checking is to rely on either actual execution of a program or a model with an execution-like operational semantics. Tools in this category include VeriSoft [16], JPF [24], Bogor [23], and CMC [21]. While some of these tools allow some kind of automated abstraction, it is not an essential part of their 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 [1, 2] 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 BLAST model checker has also been extended to produce tests suites providing full coverage with respect to a given predicate [7]. 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 recompile the program that is to be verified. As a pre-processing step, certain changes are made to the source before compilation:

1. For each function, a variable is added. This variable is set to a unique 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 roughly corresponds to the call stack, and allows CRunner to distinguish the various instances of the functions at runtime 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 with the program that is to be verified. Examples of I/O functions are `printf`, `getc`, and `time`.

Functions in the I/O library are replaced by prefixing the function name with `CRUNNER_`. For example, `printf` becomes `CRUNNER_printf`. For each I/O function within the I/O library, CRunner provides a replacement. If a function performs output, the output is discarded. If a function performs input, the model checker is called to obtain a value. To be precise, each nondeterministic choice to be made is replaced with a call to the model checker[2]:

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

[2] `_Bool` is the C99 standard Boolean type.

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

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`. The CRunner version of `fgetc` can be extended in order to catch more program errors — e.g. to assert that `stream` is actually a valid pointer to a `FILE` object.

Figure 5 shows the replacement for `fprintf`. The data that is to be written is discarded. The function `crunner_get_int` is used to obtain a return value. Again, more errors could be detected by checking the arguments passed to the function 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 prior to input library calls.

3.2 Execution with Trapping of Input

As described in the previous section, all calls to system I/O library routines are replaced with versions that invoke CRunner.

The program is executed precisely as usual until I/O is performed or an assertion is violated (as shown in Figure 2). The restriction against infinite loops without input is necessary as CRunner cannot detect and abort such behavior.

Assertions include explicit assertions included within the program being verified, and automatically generated assertions introduced by the CRunner pre-processing step.

Automatically generated assertions are used to detect errors such as dereferencing of `NULL` pointers, out of bounds array indexing, and dereferencing of objects that have exceeded their lifetime.

When an assertion violation occurs, CRunner reports the error condition and terminates. In contrast to most other software model checkers, CRunner does not produce a complete counterexample trace. Rather, it outputs the sequence of inputs necessary to produce the error. If all nondeterminism has been trapped by CRunner, this will be sufficient to reproduce the error, but will lack the detail of a model checking counterexample. In this sense, CRunner is more akin to exhaustive testing than to more traditional model checking.

If the program produces output, the output data is simply discarded. There is no need to invoke the model checker in these cases, unless the output function also returns a value that is used. File system function calls such as `remove` are treated in a similar manner.

When the program requires an input, execution is suspended and the model checker is used to guide execution. CRunner uses the abstract model in order to provide an input value to the running program. The remainder of this section describes the generation and use of the abstract model.

3.3 Generating the Abstract Model

Existential Abstraction. CRunner performs a predicate abstraction [17] of the ANSI-C program: i.e., the variables of the program are replaced by Boolean variables that correspond to predicates over the original variables. The control flow of the program remains essentially unaltered — because CRunner does not use pushdown automata to represent control flow, an inlining step (at the abstract model level, rather than at the executing program level: see Section 2) is necessary.

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. Each predicate is a function that maps concrete states $x \in S$ into Boolean values. 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 [12], 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, x is abstracted to $\hat{x}$, and x' is abstracted to $\hat{x}'$. Let R denote the transition relation of the concrete program. We denote the transition relation of the abstract program by $\hat{R}$:

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

Abstraction Using SAT. A Boolean satisfiability (SAT) solver can be used to compute $\hat{R}$ [13]. The computation relies on 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

equation constrains the before and after values so that the second value is the result of applying the operations of the basic block to the original. When CRunnerproduces this equation, any calls to the `crunner_get_` functions are replaced by unique free variables $v_1, \ldots, v_q$. A SAT solver is used to obtain all satisfying assignments in terms of the the symbolic variables, which produces a precise abstract transition relation for the code in the basic block. This technique has also been applied to SpecC [19], 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 [15] 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 execution time of the SAT-based existential abstraction typically grows exponentially with 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 conjoined. Note that this over-approximation results in additional spurious behavior. In particular, we use the set of variables mentioned in each predicate in order to group together related predicates. However, techniques for computing over-approximations of the existential abstraction are beyond the scope of this paper.

