A Tool Integrating Model Checking Into a C Verification Toolset

Subash Shankar and Gilbert Pajela

City University of New York (CUNY)
`subash.shankar@hunter.cuny.edu gpajela@gradcenter.cuny.edu`

Abstract. Frama-C is an extensible C verification framework that includes support for abstract interpretation and deductive verification. We have extended it with model checking based on counterexample guided refinement. This paper discusses our tool and outlines the major challenges faced here, and likely to be faced in other similar tools.

1 Introduction and Motivation

Program verification has a long history with a more recent growth in tools for semi-automatic and automatic verification, even though the general problem is undecidable. Three major underlying approaches are abstract interpretation [8], Floyd-Hoare logic along with its weakest precondition interpretation [10, 11, 9], and model checking. Unfortunately, no one approach can verify all programs in practice, with major tradeoffs including automatability, generality, scalability, and efficiency. In particular, while deductive verification techniques require manual guidance (*e.g.*, to identify loop invariants) but can be used for all programs (given a suitably powerful theorem prover), model checking is automatic but suffers from state space explosion.

The Frama-C toolset is an extensible framework that integrates multiple static analysis techniques including abstract interpretation and deductive verification, for C programs. We have implemented a prototype model checking plugin to Frama-C that allows the user to mix-and-match all of these verification techniques. The end goal is to provide a software verification system that can exploit the benefits of all three underlying approaches in a convenient and integrated manner so program parts can be verified using the most appropriate approach. We believe this is the first tool to combine these approaches.

2 Frama-C Overview

Frama-C is a platform for static analysis of C programs, and we outline the relevant parts in this section though the reader is referred to [12] for a more extensive discussion. It is extensible through plugins that may share information and interact through a common interface. The plugins will typically interface with C Intermediate Language (CIL) and other tool results, as supported by the Frama-C kernel. All code is open-source and written in OCaml.

Frama-C analyses generally act on specifications written using the ANSI/ISO C Specification Language (ACSL) [1]. ACSL allows for specification of contracts on functions/statements, with `requires`, `ensures`, and `assigns` clauses to express pre-conditions, post-conditions, and the set of variables potentially modified by the function/statement, respectively. Clauses are standard C expressions extended with auxiliary variables `\result` and `\old(var)` corresponding to a function return value and the pre-state value of `var`, respectively.

Frama-C comes with a number of plugins, and we are primarily interested in interfacing with two of these: `value` and `wp`. Value analysis applies forward dataflow analysis on domain-dependent abstract interpretation lattices to compute conservative approximations to all variable values. Some typical abstractions include intervals and mod fields for integers, intervals for reals, offsets into memory regions for pointers, etc. Loops must be unrolled by a constant user-selected number of iterations, which unfortunately may not be efficient for large iteration counts. It is possible (through an undocumented/unrecommended option) to perform unbounded loop unrolling, but this results in a potentially non-terminating fixed point computation. The `wp` plugin performs deductive verification based on Dijkstra's weakest precondition calculus. As with all deductive verification techniques, there are limitations imposed by undecidability and the capabilities of underlying backend engines (SMT solvers and/or proof assistants). Additionally, loops are problematic since they require the manual identification of loop invariants, and it is generally recognized that software developers are not typically adept at identifying sufficiently strong invariants.

3 Model Checking for Software Verification

Traditional model checking automatically verifies liveness/reachability and safety properties expressed in temporal logic on a state machine representing the system being verified. Since an explicit representation of the state machine is often impractical, symbolic model checking uses a symbolic representation and has been used to verify very large systems [4]. However, even small programs lead to huge state spaces, and its use is thus limited.

Counter-example guided refinement (CEGAR) alleviates this problem by applying predicate abstraction to construct and verify a Boolean program abstracting the original program [6]. Initially, the predicates used for abstraction are typically either null (thus, abstracting the program into its control flow graph) or a subset of conditions in the program/contract. If the property is verifiable in the abstraction, it must be true; otherwise, the produced counterexample is validated on the original program. If validation fails (*i.e.*, the counterexample was spurious), the counterexample is analyzed to produce additional new predicates for refining the abstraction. This verify-validate-refine cycle is iterated until the property is proven, hopefully within a reasonable number of iterations.

