HOL-Boogie — An Interactive Prover for the Boogie Program-Verifier*

Sascha Böhme[1], K. Rustan M. Leino[2], and Burkhart Wolff[3]

[1] Technische Universität München
boehmes@in.tum.de
[2] Microsoft Research, Redmond
leino@microsoft.com
[3] Universität Saarbrücken
bwolff@wjpserver.cs.uni-sb.de

Abstract. *Boogie* is a program 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.

Its verification conditions — constructed via a *wp* calculus from these annotations — are usually transferred to automated theorem provers such as *Simplify* or *Z3*. In this paper, however, we present a proof-environment, HOL-Boogie, that combines Boogie with the interactive theorem prover Isabelle/HOL. In particular, we present specific techniques combining automated and interactive proof methods for code-verification.

We will exploit our proof-environment in two ways: First, we present scenarios to "debug" annotations (in particular: invariants) by interactive proofs. Second, we use our environment also to verify "background theories", i.e. theories for data-types used in annotations as well as memory and machine models underlying the verification method for C.

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 to be tackled formally, there are meanwhile verification systems like ESC/Java [11], Why/Krakatoa/-Caduceus [10], and Boogie used both for Spec# [3,1] and C [21]. The latter system is also used in a substantial verification effort for the Microsoft Hypervisor as part of the Verisoft XT project ([5], see also `http://www.verisoft.de/`).

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

- verification attempts can be debugged by interactive proofs,
- background theories can be proven consistent,
- existing front-end compilers for Spec# and C to the Boogie-Core-Language represent an alternative to a logical embedding of these languages.

Debugging Verification Attempts. Starting to annotate a given program can lead to situations where the automated prover fails and can neither find a

* Supported by BMBF under grant 01IS07008.

O. Ait Mohamed, C. Muñoz, and S. Tahar (Eds.): TPHOLs 2008, LNCS 5170, pp. 150–166, 2008.
© Springer-Verlag Berlin Heidelberg 2008

proof nor a counterexample. All existing systems report of a degree of automation approaching 100%, causing wide-spread 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 formulation 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 [4].

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 [6] uses random-based heuristics).

An interactive proof, suitably adapted to the problems arising from automated formula generation, decomposes the verification conditions along the program structure and finally the logical structure of the annotations and can thus lead to insufficient preconditions or invariants systematically.

Consistency of Background Theories. Conceptually, the Boogie-Core-Language (called *BoogiePL*) allows for specifying transition systems; these transitions are described in terms of first-order logic over a model comprising user-defined types, constants, axioms as well as program variables. The signature and axiom set is called the "background theory" of a program. A background theory can be just program specific or programming-language specific. In the case of the *Verified C Compiler* (VCC, e.g., Boogie with C front-end, the configuration of Boogie mostly studied in this paper), the operations to be axiomatized consist of elementary operations of a machine model (called *C Virtual Machine* (CVM)), allowing reading and storing byte-wise, word-wise, and double-word-wise, doing signed and unsigned operations in bitvector arithmetic, etc. 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 abstracting from registers and jumps (which are represented by **goto**'s in BoogiePL). A crucial part of the model is concerned with the representation of memory, memory regions etc. in order to formulate frame conditions. VCC compiles an ANSI-C program into a BoogiePL program based on the CVM.

Getting an axiomatization of this size 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.

An Alternative to Embeddings. Compiling ANSI-C to a transition system described in the fairly small and logically clean BoogiePL represents an alternative to a *logical embedding* into HOL (such as, for example, [12,20] describing a small-step transition semantics for the C fragment C0 comprising only side-effect free expressions). While we still consider the logical embedding method as "near

perfection" with respect to logical foundations, it is obvious that an embedding of a more substantial C fragment is an enormous effort with questionable value. Given that ANSI-C language semantics is heavily under-specified, given that optimizing compilers tend to make their own story over the variety of "semantic deviation points", and given that a realistic concurrency model depends on the granularity of the atomic operations defined by the target assembler language, it is debatable if we should care about the sterile myth of some general C semantics or rather concentrate our efforts on a specific compiler and target assembler language. For a given compiler, one can exchange the code-generator against a BoogiePL translator, and validate compiled assembler sequences against the abstract model traces in the CVM by tests (what has been done intensively) or by simulation proofs if needed. There is meanwhile sufficient empirical evidence that a carefully constructed and tested C front-end to a verification condition generator such as Boogie can achieve a reasonable degree of trustworthiness.

