Improving the Usability of Specification Languages and Methods for Annotation-based Verification*

Bernhard Beckert, Thorsten Bormer, and Vladimir Klebanov

Institute for Theoretical Computer Science,
Karlsruhe Institute of Technology, Germany
http://formal.iti.kit.edu

Abstract. It is widely recognized that human input is indispensable in deductive verification of real-world code. Verification engineers have to guide the proof search and provide information reflecting their insight into the workings of the program. Lately we have seen a shift towards an annotation-based paradigm – sometimes called "verifying compiler" –, where this information is provided in the form of program annotations instead of interactively during proof construction.
Suspicions have been growing recently that expressing verification knowledge as annotations in their current form suffers from serious scalability and maintainability issues.
In this paper, we pinpoint some of the biggest neuralgic spots and provide recommendations to the designers of annotation-based verification systems aimed to improve usability of specification languages and methods and, thus, the tool's productivity. We clarify the different purposes that annotations can serve and show why a certain class of annotations that are not program requirements is currently indispensable for proof construction. Moreover, we discuss how the use of data abstractions can be improved in annotation-based specifications.

1 Introduction

Program annotations as a form of interaction with the software verification tool have several important advantages. They make verification attempts self-contained, as human guidance is captured textually (and typically in terms closely related to the program at hand). They also keep the program and the specification close to each other, which is helpful as both unavoidably (co-)evolve.

At the same time, suspicions have been growing recently that expressing verification knowledge as annotations in their current form suffers from serious scalability and maintainability issues. This is part of a more general concern that writing specifications may turn into a bottle-neck for program verification (this observation was also, e.g., made in [13]).

* 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.

The problems with annotation-based verification arise as the lines between (a) requirement specification, (b) auxiliary information needed for proof construction, and (c) information for proof-search guidance get blurred. Even worse, related specification parts get dispersed and different levels of abstraction intermixed.

In this paper, we pinpoint some of the biggest neuralgic spots and provide recommendations to the designers of annotation-based verification systems aimed to improve usability of specification languages and methods and, thus, the tool's productivity.

After an introduction to the typical architecture and working cycle of annotation-based verification systems (Sect. 2), we discuss in Section 3 the different purposes that annotations can serve, based on a clarification of what the notion of completeness means in this framework. We show why a certain class of annotations that are not program requirements is currently indispensable for proof construction. Users are often surprised that they cannot omit certain non-requirement annotations even for the simplest (sub-)problems. We plead that they must be better educated about the inner workings of a verification system and what kinds of annotations are indispensable in which situations. This is in conflict with the desire to enable the user to work with the verification system as a "black box", which is generally seen as an important feature of the verifying compiler paradigm.

In Section 4, we discuss how the use of data abstractions can be improved in annotation-based specifications. The lack of syntactical separation between the (abstract) requirements and the non-requirement annotations obscures the inner structure of specifications and makes them hard to understand and maintain.

Finally, in Section 5 we draw conclusions from our research and discuss future work.

The problems we discuss are not inherent to annotation-based verification (nor the verifying compiler approach), but rather due to the currently implemented design decisions of such verification systems. All of the issues mentioned in this work can be overcome by extending specification languages and methodologies. The general, central idea of annotating programs at source-code level – using a language whose syntax is closely related to the programming language – is not the source of the problems described in this paper.

2 Inside a Typical Annotation-based Verification System

Tools following the annotation-based paradigm include Spec# [1], VCC [12], Caduceus [7] and others. They are all based on powerful fully-automatic provers and decision procedures, and they support real-world programming languages such as C and C#.

Compared to fully automatic verification approaches like model checking or abstract interpretation that need no human interaction, the annotation-based paradigm allows for full functional verification of programs. But the former meth-

ods are either restricted in the expressiveness of specifications, the precision of results, or in the kind of programs that can be verified.

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.

2.1 Structure of the Toolchain

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 postconditions. 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 annotation-based verification systems 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 [6]. 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 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 [5]) together with 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.

