Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

On Essential Program Annotations and Completeness of Verifying Compilers*

Bernhard Beckert, Thorsten Bormer, Vladimir Klebanov

Dept. of Computer Science, University of Koblenz, Germany

Received: date / Revised version: date

Abstract. It is widely recognized that interaction is indispensable in deductive verification of real-world code. A verification engineer has to guide the proof search and provide information reflecting their insight into the workings of the program. Lately we have seen a shift towards a paradigm, called verifying compilers, where the required information is provided in form of program annotations instead of interactively during proof construction.

In this paper, we discuss the different purposes that annotations can serve. Based on a clarification of what the notion of completeness means in the framework of verifying compilers, we show that some auxiliary (non-requirement) annotations are (only) needed for efficiency of proof search, while others are essential for completeness, i.e., indispensable for proof construction.

1 Introduction

It is widely recognized that interaction is indispensable in deductive verification of real-world code. A verification engineer has to guide the proof search and provide information reflecting their insight into the workings of the program. Lately we have seen a shift towards a paradigm, called verifying compilers [7], where the required information is provided in form of program annotations instead of interactively during proof construction. This has some interesting consequences upon the verification process and the way annotations are used to specify programs as the lines between requirement specification and information required for proof construction and proof search guidance get blurred.

* Work partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft XT project under grant 01 IS 07008 H. The responsibility for this article lies with the authors.

In this paper, we discuss the different purposes that annotations can serve. Based on a clarification of what the notion of completeness means in the framework of verifying compilers, we show that some auxiliary (non-requirement) annotations are (only) needed for efficiency of proof search, while others are essential for completeness, i.e., indispensable for proof construction. Our considerations reveal that users have to possess a certain knowledge about the inner workings of a verifying compiler. They need to know what kinds of annotations are indispensable in which situations. This is somewhat surprising, as it contradicts the idea of freeing the user from the need to know about the inner workings of the verification system and enabling the use of the tool as a "black box," which is generally seen as a central part of the verifying compiler paradigm.

Tools following the verifying compiler paradigm include Spec# [1], VCC [9], and Caduceus [5]. They are all based on powerful fully-automatic provers and decision procedures, and they support real-world programming languages such as C and C#. The typical architecture of such systems is described in Section 2, and the possible outcomes of invoking a verifying compiler are explained in Section 3. Then, we discuss the notion of completeness in the framework of verifying compilers and give a formal definition (Sect. 4). Based on this discussion, we analyze the different kinds of annotations and which of them are essential for completeness (Sect. 5 and 6). Finally, we summarize our conclusions in Section 7.

2 Inside a Typical Verifying Compiler

In the following we describe the process of software verification with the Verifying C Compiler (VCC). Our observations (unless noted otherwise) are, however, not restricted to this particular setup.

The VCC toolchain allows for modular verification of C programs using method contracts and invariants over data structures. Method contracts are specified by pre- and post-conditions. These contracts and invariants are stored as annotations within the source code in a way that is transparent to the regular, non-verifying compiler.

As most verifying compilers today, VCC works using an internal two-stage process. The reason for this is a better separation of concerns and easy integration of different tools. We will discuss the interplay of the two stages, but many of our remarks also apply to one-stage or multi-stage approaches.

The first stage of the VCC toolchain translates the annotated C code into first-order logic via an intermediate language called BoogiePL [4]. BoogiePL is a simple imperative language with embedded assertions. From this BoogiePL representation, it is easy to generate a set of first-order logic formulas, which state that the program satisfies the embedded assertions. These formulas are called verification conditions and the stage a verification condition generator (VCG).

In the second stage, the resulting formulas are given to an automatic theorem prover (TP) resp. SMT solver (in our case Z3 [3]) together with a a background theory capturing the semantics of C's built-in operators, etc. The prover checks whether the verification conditions are entailed by the background theory. Entailment implies that the original program is correct w.r.t. its specification.

3 The Possible Outcomes of Invoking a Verifying Compiler

In practice, where the limitations of resources are relevant, the possible outcomes of a verification attempt using a two-stage verifying compiler are:[1]

1. The formulas generated by the VCG are valid, and the TP has found a proof for that. This outcome entails that the original program has the specified properties.
2. Some generated formula is not valid, and the TP has found a counter-example. This can mean two things: (a) The program is not correct w.r.t. its specification, i.e., there is an error in either the program code or the specification. (b) The program satisfies the specification, but some loop invariant or other auxiliary annotation is not strong enough and, as a consequence, some generated verification condition is not a valid formula. We will discuss this distinction in more detail in Section 5.
3. The TP runs out of resources (time or space). This can mean three things: (a) The generated formula is valid and the program is correct (as in Case 1 above), but the TP could not find a proof in the allotted time/space. (b) The formula is not valid (as in Case 2 above), but the TP could not find a counter-example. The non-validity can, again, be due either to the program being incorrect or to some auxiliary annotation being not strong enough.