Verification of the Abstract Model. The result of the SAT-based abstraction is a symbolic transition relation for each basic block of the program. The program is transformed into a guarded `goto` program, in which all control constructs (e.g. `if`, `while`, and `for` statements) are replaced by guarded `goto` commands. The basic block transitions plus a program counter (PC) are combined with this control flow representation to produce a transition relation for the entire program.

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 current input call as the initial PC*. This location can be determined at runtime using the values that are set for each function prior to function calls, as described above. By reading pointer values from memory, a very precise abstraction of this initial state can be efficiently computed.

We use the NuSMV [10] symbolic model checker to perform the actual verification. NuSMV will either verify that the abstract model does not contain any errors, or will produce an abstract counterexample (which may or may not be a possible behavior of the original program).

Because the abstraction used by CRunner is an existential over-approximation of all program behaviors, if the model checker determines that the a property violation cannot be reached from the input location, this result is reliable: from the current initial state, the original program cannot reach an error state.

If NuSMV determines that the property does hold from the current initial state, CRunner cannot conclude that the program is error free, but only that no error is reachable

from the current initial state. Other states may remain to be explored. CRunner examines the stack, just as an explicit-state model checker would after finishing with a branch of the search tree:

- If the stack contains a state, let s denote the state on top of the stack, and s' denote the current state of the program (from which model checking has just completed). CRunner has exhaustively searched all paths originating from the state s'. The path from s to s' is removed from the abstract model in order to avoid re-opening this portion of the search tree. CRunner backtracks to state s, popping the search stack and restoring the program state from s. The process repeats from the input request in s.
- If the stack contains no more states, all possible paths have been explored, and CRunner reports that the program is error free.

On the other hand, if NuSMV does discover a counterexample, CRunner must attempt to concretize this abstract error trace in order to generate actual program inputs.

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 of the abstract error trace. If this 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 [13], but the underlying principle is unchanged.

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. This fundamental issue of scalability motivates our efforts to avoid attempting to simulate the entire abstract counterexample using a theorem prover or SAT solver.

CRunner, in place of this large simulation step, attempt to continue executing the program at the current input location. The goal is to find an input value that guides the executing program along the abstract trace to the error location found in the abstract model.

CRunner produces this input value by building a simulation query, in a similar manner to existing tools. However, CRunner *limits the depth of the query* to a few steps. This should reduce the computational effort required in order to compute the simulation, but provide sufficient information to compute the next input value. This works in practice because programs often perform control flow decisions based on the input values immediately after obtaining the input.

Partial Simulation Using SAT. Let the counterexample trace have k steps, and let $k' \leq k$ be the depth (number of steps) used to obtain the input value. Simulation requires

a total of k' SAT instances. Each instance adds constraints for one more step of the abstract counterexample trace. Let V denote the set of concrete program variables. We denote the value of the (concrete) variable $v \in V$ 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 in state i in the counterexample trace. Let p_i denote the statement executed in the step i. As described above, guarded `goto` statements are used 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 a 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:

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

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

$$p_i = (\texttt{v:=exp}) \longrightarrow \Sigma_i := \Sigma_{i-1} \wedge \rho_i(v) = \rho_{i-1}(exp) \wedge \bigwedge_{u \in V \setminus \{v\}} u_i = u_{i-1}$$

As in the abstraction phase, any calls to the `crunner_get_` functions on a 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. A SAT check is only necessary when the last step 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 CRunner with a satisfying assignment containing values for all variables in $\Sigma_{k'}$. This includes, in particular, a value for the first input ν_1. CRunner uses this value as the return value of the `crunner_get_` function call, and returns control to the program being verified. Prior to returning control, CRunner saves the input value, the state, and the abstract trace 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, CRunner attempts to find another abstract error trace starting from the same concrete state.

3.5 Refining the Abstract Model

There are two ways to detect spurious behavior in the abstract model: first, as in the traditional refinement loop, the simulation step may fail.

The second way to detect spurious behavior is during execution: if the executed trace diverges from the expected abstract trace, CRunner checks to determine if the abstract trace is spurious.

Following a distinction introduced in the context of hardware verification [14], we distinguish two potential sources of spurious behavior in the abstract model:

1. The abstract counterexample may be spurious because we are not performing a precise existential abstraction. Partitioning the predicates may result in *spurious transitions* in the abstract model.
2. The abstract counterexample may be spurious because the abstraction is based on an insufficient set of predicates. This is referred to as a *spurious prefix* [14].

Microsoft's SLAM model checker uses the following approach to distinguish these two cases [3]: 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 in order to refine the approximation of the abstract transition relation.

