Challenges in Decomposing Encodings of Verification Problems

Peter Schrammel
University of Oxford, UK
peter.schrammel@cs.ox.ac.uk

Modern program verifiers use logic-based encodings of the verification problem that are discharged by a back end reasoning engine. However, instances of such encodings for large programs can quickly overwhelm these back end solvers. Hence, we need techniques to make the solving process scale to large systems, such as partitioning (divide-and-conquer) and abstraction. In recent work, we showed how decomposing the formula encoding of a termination analysis can significantly increase efficiency. The analysis generates a sequence of logical formulas with existentially quantified predicates that are solved by a synthesis-based program analysis engine. However, decomposition introduces abstractions in addition to those required for finding the unknown predicates in the formula, and can hence deteriorate precision. We discuss the challenges associated with such decompositions and their interdependencies with the solving process.

1 Introduction

Logic-based encodings of the verification problem are more and more widespread in software verification [1]. However, the generated formulae are often too large to be directly handled by the back end solver. Classical divide-and-conquer techniques suggest themselves to cope with such large problems. Work on interprocedural verification, e.g. [6, 3], follows the syntactical, procedural structure of the program to perform a decomposition of the formula. This does not seem ideal, but has been shown to significantly increase efficiency in comparison with monolithic solving.

In recent work, we used a synthesis engine [7, 2] to solve for multiple predicates at once even when they are mutually dependent. Since this scales badly to large formulae, we have to decompose the formula in order to reduce the load on the synthesis engine. However, the decomposition may introduce additional abstractions, in particular when mutually dependent predicates are concerned.

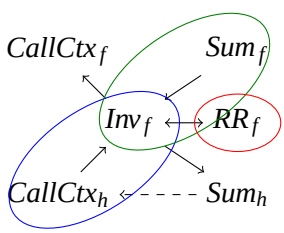
Outline We first show the encoding of a verification problem using the example of universal termination verification. Then we discuss the challenges associated with decomposing this problem and the interdependencies with the solving process.

2 Encoding

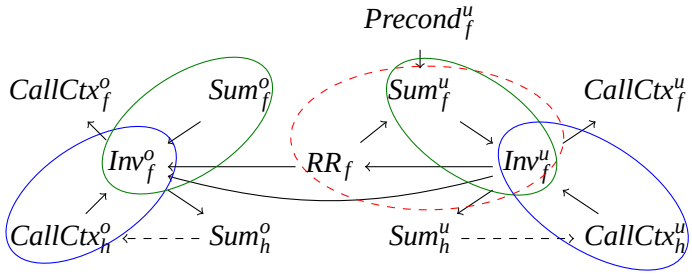
We assume that programs are given in terms of call graphs, where individual procedures f are given in terms of symbolic input/output transition systems. Formally, the input/output transition system of a procedure f is a triple of characteristic predicates for relations $(Init_f, Trans_f, Out_f)$, where $Trans_f(x,x')$ is the transition relation; the input relation $Init_f(x^{in},x)$ defines the initial states of the transition system and relates them to the inputs x^{in}; the output relation $Out_f(x,x^{out})$ connects the transition system to the outputs x^{out} of the procedure. Inputs are procedure parameters, global variables, and memory objects that are read by f. Outputs are return values, global variables, and memory objects written by f. Internal

J. P. Gallagher and P. Rümmer (Eds.): 3rd Workshop
on Horn Clauses for Verification and Synthesis (HCVS)
EPTCS 219, 2016, pp. 29–32, doi:10.4204/EPTCS.219.3

© Peter Schrammel
This work is licensed under the
Creative Commons Attribution License.



(a) Interprocedural universal termination verification problem



(b) Interprocedural sufficient preconditions for termination inference

Figure 1: Dependent predicates in the encodings and decompositions

states x are usually the values of variables at the loop heads in f. These relations are given as *first-order logic formulae* resulting from the logical encoding of the program semantics.

Let F denote the set $\{f_1,\ldots,f_n\}$ of procedures in a given program. H_f is the set of procedure calls to procedures $h \in F$ at calls sites i in procedure f. The vectors of input and output arguments ${x^{p\_in}}_{h_i}$ and ${x^{p\_out}}_{h_i}$ are intermediate variables in *Trans*. We denote the termination argument RR_f, i.e. the conditions that ensure the termination of procedure f, such as a well-founded transition invariant.

Example By encoding Hoare-style verification rules (cf. [4]) into second-order logic,[1] we obtain the following formula. Its satisfiability guarantees universal termination of the program.

$$\begin{array}{ll} \exists_2 Summary_{f_1},\ldots,Summary_{f_n} : \bigwedge_{f\in F} & \\ \quad \exists_2 Inv_f, RR_f : \forall {x^{in}}_f, x_f, {x'}_f, {x^{out}}_f : & \\ \qquad Init_f({x^{in}}_f, x_f) \Longrightarrow Inv_f(x_f) & (1) \\ \wedge \quad Inv_f(x_f) \wedge Trans_f(x_f, {x'}_f) \wedge \bigwedge_{h_i\in H_f} Summary_h({x^{p\_in}}_{h_i}, {x^{p\_out}}_{h_i}) \Longrightarrow Inv_f({x'}_f) \wedge RR_f(x_f, {x'}_f) & \\ \wedge \quad Init_f({x^{in}}_f, x_f) \wedge Inv_f({x'}_f) \wedge Out_f({x'}_f, {x^{out}}_f) \Longrightarrow Summary_f({x^{in}}_f, {x^{out}}_f) & \end{array}$$

In this formula, recursive procedures produce cyclic dependencies of their $Summary_f$ predicates. If abstractions are used to lazily solve the formula, the invariant Inv_f and the termination argument RR_f become interdependent. Similarly, invariants of nested loops are dependent on each other. Rewriting nested loops into a single loop with invariant Inv_f only "hides" these dependencies as relational dependencies between the loop variables.