See Sections 3 and 4 for some real-world examples for specification and verification of C programs with VCC.

2.2 The Possible Outcomes of Invoking an Annotation-based Verification Tool

In practice, where the limitations of resources are relevant, the possible outcomes of a verification attempt using a two-stage annotation-based verification system 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.

[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.

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 missing or 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 3.1.
3. The TP runs out of resources (time or space). This can mean two 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 verification system 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.

3 Distinguishing Different Kinds of Annotations

3.1 Annotations and their Properties

Preliminaries. In the following we assume as given a programming language, and an annotation language for expressing specifications. Which annotations are possible depends on the particular language; typical annotations are for example invariants, pre-/postcondition pairs, and assertions of various kinds. In order to easily relate alternative potential annotations for the same program, we take the view that annotations are disjoint from regular program statements. On the other hand, each annotation has an *intended context* (statement, method, class, etc.). We assume that we only deal with combinations of programs and annotations without context mismatches, i.e., annotations are compatible with the programs to which they are added. The context an annotation refers to must actually exist in the program, and the symbols used in the annotation must be defined for that context.

Definition 1 (Combination of program and annotations). *If P is a program and A is a set of annotations compatible with P, then we call the pair $P +_< A$ the* combination *of the two. The parameter $<$ fixes the order of annotations if several of them have the same intended context. We will omit the ordering whenever it is irrelevant or clear from the context and simply write $P+A$.*

Definition 2 (Annotation satisfaction). *We assume that there is a definition of when a program P satisfies a specification REQ, denoted by $\models P+REQ$.*

Definition 3 (Strength of annotations). *An annotation A is* (logically) stronger *than an annotation B, in symbols $A \Rightarrow B$, if $\models P{+}B$ holds for all programs P with $\models P{+}A$.*

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 postconditions, 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. 7).

(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.

Monotonicity of Auxiliary Annotations. A very desirable property of annotation satisfaction is monotonicity. Adding auxiliary annotations should strictly increase the strength of the specification, i.e., $\models$ should be monotonic w.r.t. adding annotations.

Definition 4 (Monotonicity of $\models$). $\models$ *is* monotonic *w.r.t. adding annotations iff, for all programs P and all specifications REQ and AUX the following holds:*

$$\textit{if} \;\; \models P{+}(REQ \cup AUX) \;\; \textit{then} \;\; \models P{+}REQ \;\; .$$

In reality, monotonicity of $\models$ w.r.t. adding auxiliary annotations is not given unless we make some restrictions. One concerns `assume` annotations that add an unchecked assumption to the following proof (and thus can make a specification weaker). Since all proved properties only hold modulo these assumptions, `assume` annotations are a correctness risk. For these reasons we always classify `assume` annotations as part of the requirement and never as auxiliary.

In the same vein, adding a formula to a precondition, and thus weakening it, violates the condition of Def. 4 and is not an acceptable way of adding auxiliary annotations.

3.2 Annotations and Existence of Proofs

To separate the annotations that are essential for the existence of a proof and the ones that are needed only for supporting proof search, one needs a clear understanding of the notion of completeness. In this section, we discuss what completeness means in the framework of annotation-based verification systems and give formal definitions.

Completeness and Relative Completeness. The classical notion of completeness for deduction systems can be adapted to annotation-based verification systems as follows:

Definition 5 (Completeness). *Let S be a verification calculus or system. S is* complete *if, for any program P satisfying its requirement specification REQ, this fact can be proved using the calculus from a fixed set of axioms Th_S. In symbols:*

$$\textit{if} \ \models P{+}REQ \ \textit{ then } \ Th_S \vdash_S P{+}REQ$$

The semantics of the programming language (used for P) and the annotation language (used for REQ) are encoded in the calculus rules $\vdash_S$ and in the background theory Th_S. The restriction of resources (time and space) of real-world systems is usually not considered for the notion of completeness.

