Open-Source Model Checking*

R. Grosu[1], X. Huang[1], S. Jain[2], and S.A. Smolka[1]
[1] Dept. of Computer Science, Stony Brook Univ., Stony Brook, NY 11794, USA
[2] Intel Corporation, Hillsboro, OR 97124, USA
E-mail: {grosu,xhuang,sumit,sas}@cs.sunysb.edu, sumit.jain@intel.com

Abstract. We present GMC[2], a software model checker for GCC, the open-source compiler from the Free Software Foundation (FSF). GMC[2], which is part of the GMC static-analysis and model-checking tool suite for GCC under development at SUNY Stony Brook, can be seen as an extension of *Monte Carlo model checking* to the setting of concurrent, procedural programming languages. Monte Carlo model checking is a newly developed technique that utilizes the theory of geometric random variables, statistical hypothesis testing, and random sampling of lassos in Büchi automata to realize a one-sided error, randomized algorithm for LTL model checking. To handle the function call/return mechanisms inherent in procedural languages such as C/C++, the version of Monte Carlo model checking implemented in GMC[2] is optimized for pushdown-automaton models. Our experimental results demonstrate that this approach yields an efficient and scalable software model checker for GCC.

1 Introduction

During the past 15 years, GCC has evolved from a modest C compiler, to a full-blown, multi-language compiler that can generate code for more than 30 target architectures. The set of programming languages handled by GCC now includes C, C++, Objective-C, Fortran, Java, and Ada. This diversity of languages and architectures has made GCC one of the most popular compilers in current use.

Traditionally, GCC has translated source code directly to RTL (register transfer level), a very low-level intermediate language, before applying any optimizations. This inevitably rendered the optimizations performed as low level, since higher-level semantic information such as data types, structures and fields were lost during translation. To remedy this situation, the Tree-SSA branch [19] of GCC has resulted in the addition of two new intermediate languages to GCC: GENERIC, which provides a common infrastructure for abstract syntax tree analysis and optimization; and GIMPLE three-address code, which provides a common infrastructure for CFG (control flow graph) analysis and optimization.

Together with their associated APIs, GENERIC and GIMPLE make Tree-SSA suitable as a platform not only for the development of high-level code-optimization techniques, but also for new static-analysis tools, applicable to all of GCC's input languages. The acceptance of the Tree-SSA branch by the open-source community has led, during the past year, to it being merged with the main line in Release Version 3.5.

In this paper, we describe a software model checker for GCC that we have designed and implemented at the Tree-SSA level. Our model checker, which we call GMC[2] for GCC-based Model Checking, is an extension of the technique of *Monte Carlo model checking* [8] to the setting of concurrent, procedural programming

* R. Grosu, X. Huang and S. Jain were partially supported by the NSF Faculty Early Career Development Award CCR01-33583.

languages. Monte Carlo model checking is a newly developed technique that utilizes the theory of geometric random variables, statistical hypothesis testing, and random sampling of lassos in Büchi automata to realize a one-sided error, randomized algorithm for LTL model checking. To handle the function call/return mechanisms inherent in procedural languages such as C/C++, the version of Monte Carlo model checking implemented in `GMC`^{2} is optimized for pushdown-automaton models.

At the heart of `GMC`^{2} is a `GIMPLE CFG` interpreter `interpret` that traverses `CFG`s using `Tree-SSA` statement iterators `succ`, `tsucc` and `fsucc`, interpreting each statement encountered according to its semantics. Of particular interest is the manner in which process creation and synchronization statements are processed, which force a return whenever a context switch is required, as well as function invocation and return statements, which induce a hierarchic structure on the hash table `GMC`^{2} utilizes for lasso detection.

`GMC`^{2} and `interpret` are part of the `GMC` suite of analysis and verification tools we are developing for the `Tree-SSA` level of `GCC`, which additionally includes an intra-procedural slicer and a BDD implementation of a symbolic-execution engine for `GIMPLE CFG`s. The tool suite is intended to provide an open-source framework for `GCC`-based static analysis and model-checking.

The main contributions of this paper can be summarized as follows.

- By virtue of being implemented at the `Tree-SSA` level, `GMC`^{2} is at once a software model checker for each of `GCC`'s 6 input languages and more than 30 target architectures.
- `GMC`^{2} is an *open-source model checker*: its integration into `GCC` renders it readily and widely accessible for usage, critique, and extension by the open-source community.
- `GMC`^{2} implements the technique of Monte Carlo model checking [8] within the setting of concurrent procedural programming languages. The version of Monte Carlo model checking implemented in `GMC`^{2} is therefore optimized for pushdown-automaton models.
- Our experimental results demonstrate that the Monte Carlo approach yields an efficient and scalable software model checker for `GCC`.

The rest of the paper develops along the following lines. Section 2 considers the technique of Monte Carlo model checking. Section 3 provides an overview of the `GCC` compilation process. Section 4 describes `GMC`^{2}, our software model checker for `GCC`, while Section 5 summarizes our experimental results. Section 6 discusses related work. Section 7 contains our conclusions and directions for future work. The `GMC`^{2} model checker is available from [21].

2 Monte Carlo Model Checking

Monte Carlo model checking [8] performs random sampling of lassos in a Büchi automaton (BA) to realize a one-sided error, randomized algorithm for LTL model checking. In this section, we provide an overview of this technique. In Section 4, we show how to extend this technique to hierarchic Büchi automata (HBA) in the context of software model checking.

