Software Model Checking by Program Specialization

Emanuele De Angelis

University 'G. d'Annunzio' of Chieti-Pescara,
Viale Pindaro 42, I–65127 Pescara, Italy
Email: `deangelis@sci.unich.it`

Abstract

We introduce a general verification framework based on program specialization to prove properties of the runtime behaviour of imperative programs. Given a program P written in a programming language L and a property φ in a logic M, we can verify that φ holds for P by: (i) writing an interpreter I for L and a semantics S for M in a suitable metalanguage, (ii) specializing I and S with respect to P and φ, and (iii) analysing the specialized program by performing a further specialization. We have instantiated our framework to verify safety properties of a simple imperative language, called SIMP, extended with a nondeterministic choice operator. The method is fully automatic and it has been implemented using the MAP transformation system [14].

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Software model checking, program specialization, constraint logic programming.

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.439

1 Introduction and problem statement

Formal verification techniques allow us to prove that software artefacts (e.g., analysis and design models and source code) satisfy some given specifications. These techniques have recently received a growing attention as the basis for a promising methodology to increase the reliability and reducing the cost of software production (e.g., by reducing time to market).

Software model checking is a body of formal verification techniques for imperative programs that combine and extend ideas and techniques developed in the fields of static program analysis and model checking (see [12] for a recent survey). In order to prove that a program satisfies a given specification, software model checking methods automatically construct a program model which is sound, in the sense that if the model satisfies the given specification, then so does the actual program. Constructing such a model is a critical aspect of software model checking, since it tries to meet two somewhat conflicting requirements. On one hand, in order to make the verification process of large programs viable in practice, it has to construct a model by abstracting away as many details as possible. On the other hand, it would be desirable to have a model which is as precise as possible to reduce the number of wrong detections. Unfortunately, even for small programs operating over integer variables, an exhaustive exploration of the state space generated by the execution of programs is practically infeasible, and simple properties such as safety (which essentially states that 'something bad never happens') are undecidable. Despite this undecidability limitation, software model checking techniques work in many practical cases.

A huge amount of imperative languages is nowadays available. These languages provide sophisticated features which continuously change. Thus, software model checkers are required to be rapidly adapted to those changes. In order to develop tools which meet such a requirement, agile development methodologies should be employed. In this paper we consider program specialization as a framework in which model checking of imperative programs may be performed in a very *agile*, effective way. Program specialization is a program transformation technique whose objective is the adaptation of a program to the context of use. In particular, it turns out to be a very flexible and general methodology through which variations of the semantics of the considered imperative language and to different logics for expressing the properties of interest may be rapidly implemented into the software model checker. Indeed, by following this approach, which can be regarded as a generalization of the one proposed in [16], given a program P written in a programming language L, and a property φ in a logic M, we can verify that φ holds for P by: (i) writing an interpreter I for L and a semantics S for M in a suitable metalanguage, (ii) specializing the interpreter and the semantics with respect to P and φ, and finally (iii) analysing the specialized program. We choose *Constraint Logic programming* (CLP), which has been shown to be a very suitable language for implementing symbolic evaluation and analysis of imperative programs [10, 11, 16, 17], as a metalanguage.

2 Background and overview of the existing literature

Constraint logic programming has been successfully applied to perform model checking of both finite state [15] and infinite state [4] systems. In [6] a framework for the verification of safety properties of infinite reactive systems based on CLP program specialization has been introduced. Moreover, in [7] it has been shown that the termination of reachability analyses of infinite state systems can be improved by encoding reachability as a CLP program and then specializing the CLP program by incorporating the information about the initial and the unsafe states. The use of CLP to analyse simple imperative programs has been proposed in [16], where a CLP-based interpreter for the operational semantics of a simple imperative language is partially evaluated w.r.t. an input program. The result of the analysis of the residual CLP program can thus be used for annotating the original imperative program with relations among variables occurring in the imperative program. In [8] a method is presented for translating imperative programs supporting heap-allocated mutable data structures and recursive procedures to CLP.

A widely used technique implemented by software model checkers (e.g. SLAM and BLAST) is the *Counter-Example Guided Abstraction Refinement* (CEGAR) [12] which, given a program P and a safety property φ, uses an abstract model $\alpha(P)$ to check whether or not P satisfies φ. If $\alpha(P)$ satisfies φ then P satisfies φ, otherwise a counterexample, i.e., an execution which makes the program unsafe, is produced. The counterexample is then analysed: if it turns out to be a real execution of P (*genuine* counterexample) then the program is proved to be unsafe, otherwise it has been generated due to a too coarse abstraction (*spurious* counterexample) and $\alpha(P)$ needs to be refined. The CEGAR approach has also been implemented by using CLP. In particular, in [17], the authors have designed a CEGAR-based software model checker for C programs, called ARMC. In [10], another CLP-based software model checker for C programs, called TRACER, is presented. It integrates an abstraction refinement phase within a symbolic execution process.

