VeriMAP: A Tool for Verifying Programs through Transformations

Emanuele De Angelis[1]*, Fabio Fioravanti[1],
Alberto Pettorossi[2], and Maurizio Proietti[3]

[1] DEC, University 'G. D'Annunzio', Pescara, Italy
{emanuele.deangelis,fioravanti}@unich.it
[2] DICII, University of Rome Tor Vergata, Rome, Italy
pettorossi@disp.uniroma2.it
[3] IASI-CNR, Rome, Italy
maurizio.proietti@iasi.cnr.it

Abstract. We present VeriMAP, a tool for the verification of C programs based on the transformation of constraint logic programs, also called constrained Horn clauses. VeriMAP makes use of Constraint Logic Programming (CLP) as a metalanguage for representing: (i) the operational semantics of the C language, (ii) the program, and (iii) the property to be verified. Satisfiability preserving transformations of the CLP representations are then applied for generating verification conditions and checking their satisfiability. VeriMAP has an interface with various solvers for reasoning about constraints that express the properties of the data (in particular, integers and arrays). Experimental results show that VeriMAP is competitive with respect to state-of-the-art tools for program verification.

1 The Transformational Approach to Verification

Program verification techniques based on *Constraint Logic Programming* (CLP), or equivalently *constrained Horn clauses* (CHC), have gained increasing popularity during the last years [2,4,9,18]. Indeed, CLP has been shown to be a powerful, flexible metalanguage for specifying the program syntax, the operational semantics, and the proof rules for many different programming languages and program properties. Moreover, the use of the CLP-based techniques allows one to enhance the reasoning capabilities provided by Horn clause logic by taking advantage of the many special purpose solvers that are available for various data domains, such as integers, arrays, and other data structures.

Several verification tools, such as ARMC [19], Duality [16], ELDARICA [13], HSF [8], TRACER [14], μZ [12], implement reasoning techniques within CLP (or CHC) by following approaches based on *interpolants*, *satisfiability modulo theories*, *counterexample-guided abstraction refinement*, and *symbolic execution* of CLP programs.

Our tool for program verification, called VeriMAP, is based on *transformation* techniques for CLP programs [5,4]. The current version of the VeriMAP can be

* Supported by the National Group of Computing Science (GNCS-INDAM).

used for verifying safety properties of C programs that manipulate integers and arrays. We assume that: (i) a safety property of a program P is defined by a pair $\langle \varphi_{init}, \varphi_{error} \rangle$ of formulas, and (ii) safety holds iff no execution of P starting from an initial configuration that satisfies φ_{init}, terminates in a final configuration that satisfies φ_{error}.

From the CLP representation of the given C program and of the property, VeriMAP generates a set of *verification conditions* (VC's) in the form of CLP clauses. The VC generation is performed by a transformation that consists in specializing (with respect to the given C program and property) a CLP program that defines the operational semantics of the C language and the proof rules for verifying safety. Then, the CLP program made out of the generated VC's is transformed by applying unfold/fold transformation rules [6]. This transformation 'propagates' the constraints occurring in the CLP clauses and derives equisatisfiable, easier to analyze VC's. During constraint propagation VeriMAP makes use of constraint solvers for linear (integer or rational) arithmetic and array formulas. In a subsequent phase the transformed VC's are processed by a *lightweight analyzer* that basically consists in a bounded unfolding of the clauses. Since safety is in general undecidable, the analyzer may not be able to detect the satisfiability or the unsatisfiability of the VC's and, if this is the case, the verification process continues by iterating the transformation and the propagation of the constraints in the VC's.

The main advantage of the transformational approach to program verification over other approaches is that it allows one to construct highly parametric, configurable verification tools. In fact, one could modify VeriMAP so as to deal with other programming languages, different language features, and different properties to be proved. This modification can be done by reconfiguring the individual modules of the tool, and in particular, (i) by replacing the CLP clauses that define the language semantics and proof rules, (ii) by designing a suitable strategy for specializing the language semantics and proof rules so as to automatically generate the VC's for any given program and property, (iii) by designing suitable strategies for transforming the VC's by plugging-in different constraint solvers and *replacement rules* (which are clause rewriting rules) depending on the theories of the data structures that are used, (iv) by replacing the lightweight analyzer currently used in VeriMAP by other, more precise analyzers available for CLP programs. These module reconfigurations may require considerable effort (and this is particularly true for the design of the strategies of Point (iii)), but then, by composing the different module versions we get, we will have at our disposal a rich variety of powerful verification procedures.