Büchi automata. A *Büchi automaton* $A = (\Sigma, Q, Q_0, \delta, F)$ is a five-tuple where: Σ is a finite *input alphabet*; Q is a finite set of *states*; $Q_0 \subseteq Q$ is the set of *initial states*; $\delta \subseteq Q \times \Sigma \times Q$ is the *transition relation*; $F \subseteq Q$ is the set of *accepting states*. We assume, without loss of generality, that every state of a BA has at least one outgoing transition, even if this transition is a self-loop.

A sequence $\sigma = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1}, ...$, where $s_0 \in Q_0$ and for all $i \geq 0$, $s_i \xrightarrow{a_i} s_{i+1} \in \delta$ is called an *infinite run* of A if the sequence is infinite and a *finite run* otherwise. An infinite run is called *accepting* if there exists an infinite set of indices $J \subseteq \mathbb{N}$, such that for all $i \in J$, $s_i \in F$.

We say that σ is *ultimately periodic* if there exist $i \geq 0$, $l \geq 1$ such that for all $j \geq 0$, $s_{i+j} = s_{i+j \bmod l}$. This means that σ consists of a finite prefix $s_0 \xrightarrow{a_0} \cdots s_{i-1} \xrightarrow{a_{i-1}}$, followed by the "infinite unfolding" of a *cycle* $s_i \xrightarrow{a_i} \cdots \xrightarrow{a_{i+l-1}} s_i$. The cycle is called *simple* if for all $0 \leq j \neq k < l$, $s_{i+j} \neq s_{i+k}$; i.e., the cycle does not visit the same node twice. In the following, we shall refer to such a reachable simple cycle as a *lasso*, and say that a lasso is *accepting* if its simple cycle contains an accepting state.

Let S be a concurrent system, A_S the BA encoding S's state transition graph, and φ an LTL property. Using the tableau method, one can construct a Büchi automaton $A_{\neg\varphi}$ accepting the same language as $\neg\varphi$ [5]. The LTL model-checking problem $A_S \models \varphi$ is then naturally defined in terms of the emptiness problem for $B = A_S \times A_{\neg\varphi}$, which reduces to finding accepting lassos in B [22].

Random lassos and hypothesis testing. Instead of searching the entire state space of B for accepting lassos, we successively generate up to M lassos of B on the fly, by performing random walks in B. The walks are *uniform* in the sense that they are generated by imposing a uniform distribution on the outgoing transitions of the current state along the walk. If the currently generated lasso is accepting, we have found a counter-example to emptiness, and stop.

To determine the number M of lassos we need to generate, we aim to answer, with *confidence* $1-\delta$ and within *error margin* ϵ, the following question: *how many independent lassos do we need to generate until one of them is accepting?* The answer is based on the theory of *geometric random variables* and *statistical hypothesis testing*. Let X be geometric random variable parameterized by the Bernoulli random variable Z (defined below) that takes value 1 with probability p_Z and value 0 with probability $q_Z = 1 - p_Z$. Intuitively, p_Z is the probability that an arbitrary lasso of B is accepting.

The cumulative distribution function of X for N independent trials of Z is: $F(N) = \mathbf{Pr}[X \leq N] = 1 - (1 - p_Z)^N$. Requiring that $F(N) = 1 - \delta$ yields: $N = \ln(\delta)/\ln(1 - p_Z)$. Given that p_Z is what we wish to determine, we assume for the moment that $p_Z \geq \epsilon$. Replacing p_Z with ϵ yields $M = \ln(\delta)/\ln(1 - \epsilon)$ which is greater than N and therefore $\mathbf{Pr}[X \leq M] \geq \mathbf{Pr}[X \leq N] = 1 - \delta$. Summarizing:

$$p_Z \geq \epsilon \quad \Rightarrow \quad \mathbf{Pr}[X \leq M] \geq 1 - \delta \quad \textbf{where} \quad M = \ln(\delta)/\ln(1 - \epsilon) \qquad (1)$$

Inequation 1 gives us the minimal number of attempts M needed to achieve success with confidence ratio δ, under the assumption that $p_Z \geq \epsilon$. The standard way of discharging such an assumption is to use *statistical hypothesis testing* (see

e.g. [18]). Define the *null hypothesis* H_0 as the assumption that $p_Z \geq \epsilon$. Rewriting inequality 1 with respect to H_0 we obtain:

$$\mathbf{Pr}[X \leq M \,|\, H_0] \geq 1 - \delta \tag{2}$$

We now perform M trials. If no counterexample is found, i.e., if $X > M$, we reject H_0. This may introduce a type-I error: H_0 may be true even though we did not find a counter-example. However, the probability of making this error is bounded by δ; this is shown in inequality 3 which is obtained by taking the complement of $X \leq M$ in inequality 2:

$$\mathbf{Pr}[X > M \,|\, H_0] < \delta \tag{3}$$

Because we seek to attain a one-sided error decision procedure, we do not consider type-II errors in our application of hypothesis testing: as soon as we find a counter-example, we stop sampling and decide (with probability 1) that $A \not\models \varphi$.