Note also the difference between $\models$ and $\vdash$: Fewer annotations are easier to satisfy by the program ($\models$), while more annotations may make it easier to find a proof ($\vdash$).

Since first-order arithmetics is undecidable, all non-trivial properties of programs are undecidable (Rice's Theorem), and all program verification systems are necessarily incomplete in the sense of Def. 5. 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 [4]. This can be formalized as follows:

Definition 6 (Relative completeness). *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*

$$\models P{+}REQ \ ,$$

there is a set Arith of valid arithmetical formulas such that

$$Th_S \cup Arith \vdash_S P{+}REQ \ .$$

Luckily, undecidability of first-order arithmetics is usually not an obstacle for verification in practice. Experience shows that the axiomatization *Th* together with the calculus rules of the theorem prover approximate arithmetics well enough and that the valid arithmetic formulas occurring in practice can be derived (which does not imply that finding a derivation is easy or possible automatically but only that a derivation exists). One has to keep in mind, that the distinction between completeness and relative completeness exists, even if the restriction to relative completeness is not a real limitation in practice.

Theoretical Completeness Arguments. Relatively complete calculi exist for many program logics. Harel gives one for first-order Dynamic Logic in [8]. Less-known is the fact that the presence of auxiliary annotations, such as loop invariants, is not a prerequisite for relative completeness.

Harel conducts his relative completeness proof by showing that program logics are no more expressive than first-order arithmetics. That is, for every program there is a first-order arithmetics formula that encodes the same relation between states that the program encodes. The less-known fact is that it is possible to effectively compute such a formula without further input. In fact, Harel's proof contains a simple algorithm that for any Dynamic Logic formula ϕ effectively computes an equivalent purely first-order arithmetics formula ϕ_A [8, Theorem 3.2]. This construction gives along the way a means to automatically compute the strongest invariant of any loop.

The algorithm is based on Gödelization, i.e., encoding a finite sequence of domain elements into one element. The generated invariant formula asserts the existence of a number encoding a sequence of states corresponding to the forthcoming computation sequence of the loop until it terminates.

Thus, theoretically speaking, the strongest loop invariant for any given loop and so the verification conditions for any given piece of code can be easily computed in polynomial time. That is not a contradiction to general undecidability of program verification, since one undecidable problem (program verification) gets transformed into another undecidable problem (deciding first-order logic with arithmetics). Still, assuming a theoretical standpoint, one can conclude that no auxiliary annotations are really needed because all information contained in annotations can easily be computed by the VCG.

In Practice: Annotation Completeness. The theoretical fact that all necessary annotations can "easily" be constructed (see above) is in practice a red herring because the constructed annotations use Gödelization and, thus, complex arithmetics. Proof obligations generated from such annotations would be impossible to discharge by existing theorem provers. For practical purposes one needs instead annotations containing all the necessary information in a clear and direct manner and not obscured by Gödelization.

Therefore, in contrast to theory, all of today's deductive verification systems presuppose certain types of additional, non-requirement annotations to be given by the user. It is neither given nor expected that an annotation-based verification system is relatively complete in the sense of Def. 6. In practice, completeness of a verification system 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 7 (Annotation completeness). *A verification system* $(\vdash_S, Th_S)$ *is* annotation complete *if, for each program* P *and specification* REQ *with*

$$\models P{+}REQ \ ,$$

there is (a) a set AUX *of annotations (not containing any* `assume` *clauses), (b) an order* $<$ *on the annotations, and (c) a set* $Arith$ *of valid arithmetical formulas such that*

$$Th_S \cup Arith \vdash_S P{+}_<(REQ \cup AUX) \ .$$

The completeness of the whole verification 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,

$$\text{if} \ \models P{+}REQ \ \text{ then } \ Th \models_{FOL} VCG(P{+}(REQ \cup AUX))$$

for some AUX. Then the TP, in its turn, must be able to prove these valid formulas:

$$Th \vdash VCG(P{+}(REQ \cup AUX)) \ .$$

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 annotation-complete system S is stronger than another annotation-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 annotation-complete; there are no different degrees of annotation completeness.

