

HOL-Boogie — An Interactive Prover-Backend for the Verifying C Compiler

Sascha Böhme · Michał Moskal · Wolfram Schulte · Burkhart Wolff

Abstract *Boogie* is a verification condition generator for an imperative core language. It has front-ends for the programming languages *C#* and *C* enriched by annotations in first-order logic, i. e. pre- and postconditions, assertions, and loop invariants. Moreover, concepts like ghost fields, ghost variables, ghost code and specification functions have been introduced to support a specific modeling methodology.

Boogie's verification conditions — constructed via a *wp* calculus from annotated programs — are usually transferred to automated theorem provers such as *Simplify* or *Z3*. This also comprises the expansion of language-specific modeling constructs in terms of a theory describing memory and elementary operations on it; this theory is called a *machine/memory model*.

In this paper, we present a proof environment, HOL-Boogie, that combines *Boogie* with the interactive theorem prover Isabelle/HOL, for a specific C front-end and a machine/memory model. In particular, we present specific techniques combining automated and interactive proof methods for code verification. The main goal of our environment is to help program verification engineers in their task to “debug” annotations and to find combined proofs where purely automatic proof attempts fail.

Keywords Isabelle/HOL, Theorem Proving, Program Verification, Memory Models, Annotation Languages

Sascha Böhme
Technische Universität München, E-mail: boehmes@in.tum.de

Michał Moskal
European Microsoft Innovation Center, Aachen, E-mail: michal.moskal@microsoft.com

Wolfram Schulte
Microsoft Research, Redmond, E-mail: schulte@microsoft.com

Burkhart Wolff
Université Paris-Sud, LRI, CNRS, Orsay, E-mail: wolff@lri.fr

1 Introduction

Verifying properties of programs at their source code level has attracted substantial interest recently. While not too long ago, “real programming languages” like Java or C have been considered as too complex for formal analysis, there are meanwhile verification systems like ESC/Java [30], Why/Krakatoa/Caduceus [29], and Boogie [6] used both for Spec# [3,4,7] and C [15,16,50]. The latter system, called “Verifying C Compiler” or VCC for short, is currently used in a substantial verification effort for the Microsoft Hypervisor as part of the Verisoft XT project [46,47].

VCC/Boogie not only supports nearly the complete ANSI-C language — a completeness that was hitherto unseen in a proof environment —, but comes with a rich annotation language including means to define auxiliary predicates, framing conditions and syntactic support for them, and ways to add types, fields in structures, function parameters and code just for the purpose of specification. These so-called *ghost types*, *ghost fields*, *ghost parameters* and *ghost code* increase the expressiveness of the annotation language in various ways while maintaining a constructive, proof-oriented style; for example, it is common practice to provide ghost code that actually constructs a witness for existential quantifiers.

Combining VCC/Boogie with an interactive prover has a number of incentives:

- existing front-end compilers for Spec# and C to the Boogie core language represent an alternative to a logical embedding of these languages,
- verification attempts can be debugged by interactive proofs, so interactive techniques may increase the verification productivity,
- combinations of structured interactive and automated techniques can make infeasible verifications feasible, especially if algorithmic problems are involved, and
- the logical foundation of the VCC/Boogie-approach can be improved.

In this paper, we present a new verification environment, called HOL-Boogie [12,13], based on VCC/Boogie and Isabelle/HOL, and discuss challenges and achievements.

1.1 VCC/HOL-Boogie as an Alternative to Embeddings

Compiling ANSI-C into a transition system described in the fairly small and logically clean Boogie core language (called *BoogiePL*) represents an alternative to a *logical embedding* into HOL. Such an embedding approach, for example, has been taken by [36,49] describing a big-step transition semantics for a C fragment comprising only side-effect free expressions. While this approach — linking a C Hoare-calculus via a proven correct, but simplistic compiler to a small-step semantics of an academic assembler — has a perfect logical foundation, it is obvious that an embedding of a more substantial C fragment is an enormous effort with questionable value. The reasons for this, reflecting C’s historical roots as “portable assembler”, are two-fold:

1. The ANSI-C language semantics is heavily under-specified; the standard is essentially a collection of requirements on a not further defined “underlying execution model”. In contrast, “real C code” as occurring in operating system level programs (device drivers, OS kernels, etc.) characteristically relies on an architecture-dependent memory/machine model, be it x86, PowerPC, VAMP [9], etc.¹ To be useful, the embedding approach has to take them into account.

¹ ... consequently it has long been considered to be “dirty” and out of the reach of formal verification in academia. . .

-
2. “Real C code” is usually compiled by aggressively optimizing compilers, which are not necessarily correct wrt. to a given memory/machine model in practice (which does not imply that concrete resulting code is incorrect). A logical embedding following the embedding approach would need to verify all these compiler-specific semantic decisions and optimizations.

These problems are solved if one uses the resulting machine-specific, optimized code of a given compiler; if verification is performed on this level, all architecture and compiler features can be taken into account. In our case, we use a slight abstraction of the resulting x86 assembler code into a machine/memory model assuming linear memory and arithmetically computed pointers into it; the distinctive feature of this model called *C Virtual Machine* (CVM) is that it supports type unions and bit fields.

1.2 HOL-Boogie as a Technology for Debugging Verification Attempts

Starting to annotate a given program will sooner or later lead to situations where the automated prover fails and can neither find a proof nor a counterexample. All existing systems report a degree of automation approaching 100%, causing widespread and understandable enthusiasm. However, there is also a slight tendency to overlook that the remaining few percent are usually the critical ones, related to the underlying theory of the algorithm rather than implementation issues like memory and sharing. Moreover, these figures tend to hide the substantial effort that may have been spent to end up with a formalization that can be finally proven automatically; there is even some empirical evidence that in the difficult cases, the labor to massage the specification can be comparable to the effort of an interactive proof [8].

The reason for a prover failure might be:

- specification-related, i. e. annotations and “background theories” (see below) are inconsistent, incomplete, or specify unintended behavior,
- program-related, e. g. a program simply does not behave as intended, or
- it can be a problem of the prover, by just using a wrong heuristics for the concrete task, or even by bad luck (e. g. Z3 [23] uses random-based heuristics).

The key advantage of HOL-Boogie is that assertions at specific program points can be tracked back to the precise set of facts valid at it; in combination with other automated techniques, these facts can be simplified and filtered to the relevant ones. Thus, insufficient preconditions or invariants can be inferred fairly efficiently.

1.3 HOL-Boogie can Increase the Overall Proof Power

Generated verification conditions can be very large — we have seen examples which were several Megabytes large (measures as size of the SMT exchange format). While there are powerful techniques to slice them into smaller pieces, it can be safely predicted that any automated procedure will reach at some point its limits, being simply flooded by the large number of facts to be taken into account. The situation is worse if there is a substantial number of auxillary definitions, representing “the theory” of an algorithm including invariants and formalizations of key concepts; in practice, these definitions are simply unfolded, and usually not in an adequate manner.

An interactive proof, suitably adapted to the problems arising from automated formula generation, can decompose the verification conditions along the program structure and finally the logical structure of the annotations as needed. Moreover, interactive theorem proving technology has developed a body of techniques to abstract and structure proof tasks, profiting from techniques developed for proofs of mathematical problems of considerable size. In particular, relations between auxiliary predicates can be exploited abstractly without actually unfolding them.

1.4 Strengthening Logical Foundations

Conceptually, BoogiePL allows for specifying programs running over an axiomatized machine/memory model, in our case the CVM. This machine model presents a (slight) abstraction over an x86 processor architecture, taking into account the processor intricacies of little-endianness, bit-padding, etc., but assuming linear memory and abstracting from registers and jumps (which are represented by `goto`'s in BoogiePL). The CVM provides also complex operations for allocation, release, move and copy memory. Furthermore, the CVM provides the critical infrastructure to state framing conditions.

Getting an axiomatization of this size (ca. 900 axioms) consistent is a non-trivial task, and for several automated and interactive provers to work together, one has to make sure that all provers agree on this axiomatization. Besides formal proofs of consistency, it can be proven formally within HOL-Boogie that a given CVM model indeed represents an abstraction of a concrete machine model, for example a precise formal model of the x86 architecture.

1.5 Contribution

We built HOL-Boogie to exploit these aforementioned incentives and to support the rich annotation language. We provided techniques for assertion tracking in an interactive setting, and an integration of the target SMT solver Z3, including techniques to maintain Z3's prover instrumentation in order to leverage a real Z3 integration into Isabelle. HOL-Boogie also provides a generic framework to support memory models; in this paper, we will discuss two particular versions of the CVM called VCC1 and VCC2 including their specific tactic support, enabling also native Isabelle proofs on the model. By a collection of smaller and larger examples, we provide evidence for the feasibility of the approach, and show examples in the data type domain where purely automated proof attempts even fail at the end of the development.

1.5.1 Outline of the Paper

After presenting the background of this work, namely Isabelle/HOL, Boogie, and the general HOL-Boogie architecture, we describe BoogiePL and its labeling techniques. We discuss our SMT solver interface in HOL-Boogie and demonstrate the “debugging approach” at a non-trivial imperative algorithm. In the next stage, we describe the CVM annotation language and the underlying machine/memory model VCC1 as well as its specific proof-support and show its use in a non-trivial C example. A further machine/memory model called VCC2 is introduced and discussed. Finally we demonstrate how machine/memory models can also be verified inside HOL-Boogie.

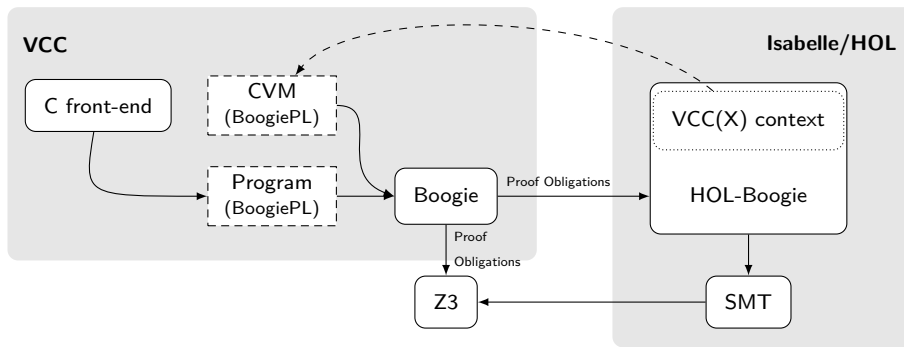


Figure 1 VCC and the HOL-Boogie Back-end

2 Background

2.1 Isabelle/HOL and the Isar Framework

Isabelle is a generic theorem prover [44], i.e. new object logics can be introduced by specifying their syntax and inference rules. Isabelle/HOL is an instance of Isabelle with Church’s higher-order logic (HOL), a classical logic with equality. Substantial libraries for sets, lists, maps, etc. have been developed for Isabelle/HOL, based on definitional techniques, allowing the use of Isabelle/HOL as a “functional language with quantifiers”.

