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 [2,4,8,17]. 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 [18], Duality [15], ELDARICA [12], HSF [7], TRACER [13], μZ [11], 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 [3,4,19]. The current version of the VeriMAP can

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

E. Ábrahám and K. Havelund (Eds.): TACAS 2014, LNCS 8413, pp. 568–574, 2014.
© Springer-Verlag Berlin Heidelberg 2014

be 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 [5]. 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 [16]. (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.

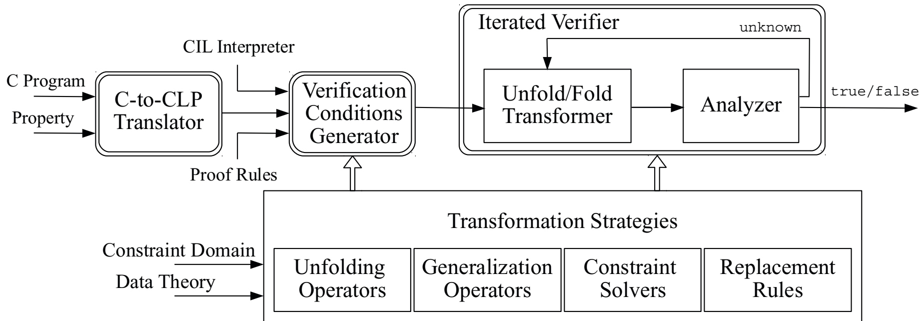

Fig. 1. The VeriMAP architecture

The *C2CLP* module is based on a modified version of the CIL tool [16]. 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 [16]. 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 [5]. Similarly to what has been proposed in [17], 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 [14], 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* [6], 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 [9], TRACER [13] and InvGen [10]. 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 [18], (ii) HSF(C) [7], and (iii) TRACER [13] using the strongest postcondition (*SPost*) and the weakest precondition (*WPre*) options.

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.

	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

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 [8]), and (iii) the theory of the data types in use, including those for dynamic data structures, such as lists and heaps.

References

1. Beyer, D.: Second Competition on Software Verification (SV-COMP 2013). In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 594–609. Springer, Heidelberg (2013)
2. Bjørner, N., McMillan, K., Rybalchenko, A.: On solving universally quantified Horn clauses. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 105–125. Springer, Heidelberg (2013)
3. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Verification of imperative programs by constraint logic program transformation. In: SAIRP 2013, Electronic Proceedings in Theoretical Computer Science, vol. 129, pp. 186–210 (2013)
4. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Verifying Programs via Iterated Specialization. In: PEPM 2013, pp. 43–52. ACM (2013)
5. Fioravanti, F., Pettorossi, A., Proietti, M.: Transformation rules for locally stratified constraint logic programs. In: Bruynooghe, M., Lau, K.-K. (eds.) Program Development in Computational Logic. LNCS, vol. 3049, pp. 291–339. Springer, Heidelberg (2004)
6. Fioravanti, F., Pettorossi, A., Proietti, M., Senni, V.: Generalization strategies for the verification of infinite state systems. Theory and Practice of Logic Programming 13(2), 175–199 (2013)
7. Grebenshchikov, S., Gupta, A., Lopes, N.P., Popeea, C., Rybalchenko, A.: HSF(C): A software verifier based on Horn clauses. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 549–551. Springer, Heidelberg (2012)
8. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI 2012, pp. 405–416. ACM (2012)
9. Gulavani, B.S., Chakraborty, S., Nori, A.V., Rajamani, S.K.: Automatically refining abstract interpretations. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 443–458. Springer, Heidelberg (2008)
10. Gupta, A., Rybalchenko, A.: InvGen: An efficient invariant generator. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 634–640. Springer, Heidelberg (2009)
11. Hoder, K., Bjørner, N., de Moura, L.: μZ– An efficient engine for fixed points with constraints. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 457–462. Springer, Heidelberg (2011)

12. Hojjat, H., Konečný, F., Garnier, F., Iosif, R., Kuncak, V., Rümmer, P.: A verification toolkit for numerical transition systems. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 247–251. Springer, Heidelberg (2012)
13. Jaffar, J., Murali, V., Navas, J.A., Santosa, A.E.: TRACER: A symbolic execution tool for verification. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 758–766. Springer, Heidelberg (2012)
14. The MAP system, `http://www.iasi.cnr.it/~proietti/system.html`
15. McMillan, K.L., Rybalchenko, A.: Solving constrained Horn clauses using interpolation. MSR Technical Report 2013-6, Microsoft Report (2013)
16. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 209–265. Springer, Heidelberg (2002)
17. Peralta, J.C., Gallagher, J.P., Saglam, H.: Analysis of imperative programs through analysis of Constraint Logic Programs. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 246–261. Springer, Heidelberg (1998)
18. Podelski, A., Rybalchenko, A.: ARMC: The logical choice for software model checking with abstraction refinement. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 245–259. Springer, Heidelberg (2007)
19. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Verifying Array Programs by Transforming Verification Conditions. In: McMillan, K.L., Rival, X. (eds.) VMCAI 2014. LNCS, vol. 8318, pp. 182–202. Springer, Heidelberg (2014)