The Monte Carlo model-checking algorithm. For a BA B, define the probability space $(\mathcal{P}(L), \mathbf{Pr})$, where $L = L_a \cup L_n$ is the set of all lassos of B and L_a and L_n are the sets of all accepting and non-accepting lassos of B, respectively. The probability $\mathbf{Pr}[\sigma]$ of a lasso $\sigma = s_0 \xrightarrow{a_0} \dots \xrightarrow{a_{n-1}} s_n$ is defined inductively as follows: $\mathbf{Pr}[s_0] = k^{-1}$ if $|Q_0| = k$ and $\mathbf{Pr}[s_0 \xrightarrow{a_0} \dots \xrightarrow{a_{n-1}} s_n] = \mathbf{Pr}[s_0 \xrightarrow{a_0} \dots \xrightarrow{a_{n-2}} s_{n-1}] \cdot \pi[s_{n-1} \xrightarrow{a_{n-1}} s_n]$ where $\pi[s \xrightarrow{a} s'] = m^{-1}$ if $s \xrightarrow{a} s' \in \delta$ and $|\delta(s)| = m$. That $(\mathcal{P}(L), \mathbf{Pr})$ is actually a probability space is established in [8].

Example 1 (Probability of lassos). Consider BA B of Figure 1. It contains four lassos, `11`, `1244`, `1231` and `12344`, having probabilities 1/2, 1/4, 1/8 and 1/8, respectively. Lasso `1231` is accepting.

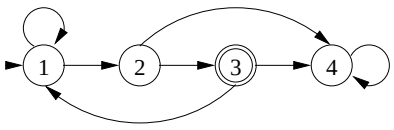

Fig. 1. Example lasso probability space.

Definition 1 (Lasso Bernoulli variable). *The* random variable Z *associated with the probability space* $(\mathcal{P}(L), \mathbf{Pr})$ *of a Büchi automaton* B *is defined as follows:* $p_Z = \mathbf{Pr}[Z = 1] = \sum_{\lambda_a \in L_a} \mathbf{Pr}[\lambda_a]$ *and* $q_Z = \mathbf{Pr}[Z = 0] = \sum_{\lambda_n \in L_n} \mathbf{Pr}[\lambda_n]$.

Example 2 (Lassos Bernoulli variable). For the Büchi automaton B of Figure 1, the lassos Bernoulli variable has associated probabilities $p_Z = 1/8$ and $q_Z = 7/8$.

Having defined Z, X and H_0, we are now ready to present our Monte Carlo decision procedure for emptiness checking of Büchi automata, called $\texttt{MC}^2$ in [8]. $\texttt{MC}^2$ consists of three statements. The first uses inequation 1 to determine the value for M, given parameters ϵ and δ. The second statement is a for-loop that successively samples up to M lassos by calling the *random lasso* (`rLasso`) routine, described in Section 4. If an accepting lasso l is found, $\texttt{MC}^2$ decides false and returns l as a counter-example. If no accepting lasso is found within M trials, $\texttt{MC}^2$ decides true, and reports that with probability less than δ, $p_Z > \epsilon$.

```
bool × lasso MC² (BA B = (Σ, Q, Q₀, δ, F), float 0 < ϵ, δ < 1)
{
  M = ln δ / ln(1 − ϵ);
  for (i = 1; i ≤ M; i++) if(rLasso(B)==(true,l)) return (false,l);
  return (true,nil); /* Pr[X > M | H₀] < δ */;
}
```

Theorem 1 ([8]). *Given a Büchi automaton B and parameters ϵ and δ, if* MC^2 *returns false, then $L(B) \neq \emptyset$. Otherwise, $\mathbf{Pr}[X > M \mid H_0] < \delta$ where $M = \ln(\delta)/\ln(1-\epsilon)$ and $H_0 \equiv p_Z \geq \epsilon$.*

MC^2 is very efficient in both time and space. The *recurrence diameter* of a Büchi automaton B is the longest loop-free path in B starting from an initial state.

Theorem 2 ([8]). *Let B be a Büchi automaton, D its recurrence diameter and $M = \ln(\delta)/\ln(1-\epsilon)$. Then* MC^2 *runs in time $O(MD)$ and uses $O(D)$ space.*

In the worst case, D is exponential in $|S| + |\varphi|$ and thus MC^2's does not improve on the space complexity of a typical model checker. In practice, however, one can expect MC^2 to perform much better than this.

3 Overview of GCC

The block diagram of Figure 2 provides an overview of the GCC compilation process from source input file to object code.

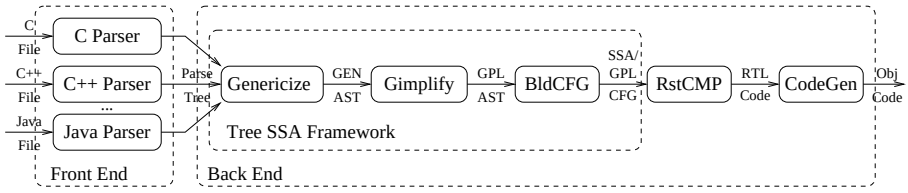

Fig. 2. Block diagram of the GCC compilation process.

The language-specific front-end of GCC translates a source input file to a (language-specific) parse tree. The back end is largely language independent, and handles code optimization and final code generation. Traditionally, GCC translated source code directly to RTL (register transfer level), a very low-level intermediate language, before applying any optimizations. This inevitably rendered the optimizations performed as low level, since higher-level semantic information such as data types, structures and fields were lost during translation.