Another interesting feature of the transformational approach is that at each step of the transformation, we get a set of VC's which is equisatisfiable with respect to the initial set. This feature allows us both (i) to compose together various verification strategies, each one being expressed by a sequence of transformations, and (ii) to use VeriMAP as a front-end for other verifiers (such as those we have mentioned above) that can take as input VC's in the form of CLP clauses. Finally, the use of satisfiability preserving transformations eases

the task of guaranteeing that VeriMAP computes sound results, as the soundness of the transformation rules can be proved once and for all, before performing any verification using VeriMAP.

2 The VeriMAP Tool: Architecture and Usage

Architecture. The VeriMAP tool consists of three modules (see Figure 1). (1) A *C-to-CLP Translator* (*C2CLP*) that constructs a CLP encoding of the C program and of the property given as input. *C2CLP* first translates the given C program into CIL, the C Intermediate Language of [17]. (2) A *Verification Conditions Generator* (*VCG*) that generates a CLP program representing the VC's for the given program and property. The *VCG* module takes as input also the CLP representations of the operational semantics of CIL and of the proof rules for establishing safety. (3) An *Iterated Verifier* (*IV*) that attempts to determine whether or not the VC's are satisfiable by iteratively applying unfold/fold transformations to the input VC's, and analyzing the derived VC's.

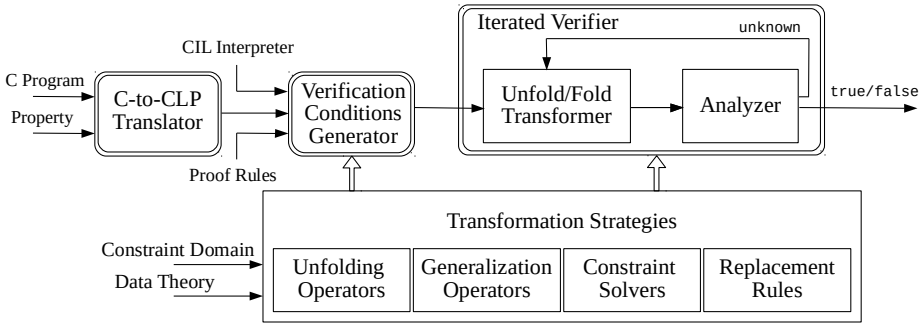

Fig. 1. The VeriMAP architecture.

The *C2CLP* module is based on a modified version of the CIL tool [17]. This module first parses and type-checks the input C program, annotated with the property to be verified, and then transforms it into an equivalent program written in CIL that uses a reduced set of language constructs. During this transformation, in particular, commands that use *while*'s and *for*'s are translated into equivalent commands that use *if-then-else*'s and *goto*'s. This transformation step simplifies the subsequent processing steps. Finally, *C2CLP* generates as output the CLP encoding of the program and of the property by running a custom implementation of the CIL visitor pattern [17]. In particular, for each program command, *C2CLP* generates a CLP fact of the form `at(L,C)`, where `C` and `L` represent the command and its label, respectively. *C2CLP* also constructs the clauses for the predicates `phiInit` and `phiError` representing the formulas φ_{init} and φ_{error} that specify the safety property.

The *VCG* module generates the VC's for the given program and property by applying a program specialization technique based on equivalence preserving unfold/fold transformations of CLP programs [6]. Similarly to what has been proposed in [18], the *VCG* module specializes the interpreter and the proof rules

with respect to the CLP representation of the program and safety property generated by *C2CLP* (that is, the clauses defining `at`, `phiInit`, and `phiError`). The output of the specialization process is the CLP representation of the VC's. This specialization process is said to 'remove the interpreter' in the sense that it removes every reference to the predicates used in the CLP definition of the interpreter in favour of new predicates corresponding to (a subset of) the 'program points' of the original C program. Indeed, the structure of the call-graph of the CLP program generated by the *VCG* module corresponds to that of the control-flow graph of the C program.