In Case 1 above, the invocation of the verifying compiler was successful—a desired but rare case in practice. Cases 2 and 3 are much more common, and the user has to analyze the problem. If they find (using the potential counter-example) that the program indeed does not satisfy the specification, the error has to be corrected. If they find that the program satisfies the specification, then new auxiliary annotations (stronger invariants, helpful lemmas, etc.) have to be added. This process is repeated until the program can be verified.

[1] We assume that the programs to be verified are of reasonable size such that only the theorem proving stage can run out of resources and not the VCG stage.

4 The Notion of Completeness for Verifying Compilers

To separate the annotations that are essential and the ones that are needed only for efficiency, one needs a clear understanding of the notion of completeness. In this section, we discuss what completeness means in the framework of verifying compilers and give a formal definition.

In general, completeness of a given calculus or system in program verification means that if a program P is correct w.r.t. its requirement specification REQ, in symbols $\models \langle P, REQ\rangle$, then this fact can be proved using the calculus resp. system, in symbols $Th \vdash \langle P, REQ\rangle$, where Th is a fixed set of axioms. The semantics of the programming language (used for P) and the annotation language (used for REQ) are encoded in the calculus rules and in the background theory Th. The restriction of resources (time and space) of real-world systems is usually not considered for the notion of completeness.

First off, because all non-trivial properties of programs are undecidable (Rice's Theorem), all program verification systems are necessarily incomplete. Instead the notion of *relative completeness* is used, i.e., completeness in the sense that the system or calculus would be complete if it had an oracle for the validity of formulas about arithmetic [2]. This can be formalized as follows:

Definition 1. Given a programming language, an annotation language, and a definition of when a program P satisfies a specification REQ, denoted by $\models \langle P, REQ\rangle$, a verification system S, consisting of $\vdash_S$ and Th_S, is *relatively complete* (w.r.t. arithmetics) if, for each program P and specification REQ with $\models \langle P, REQ\rangle$, there is a set $Arith$ of valid arithmetical formulas such that $Th_S \cup Arith \vdash_S \langle P, REQ\rangle$.

From here on, we assume that the axiomatization *Th* together with the calculus rules built into the theorem prover approximates arithmetic well enough, such that the valid arithmetic formulas that are actually needed in practice can be derived. Consequently, in the following, we do not need to distinguish between completeness and relative completeness. One has to keep in mind, though, that the distinction exists, even if we leave it out of our further considerations.

In the usual theorem proving setup, not much more is to be said about the notion of completeness, and one can fully concentrate on the problem of how to construct complete calculi and systems. The same could theoretically be demanded from verifying compilers as well. Then, no non-requirement annotations would be necessary, and the ones given would be solely for efficiency purposes. This is conceivable as, for instance, it is always possible to generate the strongest loop invariant automatically, based on Gödelization (cf. [6]). The reason it is not done in practice is that the resulting proof obligations would be extremely difficult to discharge.

Instead, all of today's deductive verification systems presuppose certain types of additional, non-requirement annotations to find proofs. It is neither given nor expected that a verifying compiler is relatively complete in the sense of Def. 1. In contrast, completeness of a verifying compiler means that if the program is correct w.r.t. its *given* requirement specification REQ, then some auxiliary specification AUX *exists* allowing to prove this.

Definition 2. A verifying compiler $S = (\vdash_S, Th_S)$, is *complete* if, for each program P and specification REQ with $\models \langle P, REQ\rangle$, there is (a) a set AUX of annotations and (b) a set $Arith$ of valid arithmetical formulas such that $Th_S \cup Arith \vdash_S \langle P, REQ{+}AUX\rangle$.

Of course, adding auxiliary annotations must strictly increase the strength of specifications, i.e., + must be monotonic:

Definition 3. The operator + for adding annotations is *monotonic* iff, for all programs P and all specifications $SPEC$ and $SPEC'$, if $\models \langle P, SPEC{+}SPEC'\rangle$ then $\models \langle P, SPEC\rangle$.

For example, adding a formula to a pre-condition, and thus weakening it, violates the condition of Def. 3 and is not an acceptable way of adding auxiliary annotations. Note also the difference between $\models$ and $\vdash$: Less annotations are easier to satisfy by the program ($\models$), while more annotations may be required to find a proof ($\vdash$).

