Integration of Bounded Model Checking and Deductive Verification

Bernhard Beckert, Thorsten Bormer, Florian Merz, and Carsten Sinz

Department of Informatics
Karlsruhe Institute of Technology (KIT), Germany
http://{formal|verialg}.iti.kit.edu

Abstract. Modular deductive verification of software systems is a complex task: the user has to put a lot of effort in writing module specifications that fit together when verifying the system as a whole. In this paper, we propose a combination of deductive verification and software bounded model checking (SBMC), where SBMC is used to support the user in the specification and verification process, while deductive verification provides the final correctness proof. SMBC provides early – as well as precise – feedback to the user. Unlike modular deductive verification, the SBMC approach is able to check annotations beyond the boundaries of a single module – even if other relevant modules are not annotated (yet). This allows to test whether the different module specifications in the system match the implementation at every step of the specification process.

1 Introduction

Deduction-based methods for software verification and systematic debugging have seen a tremendous progress in recent years. Up to the nineties, the main focus in these areas was on fundamental research – the methods developed during this period were applicable only to small, academic software systems. Since the beginning of this century, a set of new techniques emerged that puts into reach the application of these techniques to real-world software systems.

What is needed now are methods and tools to help the user in writing modular specifications for complex systems and support the verification process. The size of real-world systems, together with a large amount of interdependencies of functions and data structures in the system make them hard to specify.

Modular specification and verification techniques are essential to be able to verify large software systems for several reasons. On the one hand, current deductive verification tools do not scale to large systems if not taking advantage of modularization. On the other hand, coming up with a requirement specification of a whole system at once is often a non-trivial task and better split up into specification of smaller modules. In addition, the reasons for a failed verification attempt are also difficult to identify if modularity is not taken into account.

Therefore, in most cases, verification tools require to decompose the specification and verification task into smaller modules, e.g., by using contracts for

each module in proofs instead of the module's implementation. Some verification tools allow the user to choose whether to use a module's contract or its implementation in a proof – however, inlining the implementation is only viable for simpler modules, due to scalability issues.

The disadvantage of verifying each module in isolation is that the specification of a module may not fit the other parts of the system where the module is used. This may result in many iterations of changes to the specification of all modules until a fix-point is reached that allows verification of the full system.

On the other hand, approaches like software bounded model checking (SBMC) are able to analyze a system beyond module boundaries, though this ability does not come for free: The specification languages of SBMC tools are not as expressive, and they have less precision as compared to deductive verification systems.

In this paper, we propose a combination of deductive verification and SMBC, where SBMC is used to support the user in the specification and verification process while deductive verification provides the final correctness proof. For this, auxiliary specifications that the user adds as annotations to the system are translated into input for the SBMC tool. As the user's annotations are aimed at deductive verification, the SBMC tool may not be able to handle them, but in many cases it can provide early feedback. In particular, SBMC can check the appropriateness of the specifications beyond the boundaries of a single module – even if other relevant modules are not specified (yet). This allows to test early on whether the different module specifications in the system match the implementation at every step of the specification process.

In Section 2, we give a brief introduction to deductive verification, followed by a description of the problems encountered when deductively verifying software systems in practice in Section 3. Section 4 contains a short explanation of software bounded model checking and its relation to deductive verification, and Section 5 a brief description of how to translate from the annotation-based specification used in deductive verification into input for the BMC tool. Section 6 presents the main contribution of this work, namely the integration of annotation-based deductive verification and software bounded model checking. In Section 7 an extended example is given describing the advantages of this combination, and this example is then evaluated in Section 8. Finally, Sections 9 and 10 are concerned with related work, conclusion, and future work.

As the basis for the work presented in this paper, we used the deductive verification tool VCC [4] and the SBMC tool LLBMC [15]. Accordingly, our verification targets are C programs, and annotations are written in VCC's specification language. Nevertheless, the ideas presented here are not restricted to procedural programming languages like C, and can also be applied to programs written in object-oriented languages like Java or C++. Abstract types (interfaces), e.g., can be dealt with by providing suitable contracts for these interfaces or by giving a set of concrete instantiations. Dynamic typing can be taken into account by replacing method calls by case discrimination over the possible dynamic types. The fundamental problem of how to engineer suitable annotations for verification remains largely unchanged compared to procedural languages.

2 Basics of Deductive Verification

2.1 The Annotation-based Verification Paradigm

Annotation-based deductive verification allows to obtain a rigorous mathematical proof for the correctness of a software system w.r.t. its formal functional specification. Verification tools in this category are based on a specific style of user interaction, called *auto-active* [10]. Proof construction is not interactive, but all information needed for finding a correctness proof, including auxiliary specifications, have to be provided by the user before the tool is run – there is no provision for manual intervention during the proof construction process.

Specifications are written directly in the source code as annotations in a way that does not influence the execution of the software system. Often, these annotations feature a syntax that is close to the syntax of expressions of the target programming language, enriched by constructs of first-order logic.

A typical example for an annotation-based verification system is VCC [4], a deductive verification tool for concurrent C programs. It uses the Boogie tool [1] to generate verification conditions in first-order logic. The generated FOL formula has the property that it is unsatisfiable iff the program fulfills its specification. An automated theorem prover, in this case Z3 [5], is then used to show unsatisfiability of the formula. If the formula is satisfiable, Z3 can often find models for it, which can be translated back to traces of the program that violate the specification [12].

