Enhancement of automated static verification efficiency through manual quantifiers instantiation

Denis Buzdalov[1]
Institute for System Programming
of the Russian Academy of Sciences
Moscow, Russia
Email: buzdalov@ispras.ru

***Abstract*—Checking of a program conformity with a contract specification is a hard problem. Usually this check requires high time and memory expenses. This work describes a technique that allows to lower verification costs through usage of some information known by people who verify a program and contains suggestions of the way how to organize it.**

I. Introduction

Nowadays a lot of computer systems perform quality critical tasks. It means that bad quality of work of these systems can lead people's deaths and injuries or financial losses. That's why such systems need to be *verified*.

Verification requires checked properties and possibly some additional info to be specified in some formal language. Such collection of formal statements is called a *formal specification*. Formal specification has precisely defined semantics which is used by a verification instrument.

There is a lot of ways of verification. They vary on resources requirements and quality of a result. *Formal static verification* differs from e.g. testing in confidence of specification conformity in case of positive verification verdict. Unlike the model checking this approach can be used for verification of a target system itself but not its model. Also, this approach can be used for a wide class of target systems (*e.g. for checking of embedded systems*).

There are many types of specifications. *Contract specifications* [1] is a popular type which has a lot of supporting instruments. Contract specification is a set of statements of the following types:

- *precondition* is a condition which holding is required for an operation (function, method, subroutine) execution; *e.g. a function of real square root computation will have a non-negativity of its argument as a precondition*;
- *postcondition* is a condition which have to be held at the end of an operation execution; *the square root function can have a condition that the result multiplied by itself must be equal to the input argument as a postcondition*;
- *invariant* is a condition that have to be held at some time or on some events arising

Instruments of contract-based static verification reduce the program correctness proving task to the task of *satisfiability* of some *typed predicates* (these predicates can contain some operations and relations of some typed values).

Satisfiability is a laborious task. This task is usually not solved manually because of usually big number of predicates which satisfiability have to be proved. Also manual proofs are not stable to changes of program, so it cannot be used during the program development. That's why automatic or automated *solvers* are used for such tasks (e.g. SMT-solvers [2]).

II. Motivation

Specification itself is usually not enough for successful verification of a correct program. For example, Floyd methods [3] and Hoare rules [4] expect each loop to be marked by a loop invariant (a predicate which is hold before each loop condition check). Ability and time of proving highly depends on which loop invariants are chosen.

But even if every loop has an invariant allowing to prove a program correctness it may be not enough for the successful verification.

A person that wants to prove a program correctness has to add *lemmata* and *assertions* to a specification to help solver. These additions may reflect different properties of data and its operations which solver it not able to understand itself. *For instance, if we have a* $f : List \to Multiset$ *operation and* $List$ *and* $Multiset$ *types have an addition operation (concatenation and union correspondingly), then lemma of linearity of* f *relatively to the addition operation will have form of* $\forall l1, l2 : List \cdot f(l1 + l2) = f(l1) + f(l2)$

These lemmata are conjuncted with a precondition during proving of a fact that a precondition implies a postcondition ($p_{pre} \wedge p_{lemma1} \implies p_{post}$). This allows lemmata statements to be used in the verification process.

Assertions are similar to lemmata but unlike them are defined inside operations and hold only inside them. Consequently, assertions can represent properties connected with local data and also consider a precondition to be held (be implied by it). Assertions also are conjuncted with a precondition for the verification process.

There are some instruments that support the described correctness proving technique and can be used for the formal static verification of software e.g. frama-c [5] and Dafny [6].

Lemmata and assertions often are statements with the universal quantifier.

[1]This work is partially supported by RFBR 11-07-12075-ofi-m, 10-07-1047a, Minorbnauki RF 07.514.11.410.

One of the main difficulty of the satisfiability problem solving is a successful usage of such statements. The way how to instantiate an expression of the quantifier is not defined by algorithms that are used in solvers [7]. But effectivity of satisfiability proving highly depends on how instantiation is performed. There are some heuristic methods of instantiation that increase the proving speed [8], [9] but they give positive results not often in the practice. That's why verification usually requires high resources amount (both time and memory).

There can be a lot of lemmata but not all of them are required for a check of each postcondition. But still, solver will try to instantiate useless statements with the universal quantifier. This can considerably increase proving costs.

This work offers a technique that makes verification easier by reducing of quantified statements usage in lemmata and assertions and also by limiting of usage of useless lemmata.

III. Suggested technique

Technique is based on the fact that a person who is trying to verify a program unlike modern solvers usually knows namely which lemmata help or can help for the postcondition or invariant checking. Also he usually understands how a statement with the universal quantifier should be instantiated to use lemma properly.

Consider a lemma of linearity of a f function from the example above. To prove that removing of the first element from a list leads a multiset of its elements to decrease its size by one, it is enough to take a sublist containing only the first element of the original list as $l1$ and a tail of the list as $l2$.

To use such knowledge it is offered to rewrite lemmata that are formed like

$$\forall x_1 : X_1, x_2 : X_2, \ldots, x_n : X_n \cdot$$
$$\cdot A(x_1, x_2, \ldots, x_n) \implies P(x_1, x_2, \ldots, x_n)$$

as *pure functions* of form $t : X_1 \times X_2 \times \ldots \times X_n \to \varnothing$ and having the $A(x_1, x_2, \ldots, x_n)$ predicate as a precondition and the $P(x_1, x_2, \ldots, x_n)$ predicate as a postcondition. Pre- and postconditions have the same set of parameters because of a function purity and the fact that it does not return any value. Let's call functions of the described type that represent some lemma a *function-lemma*.