3 Decomposition

We can decompose a formula that encodes a verification problem, such as (1) above, into a sequence of subproblems that are solved by the synthesis engine. The soundness of the analysis result is ensured by (1) the soundness of the analysis of individual subproblems, (2) the soundness of the combination of the subproblem results, (3) and induction over the decomposition hierarchy. Decomposition causes the following issues: (A) It may introduce additional interdependent predicates. (B) The subproblems may be inference, and not verification problems; hence their solving requires optimisation (like our synthesis engine) instead of decision procedures.

Example In [3] we followed the classical approach of a procedural decomposition. We emulate the traversal of the call graph top-down analysing each procedure separately and propagating the summaries back up. This decomposition splits the $Summary_h$ predicate for a call to procedure h at call site i into

[1] Mind that we use the notation $\exists_2$ to stress the fact that the quantifier binds a predicate.

a *calling context* predicate $CallCtx_h$ that transfers information from the caller to the callee,[2] and a summary predicate Sum_h that transfers information from the callee to the caller. These two predicates are mutually dependent as illustrated by the cycle in the dependency graph in Figure 1a. The dashed arrows are dependencies resulting from unfolding the diagram along the call graph. The blue, green and red ellipses indicate the decomposition, i.e. predicates that are solved for at once. The algorithm in [3] uses a greatest fixed point computation to resolve this dependency. However, this is very imprecise for recursive procedures. Figure 1b shows the predicate dependencies for the inference of sufficient preconditions for termination. Without going into details (see [3]), we want to direct the attention to the dependency (red dashed ellipse) between the *under-approximating* summary Sum^u_f (of which the sufficient precondition is a projection), the termination argument, and the invariants – which is a maximisation problem.

4 Lessons Learned and Prospects

We have to accommodate the following two conflicting goals: (1) Solving as large subformulae as possible to increase precision and reduce the need for later refinement. (2) Solving as small subformulae as possible to be scalable. In Figure 1a, we solve for invariants (green) and termination arguments (red) separately because our synthesis engine currently does not support product domains that could infer both at once, each with their optimised domains, thus eliminating cyclic dependencies. Some domains require least, others greatest fixed point computations, our engine is currently unable to combine both in a single query. Moreover, programs are rarely written with verification in mind; they are often badly structured. Therefore we need a property-, precision-, and capacity-driven dynamic (de)composition to achieve goals (1) and (2). Re-partitioning the verification problem by eliminating predicates and introducing new ones seems essential. Decompositions introducing cyclic dependencies should only be used if the solving capacity is exceeded. On the other hand, precision can be increased by expansion, i.e. unrolling of loops and inlining of recursions, if the capacity allows it. Many of these issues are akin to open problems in neighbouring areas of research, e.g. [5].

References

[1] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan & Andrey Rybalchenko (2015): *Horn Clause Solvers for Program Verification*. In: *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, *LNCS* 9300, Springer, pp. 24–51, doi:10.1007/978-3-319-23534-9_2.

[2] Martin Brain, Saurabh Joshi, Daniel Kroening & Peter Schrammel (2015): *Safety Verification and Refutation by k-Invariants and k-Induction*. In: *Static Analysis Symposium*, *LNCS* 9291, Springer, pp. 145–161, doi:10.1007/978-3-662-48288-9_9.

[3] Hong-Yi Chen, Cristina David, Daniel Kroening, Peter Schrammel & Björn Wachter (2015): *Synthesising Interprocedural Bit-Precise Termination Proofs*. In: *Automated Software Engineering*, ACM, pp. 53–64, doi:10.1109/ASE.2015.10.

[4] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea & Andrey Rybalchenko (2012): *Synthesizing software verifiers from proof rules*. In: *Programming Language Design and Implementation*, ACM, pp. 405–416, doi:10.1145/2254064.2254112.

[5] Youssef Hamadi & Christoph M. Wintersteiger (2013): *Seven Challenges in Parallel SAT Solving*. *AI Magazine* 34(2), pp. 99–106.

[2] The calling context can be inferred by synthesising a predicate $CallCtx_{h_i}$ s.t. $\forall x_f, x'_f, x^{p\_in}{}_{h_i}, x^{p\_out}{}_{h_i} : Inv_f(x_f) \wedge Trans_f(x_f, x'_f) \Longrightarrow CallCtx_{h_i}(x^{p\_in}{}_{h_i}, x^{p\_out}{}_{h_i})$.

[6] Anvesh Komuravelli, Arie Gurfinkel & Sagar Chaki (2014): *SMT-Based Model Checking for Recursive Programs*. In: *Computer-Aided Verification, LNCS* 8559, Springer, pp. 17–34, doi:10.1007/978-3-319-08867-9_2.

[7] Peter Schrammel & Daniel Kroening (2016): *2LS for Program Analysis - (Competition Contribution)*. In: *Tools and Algorithms for the Construction and Analysis of Systems, LNCS* 9636, Springer, pp. 905–907, doi:10.1007/978-3-662-49674-9_56.