3 Goal of the research

The goal of our research is to introduce a software model checking *framework*, based on the specialization of CLP programs, which is *parametric* with respect to: (i) the imperative language of the programs to be verified, and (ii) the specification language of the property to be checked. Regardless of the actual languages, the software model checker consists of a *front-end* module, which handles source code of programs, and a *verification engine* module which actually performs the verification task. Since our objective is performing software model checking of real programs we have to deal with issues arising from handling programs consisting of thousands of lines of code and using advanced features of imperative languages. Handling large programs introduces scalability issues which are to be carefully considered during the realization of the front-end. Indeed, the choices made during the design of the front-end may heavily affect the performance of the verification engine. More complex issues arise from handling programs using static and dynamic data structures, procedures, concurrency and objects. Consequently, a large portion of our research activity is devoted to designing abstraction techniques to be integrated in the CLP specialization process to prove properties which range from simple safety properties, such as array bound checking, to more sophisticated properties dealing with contents of data structures (e.g., sortedness), class relations (e.g., inheritance) and object interactions (e.g., effects of method invocations on object fields) and concurrent processes.

Our first mid-term objective is to realize a software model checker for the C language, which is still very popular, especially among device drivers and operating systems developers. A renewed attention to the C language is demonstrated in the TACAS 2012 competition (`http://sv-comp.sosy-lab.org/2012/`) in which several C software model checkers have been tested on very large programs preprocessed by using the CIL (C Intermediate Language) front-end (`http://cil.sourceforge.net/`). Thus, in order to ease the comparison with other tools we have decided to instantiate our framework to perform model checking of the C language by: (i) using CIL to translate the source language, and (ii) introducing a verification engine to prove safety properties of C programs.

4 Current status of the research

As a first step of our research activity we have instantiated our framework to perform the model checking of programs written in a simple imperative language with nondeterministic choice (SIMP), which is an abstraction of a subset of the C language. In particular, we have considered integer variables and the common control flow statements: `while(b) {···}` and `if(b) {···} else {···}`. The operational, or transition, semantics of SIMP is defined in terms of a transition relation $\Rightarrow$ over *states*, that is, pairs of the form $\langle p, e\rangle$, where p is a command and e is an *environment*, i.e, a function which maps variables occurring in p to their values. A state s' is said to be *reachable* from s if $s \Rightarrow^* s'$. A *specification* S is a triple $\langle \mathit{initial\text{-}prop}, p, \mathit{unsafe\text{-}prop}\rangle$, where p is a command and *initial-prop* and *unsafe-prop* are two formulas describing environments. A state $\langle p, e\rangle$ is said to be *initial* (*unsafe*) if e satisfies *initial-prop* (*unsafe-prop*). We say that S holds, or p is *safe* w.r.t. S, if there is no unsafe state which is reachable from an initial state. Performing model checking of p consists in verifying whether or not S holds. In order to perform model checking of SIMP commands we have defined a CLP-based interpreter of the operational semantics of SIMP as follows:

```
t( s(asgn(loc(X),A),E1), s(skip,E2) ) :- aeval(A,E1,V), update(loc(X),V,E1,E2).
t( s(ite(B,S1,_),E), s(S1,E) ) :- beval(B,E).
t( s(ite(B,_,S2),E), s(S2,E) ) :- beval(not(B),E).
t( s(ite(ndc,S1,_),E), s(S1,E) ).
t( s(ite(ndc,_,S2),E), s(S2,E) ).
t( s(while(B,S1),E), s(ite(B,comp(S1,while(B,S1)),skip),E) ).
t( s(comp(skip,S),E), s(S,E) ).
t( s(comp(S1,S3),E1), s(comp(S2,S3),E2) ) :- t( s(S1,E1), s(S2,E2) ).
```

where `t(s(P,E), s(P1,E1))` holds iff $\langle$`P,E`$\rangle \Rightarrow \langle$`P1,E1`$\rangle$, that is, the execution of the command `P` in the environment `E` leads to the execution of the command `P1` in the environment `E1`. Terms of the form `s(P,E)` denote states, where `P` ranges over ground terms built out of the following functors: `asgn` for the assignment, `ite` for the if-then-else statement (`ndc` represents the nondeterministic choice operator), `while` for the while loop, `skip` for the empty statement, `comp` for the statement composition, and `E` is a list of terms encoding the environment. We also have that: (i) `aeval(A,E,V)` holds iff `V` is the value of the arithmetic expression `A` in the environment `E`, (ii) `beval(B,E)` holds iff the boolean expression `B` evaluates to **true** in the environment `E`, and (iii) `update(loc(X),V,E1,E2)` holds iff `E2` is equal to `E1` except in `X` which takes the value `V` (`loc` encodes a variable identifier).

