A Tool Integrating Model Checking into a C Verification Toolset

Subash Shankar(✉) and Gilbert Pajela

City University of New York (CUNY), New York, USA
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 [9], deductive verification based on Floyd-Hoare logic along with its weakest precondition interpretation [10–12], and model checking [7]. 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 deep user understanding to provide manual guidance (*e.g.*, to identify loop invariants) but can be used for all programs (given a suitably powerful theorem prover), model checking is completely automatic but suffers from state space explosion. Given the pros and cons of each technique, it is desirable to integrate them, enabling a 'verification engineer' to select tools as appropriate.

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, and the results integrated in a seamless fashion. 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 [13] for a more

D. Bošnački and A. Wijs (Eds.): SPIN 2016, LNCS 9641, pp. 214–224, 2016.
DOI: 10.1007/978-3-319-32582-8_15

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 and statements (among other features), and we support three types of clauses in contracts:

- `requires`: pre-condition for the contract
- `ensures`: post-condition for the contract. When in a statement contract, it is conditional on normal termination; that is, not through a `goto`, `break`, `continue`, `return`, or `exit` statement (ACSL is a rich language that also includes analogs for the abnormal termination case, but we currently do not support these).
- `assigns`: The set of variables potentially modified by the statement/function (including those modified on abnormal execution paths). If there is no assign statement, any variable is assumed to be potentially modified.

Arguments for `requires` and `ensures` clauses are standard C expressions with numerous extensions to support simpler expression of properties. In particular, we support two functions:

- `\result`: evaluates to the return value of a function
- `\at(e,id)`: evaluates to the value of the expression `e` at label `id`, where the label may be in the C program or one of 6 ACSL-defined labels. We support 4 predefined labels:
  - `Pre`: The prestate of the function, when in a statement contract.
  - `Post`: The poststate of the contract (visible in ensures clauses).
  - `Old`: The prestate of the contract (visible in ensures clauses).
  - `Here`: the prestate when in a requires clause, and the poststate when in an ensures clause.

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 to perform unbounded loop unrolling, but this results in a potentially non-terminating fixed point computation and is thus not recommended. 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 direct use is thus limited. For example, a program with just 10 32-bit variables requires $\sim 10^{96}$ states, which approaches the limits of symbolic model checking.

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 (see Fig. 1), hopefully within a reasonable number of iterations.

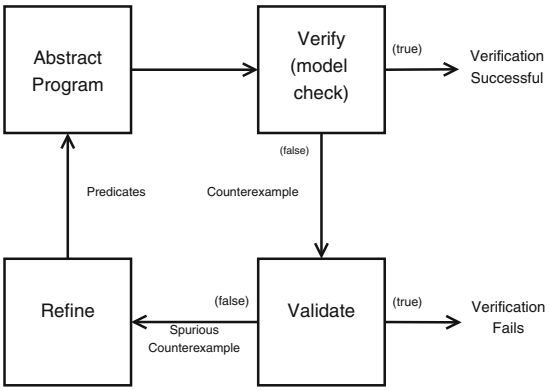

Fig. 1. CEGAR algorithm

There are several CEGAR-based tools for C program verification, with 2 common ones being 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 C99 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. 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, with clauses that may be C expressions possibly using the ACSL functions mentioned in Sect. 2. For inter-procedural verification (discussed later), we also utilize function contracts in called functions along with assigns clauses. We do not verify function contracts themselves, since Frama-C can handle those given proofs of individual statement contracts.

These form a fairly complete C and ACSL subset, though there is in principle no reason why other constructs can't be supported (provided they have well-defined semantics across C standards, and a CEGAR backend supports them).

4.1 `Cegarmc` Implementation

`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 2 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 [8], and `cegarmc` emits a 'true' or 'dont know' status depending on results.

Figure 3 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

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

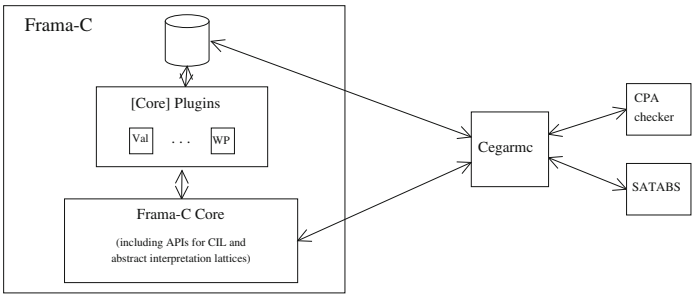

Fig. 2. System architecture

(see Sect. 4.2 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 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.