Essential and Non-essential Annotations. When a verification system is used that is annotation complete (Def. 7) but not relatively complete (Def. 6),

i.e., any annotation-based verification system, then there are *essential* auxiliary annotations that cannot be omitted without losing the existence of a proof. Besides such essential annotations there are *non-essential* annotations that are not needed for the existence of a proof but for finding it more easily.

Definition 8 (Essential annotation). *Given a verification system* $(\vdash_S, Th_S)$*, a program* P*, a specification* REQ *with* $\models P{+}REQ$*, and a set* AUX *of annotations and an order* $<$ *with*

$$Th_S \cup Arith \vdash_S P{+}(REQ \cup AUX)$$

for some set $Arith$ *of valid arithmetical formulas.*

A subset $AUX_{ess} \subset AUX$ *is* essential *if*

$$Th_S \cup Arith \nvdash_S P{+}(REQ \cup (AUX \setminus AUX_{ess})) \ .$$

Otherwise it is non-essential.

The notion of essential annotations (Def. 8) has some awkward properties, which make it difficult to recognize essential annotations in practice. It is possible that some subsets $A, A' \subset AUX$ are both essential but $A \cup A'$ is not. This happens frequently if, for example, A is needed for the proof of A' and A' is needed for the proof of A. Also, there is in general no single minimal set of essential annotations. In fact there may be completely different sets of essential auxiliary annotations for proving the same requirement that are both minimal but disjoint.

Besides the question of whether an annotation may be omitted or not, one may also be interested in the question of whether it can be replaced by a weaker annotation.

Definition 9 (Strongly essential annotation). *A subset* $AUX_{ess} \subset AUX$ *is* strongly essential *if*

$$Th_S \cup Arith \nvdash_S P{+}REQ \cup ((AUX \setminus AUX_{ess}) \cup AUX')$$

for all AUX' *that are weaker than* AUX_{ess}*, i.e.,* $AUX_{ess} \Rightarrow AUX'$*.*

While an auxiliary annotation that is strongly essential cannot be replaced by a weaker annotation, it may well be possible to replace it by an equivalent annotation that is "simpler" in some practical way not covered by Definition 9 (e.g., easier to understand for human users).

Note that even with the notion of strongly essential annotations, there is in general no single minimal set of auxiliary annotations.

It is important for a user to know which annotations are essential because during the verification process many auxiliary annotations are added. And as too many annotations clutter the program and make it harder to find a proof, users often remove unneeded annotations. This carries the danger that simple but essential annotations get removed by accident, which – as experience shows – leads to hard to solve problems in the search for a set of annotations with

which a proof can be constructed. Thus, to understand which annotations may be essential, users have to possess a certain knowledge about the inner workings of a verification system. As further discussed in Section 3.4, 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.

3.3 Possible Failures in Authoring Annotations

In the following, we use a concrete example to illustrate the three different ways in which authoring annotations may fail.

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: $\not\models P{+}SPEC$.

Consider the code in Figure 1 together with the requirement to compute the minimum of a given array of length `size`. The precondition 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 than or equal to 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 second 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 $\models P{+}REQ \cup AUX$, 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.,