The *IV* module consists of two submodules: (i) the *Unfold/Fold Transformer*, and (ii) the *Analyzer*. The *Unfold/Fold Transformer* propagates the constraints occurring in the definition of `phiInit` and `phiError` through the input VC's thereby deriving a new, equisatisfiable set of VC's. The *Analyzer* checks the satisfiability of the VC's by performing a lightweight analysis. The output of this analysis is either (i) `true`, if the VC's are satisfiable, and hence the program is safe, or (ii) `false`, if the VC's are unsatisfiable, and hence the program is unsafe (and a counterexample may be extracted), or (iii) `unknown`, if the lightweight analysis is unable to determine whether or not the VC's are satisfiable. In this last case the verification continues by iterating the propagation of constraints by invoking again the *Unfold/Fold Transformer* submodule. At each iteration, the *IV* module can also apply a *Reversal* transformation [4], with the effect of reversing the direction of the constraint propagation (either from `phiInit` to `phiError` or vice versa, from `phiError` to `phiInit`).

The *VCG* and *IV* modules are realized by using MAP [15], a transformation engine for CLP programs (written in SICStus Prolog), with suitable concrete versions of *Transformation Strategies*. There are various versions of the transformation strategies which, as indicated in [4], are defined in terms of: (i) *Unfolding Operators*, which guide the symbolic evaluation of the VC's, by controlling the expansion of the symbolic execution trees, (ii) *Generalization Operators* [7], which guarantee termination of the *Unfold/Fold Transformer* and are used (together with widening and convex-hull operations) for the automatic discovery loop invariants, (iii) *Constraint Solvers*, which check satisfiability and entailment within the Constraint Domain at hand (for example, the integers or the rationals), and (iv) *Replacement Rules*, which guide the application of the axioms and the properties of the Data Theory under consideration (like, for example, the theory of arrays), and their interaction with the Constraint Domain.

Usage. VeriMAP can be downloaded from `http://map.uniroma2.it/VeriMAP` and can be run by executing the following command: `./VeriMAP program.c`, where `program.c` is the C program annotated with the property to be verified. VeriMAP has options for applying custom transformation strategies and for exiting after the execution of the *C2CLP* or *VCG* modules, or after the execution of a given number of iterations of the *IV* module.

3 Experimental Evaluation

We have experimentally evaluated VeriMAP on several benchmark sets. The first benchmark set for our experiments consisted of 216 safety verification problems

of C programs acting on integers (179 of which are safe, and the remaining 37 are unsafe). None of these programs deal with arrays. Most problems have been taken from the TACAS 2013 Software Verification Competition [1] and from the benchmark sets of other tools used in software model checking, like DAGGER [10], TRACER [14] and InvGen [11]. The size of the input programs ranges from a dozen to about five hundred lines of code.

In Table 1 we summarize the verification results obtained by VeriMAP and the following three state-of-the-art CLP-based software model checkers for C programs: (i) ARMC [19], (ii) HSF(C) [8], and (iii) TRACER [14] using the strongest postcondition (*SPost*) and the weakest precondition (*WPre*) options.

	VeriMAP	ARMC	HSF(C)	TRACER	
				SPost	*WPre*
correct answers	185	138	160	91	103
safe problems	154	112	138	74	85
unsafe problems	31	26	22	17	18
incorrect answers	0	9	4	13	14
missed bugs	0	1	1	0	0
false alarms	0	8	3	13	14
errors (∗)	0	18	0	20	22
timeout	31	51	52	92	77
total time	10717.34	15788.21	15770.33	27757.46	23259.19
average time	57.93	114.41	98.56	305.03	225.82

Table 1. Verification results using VeriMAP, ARMC, HSF(C), and TRACER. Time is in seconds. The time limit for timeout is five minutes. (∗) These errors are due to incorrect parsing, or excessive memory requirements, or similar other causes.