Isabelle is based on the so-called “LCF-style architecture” which allows one to extend a small trusted logical kernel by user-programmed procedures in a logically safe way. Moreover, on top of the kernel, there is a generic system framework Isabelle/Isar [52] that can be compared in a rough analogy to the Eclipse programming system framework. It provides (1) a hierarchical organization of theory documents, (2) incremental document processing for interactive theory and proof development (with unlimited undo) and an Emacs-based GUI, and (3) extensible syntax for top-level commands, embedded methods and attributes, and the inner term language. HOL-Boogie is yet another instance of the Isabelle/Isar framework. It comes with a loader of the verification conditions generated by Boogie, a proof-obligation management and specific tactic support for the formulas arising in this scenario as well as interactions with external provers such as Z3 which have been integrated via the Isabelle oracle mechanism.

2.2 The VCC System Architecture

The *Verifying C Compiler* (VCC) evolved from the Spec# project [4, 7]. It comprises a C front-end supporting ANSI-C and — geared towards verification of programs close to the hardware-level — bitwise representation of e.g. integers, structs, and unions in memory. The core component of VCC is Boogie, a verification condition generator. Its input language BoogiePL provides constants, functions and first-order axioms, as well as a small imperative language with assignments, first-order assertions, unstructured **goto** and structured control constructs (**if**, **while**, **break**). From

these annotated imperative programs, Boogie computes (optimized) verification conditions over the program and the axiomatization of a *background theory*, i.e. a machine/memory model. We use Isabelle’s theory mechanism to include several such machine/memory models $VCC(X)$ alternatively, which describe in our cases linear memory (a map from references to bitvectors), allocation operations, little-endian word-wise load and store operations, and a family of word-wise operations abstracting the x86-64 processor architecture.

Boogie also provides a framework into which converters to external prover formats may be “plugged in”. Our HOL-Boogie integration is based on such a plug-in that outputs the verification conditions in a typed, special-purpose format we developed for interactive prover back-ends.

3 Foundations of Boogie and HOL-Boogie

3.1 Introduction to BoogiePL

BoogiePL is a many-sorted logical specification language extended by an imperative language with variables, contracts, and procedures.

The type system of BoogiePL has several built-in as well as user-defined types. The former cover basic types like **bool** and **int**, as well as one- and two-dimensional arrays which can be indexed by any valid type.

BoogiePL includes the following kinds of top-level declarations:

- user-defined types:

```
type Vertex;
```

- symbolic *constants* having a fixed but possibly unknown value:

```
const Infinity : int;
```

- *uninterpreted functions*:

```
function Distance(from : Vertex , to : Vertex) returns (result : int);
```

- *axioms* constraining symbolic constants and functions:

```
axiom 0 < Infinity;
```

- global variables:

```
var ShortestPath : [Vertex] int;
```

- *procedure contracts*, i.e. signatures with pre- and postconditions, and
- *implementations* of procedures.

An implementation begins with local-variable declarations which are followed by a sequence of basic blocks. We will only consider the latter in more detail here, and we omit the structured control structures, which can be desugared into the statements and **goto**’s shown here. Each basic block has a name, a body, and a possibly empty set of successors (Figure 2). Expressions are first-order logic formulas with equality and integer operations.

Semantically, each block corresponds to a transition relation over the variables of a program; **goto** statements correspond to a composition with the intersection of

```

BlockSeq ::= Block+
Block ::= BlockId : [ Statement ; ] Goto ;
Statement ::= Var := Expression
            | havoc VarId
            | assert Expression
            | assume Expression
            | call [ Var+ := ] ProcId ( Expression* )
            | Statement ; Statement
Goto ::= goto BlockId* | return

```

Figure 2 Schematic syntax of blocks in BoogiePL

the successor transition relations, loops to fixpoints: Boogie represents a partial correctness framework. The basic assertion **assert** constrains the subsequent transition, while **assume** weakens it. Pragmatically, **assert** produces obligations for the programmer, while **assume** leaves him “off-the-hook”, see, e. g., [41,43].

An assignment statement $x := E$ updates the program state by setting the variable x to the value of the expression E . The statement **havoc** x sets the variable x to an arbitrary value. The statement $S ; T$ corresponds to the relation composition. The procedure call statement, i.e. **call**, is just a short-hand for suitable **assert**, **havoc** and **assume** statements, encoding the callee’s pre- and postconditions [38]. The **return** command is a short-hand for the procedure’s postconditions and a **goto** with no successors.

BoogiePL also comes with a structured syntax with which one can express loops (**while**) and branches (**if**) directly. These can be defined as a notation for certain basic blocks; for example, the following schematic **while** loop:

```

while (G) invariant P; { B }

```

is encoded by the following basic blocks [3]:

```

LoopHead : assert P ; goto LoopBody, LoopDone;
LoopBody : assume G ; B ; goto LoopHead;
LoopDone : assume ¬G ; ...

```

More details of BoogiePL can be found in [3,24].

3.2 Generating Verification Conditions

Verification condition generation proceeds in the following steps: First, the expansion of syntactic sugar and (safe) cutting of loops result in an acyclic control-flow graph. Second, a single-assignment transformation is applied. Third, the result is turned into a passive program by changing assignment statements into **assume** statements. Finally, a verification condition of the unstructured, acyclic, passive procedure is generated by means of weakest preconditions.

We will present only the final step here, the reader interested in the first three is referred to [5]. Each basic block in a preprocessed program consists only of a sequence of **assert** and **assume** statements, followed by a final **goto** command.

For any statement S and predicate Q on the post-state of S , the weakest precondition of S with respect to Q , written $wp(S, Q)$, is a predicate that characterizes

all pre-states of S whose reachable successor states satisfy Q . The computation of weakest preconditions follows the well-known rules:

$$\begin{aligned} wp(\mathbf{assert} P, Q) &= P \wedge Q \\ wp(\mathbf{assume} P, Q) &= P \implies Q \\ wp(S ; T, Q) &= wp(S, wp(T, Q)) \end{aligned}$$

For every block

$$A : S ; \mathbf{goto} B_1, \dots, B_n;$$

an auxiliary variable $A_{correct}$ is introduced, the intuition being that $A_{correct}$ is true if the program is in a state from which all executions beginning from block A are correct. Formally, there is the following block equation:

$$A_{correct} \equiv wp(S, \bigwedge_{B \in \{B_1, \dots, B_n\}} B_{correct})$$

Each block contributes one block equation, and from their conjunction, call it R , the procedure's verification condition is

$$R \implies Start_{correct}$$

where $Start$ is the name of the first block of the procedure. The verification condition generated this way is linear in the size of the procedure [5].

3.3 Labeling in Boogie

Boogie is able to output source code locations of errors and also execution traces leading to these errors. The underlying basic idea is to enrich formulas by labels, i.e. uninterpreted predicate symbols intended to occur in counterexamples of verification conditions. In verification conditions generated by Boogie, labels are either positive (**lblpos** $L: P$) or negative (**lblneg** $L: P$). Logically, these formulas are equivalent to P ; the labels occur in counterexamples if P has the indicated sense (i.e. P or $\neg P$). Their formal definition is as follows:

$$\begin{aligned} (\mathbf{lblneg} L: P) &= P \vee L \\ (\mathbf{lblpos} L: P) &= P \wedge \neg L \end{aligned}$$

Negative labels tag formulas of assertions (including invariants and postconditions) with an abstract identifier referring to a location in the source program. If an assertion cannot be proven, the accompanying label allows Boogie to emit an error location identifying which program check failed. Positive labels tag the beginning of a block by an additional assertion which is always true. This way, execution traces contain information reflecting the order in which basic blocks were processed. If execution terminates in an error, the positive labels represent an error trace.

A more detailed description of this use of labels is found in [37].

We extended Boogie to emit real program locations instead of abstract labels. After producing a verification condition and loading it in HOL-Boogie, the program locations then occur at the formulas to be proven.

3.4 Attribution in BoogiePL

We implemented a new feature in Boogie: The top-level declarations for types, constants, functions, axioms, and global variables, can be tagged by *attributes*; previously, Boogie allowed such attributes only on quantifier expressions (so-called *triggers* [25]). Attributes are opaque for Boogie; they may carry information for external provers and may influence Boogie’s back-ends. For example, consider the following declarations:

```

function {:bvbuiltin "bvadd"} $add.u4(x:bv32, y:bv32) returns(bv32);
function {:bvbuiltin "bvadd"} $add.unchk.u8(x:bv64, y:bv64) returns(bv64);
axiom {:ignore "bvSem"} (∀ x:bv64, y:bv64 :: { $add.unchk.u8(x, y) }
    $check.add.u8(x, y) ⇒ $add.u8(x, y) = $add.unchk.u8(x, y));

```

The attributes at the function declarations direct the Z3 back-end of Boogie to translate these functions into Z3’s built-in bitvector addition operation. The axiom attribute makes the prover back-end ignore the axiom, if a particular model is assumed. The so-called *trigger* at the quantifier is directly passed to Z3, where it is used to find instantiations for quantifiers via matching currently valid hypotheses. Although the entirety of such attributes may sum up to a quite substantial and highly critical part of a background theory (Z3 usually will loop or perform very bad without them!), we will ignore them throughout the presentation of this paper. HOL-Boogie, however, has to keep track of them in order to pass sub-formulas of a verification condition to Z3.

3.5 Generic Proof Support in HOL-Boogie

HOL-Boogie comes with a set of specific tactics to manipulate verification conditions. Based on the structure of verification conditions generated by Boogie, the central tactic of HOL-Boogie, *split-vc*, extracts all assertions and associates them with their execution trace, expressed as a list of premises. Each assertion then forms a subgoal for the proof of the original verification condition.

After splitting a verification condition, each subgoal is passed to a number of sub-tactics given as arguments to *split-vc*. The set of available sub-tactics is extensible and currently comprises a generic SMT binding explained in Section 4, Isabelle’s simplifier, and a specifically tailored tactic for VCC-generated verification conditions (see Section 7.3). These tactics essentially give the “debugging flavor” to HOL-Boogie, as we will see later, because they usually discharge all subgoals except the incorrect or inconsistent ones.

4 A Generic Interface for SMT Solvers

The main task of automated program verification, i. e. deciding whether a given verification condition holds or not, is nowadays performed by SMT (satisfiability modulo theories) solvers. The underlying technology is an efficient combination of DPLL, a fast SAT solving technology, with several first-order theories generally connected by a variant of the congruence closure algorithm due to Nelson and Oppen (see [32]). Theories covered by SMT solvers comprise uninterpreted functions, linear arithmetics, (extensional) arrays, fixed-size bitvectors, and many more, but only few

of the currently available solvers support a considerable subset of these theories. Several SMT solvers also decide (some) quantified formulas using clever heuristics, despite the fact that such formulas are, in general, undecidable. Although many-sorted logics are the default type system in the SMT community, some solvers also support polymorphism [11] or a restricted kind of polymorphism for special built-in functions only.

Each of the available SMT solvers comes with its own native input format, where some differentiate between formulas and terms and others do not. In an effort to provide a standardized interface for all solvers, the SMT-LIB initiative was established. It developed the SMT-LIB format [48], which is nowadays understood by most current SMT solvers. This format is necessarily a compromise, and hence lacks features found in several solvers' native input formats. Firstly, combinations of logical theories are fixed. There is, for example, no logic comprising bitvectors, arrays, uninterpreted functions and quantifiers, which is needed for low-level C verification. Secondly, all logics are many-sorted without any kind of polymorphism, and thirdly, there is a strict separation between terms and formulas. Nevertheless, due to its broad support by solvers, any reasonable generic SMT interface should at least be based on the SMT-LIB format.