2.2 Verification Targets

In the following, we consider the verification targets to be C programs, containing a set of function definitions. We consider these functions to be the modules of the program. We say that a C function f_A depends on a function f_B iff the function body of f_A (syntactically) contains a function call to f_B. This dependency relation, together with the set of functions of the system forms a directed graph. These graphs may contain arbitrary cycles depending on the implementation of the functions. In the following, for simplicity, we assume the graphs to be acyclic. In practice, mutually recursive functions would have to be specified together in one step and the verification methodology has to ensure that no cyclic reasoning occurs. For longer cycles in the call graph, techniques such as program slicing would allow us to split the graph and consider acyclic parts separately.

Similar to the notion of a root in a tree, we define as the roots in the dependency graph any node without a parent. The depth of a node is defined as the length of the longest path from this node to any of the roots in the graph. The set of all functions that a function f depends on is called $\mathit{children}(f)$; conversely, the set of all functions that depend on f is called $\mathit{parents}(f)$.

The depth of a node can be used to introduce a (topological) ordering $\prec$ on the nodes of the graph: $f_1 \prec f_2$ iff $\mathit{depth}(f_1) < \mathit{depth}(f_2)$. In the following, we identify functions with their corresponding nodes in the dependency graph, and we use the terminology of order theory for functions where appropriate.

2.3 Annotations and Their Semantics

In modular deductive verification, the specification *SPEC* of a software system S is composed of the specifications of its modules (C functions) and data structures. We assume that *SPEC* is a set of annotations, where each annotation consists of (a) one or more expressions of the specification language (pre-/post-conditions, invariants, assertions, etc.) and (b) the position of the annotation in the program. We further assume that each annotation is local to a single function of the specified system, i.e., *SPEC* is the disjoint union $SPEC = SPEC_1 \cup \ldots \cup SPEC_n$ of specifications $SPEC_i$ for each of the functions f_i of which S consists.

The binary relation $\models$ between programs and (sets of) annotations denotes the semantics of specifications, i.e., $S \models SPEC$ iff the software system S satisfies the specification *SPEC* according to the definition of the specification language. Since the specification languages we consider are modular, we have

$$S \models SPEC \quad \text{iff} \quad f_1 \models SPEC_1, \ldots, f_n \models SPEC_n \ .$$

Also, the relation $\models$ is monotonic w.r.t. adding annotations:

$$S \models SPEC \cup SPEC' \text{ implies } S \models SPEC \ .$$

This monotonicity condition requires, for example, that a pre-condition is not considered to be an annotation on its own but only in combination with a post-condition. Adding a lone pre-condition may weaken a specification while adding a pre-/post-condition pair always strengthens it.

2.4 Deductive Verification Systems

To prove that a software system satisfies its specification, we use a verification system V (in our case the VCC tool). The relation $S \vdash_V SPEC$ denotes that V is able to prove that S satisfies *SPEC*. We assume the verification system to be sound, i.e., $S \vdash_V SPEC$ implies $S \models SPEC$.

Any such sound verification system has to be incomplete as any non-trivial system property is undecidable due to Rice's Theorem. Instead, verification systems are supposed to be *relatively complete*, in the sense that they would be complete if there was available an oracle for validity of first-order formulas with arithmetic. In practice, this is rarely an issue: the amount of verification problems where there doesn't exist a proof is negligible compared to the far larger class of problems where the performance of the verification system is the reason a proof is not found in time. For the latter case, the user of a verification tool is prepared to give the prover further hints in form of auxiliary annotations in order to be able to verify a software system to satisfy its specification.

Moreover, 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 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. In our terminology, $SPEC$ covers both types of annotations, thus in the following $SPEC$ is a synonym for $REQ \cup AUX$.

2.5 The Verification Task

Given a software system S, consisting of the functions $f_1, \ldots, f_n$ and a requirement specification REQ_S, such that $S \models REQ_S$, the task of the user is to find a set of annotations AUX_S, s.t. $S \vdash REQ_S \cup AUX_S$.

Typically, these auxiliary annotations include loop and object invariants, lemmas, as well as program code that updates a separate specification memory (which is not visible from the C program during execution).

We tacitly assume that an already proved auxiliary annotation $a \in AUX_S$ can be used in subsequent proofs for other annotations in REQ_S and AUX_S. Thus, by using such auxiliary annotations as *lemmas*, proofs for elements of REQ_S may be greatly simplified or may even become possible at all in a given verification system.

Note that both REQ_S and AUX_S are composed of specifications for each of the functions f_i in S, i.e.,

$$REQ_S = REQ_1 \cup \ldots \cup REQ_n \text{ and } AUX_S = AUX_1 \cup \ldots \cup AUX_n \ .$$

Assuming soundness of the verification system,

$$S \vdash REQ_S \cup AUX_S \text{ implies } S \models REQ_S \cup AUX_S \ ,$$

and due to the requirement that $\models$ is monotonic w.r.t. adding annotations, a solved verification task (i.e. $S \vdash REQ_S \cup AUX_S$) implies $S \models REQ_S$.