Tree-SSA. To remedy this situation, the Tree-SSA branch [19] of GCC has resulted in the addition of two new intermediate languages (ILs): GENERIC and GIMPLE [16]. Together with their APIs, these ILs make Tree-SSA suitable as a platform not only for the development of high-level code optimization techniques, but also for new static analysis tools, applicable to all of GCC's input languages. The acceptance of Tree-SSA by the open-source community has led, during the past year, to it being merged with the mainline in Release Version 3.5.

```
int main() {                              1.  int main {
    int a, b, c;                          2.      int a, b, c;
    a = 5;                                3.      int T1, T2, T3, T4;
    b = a + 10;                           4.      a = 5;
    c = b + foo(a, b);          =>        5.      b = a + 10;
    if (a > b + c)                        6.      T1 = foo(a, b);
      c = b++ / a + (b * a);              7.      T2 = b + T1;
    bar(a, b, c); }                       8.      if (a > T2) goto fi
                                          9.         T3 = b / a;
                                         10.         T4 = b * a;
                                         11.         c = T2 + T3
                                         12.         b = b + 1;
                                         13.  fi: bar (a, b, c); }
```

Fig. 3. Sample C program and corresponding GIMPLE representation.

GENERIC realizes a common infrastructure for AST-level (abstract syntax tree) analysis and optimization by providing a language-independent IL for all parse-tree constructs produced by the language-specific front ends. GIMPLE is a C-like three-address (3A) code which provides a common infrastructure for CFG-level (control flow graph) analysis and optimization. As usual, complex expressions (possibly with side effects) are broken into simple 3A statements by introducing new, temporary variables. Similarly, complex control statements are broken into simple 3A (conditional) gotos by introducing new labels.

Syntactically, GIMPLE is a subset of GENERIC. Each GIMPLE tree is used to construct a GIMPLE-CFG, which itself can subsequently be converted to static single-assignment (SSA) form. Figure 3 shows a C program and its corresponding GIMPLE representation, which preserves source-level information such as data types and procedure calls. While not shown in the example, GIMPLE types also include pointers and structures.

Once a function is translated to GIMPLE form, the Tree-SSA framework builds its associated control flow graph. Each node in the CFG is linked to a basic block (sequence of non-branching instructions), represented as a GIMPLE AST. CFG transitions correspond to (conditional) goto instructions. For example, the CFG for the GIMPLE program of Figure 3 is shown in Figure 4.

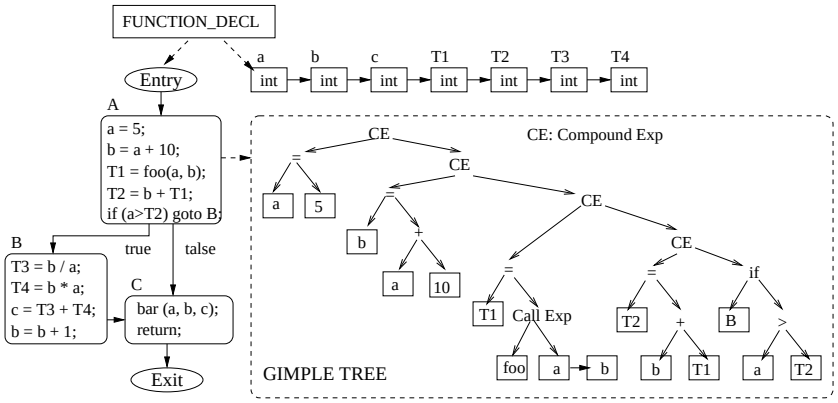

Fig. 4. Control flow graph for example C program.

The Tree-SSA API provides functions to manipulate and traverse CFGs, their associated ASTs, and their list of variables. For example, succ returns the address of the immediate successor of a non-branching statement, and tsucc and fsucc return the address of the immediate true and respectively false successor of a branching statement. Similarly, if a is a variable in a CFG, its type attribute a.type is also a GIMPLE AST containing various information such as type name, size, and alignment. Basic data-flow, control-flow, alias and reachability-analysis routines are also provided by the Tree-SSA API.

4 Monte Carlo Software Model Checking

We have implemented a software model checker for GCC based on the generic Monte-Carlo model-checking algorithm of Section 2. Our model checker, GMC^2, is applicable to any program written in one of the procedural languages supported by GCC, e.g. C. Call this program the *target program* to be verified. GMC^2 also requires as input a procedure or function, call it the *property function*, representing the LTL property of interest. The target program can contain concurrency primitives similar to those supported by the Verisoft model checker [6]. In the case of safety properties, the property function is called to check for property violations in the target program. In the case of liveness properties, the property function is called to check if an accepting state of the target program is visited infinitely often, viewing the target program as a succinct representation of a Büchi automaton.

GMC^2 operates at the Tree-SSA level and assumes that the target program and property function have been compiled into CFGs. Let P be the array of CFGs corresponding to the target program, one for each of its functions, and let φ be the CFG for the property function. At the heart of GMC^2 is a CFG interpreter that traverses the CFGs in P using Tree-SSA's statement iterators and interprets the statements contained in the CFGs according to their semantics. This allows GMC^2 to generate the random lassos of the target program on the fly.

4.1 The Main Routine

Due to space considerations, we limit our discussion to the treatment of safety properties. Given an array of P of CFGs for the target C program, a CFG φ for the C function encoding a safety property, and parameters ϵ and δ, GMC^2 successively generates at most $\ln(\delta)/\ln(1-\epsilon)$ random lassos of P; see Section 2. While generating a lasso, φ is called to check whether or not φ is violated in the newly reached program state. If so, GMC^2 stops and returns the counter-example path leading to the violating state. If all states of all sampled executions satisfy φ, GMC^2 stops and reports with confidence greater than $1-\delta$ that it rejects $\mathtt{H}_0 \doteq \mathtt{p}_z \geq \epsilon$.