Up to now, there have been some attempts to connect interactive provers with SMT solvers [26,31,39], but all of them have been specific in one way or the other. Moreover, none of them targeted Z3, the solver already used as back-end of VCC and Boogie. In contrast, our currently implemented interface to SMT solvers in Isabelle/HOL is designed to be generic in the sense that many different solver input formats can be addressed. For now, only the SMT-LIB format and the native input format of Z3, which supports more features than the former, are covered, but other input formats may be implemented as well, especially if they differ only syntactically to one of the already provided formats. Currently, our interface covers uninterpreted functions, linear integer arithmetic, (extensional) arrays, and bitvectors; moreover, support for quantifier instantiation heuristics (so-called triggers) is included.

As part of the HOL-Boogie core environment, some parts of the rich Isabelle/HOL system were mapped to SMT primitives (notably bitvectors); this provides the basis for all specific machine/memory models for C or other languages represented in BoogiePL. We treat natural numbers as SMT integers and insert conversion functions to guarantee that mapped values are always non-negative. Moreover, variables with function type are transformed into SMT arrays, i. e. reasoning about higher-order functions gets possible. Thus, at least the interface can handle quantified functions, although there are no results yet on how well SMT solvers come along with this task.

Practice shows that speeding up the interface is of paramount importance. Therefore, we implemented the Isabelle SMT interfaces as an (potentially unsound) oracle to avoid the otherwise necessary, complicated proof reconstruction due to Isabelle's LCF architecture. This approach is also guided by the fact that the back-end SMT solver is usually trusted in program verification. Moreover, we included a fast channel to pass a huge background axiomatization directly to the solver, avoiding Isabelle's trusted kernel.

Although at the beginning the aim has been to automate program verification proofs, the SMT binding may as well be applied to other problems in Isabelle/HOL. Consider for example the integer recurrence relation $x_{i+2} = |x_{i+1}| - x_i$, which is periodic with period 9. Since SMT solvers, in general, have no built-in notion of

absolute values, the function $|\cdot| :: int \Rightarrow int$ must be treated as uninterpreted with a suitable axiomatization:

$$\forall (i::int). |i| = (if\ i < 0\ then\ -i\ else\ i)$$

This definition is available in Isabelle as a theorem called *zabs-def*. The periodicity property of the recurrence relation mentioned before then looks as follows in Isabelle/HOL:

```
lemma [| x3 = |x2| - x1; x4 = |x3| - x2; x5 = |x4| - x3;
          x6 = |x5| - x4; x7 = |x6| - x5; x8 = |x7| - x6;
          x9 = |x8| - x7; x10 = |x9| - x8; x11 = |x10| - x9 |]
  ==> x1 = x10 & x2 = (x11::int)
by (smt add: zabs-def)
```

This combination of linear integer arithmetic, uninterpreted functions and quantifiers can be solved by current SMT solvers in an instant; in comparison, Isabelle's *arith* tactic takes several minutes to solve the same problem.

5 Case Study: Verifying Imperative BoogiePL Programs

Widely known and yet fairly complex, Dijkstra's Shortest Path Algorithm already poses a reasonable challenge for verification efforts. The following code, written by Itay Neeman, presents a high-level implementation of Dijkstra's algorithm, abstracting from any memory model and even shortening several initializations and assignments by logical expressions.

```
type Vertex;
const Graph: [Vertex, Vertex] int;
axiom ( $\forall x: \text{Vertex}, y: \text{Vertex} :: x \neq y \implies 0 < \text{Graph}[x,y]$ );
axiom ( $\forall x: \text{Vertex}, y: \text{Vertex} :: x = y \implies \text{Graph}[x,y] = 0$ );

const Infinity: int;
axiom  $0 < \text{Infinity}$ ;

const Source: Vertex;
var SP: [Vertex] int; // shortest paths from Source

procedure Dijkstra ();
  modifies SP;
  ensures  $\text{SP}[\text{Source}] = 0$ ;
  ensures ( $\forall z: \text{Vertex}, y: \text{Vertex} ::$ 
     $\text{SP}[y] < \text{Infinity} \wedge \text{Graph}[y,z] < \text{Infinity} \implies$ 
     $\text{SP}[z] \leq \text{SP}[y] + \text{Graph}[y,z]$ );
  ensures ( $\forall z: \text{Vertex} :: z \neq \text{Source} \wedge \text{SP}[z] < \text{Infinity} \implies$ 
    ( $\exists y: \text{Vertex} :: y \neq z \wedge \text{SP}[z] = \text{SP}[y] + \text{Graph}[y,z]$ ));

implementation Dijkstra ()
{
  var v: Vertex;
  var Visited: [Vertex] bool;
  var oldSP: [Vertex] int;

  havoc SP;
  assume ( $\forall x: \text{Vertex} :: x = \text{Source} \implies \text{SP}[x] = 0$ );
  assume ( $\forall x: \text{Vertex} :: x \neq \text{Source} \implies \text{SP}[x] = \text{Infinity}$ );

  havoc Visited;
  assume ( $\forall x: \text{Vertex} :: \neg \text{Visited}[x]$ );

  while ( $(\exists x: \text{Vertex} :: \neg \text{Visited}[x] \wedge \text{SP}[x] < \text{Infinity})$ )
    invariant  $\text{SP}[\text{Source}] = 0$ ;
```

```

invariant ( $\forall y: \text{Vertex}, z: \text{Vertex} ::$ 
   $\neg \text{Visited}[z] \wedge \text{Visited}[y] \implies \text{SP}[y] \leq \text{SP}[z]$ );
invariant ( $\forall z: \text{Vertex}, y: \text{Vertex} ::$ 
   $\text{Visited}[y] \wedge \text{Graph}[y,z] < \text{Infinity} \implies \text{SP}[z] \leq \text{SP}[y] + \text{Graph}[y,z]$ );
invariant ( $\forall z: \text{Vertex} :: z \neq \text{Source} \wedge \text{SP}[z] < \text{Infinity} \implies$ 
   $(\exists y: \text{Vertex} :: y \neq z \wedge \text{Visited}[y] \wedge \text{SP}[z] = \text{SP}[y] + \text{Graph}[y,z])$ );
{
  havoc v;
  assume  $\neg \text{Visited}[v]$ ;
  assume  $\text{SP}[v] < \text{Infinity}$ ;
  assume ( $\forall x: \text{Vertex} :: \neg \text{Visited}[x] \implies \text{SP}[v] \leq \text{SP}[x]$ );
   $\text{Visited}[v] := \text{true}$ ;
   $\text{oldSP} := \text{SP}$ ;
  havoc SP;
  assume ( $\forall u: \text{Vertex} ::$ 
     $\text{Graph}[v,u] < \text{Infinity} \wedge \text{oldSP}[v] + \text{Graph}[v,u] < \text{oldSP}[u] \implies$ 
     $\text{SP}[u] = \text{oldSP}[v] + \text{Graph}[v,u]$ );
  assume ( $\forall u: \text{Vertex} ::$ 
     $\neg(\text{Graph}[v,u] < \text{Infinity} \wedge \text{oldSP}[v] + \text{Graph}[v,u] < \text{oldSP}[u]) \implies$ 
     $\text{SP}[u] = \text{oldSP}[u]$ );
}
}

```

While developing algorithms and their specifications like the one given here, it commonly happens that, even if a program behaves as intended, its specification is incomplete or inconsistent. Indeed, when letting Boogie check the given program, it reports the following error message²:

```

Spec# Program Verifier Version 0.88, Copyright (c) 2003-2007, Microsoft.
dijkstra.bpl(34,5): Error BP5005: This loop invariant might not be
  maintained by the loop.
Execution trace:
  dijkstra.bpl(26,3): anon0
  dijkstra.bpl(33,3): anon2_LoopHead
  dijkstra.bpl(42,5): anon2_LoopBody
Spec# Program Verifier finished with 0 verified, 1 error

```

Using HOL-Boogie we can navigate to the cause for this error and inspect it. The underlying techniques, described in Section 3.5, split the verification condition into altogether 11 subgoals and pass each of them to Z3, which can discharge all of them except two. The first of the remaining subgoals, without its premises, reads as follows in HOL-Boogie :

```
assert-at Line-34-Column-5 ( SP-2 Source = 0)
```

This formula corresponds to a negatively labeled formula in the verification condition generated by Boogie. Note that *SP-2* is an inflection of the program variable *SP* holding the computed shortest paths after arbitrary runs of the **while** loop.

The subgoal found by HOL-Boogie is exactly the cause of the error reported by Boogie, as the position label indicates. The associated premises represent the complete execution trace until the point where the above invariant is checked. Among those premises, there is one remarkably similar to the subgoal above:

```
SP-1 Source = 0
```

while two premises express properties of *SP-2* stemming from the last two assumptions in the **while** loop:

² Note that the current version of Boogie (version 0.90) does not terminate within a minute for the same problem.

$$\begin{aligned} & \bigwedge u. \text{Graph}(v-1, u) < \text{Infinity} \wedge \text{SP-1 } v-1 + \text{Graph}(v-1, u) < \text{SP-1 } u \\ & \implies \text{SP-2 } u = \text{SP-1 } v-1 + \text{Graph}(v-1, u) \\ & \bigwedge u. \text{Graph}(v-1, u) < \text{Infinity} \wedge \neg(\text{SP-1 } v-1 + \text{Graph}(v-1, u) < \text{SP-1 } u) \\ & \implies \text{SP-2 } u = \text{SP-1 } u \end{aligned}$$

Based on those properties, we attempt to prove the subgoal. Note that we can use the rich proof support developed for Isabelle, including the structured proof language Isar as well as a connection to external first-order provers for filtering relevant lemmas [40]. Consider the following Isar extract:

```

boogie-vc Dijkstra
proof (split-vc try: smt simp)
  case goal1
    note 1 = ⟨SP-1 Source = 0⟩
    note 2 = ⟨ $\bigwedge u. \text{Graph}(v-1, u) < \text{Infinity} \wedge \text{SP-1 } v-1 + \text{Graph}(v-1, u) < \text{SP-1 } u$ 
       $\implies \text{SP-2 } u = \text{SP-1 } v-1 + \text{Graph}(v-1, u)$ ⟩
    note 3 = ⟨ $\bigwedge u. \text{Graph}(v-1, u) < \text{Infinity} \wedge \neg(\text{SP-1 } v-1 + \text{Graph}(v-1, u) < \text{SP-1 } u)$ 
       $\implies \text{SP-2 } u = \text{SP-1 } u$ ⟩
    show ?case
    proof (simp add: labels)
      show SP-2 Source = 0
      proof (cases)
         $G(v-1, \text{Source}) < \text{Infinity} \wedge$ 
         $\text{SP-1 } v-1 + G(v-1, \text{Source}) < \text{SP-1 } \text{Source}$ 
      case True
        with 2 have SP-2 Source = SP-1 v-1 + G(v-1, Source) by simp
        moreover from True have  $\text{SP-1 } v-1 + G(v-1, \text{Source}) < \text{SP-1 } \text{Source}$  by simp
        ultimately have SP-2 Source < 0 using 1 by simp
oops

```

We want to show that $\text{SP-2 } \text{Source} = 0$ holds, but with a case distinction and the fact noted as 2, we can deduce that $\text{SP-2 } \text{Source} < 0$. Clearly, there must be an error in the specification with respect to the implementation (or vice versa) leading to this contradiction. Indeed, since Dijkstra’s algorithm takes as input a graph with only non-negative distances, and shortest paths are computed by adding up those distances, no computed shortest path, including that for the source node, may ever be negative. The invariants of the **while** loop, however, do not maintain this property. We correct the specification by adding the following invariant:

invariant $(\forall x: \text{Vertex} :: \text{SP}[x] \geq 0)$;

This addition suffices to correct the specification; the program can now be verified automatically by Boogie and Z3.