If, on the other hand, S does not satisfy REQ_S, the verification task has no solution, i.e., no appropriate AUX_S exists. In that case, the verification system may still give the user feedback that helps to correct the requirement specification and/or the implementation.

2.6 The Modular Verification Process

The set of annotations of a function f consists of two parts: one part can be used in the correctness proof for calling functions of f (e.g., pre-/post-conditions of f), while the rest can only be used in the verification of the function f itself (e.g., loop invariants). We call the former set of annotations the *external* specification f, while the latter is named the *internal* specification of f. Which kind of annotation belongs to which of these categories is determined by the verification methodology built into the verification tool and the verification task at hand.

When verifying a function f using a modular verification approach, the external specifications of the children f' of f are used in the correctness proof of f instead of their implementation. Thus, the external parts of the auxiliary specification AUX' of f' are not only relevant for the verification of f' but can also

be used as lemmas in the proof of other functions, which in some sense breaks the modularity of the verification process. There exist dependencies between the auxiliary annotations for the different functions, which makes finding a complete set of auxiliary annotations to solve a verification task a difficult problem.

2.7 Top-down and Bottom-up Verification

The user of a deductive verification system may chose different orders in specifying and verifying the modules of a system. The extreme cases are:

Top-down verification The process starts with specifying and verifying the top-level functions with minimal depth in the dependency tree, before proceeding to verify functions with greater depth.

Bottom-up verification The process starts with specifying and verifying functions with maximal depth (leaves in the dependency tree) and proceeds to functions with smaller depth.

In an ideal world, given a prover that never fails due to time-outs, the modular software verification process would proceed top-down, starting with the requirement specification of a top-level function f. All children of f are then specified using the strongest possible contract (which by definition must be sufficient if any annotation is sufficient). Then, f can be proven to be correct with the help of auxiliary internal annotations given by the user. The process repeats with the children of f, until all functions of the system are verified to be correct w.r.t. their specifications. Similarly, this process could also be performed bottom-up.

Unfortunately, using strongest contracts is not a good idea in practice. They are (a) hard to find and (b) hard to prove. So, in practice, the solution to a verification task is a set of auxiliary annotations that are just (barely) strong enough. To support the user in the process of finding a solution and making the search less chaotic is the goal of the work presented.

3 Deductive Verification of Large Software Systems

In practice, a verification attempt may fail for a number of reasons. One significant problem is the performance of available verification tools, which may lead to time-outs. A verification attempt can have the following possible outcomes:

1. Verification of the program w.r.t. its specification succeeds.
2. Verification fails and a counterexample is returned by the prover. In this case, either the program does not satisfy the specification or the auxiliary annotations are not sufficient for the existence of a proof.
3. Verification fails because of a lack of resources (memory or time) and no indication is given whether the program is correct w.r.t. its specification.

Recall that in modular verification a function is verified using the external specifications of its children. If the verification of a function f succeeds (Case 1 above), then that does not imply that its children are correct w.r.t. their specifications. In case a child function f' does not satisfy its specification, there may

or may not be a different auxiliary specification for f' that is both satisfied by f' and sufficient to verify f.

In a similar manner, in Case 2 above, the external specifications of a child f' may be insufficient to verify f. Again, there may or may not be an alternative specification for f' that solves the problem.

When adhering strictly to a bottom-up verification process, one will never encounter the case that one of the children does not satisfy its specification, but it may very well happen that the specification of a child is insufficient to verify its parent. On the other hand, when verifying top-down, one will never end up with insufficient specifications of the children, but a specification of a child f' that is not satisfied by f' may very well occur. That is, independently of the order in which functions are specified and verified, one of the two problems remains.

Even always using the strongest possible contract for all functions is not an option here: while providing a stronger contract for a function f' may help in the verification of the parents of f', it also makes verifying f' more difficult. In practice, the user has to provide a specification for f' that is strong enough to verify all parent functions of f' and weak enough to verify f' itself.

Moreover, the logical strength of a contract is not its only relevant property but its syntactic form is just as important. As it is hard to foresee which specification of a function f is the most appropriate without paying attention to all call sites in the parents, in practice, neither a strict top-down nor a strict bottom-up approach is applied. Instead during the process a continuous adaptation of the specification takes place during which the specifications of calling and called functions are changed in alternation until verification of the software as a whole succeeds – this process is shown in Fig. 1a. A further possibility, which is not shown in the figure, is that the verification process fails because the implementation does not satisfy the requirement specification (in which case refining the annotations cannot help). Then, the implementation and/or the requirement specification need to be changed and the verification process restarted. The iterative process shown in Figure 1a is often applied locally, i.e., only one pair of caller and callee is considered at a time. As other functions may also use the callee and depend on its contract, changes in this contract may have to be propagated to various other parts of the system.

3.1 Object Orientation

Our proposed approach may also help with specifying and verifying object-oriented programs. In the following, support of our method for two particular features of OOP is examined, namely polymorphism (through class based inheritance with method overriding) complying with the Liskov substitution principle, as well as class invariants.

Polymorphism. For this, consider a class A implementing a method m and n subclasses of A called $\mathrm{B}_1, \ldots, \mathrm{B}_n$, each overriding A's implementation of method m.