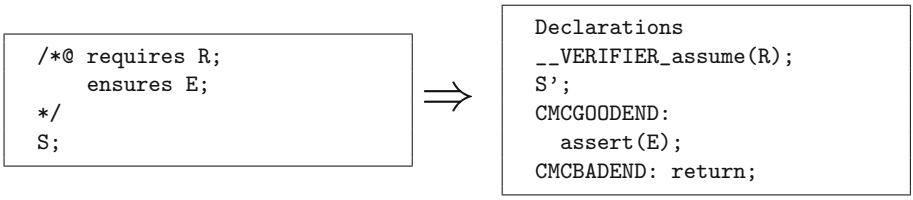

Fig. 3. Translation of statement

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 4 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 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.

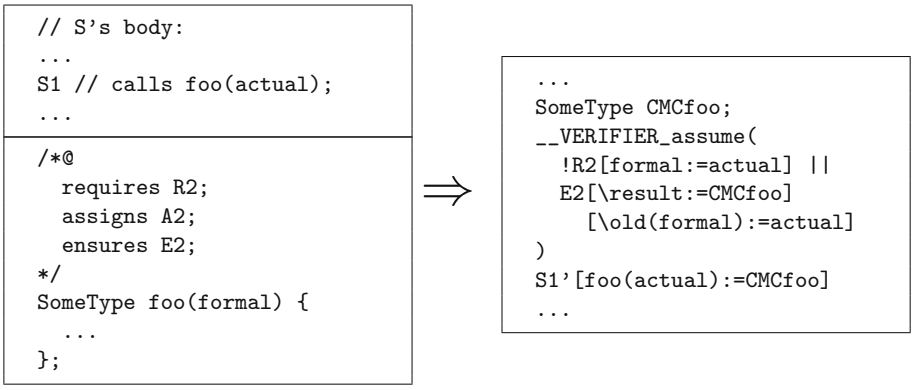

Fig. 4. Inter-procedural translation of S

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).

Figure 5 illustrates the algorithm for handling such interferences, where Θ(S,C) denotes the transformation of S illustrated in Fig. 4. The algorithm works by first identifying which variables are modified by S and any functions called in S's body, in a manner consistent with ACSL semantics for assigns clauses. This essentially amounts to using the assigns clause when one is available, and analyzing the code to determine assigned variables otherwise. Then, it checks for side effects of called functions that may potentially interfere, and aborts if so (this case results in the desired property marked with a 'dont know' status). We currently do not support pointers in assigns clauses, though there may be side effects from, for example, static variables. Finally, it calls the CEGAR checker on the transformed program. Frama-C introduces the notion of emitters, whereby a proof is marked with the name of the supporting tool, and we exploit this to mark proofs as proven by MC or MC.ind depending on whether an independence assumption needed to be made. This latter case occurs when the called function has a null body (*i.e.*, it is a stub, declared in the caller as an extern) and no assigns clause. Of course, the user may later decide to supply more information and retry for an unconditional proof.

Figure 6 illustrates this translation on a simple interprocedural program containing a statement that repeatedly decrements a positive number s until it is zero[2]. In this case, `id`'s function contract has neither an assigns clause nor a

[2] For readability reasons, only an abstracted translation is shown since the actual translation occurs at the CIL level.

Given the statement and functions (a schema is shown):

```
/*@                                  /*@
  requires R1;                       requires R2;
  assigns A1;                        assigns A2;
  ensures E1;                        ensures E2;
*/                                   */
S; // calls foo                      foo(Formals) { ... };
```

Translation:

Rename multiple calls to the same function with distinct identifiers
IndepFlag ← false
if A1 = ∅ **then**
  A1 ← variables modified in S
end if
for all calls foo(actuals) ∈ S (denote the call C) **do**
  if foo's body is null and A2 = ∅ **then**
    IndepFlag ← true
  else if foo has a body and A2 = ∅ **then**
    A2 ← variables modified in foo
  end if
  if A1 ∩ A2 ≠ ∅ **then**
    abort "Interference with Called Function"
  end if
  S ← $\Theta(S, C)$
end for
if not IndepFlag **then**
  Call CEGAR checker on program S with emitter MC
else
  Call CEGAR checker on program S with emitter MC.Ind
end if

Fig. 5. Translation for inter-procedural code

body, and `cegarmc` thus marks the proof with an MC.ind emitter. If the body for `id` (or an assigns clause) had been supplied, `cegarmc` would have been able to determine that there is no interference, and the proof would have been emitted with an MC emitter. In either case, the proof of the statement contract would be marked as conditional on the proof of `id`'s function contract, which could be proved using either `cegarmc` or other Frama-C tools such as `wp`.