The results of the experiments show that our approach is competitive with state-of-the-art verifiers. Besides the above benchmark set, we have used VeriMAP on a small benchmark set of verification problems of C programs acting on integers and arrays. These problems include programs for computing the maximum elements of arrays and programs for performing array initialization, array copy, and array search. Also for this benchmark, the results we have obtained show that our transformational approach is effective and quite efficient in practice.

All experiments have been performed on an Intel Core Duo E7300 2.66Ghz processor with 4GB of memory running GNU/Linux, using a time limit of five minutes. The source code of all the verification problems we have considered is available at `http://map.uniroma2.it/VeriMAP`.

4 Future Work

The current version of VeriMAP deals with safety properties of a subset of the C language where, in particular, pointers and recursive procedures do not occur. Moreover, the user is only allowed to configure the transformation strategies by choosing among some available submodules for unfolding, generalization, constraint solving, and replacement rules (see Figure 1). Future work will be devoted to make VeriMAP a more flexible tool so that the user may configure other parameters, such as: (i) the programming language and its semantics, (ii) the class

of properties and their proof rules (thus generalizing an idea proposed in [9]), and (iii) the theory of the data types in use, including those for dynamic data structures, such as lists and heaps.

References

1. D. Beyer. Second Competition on Software Verification (SV-COMP 2013). *TACAS'13*, LNCS 7795, pages 594–609. Springer, 2013.
2. N. Bjørner, K. McMillan, and A. Rybalchenko. On solving universally quantified Horn clauses. *SAS'13*, LNCS 7935, pages 105–125. Springer, 2013.
3. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verification of imperative programs by constraint logic program transformation. *SAIRP'13*, *Electronic Proceedings in Theoretical Computer Science*, 129, pages 186–210, 2013.
4. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying Programs via Iterated Specialization. *PEPM'13*, pages 43–52. ACM, 2013.
5. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying Programs by Transforming Verification Conditions. *VMCAI'14*, LNCS 8318, pages 182–202, 2014.
6. F. Fioravanti, A. Pettorossi, and M. Proietti. Transformation rules for locally stratified constraint logic programs. *Program Development in Computational Logic*, LNCS 3049, pages 292–340. Springer, 2004.
7. F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Generalization strategies for the verification of infinite state systems. *Theory and Practice of Logic Programming*, 13(2):175–199, 2013.
8. S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, and A. Rybalchenko. HSF(C): A software verifier based on Horn clauses. *TACAS'12*, LNCS 7214, pages 549–551. Springer, 2012.
9. S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. *PLDI'12*, pages 405–416. ACM, 2012.
10. B.S. Gulavani, S. Chakraborty, A.V. Nori, and S.K. Rajamani. Automatically refining abstract interpretations. *TACAS'08*, LNCS 4963, pp. 443–458. Springer, 2008.
11. A. Gupta and A. Rybalchenko. InvGen: An efficient invariant generator. *CAV'09*, LNCS 5643, pages 634–640. Springer, 2009.
12. K. Hoder, N. Bjørner, and L. M. de Moura. μZ - An efficient engine for fixed points with constraints. *CAV 2011*, LNCS 6806, pages 457–462. Springer, 2011.
13. H. Hojjat, F. Konecný, F. Garnier, R. Iosif, V. Kuncak, and P. Rümmer. A verification toolkit for numerical transition systems. *FM'12*, LNCS 7436, pages 247–251. Springer, 2012.
14. J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa. TRACER: A symbolic execution tool for verification. *CAV'12*, Lecture Notes in Computer Science 7358, pages 758–766. Springer, 2012.
15. The MAP system. `http://www.iasi.cnr.it/~proietti/system.html`
16. K. L. McMillan and A. Rybalchenko. Solving constrained Horn clauses using interpolation. MSR Technical Report 2013-6, Microsoft Report, 2013.
17. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. *CC'02*, LNCS 2304, pages 209–265. Springer, 2002.
18. J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of imperative programs through analysis of Constraint Logic Programs. *SAS'98*, LNCS 1503, pages 246–261. Springer, 1998.

19. A. Podelski and A. Rybalchenko. ARMC: The logical choice for software model checking with abstraction refinement. *PADL'07*, LNCS 4354, pages 245–259. Springer, 2007.