Let $S = \langle$*initial-prop*$, p,$ *unsafe-prop*$\rangle$ be a specification. We reduce the problem of verifying whether or not a command is safe to a reachability problem by using the CLP program SMC which consists of the following clauses:

```
unsafeProg :- initial(X), reachable(X).
reachable(X) :- unsafe(X).
reachable(X) :- t(X,X1), reachable(X1).
initial(s(p,E)) :- initial-prop.
unsafe(s(_,E)) :- unsafe-prop.
```

together with the clauses defining the semantics of SIMP. In the above program, `p` stands for the ground term encoding the command p, while `initial-prop` and `unsafe-prop` stand for constraints encoding the formulas *initial-prop* and *unsafe-prop*, respectively.

Our software model checking method consists in:
(Step 1) encoding the specification S into the clauses for `initial`, and `unsafe`,
(Step 2) specializing SMC with respect to `unsafeProg`, and
(Step 3) computing the least model M(SpSMC) of the specialized program SpSMC, and checking whether or not `unsafeProg` belongs to the least model M(SpSMC).

The objective of Step 2 is to modify the initial program SMC by propagating the information specified by `initial`, so that by exploiting this information, the computation of the least model M(SpSMC) may be more effective and terminate more often than the computation of the least model M(SMC). In particular, the interpretation overhead is compiled away by specialization, thereby producing a CLP program where no terms encoding the command p are present. Step 2 is performed by a rule-based program specialization strategy which makes use of the following rules: definition introduction, unfolding, folding, and clause removal. These rules preserve the least model semantics of CLP programs [5], thus, the specialization strategy yield a program which is guaranteed to satisfy the same set of properties satisfied by the original program. Indeed, we have that `unsafeProg`$\in M$(SMC) iff `unsafeProg`$\in M$(SpSMC). In order to ensure the termination of Step 2, we use suitable generalization operators, related to widen operators used in abstract interpretation techniques [2].

▶ **Example 1.** Let us consider the following specification $\langle x = 0 \wedge y \geq 0,\ p,\ error = 1\rangle$, where p stands for: `while (x < 10) { y = y+1;  x = x+1; } if ( y + x < 10) { error = 1; }`. We encode p into the term

```
comp( while( lt(loc(x),int(10)), comp(asgn(loc(y),plus(loc(y),int(1))),
                                      asgn(loc(x),plus(loc(x),int(1))) ) ),
      ite( lt(plus(loc(y),loc(x)),int(10)), asgn(loc(error),int(1)), skip) )
```

where the functor `int` encodes an integer value. By also translating the remaining components of the specification we obtain the following clauses:

```
initial(s(p,[X,Y,E])) :- X=0, Y>=0, E=0.
unsafe(s(_,[X,Y,E])) :- E=1.
```

where p stands for the term encoding p. By specializing SMC with respect to unsafeProg, we obtain the following program

```
new3(X,Y,E) :- X>=0, X<10, Y>=X, E=0, X1=X+1, Y1=Y+1, new3(X1,Y1,E).
new3(X,Y,E) :- X>=10, Y>=X, E=0, new5(X,Y,E).
new2(X,Y,E) :- X=0, Y>=0, E=0, X1=1, Y1=Y+1, new3(X1,Y1,E).
unsafeProg  :- X=0, Y>=0, E=0, new2(X,Y,E).
```

whose model is used to verify whether or not p is safe. Since the program contains no constrained fact, we have that $M(\mathsf{SpSMC})$ is empty, and thus p is safe.

In [16], the interpreter is specialized w.r.t. the input program and a static analyser for CLP programs is used to derive relations among variables occurring in the imperative program. In our method, we discover these relations *during* the specialization process by means of generalization techniques defined in terms of relations and operators on constraints such as widening, convex-hull, and well-quasi orders.