6 From CVM to the VCC1 Annotation Language

Compiling an annotated ANSI-C program into a BoogiePL program over an abstract machine/memory model adds an additional layer in the tool-stack and therefore some new complications. VCC [15,16,50] and its C front-end represents a compiler supporting a very large subset of the C language; at time being its front-end as well as its machine/memory model are under active development by Microsoft Research. The concrete version we are describing here — called VCC1 — is based on the concept of *regions*, i.e. sets of pointers representing a subdomain of the memory, which is understood as a finite map from pointers to byte sequences. Another memory

model — an ownership-based one called VCC2 [16] — will be presented later in Section 9. The difference between these two set-ups is reflected in a different front-end and another CVM axiomatization; the architecture shown in Figure 1 remains the same. In the sequel, we will outline the key features of the VCC1 CVM and its representation in BoogiePL.

6.1 The CVM Machine Model

The standard x86-64 machine operations are fairly easily described; we have to distinguish several operator families for unsigned and signed integers of various byte-lengths: `u1`, `u2`, `u4`, `u8`, `i1`, `i2`, `i4`, `i8`. Thus, we have declarations like:

```

...
function $add.i1(x: bv32, y: bv32) returns (bv32);
function $add.i2(x: bv32, y: bv32) returns (bv32);
function $add.i4(x: bv32, y: bv32) returns (bv32);
function $add.i8(x: bv64, y: bv64) returns (bv64);
function $add.u1(x: bv32, y: bv32) returns (bv32);
function $add.u2(x: bv32, y: bv32) returns (bv32);
function $add.u4(x: bv32, y: bv32) returns (bv32);
function $add.u8(x: bv64, y: bv64) returns (bv64);
...
function $cge.i4(x: bv32, y: bv32) returns (bool);
function $cge.u4(x: bv32, y: bv32) returns (bool);
...

```

and their axiomatic definitions (together, these are about 80 operations). Here, the types `bv64`, `bv32`, etc., are built-in types of BoogiePL representing bitvectors of corresponding size. All machine operations are either double-word (`bv32`) or four-word (`bv64`) aligned. Note that these assembler-like operations are modeled as *functions*, i. e. have no side-effects on registers or the like. Any side-effect will be modeled as side-effect on *memory* to be discussed later.

6.2 The CVM Elementary Memory Model

Pointers are axiomatized in the CVM model to be pairs of a *base* and an *offset* (where `$_ptr` is used as pointer constructor), and for pointers to the same base, pointer addition is defined that takes into account the size of an object to be referenced by a pointer:

```

type $ref;
type $ptr;
...
const unique $ldnull: $ptr;
axiom $ldnull == $_ptr(null, 0bv64);

function $offset(p: $ptr) returns (bv64);
function $base(p: $ptr) returns (ref);
function $add.ptr(p: $ptr, off: bv64, elsize: bv64) returns ($ptr);

axiom (∀ p: $ptr, o: bv64, sz: bv64 :: { $add.ptr(p, o, sz) }
  $add.ptr(p, o, sz) ==
  $_ptr($base(p), $add.i8($offset(p), $mul.i8(o, sz))));

```

Pointers can be converted to bitvectors of length 64; the conversion functions are omitted here.

Memory is viewed as a map from references to lists of bytes. The elementary axioms for byte-wise access capture the heart of the memory model:

```

type $memory;
...
function $get8(m:$memory, r:ref, off:bv64) returns(bv8);
function $set8(m:$memory, r:ref, off:bv64, v:bv8) returns($memory);

axiom (∀ m:$memory, r:ref, off:bv64, v:bv8 ::
  $get8($set8(m,r,off,v),r,off) == v);

axiom (∀ m:$memory, r1:ref, r2:ref, off1:bv64, off2:bv64, v:bv8 ::
  r1 ≠ r2 ⇒ $get8($set8(m,r1,off1,v),r2,off2) == $get8(m,r2,off2));

axiom (∀ m:$memory, r1:ref, r2:ref, off1:bv64, off2:bv64, v:bv8 ::
  off1 ≠ off2 ⇒ $get8($set8(m,r1,off1,v),r2,off2) == $get8(m,r2,off2));

```

A *region* denotes a set of pointers; the core operations are:

```

type $region;

function $empty() returns($region);
function $universe() returns($region);
function $region(p:$ptr, len:bv64) returns($region);
function $union(r1:$region, r2:$region) returns($region);
function $contains(small:$region, big:$region) returns(bool);
function $overlaps(r1:$region, r2:$region) returns(bool);

```

over which a small, single-typed set theory is axiomatized. Regions and the `$overlaps` operation are the main tool of this memory model to express the sharing of data structures or their absence.

6.3 The CVM x86 Memory Model

The elementary memory model forms the basis for a more complex one enabling for byte-, word-, double-word- and four-word-wise access into byte lists; the alignment reflects the behaviour of the little-endian x86-64 architecture.

```

...
function $ld.i4(m:$memory, p:$ptr) returns(bv32);
...
axiom(∀ m:$memory, p:$ptr ::
  $ld.i4(m,p) == $get8(m, $base(p), $add.i8($offset(p), 3bv64)) ++
  $get8(m, $base(p), $add.i8($offset(p), 2bv64)) ++
  $get8(m, $base(p), $add.i8($offset(p), 1bv64)) ++
  $get8(m, $base(p), $offset(p)));

function $st.i4(m:$memory, p:$ptr, v:bv32) returns($memory);
axiom {:ignore "bvInt"} (∀ m:$memory, p:$ptr, v:bv32 ::
  $st.i4(m, p, v) ==
  $set8(
    $set8(
      $set8(m,
        $base(p), $offset(p), v[8:0]),
        $base(p), $add.i8($offset(p), 1bv64), v[16:8]),
        $base(p), $add.i8($offset(p), 2bv64), v[24:16]),
        $base(p), $add.i8($offset(p), 3bv64), v[32:24]));
  ...

```

For all complex x86-64-oriented load/store operations, axiomatic definitions of this form are given; from them, a number of rules — essentially representing a complete case distinction for all `$ld.X` - `$st.X` combinations — are derived that reflect the memory behaviour including alignment issues.

6.4 The *Partial* CVM x86 Memory Model with Typed Ghost State

The elementary memory model described in Section 6.2 models essentially an infinite map from references to bytes. Of course, this is not a realistic “execution model” for C, where only finite chunks of memory can be *allocated*, and access to a chunk has to respect its size. This feature — size of allocated memory assigned to a pointer — is stored in a separate memory, the *ghost memory*. As we will see, ghost memory is not only used for internal purposes, in contrast, the user may use the ghost memory indirectly by “attaching” new information to a pointer. For example, a pointer to the head of a list may be associated with the number of nodes in that list.

The CVM state consists of two program variables (presentation simplified):

```
var $mem : $memory;
var $gmem : [$gid,<x>name]x;
```

where the ghost memory is *typed* memory. This is achieved by a slightly non-standard construct in the BoogiePL type-system, a sort of dependent type that associates to a *type key* $\langle x \rangle$ name data of a given type x ; $\$gid$ is the type of pointers into the ghost state (which were defined bijectively to pointers).

On this basis, the $\$malloc$ operation can be specified by the following contract:

```
procedure $malloc(size: bv64) returns (p: $ptr);
modifies $gmem;
ensures $only_region_changed_or_new($empty(), old($gmem), $gmem, $mem, $mem);
ensures $alloc_grows(old($gmem), $gmem)
ensures
  p ≠ $ldnull ⇒
    $base(p) ≠ null ∧ $offset(p) = 0bv64 ∧
    $sizep($gmem, p) = size ∧ $sizep(old($gmem), p) = 0bv64 ∧
    ∀ r: ref :: r ≠ $base(p) ⇒ $size(old($gmem), r) = $size($gmem, r) ∧
    ∀ q: $gid, n: <x>name :: n ≠ $size ⇒ old($gmem)[q, n] = $gmem[q, n];
ensures p = $ldnull ⇒ old($gmem) = $gmem;
```

The **modifies**-clause is treated as a conjunction of equations like $\text{old}(\$mem) = \mem , relating the old state of all program variables *not* mentioned in the clause to their new state after executing the operation. The predicates $\$only_region_changed_or_new$ and $\$alloc_grows$ control that the ghost memory remains invariant wherever it was defined and that its domain grows. The $\$malloc$ operation can fail and will return the null pointer $\$ldnull$ in this case. If not, it will return a pointer whose base is not null and “fresh”. In any case, the $\$size$ of other references will not change. A peculiar, non-standard construct is contained in the last quantified formula of this specification: a quantification ranging over all type keys (and implicitly over all types). This framing condition is a tribute to the *typed*-ness of the ghost memory and states that all “ghost attributes” different from $\$size$ are unchanged.

A key concept of the memory model is *valid* memory, a notion which is a critical pre-requisite for pointer additions and dereferentiation.

```
axiom (∀ alloc: [$gid,<x>name]x, p: $ptr, len: bv64 ::
  $valid(alloc, p, len) ⇔
    $base(p) ≠ null ∧
    $cge.u8(len, 0bv64) ∧ $cge.u8($offset(p), 0bv64) ∧
    $cge.u8($add.i8($offset(p), len), $offset(p)) ∧
    $cle.u8($add.i8($offset(p), len), $sizep(alloc, p)));
```

This definition — based on machine arithmetic tests — assures that a pointer (including its offset) added to len points to a place in memory that has been allocated before. This is due to the fact that the postcondition of $\text{malloc}(n)$ implies $\$valid(\text{alloc}, p, n-1)$

(provided that $n > 0$); key operations like `$memcpy` and `$memmove` are defined similarly.

6.5 High-Level Annotations of VCC1

Specifying contracts directly in terms of the primitives of the VCC1 memory model is quite tedious and error-prone. Therefore, a number of high-level constructs has been added to facilitate this task.

For example, there is a **writes** clause (similarly to the **modifies** clause on the BoogiePL level) specifying call-by-reference function parameters, and a **reads** clause specifying which parameters are accessed and stay unchanged. For global variables occurring in a **writes** clause the implicit equality between pre-state and post-state is omitted, enabling arbitrary changes on them.