A An example of verification using VeriMAP

In this appendix we show how to use VeriMAP on two simple examples. Let us consider the C Program listed below, stored in a file named `example1.c`. The safety property considered in this example is defined by the formulas $\varphi_{init} \equiv$ `y` $\geq$ `0` and $\varphi_{error} \equiv$ `x` $<$ `0`, which are encoded in the program as calls to the functions `__VERIFIER_assume` and `__VERIFIER_assert` according to the rules of the TACAS Verification Competition [1].

```
1  int y;
2  int incr(int z) { z = z+y; y=0; }
3  void main() {
4    int x;
5    __VERIFIER_assume(y >= 0);
6    if (x < y) {
7      x = incr(x);
8      if (x >= y)
9        goto END;
10   }
11   while (x < y)
12     x=x+1;
13   END: __VERIFIER_assert(x >= 0);
14 }
```

The function call `__VERIFIER_assume(y >= 0)` is translated into
`phiInit([...,(y,Y),...]) :- Y>=0`
and the function call `__VERIFIER_assert(x >= 0)` is translated into
`phiError([...,(x,X),...]) :- X<0`.
The argument of `phiInit` and `phiError` is a list of pairs binding the variables of the C program to the corresponding CLP variables.

By executing `./VeriMAP example1.c` we get as output: `Answer: true`, which means that the given program is safe. In the examples presented in this section, the *Unfold/Fold Transformer* module makes use of a generalization operator based on standard widening.

In the following we will see how this answer is produced. In particular, we will list the commands needed to invoke each module described in Section 2 and the output produced by each module.

C-to-CLP Translation

The *C2CLP* module is invoked by the command `./VeriMAP --c2clp example1.c`. The output is the following set of CLP clauses (`example1.pl`), which define the predicates `fun/4`, `at/2` and `gvars/1` representing the function declarations, the C statements, and the global variable declarations, respectively.

```
1  % function declarations
2  fun(incr,[id(loc(scalar(int(z))))],[],entry_point(addr(9))).
3  fun(main,[],[id(loc(scalar(int(x))))],entry_point(addr(11))).
4  % function definitions
5  at(lab(9,inst),asgn(id(loc(scalar(int(z)))),
6      aexp(plus(aexp(id(loc(scalar(int(z))))),
7      aexp(id(glb(scalar(int(y)))))))),addr(9.1))).
8  at(lab(9.1,inst),asgn(id(glb(scalar(int(y)))),
9      aexp(const(int(0))),addr(10))).
10 at(lab(10,ret),ret(aexp(const(int(0))))).
11 at(lab(11,inst),call(map__VERIFIER_assume,[bexp(gte(aexp(id(glb(scalar(int(y))))),
12     aexp(const(int(0))))))],id(undef),addr(12))).
13 at(lab(12,ifte),ite(bexp(lt(aexp(id(loc(scalar(int(x))))),
14    aexp(id(glb(scalar(int(y)))))))),addr(13),addr(18))).
15 at(lab(13,inst),call(incr,[aexp(id(loc(scalar(int(x)))))],
16     id(loc(scalar(int(x)))),addr(14))).
17 [...]          <--- missing lines here
18 at(lab(23,inst),call(map__VERIFIER_assert,[bexp(gte(aexp(id(loc(scalar(int(x))))),
19     aexp(const(int(0))))))],id(undef),addr(24))).
20 at(lab(24,ret),ret(aexp(id(undef)))).
21 at(lab(h,halt),halt).
22 % global variables
23 gvars([(id(glb(scalar(int(y)))),aexp(id(undef)))]).
```

The predicate `fun(Id,Parameters,LocalVars,EntryAddr)` represents the function definitions, where: (i) `Id` is the identifier of the function, (ii) `Parameters` is the list of the formal parameters, (iii) `LocalVars` is the list of the local variables, and (iv) `EntryAddr` is the address of the first command in the body of the function. For instance, lines 5–10 represent:

```
int incr(int z) { z = z+y; y=0; }.
```