Such lemmata representation allows to move statements that help to prove a lemma but useless for proving other lemmata, inside the function-lemma. That makes solver's task easier because it reduces count of statements that it can but shouldn't use. That allows to reduce verification costs.

Besides, this representation allows to use once proved lemma without rewriting a proof. *There's a lack of such ability in existing instruments, e.g. in frama-c [5]. This fact makes lemmata usage difficult and increases a solver's work amount.*

But still, if the lemmata meaning isn't changed, namely that all lemmata can be used for the verification, the problem of instantiation of parameters (the same problem with the problem of instantiation of quantifier variables) remains. To solve this problem it is suggested to use lemmata only in places pointed by a person that verifies a program (except situation mentioned below).

When some lemma is pointed to be used, parameters of the function-lemma have to be explicitly defined. At the place of pointing applicability of the lemma should be checked (through the function-lemma's precondition check) and if the check is successful the main lemma statement (represented by the function-lemma postcondition) have to be considered to be held (because the lemma is proved separately).

Such lemmata usage can be both standalone and inside a composite statement.

No statements with the universal quantifiers appear where they are not needed if such lemmata semantics is used. If statements with the universal quantifier are essential, then instantiation variants count is not increased comparing to the existing lemmata semantics. Moreover, verificating person can considerably decrease this count if he thinks that it is enough for proving. Consequently verification resources requirements are decreased if lemmata are used properly.

However usage of the described semantics can reduce proving quality (compared to the existing semantics) when lemmata are used improperly. To make this semantics to be not worse that the existing one, it is modified.

If proof that uses pointed by a man function-lemmata is unsuccessful, then statements of function-lemmata can be interpreted as statements with the universal quantifier and after that proving should be continued. So if a program correctness can be proved without usage of the suggested technique, then it can be done with the suggested technique too. In that case difference of verification resources expenses will be not big because the suggested technique does not increase instantiation variants count (and often decreases it up to the single one).

Right function-lemma usage can considerably decrease the verification expenses of a correct program because it eliminates usage of statements with the universal quantifier and usage of useless lemmata.

Moreover this technique allows to use named properties in a convenient way. This allows to standardize and describe lots of widely used data and operation properties.

For instance, consider a lemma of the square operation $sqr : real \to real$ being converse to the square root operation $sqrt : real \to real$. Let the lemma be named $sqrToSqrtConversity$. It has a single parameter, let it be named x. The precondition will be $x \geqslant 0$, and the postcondition will be $sqr(sqrt(x)) = x$.

Consider an operation that has an argument a and property of sqr being converse to $sqrt$ have to be held for this argument. Then the $sqrToSqrtConversity(a)$ statement should be added as a precondition. If some operation returns a numbers set s which all are numbers that the converse property have to be applicable to, then if is enough to write $\forall x \in s \cdot sqrToSqrtConversity(x)$ as a postcondition.

Consider a function evaluating $(\sqrt{x}+\sqrt{y})^2$. Then to prove a property of "$y = 0$ implies the result to be equal to x" we can use the $sqrToSqrtConversity$ lemma with the x number as a parameter (also the $sqrt(0) = 0$ property will *be required).*

IV. Case study

Considering existing instruments, Dafny [6], a static verification instrument, partially supports the suggested technique. It supports an ability of defining of functions that can be used as a function-lemma. However these functions cannot be used inside composite statements. That fact does not allow to use some technique abilities.

These limited abilities were, however, enough to prove permutation correctness of the Shell sort algorithm [10] with Sedgewick coefficients [11] using acceptable time (approx. 10 min.) and memory (approx. 300 MB) amount. Author hasn't managed to do this without the suggested technique using up to 3 hours and 4 GB of memory.

V. Conclusion

A technique that widens ability of automatic verification instruments (on appropriate instruments modernization) was suggested.

This technique was used in practice, as far as it is possible using existing instruments. The technique allowed to prove a program correctness which couldn't be done without usage of this technique.

References

[1] B. Meyer, "Design by contract," Interactive Software Engineering Inc., Tech. Rep. TR-EI-12/CO, 1986.

[2] C. Barrett, A. Stump, and C. Tinelli, "The satisfiability modulo theories library (SMT-LIB)," http://www.smtlib.org/, 2010.

[3] R. W. Floyd, "Assigning meaning to programs," in *In Proceedings of Symposium on Applied Mathematics*, vol. 19. A.M.S., 1967, pp. 19–32.

[4] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969.

[5] "Frama-c," http://frama-c.com/.

[6] "Dafny," http://research.microsoft.com/en-us/projects/dafny/.

[7] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, Jul. 1962.

[8] Y. Ge and L. Moura, "Complete instantiation for quantified formulas in satisfiabiliby modulo theories," in *Proceedings of the 21st International Conference on Computer Aided Verification*, ser. CAV '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 306–320.

[9] L. Moura and N. Bjørner, "Efficient e-matching for smt solvers," in *Proceedings of the 21st international conference on Automated Deduction: Automated Deduction*, ser. CADE-21. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 183–198.

[10] D. L. Shell, "A high-speed sorting procedure," *Commun. ACM*, vol. 2, no. 7, pp. 30–32, Jul. 1959.

[11] R. Sedgewick, "A new upper bound for shellsort," *Journal of Algorithms*, vol. 7, no. 2, pp. 159–173, June 1986.