Two common CEGAR-based tools for C program verification are SATABS [5] and Blast [2] which is now extended and embodied in the CPAchecker tool [3]. Both augment C with a `__VERIFIER_assume(expr)` statement that restricts the state space to paths in which `expr` is true (at the point of the statement), and both can be used to verify C assertions. CPAchecker is a configurable tool that allows for multiple analysis techniques, mostly related to reachability analysis. Configurations differ on underlying assumptions such as the approximation of C data types with mathematical types. CEGAR tool performances vary due largely to differing refinement strategies, and the approach in our plugin is to allow multiple user-selectable CEGAR backends. Since we wish to interact with other Frama-C tools that may be strict, we use a conservative configuration that does reachability analysis on bit-precise approximations (named `predicateAnalysis-bitprecise`), and all further mentions of CPAchecker in this paper should be understood to refer to this configuration.

4 The `cegarmc` Plugin

Our plugin, called `cegarmc`[1], verifies statements (which may of course contain arbitrarily nested statements) using SATABS and CPAchecker backends called through the Frama-C GUI. `Cegarmc` currently supports the following C-99 and ACSL constructs:

- Variables/Types: Scalars including standard variations of integers and floats, arrays, structs/unions, and pointers (to these). Automatic and static storage classes are both supported, but type attributes (*e.g.*, for alignment, storage) are not supported.
- Statements: all constructs excluding exceptions. This includes function calls.
- ACSL: Statement contracts containing `ensures` and `requires` calls. For inter-procedural verification (discussed later), we also support function contracts in called functions along with assigns clauses.

These form a fairly complete C subset, though there is in principle no reason why other constructs can't be supported (if supported by a CEGAR tool).

`Cegarmc` functions by translating the CIL representation of the statement being verified along with its ACSL contract into an equivalent well-formed single-function C program that can be verified by SATABS or CPAchecker. Figure 1 illustrates the resulting architecture. Frama-C includes a mechanism for maintaining/combining validity statuses for contracts (possibly from multiple analyses) along with dependencies between contracts [7], and `cegarmc` emits a 'true' or 'dont know' status depending on results.

Figure 2 illustrates an abstract statement and its translation, where S' is essentially the CIL version of S. Each variable that appears in S is declared in the same order (thus ensuring parsability), though not necessarily contiguously (see Section 4.1 for a discussion of resulting ramifications with respect to memory models). The labels CMCGOODEND and CMCBADEND capture normal and abnormal termination of S respectively, and S' also replaces abnormal terminations with branches to CMCBADEND (since ACSL statement contracts

[1] The tool is open-source, under standard GPG License, and available at `http://www.compsci.hunter.cuny.edu/~sshankar/cegarmc.html`

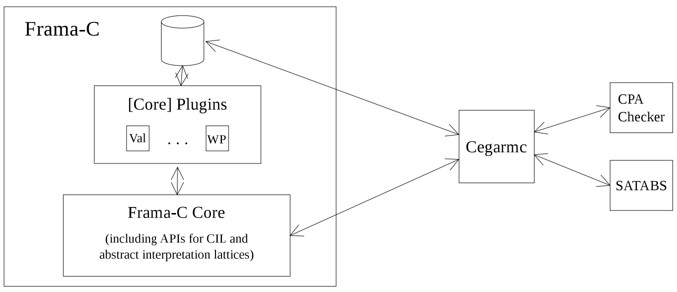


Fig. 1: System Architecture

Original Statement	**Translation**

`/*@ requires R;`
`    ensures E;`
`*/`
`S;`
	`Declarations`
	`__VERIFIER_assume(R);`
	`S';`
	`CMCGOODEND:`
	`  assert(E);`
	`CMCBADEND: return;`

Fig. 2: Translation

don't apply to abnormal terminations). Multiple requires clauses are translated to multiple assumes clauses. If there are multiple ensures clauses, this translation is repeated for each one, calling the CEGAR checker once per clause. It is easy to see that this simple translation is sound.