5 Preliminary results accomplished

We have implemented our software model checking method using the MAP transformation system [14] for the verification engine module and the Lex and Yacc tools for the front-end module (it is currently being rewritten in CIL). We have shown the effectiveness of our method by applying it to some examples (available at `http://map.uniroma2.it/smc`) taken from the literature [9, 10], and we have compared its performance with that of ARMC and TRACER. Among the wide variety of software model checkers nowadays available, we choose ARMC and TRACER because they provide CLP based implementations of two dual verification approaches: ARMC starts with a very coarse abstraction and uses counterexamples to increase the level of details of the model and, conversely, TRACER starts with a very detailed model and uses counterexamples to abstract away as many details as possible and, possibly, refines it if the model is too coarse to prove the property. Our preliminary results (see Table 1) show that our approach is viable and competitive in practice.

Table 1 Time (in seconds) for performing model checking (TRACER was run by using the option `-intp wp`). $\perp$ denotes 'terminating with error', ∞ means 'No answer within 20 minutes'.

Tool	*f2*	*substring*	*daggerP*	*seesaw*	*tracerP*	*interp*	*widen*	*selectSort*
ARMC	∞	719.39	∞	3.98	∞	0.13	∞	0.48
TRACER	1.35	227.28	1.27	1.46	1.04	1.32	1.35	$\perp$
MAP	0.21	10.20	5.37	0.03	0.03	0.06	0.07	0.06

6 Open issues and expected achievements

A challenging issue is the extension of our framework to deal with more complex language features provided by imperative languages such as arrays, lists, procedure calls, and concurrency. Moreover, since it is possible to deal with imperative languages at different levels of abstraction, it would be interesting to extend the framework to verify properties of both: (i) high-level languages with object-oriented features, such as Java, PHP, Objective C or C# [13], and (ii) low-level languages such as bytecode for Java [1] or for the .NET platform [3]. From the verification point of view, such extensions would require the design of suitable interpreters for handling the newly introduced language features, posing new theoretical and experimental challenges.

Another challenging issue is the extension of the set of properties which can be proved. In Section 3 we listed some interesting properties depending on the content of data structures, the relations among objects and among classes, and the behavior of concurrent processes. Handling these properties not only requires the investigation of suitable logics in which they may be expressed, but also raises the issue of integrating them into the verification process. In particular, it could be necessary to resort to more sophisticated logic program transformations based on the unfold/fold method.

References

1 E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla. Verification of Java bytecode using analysis and transformation of logic programs. In *Proc. of PADL'07*, volume 4354 of *LNCS*, pp. 124–139, 2007.

2 P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. of POPL'77*, pp. 238–252. ACM Press, 1977.

3 P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Proc. of POPL'11*, pp. 105–118, 2011.

4 G. Delzanno and A. Podelski. Model checking in CLP. In *Proc. of TACAS'99*, LNCS 1579, pp. 223–239, 1999.

5 S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.

6 F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL Properties of Infinite State Systems by Specializing Constraint Logic Programs. In *Proc. of VCL'01*, pp. 85-96. Revised and extended Version: Technical Report R.657, IASI - CNR, 2007, Roma, Italy.

7 F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Improving Reachability Analysis of Infinite State Systems by Specialization. In *Proc. of RP'11*, pp. 165–179, 2011.

8 C. Flanagan. Automatic software model checking via constraint logic. *Sci. Comput. Program.*, 50(1-3):253–270, March 2004.

9 B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically Refining Abstract Interpretations. In *Proc. of TACAS'08*, LNCS 4963, pp. 443–458, 2008.

10 J. Jaffar, J. A. Navas, and A. E. Santosa. TRACER: A symbolic execution tool for verification. `http://paella.d1.comp.nus.edu.sg/tracer/`.

11 J. Jaffar, J. A. Navas, and A. E. Santosa. Symbolic execution for verification. *Computing Research Repository*, 2011.

12 R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4):21:1–21:54, October 2009.

13 G.T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Form. Asp. Comput.*, 19(2):159–189, June 2007.

14 The MAP transformation system. `http://www.map.uniroma2.it/mapweb`.

15 U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model-checking. In *Proc. of CL 2000*, LNAI 1861, pp. 384–398, 2000.

16 J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of imperative programs through analysis of constraint logic programs. In *Proc. of SAS'98*, LNCS 1503, pp. 246–261,1998.

17 A. Podelski and A. Rybalchenko. ARMC: The logical choice for software model checking with abstraction refinement. In *Proc. of PADL'07*, LNCS 4354, pp. 245–259, 2007.