4.2 `Cegarmc` Issues

Although `cegarmc`'s implementation is conceptually simple, there are numerous semantic issues to ensure soundness. Additionally, there are numerous complicating underlying issues, and we highlight the major ones below. We believe these issues are also likely to be faced by other such tools.

Sample Program

```
/*@ requires n>=0;
    ensures \result == n;
*/
int id (int n);

void main() {
  int s,k;
  /*@ requires s>0;
      ensures s==0;
  */
  {
    k=1;
    while (s>0)
      s = s-id(k);
  }
}
```

Translation

```
void main() {
  auto int s,k;
  __VERIFIER_assume(s > 0);
  k=1;
  while (s>0)
    int CMCid1;
    __VERIFIER_assume(
      (!(k >= 0) ||
       (CMCid1 == k)));
    s = s-CMCid1;
    }
CMCGOODEND:
  assert(s == 0);
CMCBADEND: return;
}
```

Fig. 6. Example program and (abstracted) translation

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 C99, SATABS and CPAchecker are based on ANSI C and C11, respectively. `Cegarmc` does not account for any resulting semantic issues, and is thus not suitable for verifying any program relying on the intricacies of a particular C standard. Syntactically, `cegarmc` only supports C99 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 with the translation algorithm).

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. The only constructs for which extra variables are added are:

- Function calls: each function call results in one new variable of the return value type. Note that these variables may be local to a scope inside the verified statement, thus creating (and destroying) one new variable for each entry to the scope (*e.g.*, one per loop iteration).
- ACSL labels: For each supported label, one additional variable of the same type.

Thus, `cegarmc` is unlikely to significantly aggravate state space explosion.

4.3 Integration of Approaches

One of the major advantages of our approach is the integration of multiple verification techniques, and we believe our tool can provide a framework for research into various such types of integration. Indeed, the interprocedural approach outlined previously may be considered as a simple integration of deductive verification and model checking if the called function is verified using `wp` and the calling statement is verified using `cegarmc`.

We have also implemented one simple integration of abstract interpretation (`value`) and `cegarmc`, which we dub contextual verification. A function will typically contain multiple statements (say, statements $S_1,\ldots,S_n$), and the composition may not be provable (or take too long) using a pure CEGAR approach, partly because different predicate abstractions are needed for each constituent statement. However, value analysis may be used to determine the values of all variables in the initial state of S_k, and `cegarmc` can then be used to verify S_k's contract under this context. This is implemented through a user-selectable option, in which case `cegarmc` queries Frama-C's internal databases (see Fig. 2) for `value` results that can be used to compute S_k's initial state.

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. Thus, we use a different emitter to indicate that the proof is contextual. When reviewing the final proof, a user must then ensure that value analysis was performed before the proof (or rerun value analysis and any `cegarmc` proofs marked as contextual).

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 significant 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.

We believe that `cegarmc` is a framework for research into integrating verification approaches, both by us and others. 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 Digiteo Foreign Guest Research Grant 2013-0376D and PSC-CUNY Grant 67776-00-45. 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. Baudin, P., Cuoq, P., Filliâtre, J.-C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C specification language, version 1.8
2. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: concretizing the convergence of model checking and program analysis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 504–518. Springer, Heidelberg (2007)
3. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
4. Burch, J., Clarke, E.M., McMillan, K., Dill, D., Hwang, L.: Symbolic model checking: 10e20 states and beyond. In: Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science (LICS), pp. 428–439 (1990)
5. Clarke, E., Kroning, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
6. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
7. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)

8. Correnson, L., Signoles, J.: Combining analyses for C program verification. In: Stoelinga, M., Pinger, R. (eds.) FMICS 2012. LNCS, vol. 7437, pp. 108–130. Springer, Heidelberg (2012)
9. Cousot, P., Cousot, R.: 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), pp. 238–252 (1977)
10. Dijkstra, E.W.: Guarded commands, nondeterminacy, and formal derivation of program. Commun. ACM (CACM) **18**(8), 453–457 (1975)
11. Floyd, R.: Assigning meanings to programs. Proc. Symp. Appl. Math. **19**, 19–32 (1967)
12. Hoare, C.: An axiomatic basic for computer programming. Commun. ACM **12**(10), 576–580 (1969)
13. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c, a software analysis perspective. Form. Asp. Comput. **27**, 573–609 (2015)
14. Shankar, S.: A tool for integrating abstract interpretation, model checking, and deductive verification (presentation). In: Clarke Symposium: Celebrating 25 Years of Model Checking, Pittsburgh (2014)