Inter-procedural verification is substantially more complicated. Model checkers require callee expansion, resulting in state space explosion. Assuming a contract can be written for the callee, our approach exploits this contract to implement a form of assume-guarantee reasoning, thus avoiding state space explosion. Our basic approach is to automatically replace function calls with assumes clauses capturing the corresponding contract. Figure 3 illustrates an abstract example of this translation for the non-void 1-argument side-effect-free case, where P[x:=y] is the substitution operator that replaces all free occurrences of x in P with y. If there are multiple [syntactic] instances of calls to foo in S, distinct

Program Fragment

```
// S's body:
...
S1 // calls foo(actual);
...

/*@
  requires R2;
  assigns A2;
  ensures E2;
*/
SomeType foo(formal) {
  ...
};
```

Translation (of S1)

```
...
SomeType CMCfoo;
__VERIFIER_assume(
  !R2[formals:=actuals] ||
  E2[\result:=CMCfoo][\old(formal):=actual])
S1'[foo(actual):=CMCfoo]
...
```

Fig. 3: Inter-Procedural Translation Example

identifiers are given to each call variable (*e.g.*, CMCfoo1, CMCfoo2, . . .) – note that multiple calls themselves (*e.g.*, in a loop) are only given one variable since they are declared in a local scope/lifetime. The extensions to multi-argument and void functions are simple to see. Any proofs of S's contract are marked as conditional on foo's contract; thus, vacuous local proofs of S are possible, though the global proof would still fail since foo's contract would be false.

However, this is complicated by side-effects arising from interference between the statement and called function (*e.g.*, assigning of a static global variable). `Cegarmc` also checks for such interferences using ACSL assigns clauses to identify potentially modified variables, and proceeds with the proof only if no potential interference is found. Additionally, if no assigns clause is present, `cegarmc` attempts to determine modified variables and marks resulting proofs conditional on independence (which may be proven separately).

4.1 `Cegarmc` Issues:

There are numerous complicating issues addressed in the engineering of `cegarmc`. We believe these issues are also likely to be faced by other such tools. The major such issues are highlighted below.

Tool Philosophy: Verification tools differ on whether analyses are guaranteed correct or merely approximations, and combination techniques additionally may be based on confidences/probabilities assigned to the tools. Frama-C's combination algorithm assumes all analyses are correct, and its analyses combination algorithms result in inconsistent statuses if, for example, two plugins emit different statuses for the same contract. In contrast, many CPAchecker analyses use approximations (*e.g.*, rationals for integers) for improved efficiency. Since `cegarmc` is intended to perform seamlessly in the Frama-C platform, it uses only sound tools/configurations where possible and provides feedback otherwise (though constrained by information available in tool documentation).

Language Semantics: Whereas Frama-C supports C-99, SATABS and CPAchecker are based on ANSI C and C-11, respectively. `Cegarmc` does not account for any resulting semantic issues, and is thus not suitable for verifying any program relying on a the intricacies of a particular C standard. Syntactically, `cegarmc` only supports C-99 constructs. This [typically unstated] issue is faced by all verification tools, and even within the CEGAR tools themselves since they may use other C-targeted tools.

Memory Model: Any analysis of programs with pointers (or more precisely, pointer arithmetic) is dependent on the underlying memory model. `Cegarmc` is by its nature restricted to supporting the most restrictive memory model of tools that it interfaces with. Thus, it uses the memory model of Frama-C's value analysis, which assumes that each declared variable defines exactly one distinct base address, and a pointer is not allowed to 'jump' across base addresses (though it may, of course, still point to different elements in the same array or struct/union). Value analysis also generates proof obligations capturing such conditions, which may be independently proven. CEGAR tools also make such assumptions, though they may simply produce unsound results or be unable to prove a valid contract instead of producing a proof conditional on the obligations. Note that with this memory model, `cegarmc` need not preserve relative memory addresses (as discussed in Section 4).