Outline of the Paper. We will address the first two issues. After presenting the background of this work, namely Isabelle/HOL, Boogie, and the HOL-Boogie architecture, we present three scenarios of using HOL-Boogie and will explain the underlying machinery at need: In the first scenario, we use HOL-Boogie to verify Dijkstra's Shortest Path Algorithm given as BoogiePL program (only a high-level memory model involved). In the second scenario, we verify a C program, converted into BoogiePL, i.e. a program over the CVM. In the third scenario, we show how CVM axiomatizations can be proven consistent with HOL-Boogie, enabling a "safe mode" of C program verification.

2 Background

2.1 Isabelle/HOL and the Isar Framework

Isabelle is a generic theorem prover [17], 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, 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 [22] 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

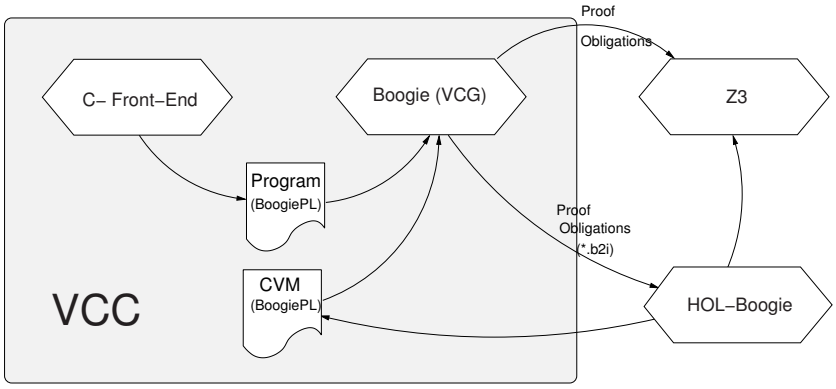

Fig. 1. VCC and the HOL-Boogie back-end

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 *Verified C Compiler*(VCC) evolved from the Spec# project (see `http://research.microsoft.com/specsharp/`, [3]). 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 and 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*. In the case of VCC, an abstract machine model is given in the background theory, describing 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 x64 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 we implemented for interactive back-ends. We also coupled HOL-Boogie to the default target prover *Z3* in such a way that formulas constructed in the former can be discharged by the latter.

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. 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 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., [15,16].

$$
\begin{array}{rcl}
\mathit{BlockSeq} & ::= & \mathit{Block}^+ \\
\mathit{Block} & ::= & \mathit{BlockId} : [\ \mathit{Statement}\ ;]\ \mathit{Goto}\ ; \\
\mathit{Statement} & ::= & \mathit{Var} := \mathit{Expression} \\
& | & \textbf{havoc}\ \mathit{VarId} \\
& | & \textbf{assert}\ \mathit{Expression} \\
& | & \textbf{assume}\ \mathit{Expression} \\
& | & \textbf{call}\ [\mathit{Var}^+ :=]\ \mathit{ProcId}\ (\ \mathit{Expression}^*\ ) \\
& | & \mathit{Statement}\ ;\ \mathit{Statement} \\
\mathit{Goto} & ::= & \textbf{goto}\ \mathit{BlockId}^* \mid \textbf{return}
\end{array}
$$

Fig. 2. Schematic syntax of blocks in BoogiePL

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 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 [14]. 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 [1]:

$$\begin{aligned} &\mathit{LoopHead} : \textbf{assert}\ P\ ;\ \textbf{goto}\ \mathit{LoopBody}, \mathit{LoopDone};\\ &\mathit{LoopBody} : \textbf{assume}\ G\ ;\ B\ ;\ \textbf{goto}\ \mathit{LoopHead};\\ &\mathit{LoopDone} : \textbf{assume}\ \neg G\ ;\ \ldots \end{aligned}$$

More details of BoogiePL can be found in [1,7].