Regardless of whether specifying bottom-up or top-down, when specifying the contract of the method m in A, two problems may occur. One is concerned with

the issue already described previously: to come up with the contract for A.m, the user has to consider all call sites of A.m. However, in our object-oriented setting, also all overridden implementations (or contracts) of A's subclasses $B_1, \ldots, B_n$ have to be taken into account. For each B_k, the precondition of A.m has to imply the precondition of B_k and the postcondition of B_k has to imply A.m's postcondition to satisfy the behavioral subtyping principle.

The last problem can be mitigated by using a technique called *lazy behavioral subtyping* described by Dovland et al. [7]: instead of using the contract of A.m as a constraint for the contracts of m in subclasses, only the properties of A.m that are actually used at the call sites in the program are required to be fulfilled by subclasses of A. To generate those properties, requirement specifications of the methods containing calls to A.m are required. In contrast, our proposed approach focuses on an earlier state in the specification process, where most requirement specifications may not yet be available.

Already while the user specifies A.m, our method would be able to give an indication whether the contract of A.m is sufficient for all call sites of A.m, as well as whether the contract is compatible to the implementations of all derived classes. The latter is implemented by inserting the appropriate method bodies of derived classes for calls to A.m in several runs of our tool. Likewise, in a bottom-up specification approach, after specifying m in any of A's derived classes B_k, this contract can be checked against the call sites and implementations of the superclass A, as well as the siblings B_j. A successful check against the siblings of B_k suggests that the contract of B_k is also a suitable contract for the implementation of the method in the superclass.

Class invariants. Another means to express functional properties of object-oriented programs are class invariants. Although the C programming language has no concept of objects, in VCC, structured data types are treated similarly and also can be annotated with invariants.

In the VCC methodology, for our purposes, such an object may have one of two states: it is either open or closed. Invariants of a closed object are known to hold throughout the execution of a (sequential) program, until it is opened, which is a prerequisite to being able to modify it. Establishing invariants in this methodology is only needed when closing an object.

Our proposed method could be extended to handle class invariants in the scope of the VCC methodology as follows: if the locations in the program are known where objects are being closed, checking invariants is reduced to checking an assertion (of the same property) at these locations. To identify these locations in the program is non-trivial in general, but for simple cases this could be approximated with the help of heuristics, e.g., always opening (closing) objects at method boundaries; or before (after) the first (last) modification to the object in a method. In cases where an invariant check fails due to wrongly placed open (close) annotations, the user may annotate the program with open (close) statements, overriding the automatically generated annotations.

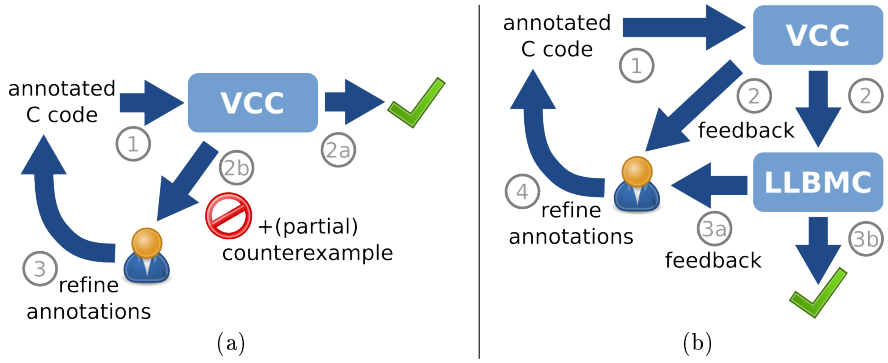

Fig. 1: (a) Normal VCC workflow and (b) counterexample guided *manual annotation* refinement (CEGMAR).

4 Software Bounded Model Checking

In contrast to deductive software verification techniques, software bounded model checking (SBMC) is based on an exhaustive search for a counterexample to the desired properties, rather than on constructing a deductive proof.

The SBMC implementation used in this work is LLBMC [15], the Low-Level Software Bounded Model Checker developed at the Karlsruhe Institute of Technology. It implements software bounded model checking of C and C++ programs and provides bit-precise reasoning, comprehensive support of the C language and a precise memory model supporting dynamic memory allocation. In the following, when we say SBMC we mean SBMC as implemented in LLBMC.

LLBMC takes an LLVM bitcode[1] file as input and first performs loop unrolling on all relevant loops, and function inlining on all called function calls. After this transformation, the resulting bitcode is translated into an intermediate logic representation (ILR). Function calls are inlined on the fly during conversion, and logic encodings of the properties to be proven are inserted into the formula. The resulting ILR formula is then simplified, translated to the logic of bitvectors and arrays and passed to an SMT solver. If the formula is satisfiable, the SMT solver's model is translated back into a readable counterexample on the LLVM level[2].

While unrolling and inlining are essential for the high performance of SBMC, they also make SBMC non-modular. Furthermore, because a fixed upper bound for loop iterations and call depths is required for the unrolling and inlining to terminate, SBMC is in general incomplete[3]. If SBMC is used for bug finding, it has been observed that low bounds are usually sufficient to detect bugs and

[1] LLBMC uses the LLVM (http://llvm.org) compiler's front-end to translate C code to LLVM-bitcode and starts bug finding on this level.

[2] See http://llbmc.org for examples and further information.