At the heart of GMC^2 is the rLasso routine for generating random lassos; rLasso conducts a random execution of the CFGs in P by interpreting their (possibly concurrent) C statements and checking for property violations.

4.2 The rLasso Random-Lasso Routine

In order to detect (global) lassos, the (concurrent) program state is stored in a *hash table* ht each time a context switch occurs. This is for efficiency purposes: the alternative, less efficient approach would be to store the program state after each statement execution. To ensure the soundness of this approach, we assume that the time between context switches is finite.

```
bool × lasso rLasso() /* global cfg array P, cfg φ */
{
  hashTbl ht = ∅; ready list ready = ∅; bool × state (f,s) = rInit();
  while (s ∉ ht) {
    insert(ht,s); if (¬f) return (true,lasso(ht));
    (f,s) = rNext(s); }
  return (false,lasso(ht));
}
```

Hash table. The ht hash table is *optimized* so that common information among global states is shared. It is also *hierarchic* in the sense that all states belonging to a callee are linked to each other so that they can easily be removed from ht when the callee returns.

The pseudo-code for the rLasso routine is given above. The first line sets ht and ready to empty, and initializes the violation flag f and the current-state variable s by calling routine rInit. The while-loop searches for (violating) lassos. If the current state s is not in ht, then it is a new state and is inserted in ht. If it is also violating, signaled by ¬f being true, then a violating lasso was found, which is returned together with the corresponding flag to GMC[2]. Otherwise, another random next state is generated by calling rNext.

4.3 Routines rInit and rNext

Given a set V of typed variables, a *valuation* (or environment) of V is a mapping of variables in V to their type-correct values. If Γ and Γ' are lists, and σ is a list element, we write $\texttt{concat}(\Gamma, \Gamma')$, $\texttt{append}(\Gamma, \sigma)$ and $\texttt{rest}(\Gamma)$ for the lists obtained by concatenating Γ and Γ', appending σ to Γ, and taking the rest of Γ, respectively, and we write $\Gamma(\texttt{i})$ for the i-th element of Γ. If Δ is a stack and ϕ a stack element, then we write $\texttt{push}(\Delta, \phi)$, $\texttt{pop}(\Delta)$ and $\Delta.\phi$ for pushing ϕ onto the stack, popping the stack, and for the topmost element on the stack, respectively. If s is a statement, i.e., AST of a CFG, then s.a is a child of s.

Program state. The state $\Sigma = (\chi, \Gamma)$ of a concurrent C program consists of a valuation χ of the *shared variables* (channels and semaphores) and a list Γ of *process states*, one for each active process. The list is ordered by the order of process creation. The state $\sigma = (\kappa, \delta)$ of a process has two components: the *control state* κ and the *data state* δ. The control state $\kappa = (\gamma, \nu)$ consists of a *function name* γ and a *statement number* ν within γ. The data state $\delta = (\pi, \beta, \Delta)$ consists of a *heap* π, a valuation of *global variables* β and a *frame stack* Δ. Each *frame* $\phi = (\kappa, \rho)$ of Δ contains a return control state κ to the caller CFG and a valuation ρ for the *local variables* of the callee CFG.

Routine `rInit`**.** Execution of `P` starts in a random state Σ_0 defined as follows. All channels in χ_0 are empty and all semaphores are 0. The process-state list Γ_0 contains only the state σ_0 of the root process. The control state κ_0 of the root process has function `main` of `P` in γ_0 and 0 in ν_0.

```
bool × prgState rInit() /* global cfg array P, cfg φ */
{
  sharedState χ = χ0; procStates Γ = ∅; frameStack Δ = ∅;
  cfgNm γ = main; stmNo ν = 0; controlState κ = (γ,ν);
  lclEnv ρ = ρ0; forall (x ∈ dom(P[γ].param)) ρ[x] = random(P[γ].param.type);
  frame φ = (trap,ρ); push(Δ,φ); dataState δ = (π0,β0,Δ);
  procState σ = (κ,δ); append(Γ,σ); prgState Σ = (χ,Γ);
  if eval(φ) return (true,Σ) else return (false,Σ);
}
```

The data state δ_0 of the root process consists of the empty heap π_0, valuation β_0 of the global variables, and stack frame Δ_0 with only frame ϕ_0 of `main` pushed. This frame has a predefined return control state `trap` (e.g. the stop point) and a valuation ρ_0 for the local variables. The valuation of the formal parameters in ρ_0 is chosen randomly within their corresponding range. Function `eval` evaluates a `CFG` in the current state and returns its value.

```
bool × prgState rNext(prgState s)
{
  /* global cfg array P, hashTbl h, CFG φ, ready list ready */
  int i = random(|ready|); int nxt = ready[i];
  return interpret(s,nxt);
}
```

Routine `rNext`**.** Routine `rNext` randomly selects one of the ready processes and interprets it by calling routine `interpret`, described next. It regains control when `interpret` reaches a concurrency statement, which requires a context switch.

4.4 Routine `interpret`

Routine `interpret` traverses the `CFG`s in `P`, using statement iterators `succ`, `tsucc` and `fsucc`, and interprets each statement according to its semantics. Of particular interest are the process creation and synchronization statements, which force a return whenever a context switch is required, as well as function invocation and return statements, which induce a hierarchic structure on the hash table.