$$Th \models_{FOL} VCG(P{+}(REQ \cup AUX)) \ ,$$

   and there is a proof for this, i.e.,

$$Th \vdash VCG(P{+}(REQ \cup AUX)) \ ,$$

   but the TP stage runs out of resources before finding a proof.

```
#define uint unsigned int

int min(int *array, uint size)
  _(requires size > 0)
  _(requires \mutable_array(array, size))
  _(ensures  \forall uint i; 0<=i && i<size ==>
                        result <= array[i])
  _(ensures  \exists uint i; 0<=i && i<size &&
                        result == array[i])
{
    uint i;
    int min;
    min = array[0];                                    // * (A) *
    for (i = 0; i < size; i++)
      _(invariant \forall uint j; 0 <= j && j < i ==>
                              array[j] >= min)
      _(invariant \exists uint j; 0 <= j && j < 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

2. Something essential is missing from AUX and at least one of the verification conditions generated by the VCG is invalid:

$$Th \not\models_{FOL} VCG(P+(REQ \cup AUX)) \ ,$$

and (because of soundness) no proof exists, that is:

$$Th \not\vdash VCG(P+(REQ \cup AUX)) \ .$$

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 second loop-invariant has been forgotten (label (B) in the program in Fig. 1). 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. By 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 are 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 verification systems 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 [10]. 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.

3.4 Improving the Annotation Languages and Methodologies

Annotation-based verification systems are currently not designed for completeness in the sense that theorem provers are (Def. 6). They are designed for completeness in a different sense (Def. 7), requiring the user as an oracle to provide sufficient auxiliary annotations in the form of, e.g., loop invariants or assertions.

Theoretically 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. It is therefore essential to provide user documentation on what kind of auxiliary annotations are needed by a verification system.

Moreover, requirement and auxiliary annotations must be syntactically distinguished. That makes specifications clearer and easier to read and understand. In certification processes it is indispensable to have a very clear understanding of which annotations form the requirement specification that has been verified.

It is preferable to keep the two in separate files: for instance, requirement annotations in the header file and auxiliary annotations in the C source file. Where no such separation is possible, keywords (in the style of visibility modifiers `public` and `private`) should be used.

4 Using Data Abstractions in Annotation-based Verification Systems

In this section, we discuss the use of data abstractions in annotation-based verification systems, and how it can be improved. For that, we first introduce parts of the VCC methodology and the VCC annotation language only as far as needed for the examples. For a more detailed description of the VCC methodology, see [3, 2].

The example we use as illustration is taken from the 1st Verified Software Competition [9]. The goal of the competition was to implement, formally specify and verify an algorithm that solves a problem defined in natural language. Our example is based upon the following requirement:

> **Problem: Searching a linked list.** Given a linked-list representation of a list of integers, find the index of the first element that is equal to zero. Show that the program returns a number i equal to the length of the list if there is no such element. Otherwise, the element at index i must be equal to zero, and all the preceding elements must be non-zero.

4.1 The VCC Approach

The particular solution presented here, consisting of a C implementation and a specification in the VCC language, was developed after the competition by the team "VC Crushers" (see [9]).[2] It is an optimized version of their competition solution – the amount of auxiliary annotations is kept to a minimum sufficient to verify the requirement specification. Another VCC formalization of the list data structure that introduces a more general abstraction suited for a large range of applications is also made available by the team (because of space restrictions, we cannot include this more general but also more complex solution here).

The concrete C implementation of the list data structure and the C method `find` that is a solution to the above problem definition is shown in Fig. 2. Annotations are given in VCC syntax and are enclosed in labelled frames throughout the code.

Linked List Data Structure. In the implementation of the linked list data structure (`struct List`), the field `data` contains the value stored in a node of the list, and the field `tail` points to the rest of the list. Note that each node of the list also stores the length of its tail (including the node itself). In this implementation, the end of a list is thus not indicated by a null pointer or a sentinel node but by the value of `length` being zero.

The semantics of the length field as well as an abstract representation of the list's contents are specified by the object invariants in the block labelled $\langle OI \rangle$: For each node, information about the elements of the sublist starting at that node is stored in the map `vals` in ghost state (a specification state separate

[2] For better readability, we have slightly modified the source code of the example.