[3] SBMC is indeed complete, if the code is guaranteed not to exceed the given bounds.

incompleteness is not an issue. For the same reason, and because we use VCC for the full verification later on, we do not care for completeness yet.

LLBMC, like most SBMC tools, provides only limited support for expressing specifications. In order to still check VCC specification with LLBMC it was therefore necessary to translate VCC specification to LLBMC input.

5 Translation of Specification into LLBMC Input

LLBMC, like most other SBMC tools, has extensive support for build-in checks, e.g., division-by-zero checks or memory access checks, but only limited support for specifying user-provided properties. For this, LLBMC provides support for `assert` statements in the C code, which can be used just like C's `assert`, though LLBMC checks that the assertion is not violated for all possible executions.

In addition to these assertions – and in contrast to C – LLBMC provides so-called assumptions (`assume`). These can be used similarly to assertions, except that violations of assumptions are not considered bugs, but instead those execution paths are simply pruned and not analyzed any further.

Both of these in combination provide basic means for expressing specifications, where assumptions are used to express pre-conditions and assertions are used for post-conditions, though compared to VCC's specification language, this method is obviously restricted. It only allows what is expressible in C, which means, e.g., that quantifiers are not (directly) supported. For an existential quantifier it is often possible, however, to mimic its behavior by providing an explicit construction of the element satisfying the given property. Finite universal quantifiers might be replaced by a loop ranging over all elements. If such a translation is not possible, our proposed method can not be applied, and the user has to use VCC alone for verifying this property and all properties depending on it.

When translating specifications, usually, a *verification driver* function is written which first executes the `assume`s corresponding to the pre-condition, then the function itself, and finally the `assert`s corresponding to the post-condition. LLBMC is then applied to this verification driver function.

To determine feasibility of an automatic translation, we developed a prototype of tool for generating `assert`s, `assume`s, and verification drivers and used it in the evaluation described in Section 8. The tool also supports loop invariants, inserting appropriate `assert` statements before the loop, and at the end of each copy of the loop body. Invariants of data structures are currently not handled, though one possible option to implement this feature is discussed in Sec. 3.1. Currently work is being done on LLBMC to allow expressing additional properties supported by VCC, e.g. by VCC's `mutable` or `writes`.

Figure 2 shows an exemplary excerpt from the VCC specification of the function `copyNoDuplicates` (from the example in Section 7) and the result of a manual translation of that specification into LLBMC input. We expect to be able to estimate better how well translation of specifications is possible in general, and to what degree it can be automated, once our prototype has matured and once LLBMC comes closer to VCC expressiveness in specification.

```
//no 'new' items in result
_(ensures \result != NULL ==>
  ∀ uint i; i < \result->count ==>
    (∃ uint j; j < source->count ∧
      \result->items[i] ==
        source->items[j]))
```

```
cnt = result->count;
//no 'new' items in result
if (result != NULL)
  for (i = 0; i < cnt; ++i) {
    int found = 0;
    for (j = 0; j < cnt; ++j)
      if (result->items[i] ==
          source->items[j])
        result = 1;
    assert(found == 1);
  }
```

Fig. 2: Exemplary excerpt from the requirement specification of the function `copyNoDuplicates` (left) and its translation into LLBMC input (right).

6 The Integrated Verification Process

In the following, we describe how to integrate software bounded model checking into the annotation-based deductive verification process, thereby taking advantage of the strengths of both methods. As said above, we use the tools VCC and LLBMC to illustrate our approach.

The central idea of our method is to use SBMC to support the process of finding a set of auxiliary annotations, AUX, for a given system S that allows the deductive verification tool to prove that S satisfies its requirement specification REQ. The resulting integrated process is illustrated in Fig. 1b. The name CEGMAR is inspired by the counterexample-guided abstraction refinement (CEGAR) technique [3] used in model checking – though in CEGMAR annotations are refined instead of abstractions.

Note, that we do not use the term refinement in its strict mathematical meaning here. Instead we have a more colloquial interpretation in mind, where refinement simply means iterations towards a specification which is fit for its purpose. Also note that this kind of refinement contains a manual component, which makes CEGMAR a machine-supported verification process, not a fully automatic algorithm.

CEGMAR aims at finding suitable auxiliary specifications for the full system S, but at any given point in time, some function f is in the focus of the process. The process starts from the given requirement specification REQ_f for a function f and a (possibly empty) set AUX_f of auxiliary annotations.

In Step 1 (Fig. 1b), the annotated C code relevant for proving f correct w.r.t. its (requirement and auxiliary) specification is passed to VCC for verification. The result of VCC's verification attempt for f is given to LLBMC (Step 2 in Fig. 1b). Then, in case both VCC and LLBMC agree that f satisfies its specification, the refinement-loop for f ends successfully (Step 3b). After this, some other function moves into focus or, if all functions have been verified, the verification task has been successfully completed.

Otherwise, if one of the tools (or both) fails to verify f, the user has to refine some of the auxiliary annotations (Step 4), using the feedback of VCC and LLBMC (from Step 2 resp. 3a). After refining the auxiliary annotations in Step 4, the next iteration starts with Step 1. If changing the auxiliary specifications is not sufficient according to the feedback from the tools, i.e., there is a problem in the implementation or the requirement specification, then the refinement loop for f terminates and can only be restarted after the implementation and/or the requirements have been fixed.