Efficiency: Our goal in `cegarmc` (at least in the initial prototype) is to integrate existing CEGAR-based model checkers into a verification toolset, enabling further research in integrated multi-technique verification. Since model checking efficiency is determined primarily by the backend CEGAR tools, the appropriate measure of efficiency is the number of extra variables added by our translation. `Cegarmc` adds extra variables only for auxiliary ACSL variables and function calls (see Sections 2 and 4, respectively), and is thus unlikely to significantly aggravate state space explosion.

Contextual Verification: Strictly speaking, a statement contract is a standalone entity, and all information about the statement's initial state should be reflected in its requires clauses. However, this complicates the verification process in practice as users may wish to write statement contracts without detailing all initial state information. `Cegarmc` optionally supports such a model, with the caveat that any changes in the program may invalidate existing `cegarmc` proofs.

5 Conclusions and Further Research

As mentioned earlier, the `cegarmc` prototype covers a fairly complete C subset. Its performance is almost completely dependent on that of the CEGAR model checker (which is in general highly variable), and `cegarmc` does not add inefficiencies. Although our primary goal is to enable the convenience of model checking in a powerful multi-approach system, we believe that we have also increased the power of CEGAR tools. In particular, contextual verification allows for CEGAR verification of program parts within procedures, while our inter-procedural approach enables verification without the typical state space explosion.

`Cegarmc` is also a framework for much further research. In particular, we plan on integrating different verification approaches to: 1) more fully automate the integration of deductive verification and model checking, 2) exploit abstract interpretation and deductive verification techniques to configure CEGAR tools for better performance, and 3) combine partial results from different techniques for more complete verification.

Acknowledgements

This project was partially supported by a Foreign Guest Researcher grant from the Digiteo Foundation in Saclay, France, and a PSC-CUNY research grant. Much thanks is due to Zachary Hutchinson, who contributed to some parts of the code. We would also like to thank the entire Frama-C team for invaluable guidance without which this tool would not have been possible.

References

1. P. Baudin, P. Cuoq, J.-C. q Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. ACSL: ANSI/ISO C specification language, version 1.8.
2. D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Computer Aided Verification (CAV)*, pages 504–518, 2007.
3. D. Beyer and M. E. Keremoglu. CPAChecker: A tool for configurable software verification. In *Computer Aided Verification (CAV)*, pages 184–190, 2011.
4. J. Burch, E. M. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10e20 states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 428–439, 1990.
5. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 570–574, 2005.
6. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification (CAV)*, pages 154–169, 2000.
7. L. Correnson and J. Signoles. Combining analyses for c program verification. In *International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, pages 108–130, 2012.
8. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.
9. E. W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of program. *Communications of the ACM (CACM)*, 18(8):453–457, August 1975.
10. R. Floyd. Assigning meanings to programs. *Proceedings of Symposia in Applied Mathematics*, 19:19–32, 1967.
11. C. Hoare. An axiomatic basic for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.
12. F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c, a software analysis perspective. *Formal Aspects of Computing*, 27:573–609, March 2015.

A Tool Integrating Model Checking Into a C Verification Toolset Oral Presentation Plan

Subash Shankar and Gilbert Pajela

City University of New York (CUNY)
`subash.shankar@hunter.cuny.edu` `gpajela@gradcenter.cuny.edu`

Abstract. This discusses an informal plan for oral presentation of the tool as requested in the call for papers.

Outline of Presentation

The presentation consists of the following parts with estimated time in parentheses:

1. (15%) An outline of major verification approaches along with the pros/cons of each to motivate the need for model checking. This includes a brief description of the tools Frama-C/ACSL, CPAchecker, and SATABS, focusing on portions used in our approach.
2. (30%) The implementation of our tool, including the translations, especially for the interprocedural case.
3. (40%) Examples that illustrate the basics as well as some of the issues discussed in the paper. In particular, this includes contextual verification and interprocedural verification. These examples will be run live using the tool, and are listed below.
4. (15%) Outline of ongoing and future research by us. We also want to emphasize that the tool provides a framework for others to research the integration of multiple verification approaches, and will mention helpful usage information.