Since `interpret` may generate several states before it returns, it has to check whether property φ is true in all of them. Properties to be checked may also be inserted in a program, as `assert(p)` statements. The interpreter then checks whether predicate `p` is true in the current state and returns with a violation if this is not the case.

The pseudo-code for `interpret` is given below. Its body is an infinite loop, which according to the type of the current statement ν within the current `CFG` $P[\gamma]$, undertakes the actions defining the semantics of the statement. Due to space limitations, we consider a representative susbset of statement types, which does not include heap and pointer manipulation statements.

```
bool × prgState interpret(prgState Σ, int i)
{
  /* global cfg array P, hashTbl ht, cfg φ, ready list ready */
  channels χ = Σ.χ; procStates Γ = Σ.Γ; procState σ = Γ[i];
  while (true) {
    cfgNm γ = σ.κ.γ; stmtNo ν = σ.κ.ν;
    frameStack Δ = σ.δ.Δ; globalEnv β = σ.δ.β;
    switch (P[γ][ν].type) of
    if:       /* if e goto t */ {
      ν = (eval(P[γ][ν].exp)) ? tsucc(P[γ][ν]) : fsucc(P[γ][ν]); }
    assert:  /* assert(e) */ {
      if (!eval(P[γ][ν].exp)) return (false,Σ); ν = succ(P[γ][ν]); }
    assign:  /* x = rhs */ {
      if (P[γ][ν].rhs.type == expr) { /* rhs == e */
        ν = succ(P[γ][ν]); (Δ.ρ:β)[P[γ][ν].var] = eval(P[γ][ν].rhs); }
      else if (P[γ][ν].rhs.fnc == toss) { /* rhs == toss(e) */
        ν = succ(P[γ][ν]);
        (Δ.ρ:β)[P[γ][ν].var] = random(eval(P[γ][ν].rhs.exp)); }
      else if (P[γ][ν].rhs.fnc == fork) { /* rhs == fork() */
        ν = succ(P[γ][ν]); (Δ.ρ:β)[P[γ][ν].var] = 0;
        append(Γ,((γ,ν),(π,β,Δ))); (Δ.ρ:β)[P[γ][ν].var] = |Γ|-1; }
      else if (P[γ][ν].rhs.fnc == recv) { /* rhs == recv(c) */
        c = P[γ][ν].rhs.chnl;
        if (empty(χ.c)) {append(χ.c.swait,i); return (true,Σ); }
        ν = succ(P[γ][ν]); (Δ.ρ:β)[P[γ][ν].var] = fst(χ.c.queue));
        rest(χ.c.queue); concat(ready,χ.c.swait); χ.c.swait = ∅; }
      else { /* rhs == f(a) */
        a = eval(P[γ][ν].rhs.act); κ = (γ,ν);
        γ = P[γ][ν].rhs.fnc); ν = 0;
        push(Δ,(κ,ρ_{γ,0})); (Δ.ρ)[P[γ].fpar] = a; }
    return:  /* return e */ {
      e = eval(P[γ][ν].exp); (γ,ν) = Δ.κ; popLocal(ht); pop(Δ);
      (Δ.ρ:β)[P[γ][ν].var] = e; }
    send:    /* send(c,e) */ {
      c = P[γ][ν].rhs.chnl;
      if (full(χ.c)) {append(χ.c.rwait,i); return (true,Σ);}
      ν = succ(P[γ][ν]); append(χ.c.queue,eval(P[γ][ν].rhs.exp));
      concat(ready,χ.c.rwait); χ.c.rwait = ∅;
    }
    σ = ((γ,ν),(π,β,Δ)); Γ[i] = σ; Σ = (χ,Γ);
    if (!eval(φ)) return (false,Σ);
  }
}
```

For the sequential intra-procedural group of statements, we discuss the interpretation of `if` and (simple) `assignment`. The former evaluates the predicate in

the current state and branches to the appropriate location by modifying ν. The latter evaluates the right-hand side expression and updates the corresponding local environment $\Delta.\rho$ (within the frame on the top of the frame stack) or global environment β, on the location given by the left-hand side variable, accordingly. By writing $\Delta.\rho : \beta$ we mean that both valuations are considered and that $\Delta.\rho$ has precedence over β; i.e., we first search the variable in the local valuation.

The modeling and verification statements presented are `toss` and `assert`. For `toss`, the interpreter first evaluates the argument expression to obtain a value v, and then it randomly generates a number within the range $[0, v]$. The obtained number is assigned to the location given by the left-hand side variable, in either $\Delta.\rho$ or β. For `assert`, the interpreter checks whether the predicate is true in the current state. If this is not the case, it returns `false` and Σ. Otherwise, it continues with the next statement by updating ν.

The inter-procedural statements presented are `call` and `return`. For `call`, a new frame $\phi = (\kappa, \rho)$ is allocated on top of the frame stack Δ; κ is the current control state; ρ has the local variables of the target function γ initialized accordingly by $\rho_{\gamma,0}$ and the formal parameters evaluated in the current state. Control is then moved to the callee by updating γ and ν accordingly. For `return`, the interpreter does the following. First, it evaluates the return expression in a temporary variable. It then restores the control state from the frame stack, pops the frame stack and erases all the states corresponding to the callee from the hash table. Finally it assigns the temporary variable, to the location given by the variable of the statement pointed to by the control state, in either $\Delta.\rho$ or β.