In Step 2, using VCC, the correctness of f is only proven *locally*, i.e., the external specifications for $children(f)$ are used without checking that they are satisfied. Feedback from VCC is either (a) the statement that f is correct w.r.t. its specification or (b) a list of annotations that cannot be proven (possibly together with counterexamples). In contrast, LLBMC is used in our integrated approach to check correctness of a function f *globally*, i.e., the implementation of all functions called by f (directly or indirectly) is taken into account. We identified three different properties to be checked with LLBMC:

A. f satisfies its specification;
B. all functions in $children(f)$ satisfy their external specifications;
C. the pre-condition of f holds at all points where f is called in $parents(f)$ (invocation contexts).

Each of the three checks A–C has three possible outcomes: either (a) the property in question holds up to a certain bound on the length of traces, or (b) LLBMC provides an error trace falsifying the property, or (c) there is a time out.

The three checks differ in which part of the implementation and annotations are given to LLBMC, as well as the consequences of the check for the verification process, as described in the following.

A: Checking that f satisfies its specification. For this, the implementations of f and all descendants of f are passed to LLBMC for model checking. The implementation of f is checked w.r.t. all annotations of f that VCC reports to be violated. LLBMC can provide the user with feedback on which unproven specifications are indeed violated by the implementation and which are likely satisfied (because no counterexample was found within the given bound), but just not provable by VCC without refining the annotations.

Even if VCC could verify that f is correct (based on the external specification of functions called by f), LLBMC is still applied. This allows to discover cases where VCC's correctness proof for f only succeeded due to an erroneous external specification of a child of f.

B: Checking that child functions satisfy their external specification. The implementations of all descendants of f are passed to LLBMC for model checking. The functions in $children(f)$ are checked to satisfy their external specifications. This check helps to rule out correctness proofs for f that are erroneous because they rely on faulty specifications of f's children.

C: Checking Invocation Contexts. For each function $g \in parents(f)$, the implementations of g and all descendants of g are passed to LLBMC. Here, the property to check is the pre-condition of f. Checking the invocation contexts

helps to avoid writing specification for f that cannot be used in the proofs for other functions in the system.

Note that in all these cases, LLBMC is not used to check whether a function satisfies an annotation in general, but to check that the function satisfies the annotation in the context in which it is called. The context may be defined by the function's pre-condition or by the context in one of its parents.

The benefit of the proposed integration of SBMC into deductive verification results mainly from the fact that SBMC is not modular. During the verification of a function f, LLBMC uses the implementation of the children and parents of f instead of (only) using the external specification of the children as VCC does. Because of this difference, LLBMC and VCC can provide the user with different information about the functions and their annotations (e.g. counterexamples). For the verification process, the information provided by LLBMC is a valuable addition to the information provided by VCC.

7 A Typical Specification Scenario

7.1 The System to be Verified

In the following we present an example that demonstrates the issues of modular verification mentioned before and how integration of software bounded model checking into the verification process can help attenuate these.

Consider the following C data structure implementing a sequence data type:

```
1 typedef struct queue_t {
2     int *items;
3     int count, capacity;
4 } queue, *pQueue;
```

Here, `count` denotes the length of the sequence and `capacity` the fixed size of memory that has been allocated to store the items of the sequence. In our case, the items of the sequence are integers and are stored in the array that starts at the memory address `items`.

The top-level function we want to verify is `copyNoDuplicates` (see Fig. 3), but in total there are three functions involved in the verification process:

- `pQueue copyNoDuplicates(pQueue dst, pQueue src)`
  Given a queue *dst* and a queue *src*, this function modifies *dst* so that it is a copy of *src*, except that duplicate elements of *src* occur only once in *dst*.
- `pQueue initQueue(int capacity)` (not shown)
  Allocates memory for a new queue structure as well as the appropriate amount of memory for storing `capacity` number of items. It also initializes the queue data structure to correspond to the empty sequence. This function is called by `copyNoDuplicates`.
- `void insert(pQueue q, int val)` (shown in Fig. 3)
  Inserts an item *val* into a queue q in such a way that if the queue is in ascending order, it remains ordered. This function is called by `copyNoDuplicates`.

```
1  pQueue copyNoDuplicates(pQueue dst, pQueue src) {
2    dst->count = 0;
3    for (int i = 0; i < src->count; i++) {
4      int sVal = src->items[i];
5      int j = 0, contained = 0;
6      while(j < dst->count && dst->items[j] <= sVal) {
7        if (dst->items[j] == sVal) {
8          contained = 1;
9          break;
10       }
11       j++;
12     }
13     if (!contained) insert(dst, sVal);
14   }
15   return dst;
16 }
17
18 void insert(pQueue q, int val) {
19   if (q->count == q->capacity) return;
20   int i,j;
21   for (i = 0; i < q->count && val > q->items[i]; i++) {}
22   for (j = q->capacity-1; j>i; j--)
23     { q->items[j] = q->items[j-1]; }
24   q->items[i] = val; q->count++;
25 }
```

Fig. 3: Implementation of `copyNoDuplicates` and `insert`.