Writing a contract directly on the level of the VCC1 memory model, say, for a C function that computes the maximum of a given array, might look like this:

```
static UINT8 maximum(UINT16 arr[], UINT64 len)
  requires(valid(arr, len * sizeof(UINT16)))
  ensures(valid(arr, len * sizeof(UINT16)))
  ensures(arr == old(arr))
  ensures( $\forall$ (UINT64 i; i < len  $\implies$  arr[i] == old(arr[i])))
```

The built-in function **sizeof**, a compiler-dependent feature, computes for each type the number of bytes which are used to represent it. The precondition and the first two postconditions represent a common pattern in specifications; a short-cut for them are SAL annotations which have been extensively used in the development of the Vista code-base [18,20]. With the help of these annotation short-cuts, a full example for unsigned word-arrays of size bounded by 2^{40} reads as follows:

```
#include "vcc.h"
...
static UINT8 maximum(__in_ecount(len) UINT8 arr[], UINT64 len)
  requires(0 < len  $\wedge$  len < (1UI64 << 40))
  ensures( $\forall$ (UINT64 i; i < len  $\implies$  arr[i]  $\leq$  result))
  ensures( $\exists$ (UINT64 i; i < len  $\wedge$  arr[i] == result))
{
  UINT8 max = arr[0];
  UINT64 p;
  spec(UINT64 witness = 0;)

  for (p = 1; p < len; p++)
    invariant(p  $\leq$  len)
    invariant( $\forall$ (UINT64 i; i < p  $\implies$  arr[i]  $\leq$  max))
    invariant(witness < len  $\wedge$  arr[witness] == max)
    {
      if (arr[p] > max) { max = arr[p]; speonly(witness = p);}
    }
  return max;
}
```

The natural specification for the postcondition “There is a position i such that `result` equals the array at that position” requires that there is an easy match against a witness. In this case, verification succeeds as appropriate ghost code is added to construct it: The declaration `spec(UINT64 witness = 0;)` introduces a local ghost variable, which is only visible inside ghost code, and the ghost code assignment `speonly(witness=p;)` updates that variable whenever a new candidate is found.

The compiled BoogiePL program is significantly larger (about 2400 lines), since it contains the axiomatization of the CVM. In order to give an impression of its abstraction level, we show some code resulting from the **invariant**'s:

```

invariant $cle.u8(p, len);
invariant ( $\forall i: \text{bv64} :: \$_{\text{inrange}}.u8(i) \implies \$_{\text{clt}}.u16(i, p) \implies$ 
  $cle.u4($ld.u1($mem, $add.ptr(arr, i, 1bv64)), max));
invariant $clt.u8(witness, len)  $\wedge$ 
  $ld.u1($mem, $add.ptr(arr, witness, 1bv64)) = max;
free invariant $only_region_changed_or_new(
  old($region(arr, $mul.u8(len, 1bv64))),
  old($gmem), $gmem, old($mem), $mem);
invariant $alloc_grows(#temp10, $gmem);

```

and the resulting verification conditions — which have all architecture-specific pad-dings resolved — are on the same level. However, as we will see, representation in terms of the Isabelle bitvector library will drastically simplify this type of formulae.

Another important concept that we introduce here in more detail is the concept of *ghost fields*: For the purpose of specification, it is possible to annotate data structures with additional fields. Conceptually, such ghost fields denote “attributes” of data cells in memory. Consider, for example, a list node for which we want to specify the number of nodes in that list (**length**) and the set of contained nodes (**entries**, a set of pointers to **void** to be casted accordingly):

```

struct list_node {
  int val;
  struct list_node * next;
  spec(unsigned int length;)
  spec(Set entries;)
};

```

By suitable ghost code, such ghost fields may satisfy stronger object invariants; in particular, typical auxiliary functions computing the values of these ghost-fields can be avoided. Technically, the given ghost fields are represented by two new type keys, $\$_{\text{list_node_length}}$ and $\$_{\text{list_node_entries}}$, to be used for accessing and updating ghost memory. Note that the postcondition part from the $\$malloc$ contract:

$$\forall q: \$gid, n: \langle x \rangle \text{name} :: n \neq \$_{\text{size}} \implies \text{old}(\$gmem)[q, n] = \$gmem[q, n];$$

also ranges over these *new*, user-defined type keys and, consequently, also implies the necessary framing equalities for them.

7 The VCC1 Plugin in HOL-Boogie

7.1 Handling Bitvectors

Many machine operations shown in Section 6.1 are tagged by attributes (cf. Section 3.4) resulting in their direct map to built-in operations in Z3. HOL-Boogie uses these attributes in the same way, but maps bitvector functions to an Isabelle/HOL formalization of polymorphic, fixed-size bitvectors [22]. This has two consequences: First, since the Isabelle/HOL bitvector theory is based entirely on axiomatic definitions and derived rules, reasoning about bitvector operations is sound. Second, the syntax of bitvector operations strongly resembles their C code counterparts. Consider, for example, the three loop invariants of the **maximum** function (Section 6.5).

Here are the corresponding goals in HOL-Boogie (the locally valid premises have been dropped):

```

assert-at Line-19-Column-5 (p-1 ≤ len)
assert-at Line-20-Column-5 (∀ i. $min.u8 ≤ i ∧ i ≤ $max.u8 ⇒ i < p-1 ⇒
  $ld.u1($mem, $add.ptr(arr, i, 1)) ≤ max-3)
assert-at Line-21-Column-5 (witness-1 < len ∧
  $ld.u1($mem, $add.ptr(arr, witness-1, 1)) ≤ max-3)

```

Note that the machine operations `$cle.u8` or `$mul.u8` disappeared again and have been replaced by standard mathematical symbols referring to operations of the Isabelle bitvector library; the implicit types still distinguish them appropriately. Thus, even on the representation level, a significant simplification and conciseness is achieved while maintaining the necessary semantic precision.

7.2 Treating Typed Ghost Memory

Conceptually, ghost memory is modeled in BoogiePL as array (or function) from pointer and type key to the corresponding type of that type key. Quantifications and comparisons on type keys are allowed. Since the type system of Isabelle/HOL does not support dependent types, let alone predicative comparisons on types as used in the shown axioms, we need a different way to represent ghost memory and its access. As a pre-requisite, we abstract the BoogiePL type `:[$gid,<x>name]x` by the type *heap*.

The key observation is that there are only *universal* quantifications over type keys, and they all occur outermost and satisfy the main restrictions as “shallow type quantifications” for Hindley-Milner type-systems. Furthermore, for any concrete program there is only a *finite* number of type keys possible. This gives rise to the idea that such quantifications can be viewed as *axiom schemes*, and constants containing type keys in their types as *constant declaration schemes*; for a given program, these schemes can be expanded.

This means that for the select operation `$gmem[gp,tt]` and for the update operation `$gmem[gp,tt] := E` we have to introduce a constant family. For concrete type keys (such as `$size`), this implies the declaration:

```

consts
  select-S-size :: heap ⇒ Sgid ⇒ 64 word
  store-S-size :: heap ⇒ Sgid ⇒ 64 word ⇒ heap

```

together with an Isabelle pretty-printer configuration resulting in output such as $(\$-size) Sgmem[: p:]$ or $Sgmem[: p \rightarrow (\$-size) E:]$, respectively.

Typical access and update in BoogiePL:

```

$gmem1[p, $size] := 1bv64 + $gmem0[q, $size];

```

look now as follows in HOL-Boogie:

```

Sgmem1 = Sgmem0[: p → ($-size) (1 + ($-size) Sgmem0[: q :]) :]

```

The “classical” BoogiePL axiom of the ghost memory theory:

```

(∀ gp:$gid, n1:<x>name, n2:<x>name, h:[$gid,<x>name]x::
  n1 ≠ n2 ⇒ h[gp,n1][gp,n2] = h[gp,n2])

```

is now expanded to

```

(∀ gp:$gid, h:[$gid,<x>name]× ::
  (False ⇒ h[gp,$size][gp,$size] = h[gp,$size])) ∧
  (False ⇒ h[gp,$tag][gp,$tag] = h[gp,$tag])) ∧
  (True ⇒ h[gp,$size][gp,$tag] = h[gp,$tag])) ∧
  (True ⇒ h[gp,$tag][gp,$size] = h[gp,$size]))

```

where `$tag` is assumed to be the only type key besides `$size`. Elementary logical simplification and representation in terms of the select and update operator family described above yields the axiom set used in HOL-Boogie. Note that the critical framing equalities of `$malloc`'s postcondition (see discussion in Section 6.4) are similarly generated.

Our encoding scheme makes no assumption over the type of the type keys; in principle, this can be anything, like for example sets over relations over integer numbers. One might ask if this generality can lead to deep inconsistencies for the generated axiom schemes, and how they can be ruled out systematically. A possible answer is the datatype package described in [14], where exactly this axiom system is automatically derived entirely from definitional axioms. This approach can also pave the way for incremental and modular proof techniques.

7.3 VCC1 Specific Tactic Support

Just dumping 900 axioms into an automated proof procedure does neither work for Z3 nor for Isabelle's tactics *metis* [34] or *auto*; for both provers, the axiom sets have to be carefully instrumented. In the case of Z3, this has been done by a substantial amount of work on the axiom annotations and the triggers (recall Section 3.4). In the case of Isabelle, axioms have to be syntactically massaged to fit to Isabelle's preferred syntactic rule format. Moreover, rules have to be classified to be simplification rules, introduction rules, elimination rules, etc., and to be collected in appropriate rule sets used by certain tactics to be processed in a particular order. Sometimes, some derived rules proved necessary for a better degree of automation.

As a result, we developed the new tactic *vcc1-auto*, which can also be applied as sub-tactic for *split-vc*. Together with *metis*, we had been able to natively show in Isabelle most of the verification conditions which could be proved by Z3. However, *vcc1-auto* is usually slower, and proof scripts tend to profit from filtering relevant assumptions by prior Z3 proof attempts. Nevertheless, producing native proofs is a viable option both for increasing confidence in the tool integration as well as having technical independence from external provers. Note that *vcc1-auto* may also be extended easily.

7.4 A Generic Infrastructure for Memory Models

The *VCC1 context* is an Isabelle/HOL *theory* on top of HOL-Boogie and contains axiomatization for a concrete machine/memory model, its instrumentation as well as specific tactics such as *vcc1-auto*, and code to treat the specialities of the typed ghost memory.

Thus, other theories for other machine/memory models may be added easily to HOL-Boogie. Besides the natural candidate VCC2, the current head of the development, it is also conceivable to add memory models for C# programs that are also based on the Boogie verification condition generator.

8 Case Study: Verifying Circular Singly-Linked Lists in VCC1

8.1 Annotating the Data Structure

Linked lists are standard data structures in C, possibly showing up in nearly all programs written in C. Their implementation is simple, but specifying correctness — especially in a way suitable for automated solvers — is not trivial. This fact makes them a good case study for HOL-Boogie. Here, we restrict to circular singly-linked lists and to two basic operations on them, which can be implemented in C as follows:

```

typedef struct _Entry
{
    struct _Entry * next;
} Entry, *List;

void initialize_list(List l)
{
    l->next = l;
}

void insert_entry(List l, Entry* entry)
{
    entry->next = l->next;
    l->next = entry;
}

```

Note that, for simplicity, list entries do not contain any data except for a pointer to the successor entry.