The predicate `at(Lab,Cmd)` represents a C statement, where: (i) `Lab` is of the form `lab(Addr,Type)` and represents the label of the command (in particular, `Addr` and `Type` represent the address of the entry point and the command type, respectively), and (ii) `Cmd` represents the given C command. For instance, lines 13–14 represent the *if-then-else* (`ite/3`) at lines 6–10 of the given C program. The first argument of `ite` represents the expression of the statement, where: (i) `lt` represents the '$<$' operator, (ii) `bexp` and `aexp` represent boolean and arithmetic expressions, respectively, and (iii) `loc` and `glb` represent local and global variable identifiers, respectively. The second and third arguments of `ite` represent the address of the first instruction of the *then* and *else* branches, respectively.

The predicate `gvars(GlbList)` represents the list of global variables. In the example we have a single global variable `id(glb(scalar(int(y))))` which is uninitialized (see `aexp(id(undef))`).

Verification Conditions Generation

By executing the command `./VeriMAP --vcg example1.c` the verification process stops after the execution of the *VCG* module. This module specializes the following proof rules for safety checking:

```
1 safe :- \+ unsafe.
2 unsafe :- elem(X,initial), reachable(X,U).
3 reachable(X,U) :- elem(X,error).
4 reachable(X,U) :- tr(X,Y), reachable(Y,U).
5 elem(X,initial) :- phiInit(X).
6 elem(X,error) :- phiError(X).
```

where `\+` denotes negation, `tr` denotes the transition relation that defines the *CIL Interpreter*, `elem(X,initial)` and `elem(X,error)` denote the properties that characterize the initial and error configurations, respectively. Thus, `safe` holds if and only if there exists no error configuration which is reachable from some initial configuration. The predicate `tr/2` is defined as follows:

```
t1. tr(cf(cmd(L,asgn(X,expr(E)),L1),D,T), cf(cmd(L1,C),D1,T1)):-
      loc_env(T,S), eval(E,D,S,V),
      update(D,T,X,V,D1,T1), at(L1,C).
t2. tr(cf(cmd(L,asgn(X,call(F,Es)),L1),D,T),cf(cmd(FL,C),D,[frame(L1,X,FEnv)|T])):-
      loc_env(T,S), eval_list(Es,D,S,Vs),
      build_funenv(F,Vs,FEnv), firstlab(F,FL), at(FL,C).
t3. tr(cf(cmd(L,ret(E)),D,[frame(L1,X,S)|T]),cf(cmd(L1,C),D1,T1)):-
      eval(E,D,S,V), update(D,T,X,V,D1,T1), at(L1,C).
t4. tr(cf(cmd(L,ite(E,L1,L2)),D,T),cf(cmd(L1,C),D,T)):-
      loc_env(T,S), beval(E,D,S), at(L1,C).
t5. tr(cf(cmd(L,ite(E,L1,L2)),D,T),cf(cmd(L2,C),D,T)):-
      loc_env(T,S), beval(not(E),D,S), at(L2,C).
t6. tr(cf(cmd(L,goto(L1)),D,T),cf(cmd(L1,C),D,T)):- at(L1,C).
t7. update(D,T,X,V,D1,T):- global(X), update_global(D,X,V,D1).
t8. update(D,T,X,V,D,T1):- local(X), update_local(T,X,V,T1).
```

We have the clauses for: (i) assignments to global and local variables (clause t1), (ii) function calls and returns (clauses t2 and t3), (iii) conditionals (clauses t4 and t5), and (iv) jumps (clause t6). The predicates for evaluating expressions and for updating the environment, have specific versions that deal with integers and arrays.