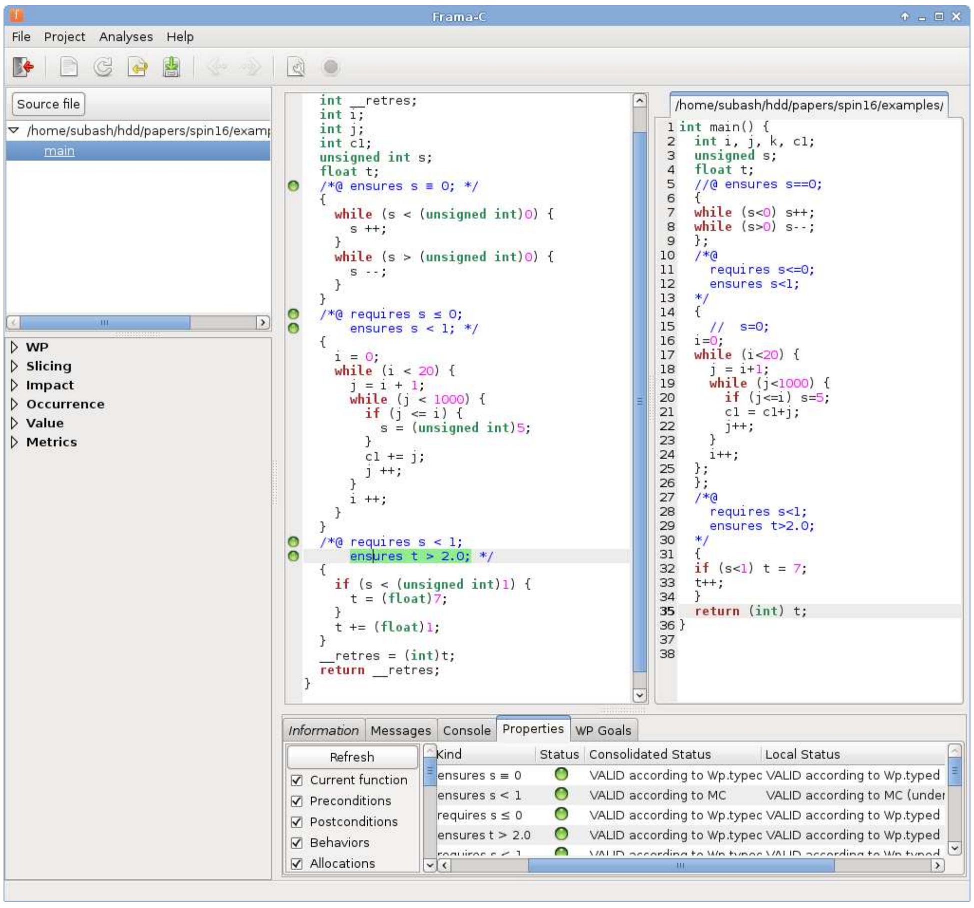
Examples

The examples alluded to above are:

1. A simple nested loop example. The goal of this example is to show the tool interface. The properties to be proven are simple with CEGAR, while value analysis would require unrealistically many unrollings and weakest preconditions would require the manual identification of loop invariants.

2. An example that illustrates the verification of a multi-statement program using multiple approaches. One loop would be proven with our tool, while some straight-line arithmetic code would be proven using deductive verification. The resulting proofs are interdependent (and marked so), and the demo would also show these dependences and how they are automatically discharged after all proofs are complete. For example, Figure 1 illustrates a program with 3 statements, where the second contract is proven by `cegarmc` while the others are proven by `wp`. The resulting proof dependence tree is shown in Figure 2.
3. An example that illustrates the use of contextual verification. This example will be a statement that is false by itself but true in the context of the program. Value analysis will be used to generate a context, and the statement will be proven using `cegarmc`. Of course, the statement may be something that isn't provable using `value` or `wp` (as shown above).
4. An example that illustrates how our compositional approach to interprocedural verification is more powerful than what model checkers can do by themselves. Figure 3 shows one such example. The verification of the statement contract in main requires .503 s. and 8 CEGAR iterations using CPAchecker, while it would have taken 25.097 s. and 16 CEGAR iterations for the entire program (both on the same machine/settings). The proof is of course conditional on foo's proof, but that is easy to verify by either `wp` (if a loop invariant is supplied) or `cegarmc` (if an equivalent statement contract on the function's body is stated). While this 50-fold improvement should not be surprising (and can be made arbitrarily larger depending on n's value), it illustrates the convenience of automatically breaking down a complicated inter-procedural verification into manageable smaller verification tasks.