A specification for the two operations can be given in terms of graphs, a natural abstraction of linked data structures. Consider the set of natural numbers from 0 to $n - 1$, where n is the number of list entries, as vertices of a graph, and take the successor relation on the first n natural numbers as edges of the graph. With a further edge from $n - 1$ to 0, we get a simple model for circular singly-linked lists, if we additionally specify a mapping from pointers to natural numbers and then verify that the relation induced by the entries' `next` pointers corresponds to the edges of our graph. Then, the function `initialize_list` has to establish this invariant to be correct. The function `insert_entry` has to maintain this invariant while extending the graph with a vertex corresponding to the inserted entry.

For every list, we restrict the domain of the according mapping to the set of list entries. Additionally, we maintain a distinguished head entry and, because sets in VCC's background axiomatization do not support cardinalities, the number of entries in the list. The usual approach to keep shared data for a list is to define a separate structure and let every list entry point to that shared structure. It would be natural to keep the mapping from entries to natural numbers also in this shared structure, but it turns out that storing these numbers along with each corresponding entry makes the specification and verification simpler. To avoid arithmetic overflows, we limit the size of the list, and thus the maximum values taken by the mapping's range, to an arbitrarily chosen value.

The annotated version of the list data structure then looks as follows:

```

#define MAX_LIST_SIZE 10000
typedef struct _Entry Entry, *List;

speonly(
    typedef struct _Meta
    {

```

```

    spec(unsigned int size;)
    spec(Entry* head;)
    spec(Set entries;)
  } Meta;
)

typedef struct _Entry
{
  struct _Entry* next;
  spec(Meta* meta;)
  spec(unsigned int index;)
} Entry, *List;

```

8.2 Annotating the Code

Based on the above annotated data structure, we formulate the validity of a circular singly-linked list using the graph model and the mapping from entry pointers to natural numbers developed before. Since the following invariant is rather long, we present it piece-wise. To be valid, a list l has to fulfill the following properties³:

```

spec(bool is_valid_list(List l)
  reads(*l)
  returns(

```

- There is exactly one meta structure instance for the list, i.e. every list entry points to the same meta structure:

$$\forall(\text{Entry* } e1; \text{Entry* } e2; \text{set_in}(e1, l \rightarrow \text{meta} \rightarrow \text{entries}) \wedge \text{set_in}(e2, l \rightarrow \text{meta} \rightarrow \text{entries}) \implies e1 \rightarrow \text{meta} = e2 \rightarrow \text{meta}) \wedge$$

- Every list entry is contained in the set `entries` (the domain of our model). In particular, the distinguished head and l itself are in that set:

```

set_in(l, l->meta->entries) ^
set_in(l->meta->head, l->meta->entries) ^
forall(Entry* e;
  set_in(e, l->meta->entries) ==>
  set_in(e->next, l->meta->entries)) ^

```

- The mapping from entries to numbers is bijective. We state this in the following two properties. Firstly, the range of the mapping is the set of natural numbers from 0 to $n - 1$, where n is the size of the list. Note that the lower border is already guaranteed by the index type. Secondly, the mapping from entries to numbers is injective.

```

forall(Entry* e;
  set_in(e, l->meta->entries) ==> e->index < l->meta->size) ^
forall(Entry* e1; Entry* e2;
  e1 != e2 ^ set_in(e1, l->meta->entries) ^
  set_in(e2, l->meta->entries) ==>
  e1->index != e2->index) ^

```

³ Note that also specification functions require frame conditions. Here, `reads(*l)` specifies that the content of l is only read and will remain unchanged therefore.

- The model corresponds to the structure of the list, i.e. the relation induced by the entries' next pointers is reflected by the successor relation on natural numbers. Furthermore, the index of the head's predecessor is the greatest number of the mapping's range, or, in terms of the graph model, there is an additional edge from the last to the first entry.

```

(l->meta->head->index == 0) ∧
∀(Entry* e;
  set_in(e, l->meta->entries) ⇒
    (e->next == l->meta->head ⇔ e->index == l->meta->size - 1) ∧
    (e->next ≠ l->meta->head ⇔
      e->index + 1 == e->next->index))
;)

```

This invariant specifies the correspondence between a list and its abstract model, but it does not cover memory properties. The following invariant does just that. It states that the list's meta structure and all entries are valid memory regions and that none of these regions pairwise overlap:

```

spec(bool is_valid_memory(List l)
reads(*l)
returns(
  valid(l->meta, sizeof(Meta)) ∧
  ∀(Entry* e;
    set_in(e, l->meta->entries) ⇒
      valid(e, sizeof(Entry)) ∧
      disjoint(region(e, sizeof(Entry)),
        region(l->meta, sizeof(Meta)))
  ) ∧
  ∀(Entry* e1; Entry* e2;
    e1 ≠ e2 ∧ set_in(e1, l->meta->entries) ∧
    set_in(e2, l->meta->entries) ⇒
      disjoint(region(e1, sizeof(Entry)), region(e2, sizeof(Entry)))
  )
;)

```

The annotation of the implementation requires yet two further specification functions, which we only mention here without showing the corresponding code. The property `is_fresh_for(e, l)` specifies that the entry `e` is a valid region and does not overlap with the meta structure of the list `l` or any of `l`'s entries. By `list_footprint(l)`, we refer to the union of all entry regions of the list `l`.

With these specification functions at hand and the model in mind, it is straightforward to annotate `initialize_list`:

```

void initialize_list(List l)
requires(valid(l, sizeof(Entry)))
writes(*l)
allocates(region(l->meta, sizeof(Meta)))
ensures(isfresh(l->meta))
ensures(is_valid_memory(l))
ensures(is_valid_list(l))
ensures(l->meta->head == l)
ensures(l->meta->size == 1)
ensures(set_in(l, l->meta->entries))
{
  l->next = l;

  speonly(
    Meta* meta = (Meta*) malloc(sizeof(Meta));
    assume(meta ≠ NULL);
    l->meta = meta;
    l->meta->entries = set_singleton(l);
    l->meta->size = 1;
  )
}

```

```

    l->meta->head = l;
    l->index = 0;
  )
}

```

Note that, since we require a new instance of the meta structure to be associated with the initialized list, we need to allocate a block of memory for it, even if the structure resides in ghost memory.

In contrast to `initialize_list`, properly annotating `insert_entry` is more involved. The specification becomes complicated because the mapping from entries to indices has to be updated; the index of every entry following the new one up to the last entry has to be increased by 1. Looping through the list requires a suitable set of invariants, and we used HOL-Boogie in the way described in Section 5 to discover the necessary ones. The fully annotated function then looks as follows:

```

void insert_entry(List l, Entry *entry)
  requires(is_valid_memory(l))
  requires(is_valid_list(l))
  requires(l->meta->size < MAX_LIST_SIZE)
  requires(is_fresh_for(entry, l))
  writes(*l, *(l->meta), list_footprint(l), *entry)
  ensures(is_valid_memory(l))
  ensures(is_valid_list(l))
  ensures(l->meta->size == old(l->meta->size) + 1)
  ensures(set_equal(l->meta->entries,
    set_union(old(l->meta->entries), set_singleton(entry))))
{
  speconly(
    entry->index = l->index + 1;
    entry->meta = l->meta;

    Entry* current = l;
    Set visited = set_empty();
    while (¬set_equal(visited, l->meta->entries))
      invariant(l->meta == old(l->meta))
      invariant(l->meta->entries == old(l->meta->entries))
      invariant(∀(Entry* e; set_in(e, l->meta->entries) ⇒
        valid(e, sizeof(Entry))))
      invariant(set_in(current, l->meta->entries))
      invariant(set_subset(visited, l->meta->entries))
      invariant(valid(entry, sizeof(Entry)))
      invariant(l->meta->size == old(l->meta->size))
      {
        if (current->index ≥ entry->index) current->index++;
        visited = set_union(visited, set_singleton(current));
        current = current->next;
      }
  )

  l->meta->entries = set_union(l->meta->entries, set_singleton(entry));
  l->meta->size++;
)

  entry->next = l->next;
  l->next = entry;
}

```

8.3 Verifying Circular Singly-Linked Lists

To inspect the subgoals stemming from a verification condition — and to verify them completely with Isabelle’s built-in tools —, one can apply the *split-vc* tactic

without any sub-tactic. Then, the proof of `initialize_list`, for example, boils down to 14 subgoals, the first two of which are (omitting the premises here):

```

assert-at Line-81-Column-3
  $valid($gmem, l, 8)

assert-at Line-81-Column-3
  $contains(
    $region(l, 8),
    $union($writes-0, $union($allocated-0, $localAllocated-0)))

```

Note that HOL-Boogie is able to track the location of assertions back to the original source code. In this case, line 81 reads as follows:

```
l->next = l;
```

Thus, the first of the above subgoals asserts that `l` is a valid memory region of 8 bytes, i. e. the next pointer of `l` can be accessed. The second subgoal asserts that `l` is in the `writes` set of the function, i. e. modifying the next pointer of `l` complies with the frame conditions of `initialize_list`. Verifying the above two subgoals, and at the same time another nine of the altogether 14 subgoals of `initialize_list`, can easily be done by applying the `vcc1-auto` sub-tactic for `split-vc`. Proofs for the remaining subgoals involve unfolding the definitions of involved, user-defined specification functions, which cannot easily be automated. The complete proof of `initialize_list`'s correctness in Isabelle's structured proof language Isar reads as follows:

```

boogie-vc initialize-list
proof (split-vc try: vcc1-auto)
  case goal1 with lemma-1 show ?case by (auto simp add: ...)
next
  case goal2 then show ?case
    unfolding is-valid-list-def ...
    by (split-vc try: vcc1-auto)
next
  case goal3 then show ?case
    unfolding is-valid-memory-def ...
    proof (split-vc try: vcc1-auto)
      case goal1
      then have e = l by (simp add: ...)
      with goal1 show ?case
        by (cases $base(l) = $base(malloc-result)) (auto simp add: ...)
    qed
  qed
qed

```

Finding the proofs for the last two subgoals was guided by the formula's structure, stepwise unfolding definitions and applying selected rules of the VCC1 context (see Section 7.4)

Similarly as in Section 5, we may also apply the SMT tactic to verify `initialize_list`, now in combination with the purely Isabelle-based, automated tactic applied before:

```

boogie-vc initialize-list
proof (split-vc try: smt)
  case goal1 then show ?case
    unfolding is-valid-list-def ...
    by (split-vc try: smt vcc1-auto)
  qed
qed

```

The verification of `insert_entry` is even easier, as all 43 subgoals can already be discharged by the SMT tactic:

```
boogie-vc insert-entry
by (split-vc try: smt)
```

Note that VCC1 fails to prove this verification condition.

9 The VCC2 machine/memory model

The VCC2 machine/memory model [17] follows a completely different verification methodology than VCC1. Instead of regions, VCC2 uses a more high-level, typed memory model based on ownerships and allowing two-state object invariants. Despite this major difference, the overall architecture of the tool chain (see Figure 1) and the annotation language (with pre- and post-conditions, writes- and reads-clauses, see Section 6.5) remains the same.

9.1 Typed Memory Model

Memory is modeled as a map from pointers to values, where pointers are represented as pairs of a type and a reference. This allows for distinguishing between `&b` and `&b.a` in the following code:

```
struct A { int x; };
struct B { A a; int y; } b;
```

When seen as references in the light of pure pointer arithmetic, `&b` and `&b.a` are equal (that is, they point to the same memory location), but as pointers in the sense of the VCC2 model, they are unequal since their types differ. This latter fact is exploited when introducing ownerships and invariants in the next section.