The completeness of the whole verifying compiler process depends on completeness of the components of the toolchain. As already described, the toolchain usually consists of a VCG stage and an automated theorem proving or SMT backend. The VCG must be able to generate valid formulas provided the auxiliary annotations are sufficiently strong, i.e, if $\models \langle P, REQ\rangle$ then $Th \models_{FOL} VCG(P, REQ{+}AUX)$ for some AUX. Then the TP, in its turn, must be able to prove these valid formulas: $Th \vdash VCG(P, REQ{+}AUX)$.

The users, who serve as an oracle for finding auxiliary annotations that are strong enough to prove a given program correct, are not relevant for the completeness as long as they are considered to be omniscient and always find the required annotation (provided it exists). In practice, of course, users are not omniscient. They may very well fail to find the required auxiliary annotation, which may lead to a failure in the verification process even if the verification system is complete.

Note that, if one complete system S is stronger than another complete system S' because it can automatically derive additional annotations (it may, e.g., include a generator for loop invariants), then life is easier for the user of S; proofs will be found more often using S and with less effort (less auxiliary annotations). Nevertheless, both systems S and S' are complete; there are no different degrees of completeness.

5 Different Purposes of Program Annotations

Annotations can serve distinctively different purposes, though sometimes several different ones simultaneously. The following classification of annotations is neither syntactic nor semantic, but concerns rather the pragmatics of their use and the intentions of their author.

Requirement Annotations. Requirement annotations constitute the specification of the program. They assure the behavior of the program (module) towards its environment. They are the reason for performing verification. Typical requirement annotations are pre- and post-conditions, class invariants, or resource consumption limits. They are visible externally and cannot be changed easily.

Auxiliary Annotations. Auxiliary annotations are used to guide the proof search. They are usually not part of program requirements. As long as they satisfy their purpose, auxiliary annotations can be changed anytime without notice. We further distinguish two subclasses of auxiliary annotations:

(a) The first subclass is necessary merely for efficiency reasons. It encompasses lemmas, intermediate assertions, quantifier instantiation triggers, and the like. These annotations are not necessary for completeness. They can always be made obsolete by increasing the space/time available for proof search or by advances in SMT prover technology. Another purpose of annotations from this subclass is to inspect the proof state. For this, the user temporarily adds auxiliary annotations to get information about implicit "knowledge" of the proof system at particular points in the proof search – in order to eventually come up with the right auxiliary annotations needed to complete the proof (as defined in Def. 2). Due to

the design philosophy of verifying compilers, the user is presented with only a limited view on the resulting proof obligations and the proof search.

(b) The other subclass of auxiliary annotations are essential annotations. Getting them right is essential for completeness, the very existence of a correctness proof. The most prominent essential annotations are loop invariants. Further auxiliary annotations that can be essential are data-structure invariants and abstractions, ownership annotations, and framing conditions.

6 Possible Problems with Annotations

Annotations and Program Code Can Be In Conflict. A program P and an annotation $SPEC$ are in conflict if the program does not fulfill the specification: $\not\models \langle P, SPEC\rangle$.

Consider the code in Figure 1 together with the requirement to compute the minimum of a given array of length `size`. The pre-condition of the method (keyword `requires`) states that `array` points to a C array in memory with positive length `size`, which is not modified outside the current thread (the latter enables sequential reasoning). The post-condition of the method (keyword `ensures`) states that the result of the method is (a) less or equal than all elements and (b) contained in the array. We assume in the following that this is the right set of requirement annotations.

One possible error that could occur in the program is that the variable `min` has never been initialized (line labeled (A) missing). The resulting program is legal C code, but depending on the random initial value of `min` and the contents of the array, may fail to compute the minimum, and it does not satisfy the annotations.

For this conflict, the VCC system is able to provide a counter-example. It demonstrates that the third loop invariant does not hold when the loop is entered. The variable assignment returned as counter-example is: `size = 1`, `min = 0`, `array[0] = 1`.

Annotations Can Be Too Weak. An auxiliary annotation AUX is too weak if $\models \langle P, REQ{+}AUX\rangle$, i.e., the program is correct w.r.t. the specification, but this cannot be shown. There are now two cases to distinguish:

1. The VCG produces valid verification conditions, i.e., $Th \models_{FOL} VCG(P, REQ{+}AUX)$, and there is a proof for this, i.e., $Th \vdash VCG(P, REQ{+}AUX)$, but the TP stage runs out of resources before finding it.
2. Something essential is missing from AUX and at least one of the verification conditions generated by the VCG is invalid: $Th \not\models_{FOL} VCG(P, REQ{+}AUX)$, and (because of soundness) no proof exists, that is: $Th \nvdash VCG(P, REQ{+}AUX)$.

In Case (1), no counter-example is available and the user has limited recourse – to assist the user, VCC provides tools for inspecting the duration of proof attempts for single proof obligations as well as identifying axioms that are "costly" for the prover to instantiate, leading to an inefficient proof search. In Case (2), a counter-example for the validity of the verification condition may be constructed. We give an example for this latter case.