Depending on the allotted time, some of these examples may be combined.



```
int __retres;
int i;
int j;
int cl;
unsigned int s;
float t;
/*@ ensures s ≡ 0; */
{
  while (s < (unsigned int)0) {
    s ++;
  }
  while (s > (unsigned int)0) {
    s --;
  }
}
/*@ requires s ≤ 0;
    ensures s < 1; */
{
  i = 0;
  while (i < 20) {
    j = i + 1;
    while (j < 1000) {
      if (j <= i) {
        s = (unsigned int)5;
      }
      cl += j;
      j ++;
    }
    i ++;
  }
}
/*@ requires s < 1;
    ensures t > 2.0; */
{
  if (s < (unsigned int)1) {
    t = (float)7;
  }
  t += (float)1;
}
__retres = (int)t;
return __retres;
}
```

```
int main() {
  int i, j, k, cl;
  unsigned s;
  float t;
  //@ ensures s==0;
  {
  while (s<0) s++;
  while (s>0) s--;
  };
  /*@
    requires s<=0;
    ensures s<1;
  */
  {
    //  s=0;
  i=0;
  while (i<20) {
    j = i+1;
    while (j<1000) {
      if (j<=i) s=5;
      cl = cl+j;
      j++;
    }
    i++;
  };
  };
  /*@
    requires s<1;
    ensures t>2.0;
  */
  {
  if (s<1) t = 7;
  t++;
  }
  return (int) t;
}
```

☑ Current function
☑ Preconditions
☑ Postconditions
☑ Behaviors
☑ Allocations

Kind	Status	Consolidated Status	Local Status
ensures s ≡ 0	●	VALID according to Wp.typec	VALID according to Wp.typed
ensures s < 1	●	VALID according to MC	VALID according to MC (under
requires s ≤ 0	●	VALID according to Wp.typec	VALID according to Wp.typed
ensures t > 2.0	●	VALID according to Wp.typec	VALID according to Wp.typed
requires s < 1	●	VALID according to Wp.typec	VALID according to Wp.typed

Fig. 1: Example

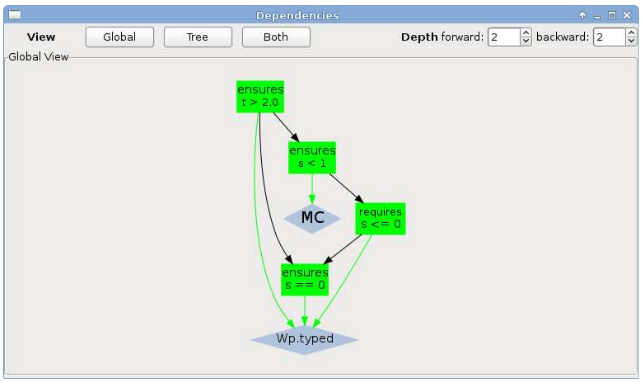


Fig. 2: Proof Dependence Tree

```
/*@ requires k>=1;
    ensures \result == 2*k;
@*/
int foo (int k) {
  int i, s;
  s = 0;
  for (i=1; i <= k; i++)
    s += 2;
  return s;
}

void main() {
  int i, n, s;
  //@ ensures s == n*(n+1);
  {
    n = 5;
    s = 0;
    for (i=1; i <= n; i++)
      s += foo(i);
  }
}
```

Fig. 3: Example - Interprocedural Verification