There are two peculiarities about this implementation of `copyNoDuplicates` that the verification engineer might not be aware of:

1. The implementation relies on `insert` retaining sortedness of the queue `dst`. This is because the algorithm stops searching for a matching element as soon as a greater element is encountered. Note that sortedness is not an invariant of the queue data structure, so queue `src` may be unsorted.
2. Inserting into a queue fails silently if the capacity of the queue is reached.

In the following, we will use the example to show why a user who is not supported by software bounded model checking will have trouble identifying these problems during verification, independently of whether a top-down or a bottom-up approach is chosen.

7.2 Bottom-up Verification of `copyNoDuplicates`

When verifying bottom-up, the first two functions that are to be verified correct in our case are `initQueue` and `insert`. Because we are mainly interested in

interaction between `copyNoDuplicates` and `insert`, from now on we assume that `initQueue` has been verified and does not need to be considered further.

The second issue mentioned in the previous subsection (finite capacity of the queue) is identified quickly with a bottom-up approach and therefore not further discussed. The first issue (sortedness) is considerably harder to identify, though.

Suppose that no requirement specification is given for `insert`, so any specification of `insert` is auxiliary. It is likely that the user correctly specifies that the item passed to `insert` is indeed inserted in the given queue. However, it is also likely that the verification engineer on the first try is not aware of the importance of sortedness and, consequently, the specification does not mention that insertion of elements retains sortedness of the queue. In that case, the specification of `insert` is not strong enough and verification of `copyNoDuplicates` will fail. But that is only noticed after `insert` has been successfully verified and the verification process has moved on to `copyNoDuplicates`.

Once `copyNoDuplicates` could not be proven correct, the verification engineer has to identify the cause for this. The too weak specification of `insert` is hard to spot, and the user may be tempted to believe `copyNoDuplicates` is not correctly implemented. The counterexample provided by VCC usually does not contain all necessary information to understand the issue.

LLBMC on the other hand states, using Check A from Section 6, that `copyNoDuplicates` does indeed satisfy its requirement specification and the relevant loop invariants – at least up to a certain size of the queue[4]. This indicates to the user that `copyNoDuplicates` is likely correct and the problem is either that the specification of `insert` is too weak or the auxiliary annotations are not enough to allow VCC to verify the property. This is an important cue towards the right direction and can therefore speed up the verification process.

7.3 Top-down Verification of `copyNoDuplicates`

In a top-down verification approach, the top-level function `copyNoDuplicates` is the first to be verified. The first issue (sortedness of the queue) is found early in the process, as the verification of `copyNoDuplicates` will already uncover it. Instead, the second issue (finite capacity of the queue) is now causing problems.

Consider a requirement specification of `copyNoDuplicates` consisting of an empty pre-condition and the following post-condition stating that all elements of the source queue are also contained in the resulting queue (in fact, this is only part of the actual requirement specification because it does not state that the result should not contain duplicates):

```
1 ∀i; i ≥ 0 ∧ i < source->count ⟹
2     ∃j; j ≥ 0 ∧ j < \result->count ∧
3         source->items[i] == \result->items[j]
```

In order to verify the implementation of `copyNoDuplicates` to satisfy this requirement, the user has to provide auxiliary specifications for the helper functions

[4] This size is determined by the bound applied during model checking.

`initQueue` and `insert`. The verification of these auxiliary specifications is postponed in the top-down approach until after the contract of `copyNoDuplicates` is proven. Nevertheless they are already used in the proof for `copyNoDuplicates`.

If the user annotates `insert` he/she may easily overlook the case where the queue has reached its capacity and insertion of yet another element fails (signaled by `insert` by returning an error code). Now, because of this omission, the specification of `insert` is too strong, which allows VCC to prove the contract of `copyNoDuplicates` – even though it is in fact not satisfied. Only when the verification of the contract of `insert` fails, this error is detected.

Then, after fixing the specification of `insert`, the verification engineer has to go back to `copyNoDuplicates` and re-verify that function, taking the modified specification of `insert` into account. In practice, the top-down approach results in numerous iterations until a function and all of its children are verified.

Using LLBMC can help resolve this issue early on, as LLBMC directly takes the implementation of `insert` into account, and not just its (too strong) specification. LLBMC uncovers the problem as soon as `copyNoDuplicates` is checked – even though VCC cannot detect any problem at this point.

8 Evaluation

In order to evaluate the proposed methodology for non-trivial programs, and because manual translation of specifications is too labor-intensive, a tool for automatically translating VCC specifications to LLBMC was needed. Therefore, we implemented a prototype for such a tool, based on a modified version of the clang compiler front-end[5], which currently supports a very small, though already useful, subset of the VCC specification language.

Given the tool's translation of the specification in the above example, it was sufficient to supply a manually written verification driver to LLBMC, coming in at 19 lines of C code and consisting mostly of code initializing a valid, though indefinite instance of a queue. Further automation is easily possible and should eliminate manual translations completely.

LLBMC was applied on this example and the tool found, as expected, a discrepancy between code and specification, which comes into effect when the capacity of the destination queue is smaller than the capacity of the source queue. Because of the way SBMC works, this was possible even though no invariants for the involved loops were specified yet. VCC, lacking these invariants, could not yet be used to check the properties.

While this proves the concept, the time LLBMC took for finding the counterexample in this example is somewhat long (12 minutes).[6] Times for other examples are similar. The use case for our tool is to run it in the background while the user develops a specification. This requires feedback within seconds. The next step in our research is to optimize LLBMC for this kind of application, so that larger and more complex examples can be handled.