The concurrency primitives considered so far include process creation, channels and semaphores. For simplicity, we only discuss `fork`, `send` and `recv`. The other are treated in a similar manner. The `fork` statement is handled by creating a new process (state) in Γ which is identical to the current except for the value assigned to the variable on the left-hand side of the fork assignment statement. This is zero for the child process and the index in Γ of the new process for the parent process. The `send` statement is treated as expected. If the channel is full, the process is put in the send wait queue of the corresponding channel, and control is returned to `rNext`. Otherwise, the message expression is evaluated and appended to the channel. Moreover, the process waiting in the receive queue of the channel is awaken, by moving it to the ready list. The `recv` primitive is treated in a similar way.

5 Experimental Results

To assess the performance and scalability of `GMC`^{2}, we compared it to `VeriSoft`, a popular software model checker from Lucent Technologies [6], on two C benchmarks: dining philosophers and the Needham-Schroeder. `VeriSoft` and `GMC`^{2} were given the same C source files as input, each of which can be downloaded from [21]. We also ran `GMC`^{2} on the TCAS benchmark. All `GMC`^{2} experiments were performed on an Athlon 2600+ MHz processor with 1GB RAM running Linux 2.6.5.

Dining philosophers. For this classical synchronization problem, we used a faulty *symmetric* but fair variant, where the number of philosophers varied

from 4 to 16. The safety property we checked was *deadlock freedom*. Our experimental results are given in Table 1. The meaning of the column headings is the following: phi. is the number of philosophers; time is the execution time in mins:secs; ce.len is the length of the counter-example found; states is the number of states `VeriSoft` visited until finding an error; transitions is the number of transitions that `VeriSoft` traversed. The `VeriSoft` experiments were performed on Sun Sparc Ultra-5.10 server running SunOS 5.6. Our experience shows that the Athlon/Linux environment performs approximately 3.4 times faster than the Sparc/SunOS environment.

	`GMC`^{2}			`VeriSoft`		
phi.	time	samples	ce.len.	time	states	transitions
4	0:00.07	2	12	0:00.61	16	37
6	0:00.11	4	12	0:16.60	773	1171
8	0:00.78	11	20	2:57.29	5431	8449
10	0:02.17	31	24	10:41	17908	31433
12	0:04.82	24	27	$>$ 2hr	N/A	N/A
14	0:06.22	22	44	$>$ 2hr	N/A	N/A
16	0:11.56	14	32	$>$ 2hr	N/A	N/A

Table 1. Deadlock freedom for the symmetric and fair C implementation.

Needham-Schroeder protocol. This classic public-key protocol provides mutual authentication for two parties, before they engage in a transaction. In 1995, Lowe first reported a flaw in the protocol [14], by exhibiting an attack involving six message exchanges. Suppose A is the initiator, B is the responder and I is the intruder. Then the attack is as follows: (i) A sends a nonce to I. (ii) I sends same nonce to B. (iii) B sends the above received nonce and its new nonce to I. (iv) I sends the above received message to A. (v) A validates the authenticity of I and sends the second nonce from the message back to I. (vi) I sends this nonce back to B which now also validates I. We checked for the existence of the above attack in a C implementation of the protocol we obtained from Patrice Godefroid, who we greatfully acknowledge. `GMC`^{2} found it in 6 hours and 37 minutes after having checked 10,682,639 lassos.

The same example and implementation was used in [7] to evaluate a novel genetic algorithm. The time usage reported there is 2 hours and 33 minutes to find 3 errors, which is superior to `GMC`^{2} on this benchmark. They also attempted exhaustive and randomized search algorithms on this C program, neither of which could find an error in 8 hours. Their experiments were performed on a Pentium III 700 MHz processor with 256 MB RAM. Unfortunately, the genetic version of `VeriSoft` is not publicly available, and we could not reproduce this result on our own machine. Its superior performance might be explained by the sequential nature of the protocol implementation, which essentially executes only one round of a reactive system. In this round, the system either deadlocks, produces a counterexample or it behaves correctly. Hence, lasso search seems to be less useful in this case than applying genetic heuristics.

TCAS. The traffic alert and collision avoidance system (`TCAS`) is used on board all US commercial aircrafts. It continuously monitors radar information to sense

whether a neighboring aircraft could become a threat by getting too close. Such an aircraft is said to be an "intruder", which is entering the protected zone. In this situation `TCAS` issues a *traffic advisory* (`TA`) and estimates the time remaining until the two aircrafts reach the closest point of approach. Such estimates are used to compute the vertical separation between the two aircraft assuming that the controlled aircraft maintains its current trajectory. Depending on the results obtained, `TCAS` issues a *resolution advisory* (`RA`) suggesting the pilot to climb or to descend.

property	rule	$\texttt{GMC}^2$		
		bugs found	time	samples
Safe Advisory Selection	1	No	0.23	1278
	2	Yes	0.03	147
Best Advisory Selection	1	No	0.25	1278
	2	Yes	0.04	206
Avoid unnecessary crossing	1	Yes	0.01	36
	2	Yes	0.03	180
No Crossing Advisory Selection	1	Yes	0.01	27
	2	Yes	0.01	8
Optimal Advisory Selection	1	No	0.23	1278
	2	Yes	0.06	217

Table 2. Running time of $\texttt{GMC}^2$ for `TCAS`.