By default, aliasing between pointers of different types is not allowed (the only exception being *memory reinterpretations* to be explained later), that is, if `T` and `S` are distinct types, then modifying the value of a pointer of type `T*` must not change the value of a pointer of type `S*`. We ensure this by introducing the set of non-overlapping top-level valid structs. Moreover, we restrict heap access to *valid* locations only: A field access on a valid pointer yields a valid pointer (the address of the field is valid, not the pointer possibly stored in the field), and each valid pointer can only be constructed by a unique sequence of valid accesses. Hence, we disallow aliasing between fields of different structs or different fields of the same struct.

To handle unions and operations like byte-wise copy of memory, as well as the complicated pointer arithmetic used inside the memory allocator, we introduce an explicit *memory reinterpretation*. Reinterpreting `T *t` as `(S*)t` proceeds in two steps. First, we assign all fields of `(S*)t` to values which result from an coercion over the underlying bit-representations from values of fields in `t`; second, we replace `t` by `(S*)t` in the set of top-level valid structs.

Memory reinterpretation is essentially confined to the memory allocator, to generic byte-wise operations and to unions (whose fields reside at the same location in memory, i.e. changing one field effects all over fields). Memory reinterpretation allows for a precise treatment of such special cases while providing a representation that can be handled by the prover efficiently in the common case, where the type information can be used to partition the flat `C` memory into separate objects.

9.2 Invariants and Ownership

Similar as in the VCC1 model, we maintain ghost fields for every object. The boolean ghost field *closed* is one of them and expresses, when set, that an object is not under construction or in the middle of a change.

Based on this flag, we introduce *two-state invariants* for every type, i.e. formulas holding for closed objects and preserved across system transitions. For example, we can state:

```

struct S {
  volatile int x;
  invariant(0 ≤ x ∧ x < 100)
  invariant(old(x) ≤ x)
};

```

which will allow the field *x* to grow only, as long as the object stays closed; moreover, *x* is forced to stay within the bounds of 0 and 99. Thus, invariants restrict the set of “proper” instances of a type as well as the possible changes of such instances.

An invariant of an object *q* is said to be *admissible* if it cannot be broken by changes to objects other than *q*, provided that those objects are not closed or the changes preserve their invariants. Consider, for example, the following structure:

```

struct T {
  struct S *s;
  int y;
  invariant(y ≤ s->x)
};

```

The invariant of *T* is admissible (modulo closedness of *s*, see below), because the invariant of *S* guarantees that *s->x* only grows; thus, any change of *s->x* maintains the invariant of *T*. For the same reasons, the invariant *y == s->x*, for example, would not be admissible.

Invariant admissibility is enforced by generating a proof obligation for every invariant, to be proven separately from the properties of the actual executable code. Admissible invariants can then be checked locally, i.e. upon an update the only invariant that might be broken is the one of the object being updated.

Since an object can only depend on invariants of closed objects, it must be possible for an object to establish closedness of other objects. We use the concept of *ownership* for this purpose: If *q* owns *p* then we can assume, when checking admissibility of *q*’s invariant, that *p* is closed. Ownership is expressed by attaching to each object a special *owner* ghost field pointing to the owning object.

Methodologically, we will enforce that each object can only have one owner, which is common in ownership systems. This restriction can be mitigated by means of a *handle* object, specified in the following way:

```

struct Handle {
  struct Resource *resource;
  invariant(closed(resource) ∧ (this in resource->handles))
};
struct Resource {
  volatile Set handles;
  invariant(old(closed(this)) ∧ closed(this) ⇒ handles == set_empty())
  invariant(∀(struct Handle *h;
    (h in old(handles)) ∧ ¬(h in handles) ⇒ ¬closed(h)))
  invariant(old(handles) == handles ∨ inv2(owner(this)))
};

```

Although the handle does not own the resource, it guarantees in its invariant that the resource is closed; moreover, since a handle is not the owner of a resource, there may be multiple handles on the same resource. For the invariant of the handle to be admissible, we let the resource keep track of outstanding handles. The resource guarantees that (1) it will only open if there are no outstanding handles, (2) if a handle is closed and points at this resource, it will not be removed from the set of outstanding handles, and finally (3) the handle set can only be updated if the owner of the resource complies with that update. The last condition allows the owner of the resource to control the set of handles, so the resource can be disposed, once there are no more handles.

We additionally introduce *claims*, which are built-in handles on multiple objects, possibly stating additional properties. These properties are much like invariants of the claim objects, subject to admissibility checks, but can combine information from invariants of multiple objects. Claims allow for expressing complex ownership dependencies as well as for capturing and passing around information about the system state.

Two-state invariants are also used to describe concurrent, atomic changes to objects. This allows for verification of lock-free data structures (like spin locks, reader-writer locks, volatile stacks and other custom concurrency protocols), with useful external specification. For example, our specification for the lock is much like the one in Concurrent Spec# [35], where the lock is introduced as a primitive.

10 Proving Consistency and Refinements of Machine/Memory Models

At present, the axiomatization of VCC1 — called Prelude Version 61 — consists of about 900 axioms, where a certain number of axioms is not even taken into account since they are “built-in” into the target prover; for example, reflexivity of equality or the laws of arithmetic. There had been a number of errors in the current and similar formalizations of background theories; and consistency is even a greater issue if Boogie is used with *different* machine/memory models. Since the abstraction level of a machine model is tantamount for deduction efficiency, more refined models should be used only when inherently needed. This is the case if, for example, the allocation function itself must be verified, which is atomic in a more abstract model, or when inherently untyped memory is required such as in unions, where everything is translated into bitvectors.

From the perspective of a HOL system, proving the consistency of a complex first-order system is not exactly an easy task, but at least routine: Just build up a theory by conservative extensions, i. e. by constant or type definitions, and derive all the “axioms” from it. In the sequel, we report on a verification of a previous version of the CVM model (Prelude Version 3.0). Since the CVM models are rapidly changing at present, we plan to repeat this effort at a later stage.

The conservative theory for Prelude 3.0 is constructed as follows: First, a simple bitvector library is built; bitvectors are represented as lists of boolean, and operations like *length*, *extract*, and *concat* were defined as usual. Since the CVM operations work only in byte and word formats, the necessary side-conditions referring to *length* can be omitted if these formats are already expressed at the type level, for example:

```
typedef bv32 = {x :: bool list. length x = 32 }
```

Arithmetic operations for signed and unsigned integers are defined over *bv32*, as well as bitwise conjunction or disjunction. For example, consider the definition:

```
constdefs shr-i4 :: [bv32, bv32] => bv32
            shr-i4 v w ≡ Abs-bv32 (bv-shr (Rep-bv32 v) (Rep-bv32 w))
```

where *bv-shr* is defined on bitvectors directly representing the usual intuition “division by two” (omitted here). Moreover, following the conventions on signs of the x86 architecture, it is enforced that the most significant bit is replicated and the size of the bitvector remains identical.

Similarly, the type of pointers *ptr* is introduced as a pair of unsigned 64 bit integers (references called *ref*) and an integer; the former is called the *base* and the latter the *offset*. On *ptr*’s, pointer arithmetic operations are defined allowing byte-wise addressing of memory. The core of the memory model is:

```
typedef memory = {x :: ref => Bitvector. True}
types state = (ref => bool) × memory
```

The pivotal concept of a valid reference, as discussed previously, can then simply be defined. Definitions for *malloc* and *free* are straightforward.

We implemented a compiler that takes a *Boogie-Configuration* — i.e. a list of theorems, their names, and attributes — and compiles this information into a BoogiePL background theory file. In particular, attributes are generated that correspond to Isabelle’s internal naming in the theory, for example:

```
axiom { : Isabelle "id_⊔prelude.basics_axms_1" } (∀ x : int :: exp(x, 0) = (1));
```

Since Boogie re-feeds attributes to its target provers, HOL-Boogie can check that every axiom in the background theory of a verification condition indeed exists (and by construction is derived) in its own logical environment; thus, a *strict checking* mode can be implemented that makes sure that a verification in an external prover is based only on a consistent axiomatization. With respect to a conservative model of the ghost state, we suggest to use the HOL-OCL datatype package (see Section 7.2).

11 Conclusion

We have presented a novel Isabelle/HOL-based proof environment, called HOL-Boogie, that is integrated into a verification toolchain for imperative programs, in particular C. This toolchain ranges from an industry-strength IDE, into which VCC is integrated (not described here), over Boogie together with its CVM model, to HOL-Boogie into which a family of SMT solvers has been integrated, most notably Z3. VCC1 leverages a rich annotation language whose distinctive feature is the support of type unions and bit fields.

The verification conditions generated over this particularly fine-grained memory model are already remarkably large (the overall size of the exchange file between VCC and Isabelle for our linked list example is already over 600KB), and so are the generated proof obligations. In contrast to wide-spread belief, this does not rule out the use of an *interactive* theorem prover: for the linked-list example, several man-months have been invested in the classical “fix-and-verify”-cycle using VCC(X)/Z3, while with HOL-Boogie essentially one man-month has been spent. From our comparative experience with these toolchains, we believe that this situation will be similar if other algorithmic problems are attacked (open problems at present in the Verisoft XT

project [47] are red-black-trees, complex, highly efficient sorting-algorithms, complex refinements of data-structures. . .). Even if one ends up with an automated verification of an algorithm, surprisingly, an interactive proof environment can be extremely helpful when debugging the specifications.

Of course, dealing with generated formulas interactively results in technical challenges that need to be addressed. Key issues of our integration are:

1. the support of labels and positions at the proof level, which enables tracking back missing properties to assertions in the source,
2. specific tactic support for decomposition of verification conditions in a way stable under certain changes of the source,
3. specific tactic support for the automated discharge of CVM-related proof obligations (native Isabelle proofs),
4. the seamless integration of target SMT solvers in order to discharge as many subgoals as possible, and
5. mechanisms to exchange meta-information (attributes) between provers.

As a by-product, HOL-Boogie can also be used to verify parts of VCC’s meta-theory, such as proving the (relative) consistency of the axiomatization of the underlying machine/memory model or its refinement towards a more detailed machine model.

11.1 Related Work

As such, combining an interactive prover with a Boogie-like VCG is not a new idea. In Figure 1, just replace C-Front-End by *Caduceus* [28], Boogie by *Why* [27], and Z3 by the default prover ERGO, and one gets (nearly) the architecture of the Why/Caduceus system [29]. The memory model presented here is similar to the embedding of C in Coq developed as part of the ongoing certification of a moderately-optimising C compiler [10]. However, its interactive prover-back-end based on Coq does not support labeling and has no integration of the target SMT solver.

The architecture of VCC is also similar to the architecture of Escher’s C compiler [19]; among other things, we adopted their idea of allowing the user to add mathematical theories to C. KeY-C [42] is a verifier for C that uses dynamic-logic instead of our first-order framework. Another approach to structure C’s memory model was recently performed as part of the L4 kernel verification [51]. They embed separation logic in HOL to achieve better alias control. Except for the L4 kernel verifier, all these verifiers can not handle unions and bitfields. The SPARK programming language, a subset of Ada, has its own verifier [2]. SPARK avoids the issues with anti-aliasing and dangling pointers by disallowing allocation at run time entirely.