The generation of the VC's is performed by specializing the proof rules and the interpreter with respect to the set of CLP clauses produced by applying *C2CLP* to `example1.c`, that is, the clauses for `at`, `phiInit`, and `phiError`. In the following we present an excerpt of the log file produced by invoking the *VCG* module. The first part shows an application of the unfolding, definition, and folding transformation rules. The second part shows the specialized program that represents the VC's.

```
 7  RESULTS of Iteration #6
 8  -unfold-
 9 new7(A,B) :- A-B>=1,
10   reachable(cf(cmd(lab(13,inst),
11     call(fun,[aexp(id(loc(scalar(int(x)))))],id(loc(scalar(int(x)))),addr(14))),
12          ([(int(y),A)],[((undef,addr(h)),[(int(x),B)])])),error).
13  -define-
14 new9(A,B) :- reachable(cf(cmd(lab(13,inst),
15     call(fun,[aexp(id(loc(scalar(int(x)))))],id(loc(scalar(int(x)))),addr(14))),
16         ([(int(y),A)],[((undef,addr(h)),[(int(x),B)])])),error).
17  -fold-
18 new7(A,B) :- A-B>=1, new9(A,B).
19
20 [...]         <--- missing lines here
21 Transformed program:
22 new15(A,B) :- A-B=<0, new12(A,B).
23 new15(A,B) :- A-B>=1, new10(A,B).
24 new13(A,B,C) :- B=0.
25 new12(A,B) :- C=1, B>=0, new13(A,C,B).
26 new12(A,B) :- C=0, B=< -1, new13(A,C,B).
27 new10(A,B) :- C=1+B, A-B>=1, new10(A,C).
28 new10(A,B) :- A-B=<0, new12(A,B).
29 new9(A,B) :- C=0, D=0, new15(D,C).
30 new7(A,B) :- A-B>=1, new9(A,B).
31 new7(A,B) :- A-B=<0, new10(A,B).
32 new6(A,B,C) :- new6(A,B,C).
33 new4(A,B,C) :- B=0, new6(A,B,C).
34 new4(A,B,C) :- B=< -1, new7(A,C).
35 new4(A,B,C) :- B>=1, new7(A,C).
36 new3(A,B) :- C=1, A>=0, new4(A,C,B).
37 new3(A,B) :- C=0, A=< -1, new4(A,C,B).
38 new2(A) :- new3(A,B).
39 unsafe :- new2(A).
40 safe :- \+unsafe.
```

Unfold/Fold Transformation

By executing `./VeriMAP --transform example1.c` the verification process stops after one execution of the *Unfold/Fold Transformer* module.

This module transforms the VC's generated as output by the *VCG* module. In order to maximize code reuse, the VC's are first converted into a transition relation representation.

The excerpt of the log file below shows the part of the transition relation `tr` (lines 42–45) corresponding to the clauses listed at lines 25–28, and the two `elem` facts (line 47 and 48) corresponding to the clauses at lines 24 and 39, respectively. Lines 51–64 list the transformed program, and lines 66–68 give some statistics about the transformation (in particular, the number of Unfold-Definition-Fold

cycles, the number of clauses introduced by the definition rule, and the time required by the transformation process).

```
41 Initial program:
42 tr(s(new12,A,B),s(new13,A,C,B)) :- C=1, B>=0.
43 tr(s(new12,A,B),s(new13,A,C,B)) :- C=0, B=< -1.
44 tr(s(new10,A,B),s(new10,A,C)) :- C=1+B, A-B>=1.
45 tr(s(new10,A,B),s(new12,A,B)) :- A-B=<0.
46 [...]         <--- missing lines here
47 elem(s(new13,A,B,C),initial) :- B=0.
48 elem(s(new2,A),error).
49
50 Transformed program:
51 new13(A,B) :- A=0, B=0, new11(A,B).
52 new11(A,B) :- C=1, A-B=<0, A>=0, new12(A,C,B).
53 new10(A,B) :- A-B=<0, A>=0, new11(A,B).
54 new9(A,B) :- C=0, D=0, A>=0, A-B>=1, new13(C,D).
55 new8(A,B) :- A>=0, A-B>=1, new9(A,B).
56 new8(A,B) :- A-B=<0, A>=0, new10(A,B).
57 new6(A,B,C) :- B=0, A=< -1, new6(A,B,C).
58 new5(A,B,C) :- B=0, A=< -1, new6(A,B,C).
59 new4(A,B,C) :- B=1, A>=0, new8(A,C).
60 new3(A,B) :- C=1, A>=0, new4(A,C,B).
61 new3(A,B) :- C=0, A=< -1, new5(A,C,B).
62 new2(A) :- new3(A,B).
63 unsafe :- new2(A).
64 safe :- \+unsafe.
65
66 #UDF-iteration(s): 13
67 #definitions: 13
68 Elapsed time 10ms
```

Analysis

As a last step, the `./VeriMAP example1.c` command invokes the *Analyzer* module which detects the absence of facts in the transformed CLP program (lines 51–64). Thus, no unfolding will ever derive a fact for the predicate `unsafe`, and hence the predicate `safe` is true. The *Analyzer* module produces the output `Answer: true`, meaning that the program in `example1.c` is safe.