We have verified the `RA` component from Georgia Techs Siemens suite [20], with respect to the specifications in [3]. Each property is verified by checking the satisfiability of two rules, with specific initial values for variables. The details of these rules, initial conditions on values and the properties, can be found in [3]. Our experimental results are presented in Table 2, where the meaning of the column headings is as follows: property name; corresponding rule number; indication of whether or not $\texttt{GMC}^2$ found a counter-example; time usage in seconds within which either a counter-example was found or a predefined number of samples was reached; if a counter-example was found, the last colum gives the number of samples taken to that point; otherwise it is the predefined number of samples to be taken: 1,278 corresponding to $\delta = 0.1$ and $\epsilon = 1.8 \times 10^{-3}$.

6 Related Work

At the confluence of model checking, static analysis and theorem proving, *software model checking* has become an area of intense research. Given a software system S, software model checkers either work directly on S, or extract a model M from S and apply more traditional model-checking techniques to M. The software model checkers most closely related to $\texttt{GMC}^2$ are those for concurrent procedural languages, such as C/C++, and include `VeriSoft` [6], Spin [11], Blast [10], Magic [2] and C Wolf [4]. In the case of `VeriSoft`, a randomized search strategy based on genetic algorithms has been developed to guide state-space search towards error states [7]. A comparison of the relative performance of $\texttt{GMC}^2$ and `VeriSoft` is given in Section 5.

The Cooperative Bug Isolation (CBI) project at Berkeley performs compile-time instrumentation on a a number of large open-source projects and distributes the resulting binaries [13]. Information is then be gathered about how many times a program terminated successfully or not. Subsequent statistical analysis is used to isolate erroneous code segments. In contrast, GMC^2 is a model checker embedded at the Tree-SSA level of the open-source GCC compiler.

Other researchers have developed randomized methods for software verification and analysis. The key idea behind the program-analysis technique of *random interpretation* [9] is to execute a code fragment on a few random inputs in a contrived manner, which includes giving random linear interpretations to the operators in the program. Both branches of a conditional are executed on each run and at joint points, a random affine combination of the joining states is performed. In the branches of an equality conditional, the data values are adjusted on the fly to reflect the truth value of the guarding boolean expression.

In [17], Monte Carlo and abstract-interpretation techniques are used to analyze programs whose inputs are divided into two classes: those that behave according to some fixed probability distribution and those considered nondeterministic.

7 Conclusions

We have presented GMC^2, a software model checker for GCC based on the technique of Monte Carlo model checking. By virtue of being implemented at the Tree-SSA level, GMC^2 is at once a model checker for each of GCC's 6 input languages, including C and C++, and more than 30 target architectures. GMC^2 is also an open-source model checker, and therefore readily and widely accessible for usage, critique, and extension by the open-source community.

An important advantage of our approach concerns the treatment of *pointers* in GMC^2. Basically, it is much easier to *interpret* pointer operations than it is to statically *analyze* them.

As ongoing and future work, we are in the process of creating a *software model checking* branch of GCC for the public distribution of the GMC tool suite. Also, we are developing automated abstraction [1] and interpolation techniques [15] to handle programs with infinite-domain variables. Currently, we are manually applying a form of bounded-range abstraction [12], for example, on the TCAS benchmark.

References

1. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. PLDI 2001, SIGPLAN Notices* 36(5), pages 203–213, 2001.
2. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *Transactions on Software Engineering*, 30(6):388–402, June 2004.

3. A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzè. Using symbolic execution for verifying safety-critical systems. In *ESEC/FSE-9: Proc. 8th European Software Engineering Conference*, pages 142–151. ACM Press, 2001.
4. D. C. DuVarney and S. P. Iyer. C Wolf – a toolset for extracting models from C programs. In *FORTE 2002*, pages 260–275, 2002.
5. R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.
6. P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of 24th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL '97)*, pages 174–186. ACM Press, 1997.
7. P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. *STTT*, 6(2):117–127, 2004.
8. R. Grosu and S. A. Smolka. Monte carlo model checking. In *Proc. 11th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, pages 271–286. Springer-Verlag, 2005.
9. S. Gulwani and G. C. Necula. Precise interprocedural analysis using random interpretation. In *Proc. 32nd Annual ACM Symposium on Principles of Programming Languages*. ACM, January 2005.
10. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN 2003)*, pages 235–239. Lecture Notes in Computer Science 2648, Springer-Verlag, 2003.
11. G. J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.
12. D. Kroening, J. Ouaknine, S. A. Seshia, and O. Strichman. Abstraction-based satisfiability solving of Presburger arithmetic. In *CAV 2004*, 2004.
13. B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Public deployment of cooperative bug isolation. In *Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*, 2004.
14. G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, pages 131–133, 1995.
15. K. L. McMillan. Applications of Craig interpolants in model checking. In *TACAS 2005*, pages 1–12, 2005.
16. J. Merrill. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In *Proceedings of the GCC Developers Summit3*, pages 171–180, May 25-27, 2003.
17. D. Monniaux. An abstract Monte-Carlo method for the analysis of probabilistic programs. In *Proc. 28th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 93–101. ACM Press, 2001.
18. A. M. Mood, F.A. Graybill, and D.C. Boes. *Introduction to the Theory of Statistics*. McGraw-Hill Series in Probability and Statistics, 1974.
19. D. Novillo. Tree SSA: A New Optimization Infrastructure for GCC. In *Proceedings of the GCC Developers Summit3*, pages 181–193, May 25-27, 2003.
20. G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *Software Engineering*, 24(6):401–419, 1998.
21. Stony Brook University. GCC Open-Source Software Model-Checking Tool Kit. `http://www.cs.sunysb.edu/~gmc`.
22. M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 332–344, 1986.