[5] The clang compiler front-end is available at http://clang.llvm.org

[6] We also tried CBMC-3.8 on this example, but it choked on the input file.

9 Related Work

Our work is related to previous work about combinations of model checking and deductive verification, improvements to the software verification process, as well as to tools and techniques that help understand failed verification attempts.

Various combinations of model checking and deductive verification have been proposed and studied in the past, e.g., [2, 9]. An extensive overview of the work published until 2000, with a focus on the verification of reactive systems,[7] is given in [17]. Since then, research in this area seems to have slowed down.

Some papers use a combination of model checking and deductive verification techniques to improve performance and generality of existing verification tools, such as [14]. Others use deductive methods specifically to extend model checking approaches to infinite-state systems, e.g. [11, 16, 6].

Most of these papers focus on improving performance of the tools or creating more powerful verification tools. In contrast to this we focus entirely on improving the process of annotation-based deductive software verification.

On a different note, Müller and Ruskiewicz use debuggers to address the problem of understanding failed verification attempts [13]. They, too, provide concrete counterexamples that illustrate why verification did not succeed. While error traces are an important part of our contribution to the deductive verification process, the proposed process is not restricted to counterexamples.

Similarly, Vanoverberghe et al. use symbolic execution techniques to generate test cases when the prover fails [18]. They also generate test cases that can be executed in a debugger to analyze the failed verification for a concrete example.

Alex Groce et al.'s tool `explain` [8] provides additional information about counterexamples generated by the bounded model checker CBMC. The tool itself uses CBMC's software bounded model checking engine to generate executions similar to an existing counterexample that do not fail, and additional counterexamples that are as different as possible but do still fail. The approach does not seem to be directly applicable to deductive verification techniques due to the lack of a concrete counterexample to start with.

10 Conclusion and Future Work

Integrating software bounded model checking into the deductive verification process can give the user of deductive verification tools early feedback, thereby decreasing the verification effort and improving the overall verification process. We showed that SBMC can help in finding insufficient or wrong annotations. An example was provided showing that SBMC can help both in top-down and bottom-up verification. It can also help in identifying specifications that are either too weak or too strong. In the future we plan to further integrate VCC and LLBMC, so that specifications in VCC can be fully automatically translated

[7] The combination of temporal logic properties and an infinite-state system makes reactive systems a fitting application for combinations of model checking and deductive verification.

into to LLBMC input, so that larger case studies can be carried out. Ideas from the area of instrumenting code for run-time checking specifications will be useful here. Furthermore, we're already working on extend LLBMC's specification language in order to allow for a more complete translation of VCC specifications. In the long term, LLBMC could also be used to automatically derive simple annotations – such as non-nullness of pointers and simple ownership relations – thereby speeding up the specification process.

References

1. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO 2005*, pages 364–387, 2005.
2. S. Berezin, K. McMillan, and C. B. Labs. Model checking and theorem proving: a unified framework. Technical report, Carnegie Mellon Univ., 2002.
3. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV'00*, pages 154–169, 2000.
4. E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, T. Santen, M. Moskal, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs'09*, pages 23–42, 2009.
5. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS'08*, pages 337–340, 2008.
6. G. Delzanno and A. Podelski. Constraint-based deductive model checking. *Software Tools for Technology Transfer*, 3(3):250–270, 2001.
7. J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Lazy behavioral subtyping. *Journal of Logic and Algebraic Programming*, 79(7):578 – 607, 2010.
8. A. Groce, D. Kroening, and F. Lerda. Understanding counterexamples with explain. In *CAV'04*, pages 453–456, 2004.
9. Y. Kesten, A. Klein, A. Pnueli, and G. Raanan. A perfect verification: Combining model checking with deductive analysis to verify real-life software. In *FM'99*, pages 173–194, 1999.
10. K. R. M. Leino and M. Moskal. Usable auto-active verification. Technical Report Manuscript KRML 212, Microsoft Research, 2010.
11. Z. Mann, A. Anuchitanukul, N. Bjorner, A. Browne, E. Chang, M. Colon, L. de Alfaro, H. Devarajan, H. Sipma, and T. E. Uribe. STeP: The Stanford Temporal Prover. Technical report, Stanford Univ., 1994.
12. M. Moskal. *Satisfiability Modulo Software*. PhD thesis, Univ. of Wrocław, 2009.
13. P. Müller and J. N. Ruskiewicz. Using debuggers to understand failed verification attempts. In *FM'11*, pages 73–87, 2011.
14. A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In *CAV'96*, pages 184–195, 1996.
15. C. Sinz, S. Falke, and F. Merz. A precise memory model for low-level bounded model checking. In *SSV'10*, 2010.
16. H. Sipma, T. E. Uribe, and Z. Manna. Deductive model checking. *Formal Methods in System Design*, 15(1):49–74, 1999.
17. T. E. Uribe. Combinations of model checking and theorem proving. In *FroCoS'00*, pages 151–170, 2000.
18. D. Vanoverberghe, N. Bjørner, J. D. Halleux, W. Schulte, and N. Tillmann. Using dynamic symbolic execution to improve deductive verification. In *SPIN'08*, pages 9–25, 2008.