Assume that the third loop-invariant has been forgotten (label (B) in the program). Without that invariant, the system cannot verify the second post-condition. The generated counter-example is still the same as above, but this time it shows that the loop invariants (after the loop terminates) do not logically entail the post-condition.

Annotations Can Be Inadequate. An annotation is inadequate when it does not mean what its author thinks it means. Verification of inadequate annotations will thus not have the expected impact in the real world. Per its very nature, user input cannot easily be verified or tested for adequacy. But, apart from many systematic approaches for elicitation of requirements (which we will not cover here), there is a number of ways in which verification technology can assist its user to formulate meaningful specifications.

First, the builders of verification systems can work on formalisms that do not make it unnecessarily hard for the users to express their exact intentions. Second, the verification systems can produce a proof or a trace to justify the result. Inspection of the proof is a very effective—if costly—measure to combat misunderstandings in the meaning of the proof obligation. There are reports that users of verifying compilers monitor the prover running time to detect verification based on inadvertently inconsistent specifications (a particular case of inadequacy). In addition, VCC can check for inconsistencies in the specification by trying to prove $false$ at the different execution branches of the program – this of course can also only give an indication whether the specification is consistent or not.

Third, a whole new class of sanity checks based on mutation has been developed lately for automated program verification with model checking [8]. After a successful verification attempt, the query (the program or the specification) is mutated and the deduction is repeated. If verification succeeds again, then the mutated part of the query probably plays no role in determining the outcome. This indicates a problem with the query.

7 Lessons Learned

Throughout this paper, the distinction of requirement annotation and auxiliary annotation plays an important role, and it moreover plays an important role in practice.

Verifying compilers are currently not being designed for completeness in the sense that theorem provers are (Def. 1). They are designed for completeness in a different sense (Def. 2), requiring the user as an oracle to

```
int min(int *array, unsigned int size)
  requires(size > 0)
  requires(wrapped(as_array(array, size)))
  ensures(forall(int i; 0<=i && i<(int)size ==> result <= array[i]))
  ensures(exists(int i; 0<=i && i<(int)size && result == array[i]))
{
    int i, min;
    min = array[0];                                                    // * (A) *
    for (i = 0; i < (int)size; i++)
      invariant(0 <= i && i <= (int)size)
      invariant(forall(int j; 0 <= j && j < i ==> array[j] >= min))
      invariant(exists(int j; 0 <= j && j < (int)size && min == array[j]))   // * (B) *
    { if (array[i] < min) min = array[i]; }
    return min;
}
```

Fig. 1. Computing the smallest element of an array by simple iteration

provide sufficient auxiliary annotations in form of, e.g., loop invariants or assertions.

In theory the user could always give auxiliary annotations of maximal strength (i.e., logically entailing all other possible annotations), but this is not feasible in practice. Instead, one is interested in a weak set of auxiliary annotations that is still sufficient. Consequently, it is extremely important for the user to have knowledge about which kind of annotations are essential for the given VCG—even in cases where the requirement to be verified is comparatively simple. Without that knowledge they may continue to add the wrong annotations.

This need is to some extent in conflict with the verifying-compiler paradigm, which assumes that the user does not need to know about the inner workings of the verification system and calculus and can use the system as a "black box." At this point, stronger communication about the richness and significance of auxiliary annotations would be beneficiary. The verification systems should also provide better round-trip feedback to the user to discern and fix different kinds of annotation-related problems we have described.

We also suggest to enrich the annotation languages with a syntactical way (e.g., a key word) to distinguish between the two kinds of annotations. That would make annotations clearer and easier to read and understand. Moreover, annotations that are known to be purely auxiliary can be changed without harm during the verification process or when the implementation is changed.

References

1. M. Barnett, R. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS), International Workshop, 2004, Marseille, France, Revised Selected Papers*, LNCS 3362, pages 49–69. Springer, January 2005.
2. S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computing*, 7(1):70–90, 1978.
3. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc., 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, Budapest, Hungary*, LNCS 4963, pages 337–340. Springer, 2008.
4. R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.
5. J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. In *Formal Methods and Software Engineering*, LNCS 3308, pages 15–29. Springer, 2004.
6. D. Harel. *First-Order Dynamic Logic*. Springer, 1979.
7. C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.
8. O. Kupferman. Sanity checks in formal verification. In *Proceedings, 17th International Conference on Concurrency Theory*, LNCS 4137, pages 37–51. Springer, 2006.
9. W. Schulte, X. Songtao, J. Smans, and F. Piessens. A glimpse of a verifying C compiler. In *Proceedings, C/C++ Verification Workshop*, 2007.