Iterated Verification

Now we consider a second C program (in file `example2.c`)

```
1 int x=0, y=0, n;
2 while (x < n) {
3   x = x+1;
```

```
4   y = y+x;
5 }
6 __VERIFIER_assert( x<=y );
```

By executing the command `./VeriMAP example2.c` we get: `Answer: unknown`. Indeed, at the end of the process we derive the following program:

```
1  new10(A,B,C,D) :- D=0, B>=0, A-B>=1, A-C>=0.
2  new8(A,B,C) :- D=1, A-B=<0, A>=0, A-C>=0, new9(A,B,C,D).
3  new8(A,B,C) :- D=0, B>=0, A-B>=1, A-C>=0, new10(A,B,C,D).
4  new5(A,B,C) :- A=0, B=0, D=1, C=<0, new6(A,B,C,D).
5  new4(A,B,C) :- A= -1+D, B=E-D, D>=1, C-D>=0, E-D>=0, new4(D,E,C).
6  new4(A,B,C) :- A>=0, B>=0, A-C>=0, new8(A,B,C).
7  [...]         <--- missing lines here
8  new2(A,B,C) :- A=0, B=0, new3(A,B,C).
9  unsafe :- A=0, B=0, new2(A,B,C).
10 safe :- \+unsafe.
```

where the presence of the constrained fact at line 1 allows the lightweight analyzer to give neither the answer `true` nor the answer `false`. Thus, the *IV* module performs one more invocation of the transformation and analysis submodules. (The first step of the *Unfold/Fold Transformer* is an application of the *Reversal* transformation to enable the propagation of the constraints occurring in the definition of `phiError`).

The excerpt of the log file reported below shows some information about the transformation performed at the second iteration of *IV*.

```
11 Initial program:
12 tr(s(new2,A,B,C),s(new3,A,B,C)) :- A=0, B=0.
13 [...]         <--- missing lines here
14 tr(s(new4,A,B,C),s(new4,D,E,C)) :- A= -1+D, B=E-D, D>=1, C-D>=0,
       E-D>=0.
15 tr(s(new4,A,B,C),s(new8,A,B,C)) :- A>=0, B>=0, A-C>=0.
16 tr(s(new5,A,B,C),s(new6,A,B,C,D)) :- A=0, B=0, D=1, C=<0.
17 tr(s(new8,A,B,C),s(new9,A,B,C,D)) :- D=1, A-B=<0, A>=0, A-C>=0.
18 tr(s(new8,A,B,C),s(new10,A,B,C,D)) :- D=0, B>=0, A-B>=1, A-C>=0.
19 elem(s(new2,A,B,C),initial) :- A=0, B=0.
20 elem(s(new10,A,B,C,D),error) :- D=0, B>=0, A-B>=1, A-C>=0.
21
22 Transformed program:
23 new3(A,B,C) :- B>=0, A-B>=1, A-C>=0, new4(A,B,C).
24 new2(A,B,C,D) :- D=0, B>=0, A-B>=1, A-C>=0, new3(A,B,C).
25 unsafe :- A=0, B>=0, C-D>=0, C-B>=1, new2(C,B,D,A).
26 safe :- \+unsafe.
27
28 #definitions: 5
29 #UDF-iteration(s): 5
30 Elapsed time 10ms
```

Since the transformed CLP program contains no constrained facts, the *Analyzer* module concludes that the program of `example2.c` is safe and returns `Answer: true`.

The iterated verification shown here has been performed by executing the command `./VeriMAP --iterations=2 example2.pl` (where '`--iterations=2`' specifies that the maximal number of iterations to be executed is 2).