CRunner, in contrast, following Wang, et al. [14] first checks whether any transition in the abstract trace is spurious. If so, CRunner refines the abstract model. The conflict graph from simulation is analyzed in order to eliminate multiple spurious transitions with one SAT solver call [14]. The details of this refinement process, as applied to software, are 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, CRunner computes new predicates by means of weakest preconditions in a similar fashion to the various existing predicate abstraction tools.

4 Experimental Results

We applied out prototype CRunner tool to a number of ANSI-C programs. All experiments were performed on a 1.5 GhZ AMD machine with 3 GB of RAM, running Linux.

We first investigated a scalable, artificial example in which a buffer overflow occurs 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 an error. Furthermore, they rely on using a theorem prover or SAT solver to concretize a large abstract trace. The execution time of these tools is typically exponential in n.

Table 1 contains execution times for the artificial benchmark for various increasing values of n. Times are presented for CRunner and for a conventional implementation using the SMV model checker and a SAT-based abstraction-refinement. Note that for this example, most of the results used by CRunner can be cached, avoiding multiple expensive model checking or simulation runs. Even for very large n, the execution time is completely dominated by the compilation.

On the other hand, the performance of the conventional implementation degrades very quickly. It must refine the abstraction n times, adding a single new predicate for the array index in each step. The final result is a complete counterexample trace, rather than a set of inputs, but the cost for this precision is very high.

We also report the time to verify a correct version of the code in which a guard is added to prevent the buffer overflow. The conventional refinement loop is faster in this case, as no compilation is needed. However, after compilation, CRunner is comparable. For correct code, neither approach depends on the value of n.

Table 1. Comparison of CRunner prototype and conventional abstraction-refinement on an artificial example with a buffer overflow after n bytes of input from a file. The execution times include the compilation time for CRunner. A star * denotes a time-out

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

We also experimented with more realistic open source examples. Spamassassin is a tool for filtering email messages. Most of the system is written in Perl, but the front-end is coded 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 BSMTP. Due to the large size of the buffer (1024), a long input stream is required to trigger the bug. The conventional predicate refinement loop could not detect this overflow error in a reasonable amount of time. CRunner 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. For example, due to a faulty type conversion, the ASCII character 255 was used to exploit older versions of `sendmail`. CRunner was able to generate the necessary input sequence to trigger this 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

This paper presents a variation of the counterexample guided predicate abstraction framework introduced for software by Ball and Rajamani [4]. Abstraction-refinement based model checkers have traditionally dealt poorly with complex data types and lengthy counterexamples. 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 well with very large state spaces. We therefore combine techniques from abstraction-refinement and explicit state model checking: exploration, meant to discover errors, is based on actual program execution, guided by abstraction that can prove the program correct, or prune the search space.

Experimental results indicate that no advantage over the existing methods in the case of correct programs, but demonstrate the power of our approach for finding errors.

Extending this work to a larger body of operating system libraries and allowing for some form of message-passing concurrency are topics for future research. We would also like to investigate whether predicate selection can be improved by using information from the program execution.

References

1. T. Ball. Abstraction-guided test generation: A case study. Technical Report 2003-86, Microsoft Research, November 2003.
2. T. Ball. A theory of predicate-complete test coverage and generation. Technical Report 2004-28, Microsoft Research, April 2004.
3. T. Ball, B. Cook, S. Das, and S. 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.
4. T. Ball and S. Rajamani. Boolean programs: A model and process for software analysis. Technical Report 2000-14, Microsoft Research, February 2000.
5. T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop on Model Checking of Software*, pages 103–122, 2001.
6. T. Ball and S. Rajamani. Generating abstract explanations of spurious counterexamples in C programs analysis. Technical Report 2002-09, Microsoft Research, January 2002.
7. D. Beyer, A. Chlipala, T. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *International Conference of Software Engineering*, 2004. To appear.
8. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, June 2004.
9. 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.
10. 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.
11. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.
12. E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. In *POPL*, January 1992.

13. 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.
14. E. Clarke, M. Talupur, and D. Wang. SAT based predicate abstraction for hardware verification. In *Proceedings of SAT'03*, May 2003.
15. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, July 2003.
16. P. Godefroid. VeriSoft: a tool for the automatic analysis of concurrent reactive software. In *Computer Aided Verification*, pages 172–186, 1997.
17. 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.
18. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Principles of Programming Languages*, pages 58–70, 2002.
19. H. Jain, D. Kroening, and E. Clarke. Verification of SpecC using predicate abstraction. In *MEMOCODE 2004*. IEEE, 2004.
20. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata- Theoretic Approach*. Princeton University Press, 1995.
21. 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.
22. T. B. 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.
23. 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.
24. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.