```
typedef struct List {
  int data;
  struct List *tail;
  unsigned length;                                                    ⟨OI⟩
  _(ghost int vals[unsigned])
  _(invariant vals == \lambda unsigned i;                          //(I1)
      i < length ? (i == 0 ? data
                           : tail->vals[i - 1])
                 :  0
   )
  _(invariant length == 0                                          //(I2)
      || (\mine(tail) && length == tail->length + 1))
} List, *PList;

unsigned find(PList l)                                                ⟨MC⟩
  _(requires \wrapped(l))
  _(ensures \result <= l->length)
  _(ensures \result < l->length
              ==> l->vals[\result] == 0)
  _(ensures \forall unsigned i; i < \result
              ==> l->vals[i] != 0)
{
  PList p;
  for (p = l; p->length != 0; p = p->tail)                            ⟨LI⟩
    _(invariant p->length <= l->length)                            //(I3)
    _(invariant p \in \domain(l))                                  //(I4)
    _(invariant p->vals == \lambda unsigned j;                     //(I5)
         j < p->length
            ? l->vals[l->length - p->length + j]
            : 0)
    _(invariant \forall unsigned j;
         j < l->length - p->length ==> l->vals[j] != 0)
  {                                                                   ⟨AN⟩
_(assert \forall unsigned j; j < p->tail->length
           ==> p->tail->vals[j] == p->vals[j + 1])
    if (p->data == 0) {
      break;
    }
  }
  return l->length - p->length;
}
```

Fig. 2: Annotated C source code of `find`.

from the normal C memory). The invariant I_1 defines the abstraction relation between the list and its abstraction `vals`. The abstraction from linked list to array is needed because the built-in data type array allows quantification and recursion over the elements of the list. A direct quantification over the elements is not possible in first-order logic because reachability is not first-order definable.

To be able to access all elements of the list (via the `data` field of the structure), each node is given exclusive ownership to the next node in the list (`mine`-annotation of invariant I_2). In addition, the invariant I_2 implies acyclicity of the list, as length is bounded and decreasing for each element that can be reached via the `tail` pointer.

Implementation of the `find` Algorithm. Using the C data structure `List`, the `find` method can be implemented by iterating over the list's elements via the tail pointer in a `for`-loop.

Annotations are used within the method at three locations – namely for the method contract $\langle MC \rangle$, the loop invariant $\langle LI \rangle$ and as an auxiliary annotation inside the loop body $\langle AN \rangle$.

The method contract in block $\langle MC \rangle$ specifies that the behavior of the method conforms to its specification in natural language. The pre- and postconditions $\langle MC \rangle$, together with the invariants $\langle OI \rangle$ of the list data structure, capture the intended semantics of `find`. $\langle MC \rangle$ and $\langle OI \rangle$ constitute the methods *requirement specification*.

To be able to verify the implementation of `find` to be correct w.r.t. to its requirement specification, a set of four essential auxiliary annotations has to be provided the in form of loop invariants (block $\langle LI \rangle$).

The invariants I_3 and I_5 state that the abstraction of the "iterator" `p` is always a suffix of the abstraction of the input list `l`.

As each access to a list element `p->data` inside the loop has to be shown to be a valid access, additional justification has to be provided to VCC to be able to prove this. This justification is given with the invariant I_4 – the property depends on the fact that `p` is also a sublist of `l` in the concrete representation.

Even in this simple example, a non-essential auxiliary annotation is needed at location $\langle AN \rangle$ in order to be able to show that the loop body preserves the third loop invariant. It asserts that the `vals` fields related to two adjacent nodes are abstractions of the appropriate sublists starting at those nodes.

In order to come up with the right auxiliary annotations, in this case, the user has to know about the inner workings of VCC, respectively the strengths of the underlying SMT solver Z3.

4.2 Separation of Concerns: Annotation-based Verification and Algebraic Specifications.

In the example shown in the previous section, annotations with different purposes are intermingled. In the following, we suggest to use techniques known from

abstract data type specifications to provide a clean separation of requirement and auxiliary annotations, and to make the specification more readable.

Assuming that we have defined an abstract data type `IntList`, an abstraction function `abs` from concrete linked lists to abstract lists, as well as an abstract (specification) function `absfind` on `IntList`, a good method contract for `find` could look like this:

```
unsigned find(PList l)                                          ⟨MC⟩
  _(requires \wrapped(l))
  _(ensures \result == absfind(abs(l)))
```

This contract is very compact and easy to understand. It simply states that `find` returns the same integer value that is the result of the abstract operation `absfind` on the abstraction of the input list.

In the VCC example presented in Sect. 4.1, the equivalent of `absfind` is implicitly given by postconditions $\langle MC \rangle$ of method `find`. The abstraction function `abs` is concealed within the definition of structure `List`, namely in the invariants in $\langle OI \rangle$.

Of course, to complete the specification, we now have to define `IntList` and `absfind`. The required syntax is not available in VCC (yet). We suggest to use a syntax for abstract data type definitions based on the Common Algebraic Specification Language (CASL) [11]. A possible definition then would look like this:

```
spec IntList =
    free type List ::= nil | cons(Int; List)
then vars i, e: Int; l, l': List
    op absfind : List -> Int
      * absfind(nil) = 0
      * absfind(cons(e, l)) = 1 when e = 0
                              else absfind(l) + 1
    ops append : List x List -> List
        tail : List -> List
      * append(nil, l) = l
      * append(cons(e,l'), l) = cons(e, append(l',l))
      * tail(nil) = nil
      * tail(cons(e,l)) = l
    within implementation find
end
```

To be able to specify the implementation of `find` with the help of the abstract data type, the data type is equipped with the externally visible operation `absfind` that captures the semantics of `find` according to the problem definition. In addition, the two operations `append` and `tail` are defined with the usual semantics.

Compared to the rudimental abstraction given by the map `vals` in Sect. 4.1, this specification is additional overhead in terms of lines of code. However, the

compactness of the map specification is partly due to the fact that maps are a built-in feature of the verification tool. Furthermore, the flexibility offered by defining arbitrary abstract data types in our opinion clearly outweighs the annotation overhead, as we can choose the abstract data type representation that matches the implementation data types best. Lastly, the above definition is to a large extent reusable and can be seen as a sort of library definition.

To relate the concrete implementation data type `List` to its abstract data type counterpart `IntList`, we have two options: (A) defining an abstraction function `abs` from `List` to `IntList` (as already mentioned above), and (B) axiomatizing the abstraction using concrete implementations for the (abstract) constructors `nil` and `cons`.

For option (A), the abstraction function `abs` is defined in terms of the concrete implementation details (i.e., field `data` and pointer `tail`):

⟨*OI*⟩

```
_(spec IntList abs(PList l)
    returns(l->length == 0 ? nil :
                  cons(l->data, abs(l->tail)))
 )
```

With this definition, one of our goals is already achieved, namely providing a clearly differentiated and discernible specification construct that couples the concrete and abstract data types. However, this definition does not hide the implementation details of the linked list data structure from callers of the `find` method.

The alternative solution (B) uses concrete implementations of the constructors of the list data structures (not shown in this paper). Then, no explicit definition of the abstraction function as in (A) is needed. The relation between the abstract and the concrete constructors can, for example, be specified as follows:

⟨*OI*⟩

```
PList cons(int e, PList l)
  _(requires \wrapped(l))
  _(ensures abs(\result) == cons(e, abs(l)) )
```

Note that the above method contract of `cons` does not use any implementation details of the linked list data type in C, so that these details do not become part of the requirement specification.

Regardless of which alternative (A) or (B) is chosen, due to the separation of abstract and concrete representation of the list data type, the annotation overhead of the `List` data structure can be reduced:

```
typedef struct List {
  int data;
  struct List *tail;
  unsigned length;
```

```
  _(invariant \exists IntList l; abs(this) == l
       && (abs(this) == nil || \mine(tail)))
} List, *PList;
```

Only one invariant of `list` remains that is concerned with the state of the structure according to the VCC methodology (keyword `mine`), as well as enforcing the existence of an abstract element corresponding to the concrete instance of the list (which rules out cyclic linked lists).

Using the two "library" functions `tail` and `append`, almost all auxiliary annotations of our example can be simplified – for the loop invariants at location $\langle LI \rangle$ the new annotations are:

```
    _(invariant p \in \domain(l))
    _(invariant \exists IntList front;
        append(front, abs(p)) == abs(l)
        && absfind(f) == 0)
```

Furthermore, the single non-essential annotation at location $\langle LI \rangle$ becomes:

```
    _(assert abs(p->tail) == tail(abs(p)))
```

5 Conclusions and Future Work

We have described and analyzed problematic properties of current annotation-based specification practices. A large part of the problem is the non-discriminatory use of the same language to specify both requirements and auxiliary (non-requirement) annotations. Among the latter some are (only) needed for efficiency of proof search, while others are essential for completeness, i.e., indispensable for proof construction. In practice, we often encountered confusion as to what the important notion of completeness means in the framework of verifying compilers. We have provided a clarification in Section 3.2. We have recommended measures to alleviate the specification bottleneck in Sections 3.4 and 4.2. Beyond that, we see the following issues as worth further exploration.

The distinction between requirement and auxiliary annotations is always relative to a module boundary. Some aspects of modularity are imposed by the programming language, while others have to be defined by the specification language and verification calculus. The designers of the latter should make their modularity concepts (syntactically) explicit and better educate the users about them. A hiding operator for annotations may be appropriate when composing modules.

It is important to keep related parts of specifications together and not mix different levels of abstraction. This tenet is frequently violated by the widespread practice of mixing ghost code (providing an abstract specification) and real code (implementing it) – often on the level of individual statements. While a separation is desirable, it is not yet clear how to disentangle the two.

The verification system should track (and disclose) dependencies between annotations. All annotations should carry a unique textual identifier (label). For any given annotation, the user must be informed about which other annotations are necessary to prove it. One way to accomplish this is by automated deduction on the part of the system. Another way is to demand that the user explicitly attach to each annotation a set of labels naming other annotations that are to be considered as premisses in the proof.

References

1. M. Barnett, K. R. M. 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. B. Beckert and M. Moskal. Deductive verification of system software in the Verisoft XT project. *KI*, 2009. Online first version available at SpringerLink.
3. E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42, Munich, Germany, 2009. Springer. Invited paper.
4. S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computing*, 7(1):70–90, 1978.
5. 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.
6. 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.
7. 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.
8. D. Harel. *First-Order Dynamic Logic*. Springer, 1979.
9. V. Klebanov, P. Müller, N. Shankar, G. T. Leavens, V. Wüstholz, E. Alkassar, R. Arthan, D. Bronish, R. Chapman, E. Cohen, M. Hillebrand, B. Jacobs, K. R. M. Leino, R. Monahan, F. Piessens, N. Polikarpova, T. Ridge, J. Smans, S. Tobies, T. Tuerk, M. Ulbrich, and B. Weiß. The 1st Verified Software Competition: Experience report. In M. Butler and W. Schulte, editors, *Proceedings, 17th International Symposium on Formal Methods (FM)*, volume 6664 of *LNCS*. Springer, 2011. Materials available at `www.vscomp.org`.
10. O. Kupferman. Sanity checks in formal verification. In *Proceedings, 17th International Conference on Concurrency Theory*, LNCS 4137, pages 37–51. Springer, 2006.
11. P. D. Mosses, editor. *CASL Reference Manual – The Complete Documentation of the Common Algebraic Specification Language*, volume 2960 of *Lecture Notes in Computer Science*. Springer, 2004.
12. W. Schulte, X. Songtao, J. Smans, and F. Piessens. A glimpse of a verifying C compiler. In *Proceedings, C/C++ Verification Workshop*, 2007.
13. A. Zeller. Mining specifications: A roadmap. In *Proceedings, The Future of Software Engineering, Zurich, Switzerland*, pages 173–182. Springer, 2010.