There is a quite substantial body on programming language embeddings into HOL, be it *shallow* [49,45] or *deep* [36]. In particular, Leinenbach [36] provides a small-step semantics for a language C0, which has been used for system level verification, and Schirmer [49] derives a (shallow-ish) Hoare-Logic from this semantics and formally developed a verification condition generator. C0 assumes a typed memory model. While C0 has been used in substantial case studies [21,1], the limited language fragment restricts its use to “code designed for verification”, i. e. academic projects. These limitations were partially overcome within the current L4.VERIFIED project [33], where a trusted pre-compiler to C0 is used to enlarge the supported C fragment; the code to be verified is about 10kloc. Still, VCC is designed to push these

limits substantially further; in the Hypervisor Verification Project [47], the goal is to adapt the VCC toolchain to cope with 100kloc of production code of a commercial OS System.

11.2 Future Work

We see the following directions for future work:

1. **More Stable Proof Formats:** In our scenario, where the specification of a program is essentially re-constructed post-hoc, it is the annotations that change constantly under development. This means that positions of assertions change easily, which can (but must not) have influence on proofs resulting from previous proof attempts. A proof style using control-flow labels (as generated by Boogie) would be more stable under changes as our predominately used technique.
2. **Alternative machine/memory models:** The verified C background theory containing the memory and machine axiomatization is currently rapidly evolving; at a later stage, we would like to verify the soundness of the VCC2 model [16]. From our experience, this is a substantial task (several man-months), but routine.

Acknowledgements Sascha Böhme was supported by the German Federal Ministry of Education and Research under grant 01IS07008.

References

1. E. Alkassar, M. A. Hillebrand, D. C. Leinenbach, N. W. Schirmer, A. Starostin, and A. Tsyban. Balancing the load: Leveraging semantics stack for systems verification. *Journal of Automated Reasoning (JAR)*, 42(2-4):389-454, 2009.
2. J. Barnes and J. G. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
3. 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*, LNCS, vol. 4111, pp. 364-387. Springer, 2006.
4. M. Barnett, M. Fähndrich, K. R. M. Leino, F. Logozzo, P. Müller, W. Schulte, H. Venter, and S. Xia. Spec#. Microsoft Research, Redmond, 2008. <http://research.microsoft.com/specsharp>.
5. M. Barnett and K. R. M. Leino. Weakest-Precondition of Unstructured Programs. In *PASTE '05*, pp. 82-87. ACM, 2005.
6. M. Barnett, K. R. M. Leino, M. Moskal, and P. Rümmer. Boogie program verification. Microsoft Research, Redmond, 2008. <http://research.microsoft.com/boogie/>.
7. M. Barnett, K. R. M. Leino, and W. Schulte. The SPEC# Programming System: An Overview. In *CASSIS 2004*, LNCS, vol. 3362, pp. 49-69. Springer, 2005.
8. D. Basin, H. Kuruma, K. Miyazaki, K. Takaragi, and B. Wolff. Verifying a signature architecture: A comparative case study. *Formal Aspects of Computing*, 19(1):63-91, 2007.
9. S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W. J. Paul. Putting it all together: Formal verification of the VAMP. *International Journal on Software Tools for Technology*, 8(4-5):411-430, 2006.
10. S. Blazy and X. Leroy. Formal Verification of a Memory Model for C-Like Imperative Languages. In K.-K. Lau and R. Banach, eds., *ICFEM, Lecture Notes in Computer Science*, vol. 3785, pp. 280-299. Springer, 2005.
11. F. Bobot, S. Conchon, E. Contejean, and S. Lescuyer. Implementing Polymorphism in SMT solvers. In C. Barrett and L. de Moura, eds., *SMT 2008: 6th International Workshop on Satisfiability Modulo*. 2008.
12. S. Böhme. HOL-Boogie, 2008. <http://www4.in.tum.de/~boehmes/hol-boogie.xhtml>.

13. S. Böhme, K. R. M. Leino, and B. Wolff. HOL-Boogie — An Interactive Prover for the Boogie Program-Verifier. In O. A. Mohamed, C. Muñoz, and S. Tahar, eds., *TPHOLs, Lecture Notes in Computer Science*, vol. 5170, pp. 150–166. Springer, 2008.
14. A. D. Brucker and B. Wolff. An extensible encoding of object-oriented data models in HOL with an application to IMP++. *Journal of Automated Reasoning (JAR)*, 41(3–4):219–249, 2008.
15. 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 *Theorem Proving in Higher Order Logics (TPHOLs 2009), Lecture Notes in Computer Science*, vol. 5674. Springer, Munich, Germany, 2009. Invited paper, to appear.
16. E. Cohen, M. Moskal, W. Schulte, and S. Tobies. A practical verification methodology for concurrent programs. Tech. Rep. MSR-TR-2009-15, Microsoft Research, 2009. Available from <http://research.microsoft.com/pubs>.
17. E. Cohen, M. Moskal, W. Schulte, and S. Tobies. A precise yet efficient memory model for C. In *4th International Workshop on Systems Software Verification (SSV 2009)*, Electronic Notes in Theoretical Computer Science. Elsevier Science B.V., 2009. To appear.
18. M. Corp. Visual Studio 2005 Developer Library. Online Documentation, 2005. [http://msdn.microsoft.com/en-us/library/ms235402\(vs.80\).aspx](http://msdn.microsoft.com/en-us/library/ms235402(vs.80).aspx).
19. D. Crocker and J. Carlton. Verification of C Programs using Automated Reasoning. In *SEFM '07: Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods*, pp. 7–14. IEEE Computer Society, 2007.
20. M. Das. Formal Specifications on Industrial-Strength Code—From Myth to Reality. In T. Ball and R. B. Jones, eds., *CAV, Lecture Notes in Computer Science*, vol. 4144, p. 1. Springer, 2006.
21. M. Daum, J. Dörrenbächer, and B. Wolff. Proving fairness and implementation correctness of a microkernel scheduler. *Journal of Automated Reasoning (JAR)*, 42(2–4):349–388, 2009.
22. J. E. Dawson. Isabelle Theories for Machine Words. In *Seventh International Workshop on Automated Verification of Critical Systems (AVOCS'07)*. Elsevier, 2007. To appear.
23. L. de Moura and N. Björner. Z3: An efficient SMT solver. In *TACAS 2008*, LNCS, vol. 4963, pp. 337–340. Springer, 2008.
24. R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Tech. Rep. 2005-70, Microsoft Research, 2005.
25. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A Theorem Prover for Program Checking. *Journal of the ACM*, 52(3):365–473, 2005.
26. L. Erkök and J. Matthews. Using Yices as an automated solver in Isabelle/HOL. In *Automated Formal Methods'08, Princeton, New Jersey, USA*, pp. 3–13. ACM Press, 2008.
27. J.-C. Filliâtre. Why: A multi-language multi-prover verification condition generator. Tech. Rep. 1366, LRI, Université Paris Sud, 2003.
28. J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. In *ICFEM 2004*, LNCS, vol. 3308, pp. 15–29. Springer, 2004.
29. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV 2007*, LNCS, vol. 4590, pp. 173–177. Springer, 2007.
30. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI 2002*, pp. 234–245. ACM, 2002.
31. P. Fontaine, J.-Y. Marion, S. Merz, L. Prensa Nieto, and A. Tiu. Expressiveness + Automation + Soundness: Towards Combining SMT Solvers and Interactive Proof Assistants. In H. Hermanns and J. Palsberg, eds., *12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems - TACAS'06, 03/2006, Lecture Notes in Computer Science*, vol. 3920, pp. 167–181. Springer, 2006.
32. H. Ganzinger, G. Hagen, R. Nieuwenhuis, and C. Tinelli. DPLL(T): Fast Decision Procedures. In *Proceedings of the 16th International Conference on Computer Aided Verification, CAV'04*, pp. 175–188. Springer, 2004.
33. G. Heiser, K. Elphinstone, I. Kuz, G. Klein, and S. M. Petters. Towards trustworthy computing systems: taking microkernels to the next level. *SIGOPS*, 41(4):3–11, 2007.
34. J. Hurd. First-order proof tactics in higher-order logic theorem provers. In M. Archer, B. D. Vito, and C. Muñoz, eds., *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*, no. NASA/CP-2003-212448 in NASA Technical Reports, pp. 56–68. 2003.
35. B. Jacobs, J. Smans, F. Piessens, and W. Schulte. A simple sequential reasoning approach for sound modular verification of mainstream multithreaded programs. *Electr. Notes Theor. Comput. Sci.*, 174(9):23–47, 2007.

-
36. D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In *SEFM 2005*, pp. 2–12. IEEE, 2005.
 37. K. R. M. Leino, T. Millstein, and J. B. Saxe. Generating error traces from verification-condition counterexamples. *Science of Computer Programming*, 55(1-3):209–226, 2005.
 38. K. R. M. Leino, J. B. Saxe, and R. Stata. Checking Java programs via guarded commands. In *FTfJP 1999*, Tech. Rep. 251. Fernuniversität Hagen, 1999.
 39. S. McLaughlin, C. Barrett, and Y. Ge. Cooperating Theorem Provers: A Case Study Combining HOL-Light and CVC Lite. *Electronic Notes in Theoretical Computer Science*, 144(2):43–51, 2006.
 40. J. Meng and L. C. Paulson. Lightweight Relevance Filtering for Machine-Generated Resolution Problems. In *ESCoR: Empirically Successful Computerized Reasoning*, pp. 53–69. 2006.
 41. C. Morgan. The specification statement. *ACM TOPLAS*, 10(3):403–419, 1988.
 42. O. Mürk, D. Larsson, and R. Hähnle. KeY-C: A Tool for Verification of C Programs. In F. Pfenning, ed., *CADE, Lecture Notes in Computer Science*, vol. 4603, pp. 385–390. Springer, 2007.
 43. G. Nelson. A generalization of Dijkstra’s calculus. *ACM TOPLAS*, 11(4):517–561, 1989.
 44. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, LNCS, vol. 2283. Springer, 2002.
 45. M. Norrish. *C formalised in HOL*. Ph.D. thesis, Computer Laboratory, University of Cambridge, 1998.
 46. W. Paul, T. Santen, and S. Tobies. Verifying 50000 lines of code. *Futures — Microsoft’s European Innovation Magazine*, pp. 42–43, 2008.
 47. W. Paul, T. von der Rhieden, T. Santen, and W. Schulte. The Verisoft XT Project. Universität des Saarlandes, 2007.
 48. S. Ranise and C. Tinelli. The SMT-LIB standard: Version 1.2. Tech. rep., Dept. of Comp. Sci., The University of Iowa, 2006. <http://www.smt-lib.org>.
 49. N. Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. Ph.D. thesis, Technische Universität München, 2006.
 50. W. Schulte, S. Xia, J. Smans, and F. Piessens. A glimpse of a verifying C compiler (extended abstract). In *C/C++ Verification Workshop*. 2007.
 51. H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, eds., *POPL*, pp. 97–108. ACM, 2007.
 52. M. Wenzel and B. Wolff. Building Formal Method Tools in the Isabelle/Isar Framework. In *TPHOLs 2007*, LNCS, vol. 4732, pp. 351–366. Springer, 2007.