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 [2]. 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 following well-known rules:

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

For every block

$$A : S\ ;\ \textbf{goto}\ B_1, \ldots, B_n;$$

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:

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

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. Note that the verification condition generated this way is linear in the size of the procedure.

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** $L\colon\ P$) or negative (**lblneg** $L\colon\ P$). 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:

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

Negative labels tag formulas of assertions (including invariants and postconditions) with their 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 [13].

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. For example, an attributed axiom looks as follows: **axiom** {attr1} ... {attrN} P. These attributes are opaque for Boogie; they may carry information for external provers and may influence Boogie's back-ends. In the case of Z3, for example, attributes are used to tag some axioms as built-in to Z3.

The attribution mechanism provided by Boogie is flexible enough to add new attributes for any prover back-end.

4 Scenario I: Interactive Verification of Algorithms

4.1 Dijkstra's Shortest Path Algorithm

Widely known and yet fairly complex, this algorithm already poses a reasonable challenge for verification efforts. The following code, written by Itay Neeman,

```
type Vertex;
const Graph: [Vertex, Vertex] int;
axiom (∀ x: Vertex, y: Vertex :: x ≠ y ⟹ 0 < Graph[x,y]);
axiom (∀ x: Vertex, y: Vertex :: x == y ⟹ Graph[x,y] == 0);

const Infinity: int;
axiom 0 < Infinity;

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

procedure Dijkstra();
  modifies SP;
  ensures SP[Source] == 0;
  ensures (∀ z: Vertex, y: Vertex ::
    SP[y] < Infinity ∧ Graph[y,z] < Infinity ⟹ SP[z] ≤ SP[y] + Graph[y,z]);
  ensures (∀ z: Vertex :: z ≠ Source ∧ SP[z] < Infinity ⟹
    (∃ y: Vertex :: y ≠ z ∧ SP[z] == SP[y] + Graph[y,z]));

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

  havoc SP;
  assume (∀ x: Vertex :: x == Source ⟹ SP[x] == 0);
  assume (∀ x: Vertex :: x ≠ Source ⟹ SP[x] == Infinity);

  havoc Visited;
  assume (∀ x: Vertex :: ¬Visited[x]);

  while ((∃ x: Vertex :: ¬Visited[x] ∧ SP[x] < Infinity))
    invariant SP[Source] == 0;
    invariant (∀ y: Vertex, z: Vertex ::
      ¬Visited[z] ∧ Visited[y] ⟹ SP[y] ≤ SP[z]);
    invariant (∀ z: Vertex, y: Vertex ::
      Visited[y] ∧ Graph[y,z] < Infinity ⟹ SP[z] ≤ SP[y] + Graph[y,z]);
    invariant (∀ z: Vertex :: z ≠ Source ∧ SP[z] < Infinity ⟹
      (∃ y: Vertex :: y ≠ z ∧ Visited[y] ∧ SP[z] == SP[y] + Graph[y,z]));
  {
    havoc v;
    assume ¬Visited[v];
    assume SP[v] < Infinity;
    assume (∀ x: Vertex :: ¬Visited[x] ⟹ SP[v] ≤ SP[x]);
    Visited[v] := true;
    oldSP := SP;
    havoc SP;
    assume (∀ u: Vertex ::
      Graph[v,u] < Infinity ∧ oldSP[v] + Graph[v,u] < oldSP[u] ⟹
        SP[u] == oldSP[v] + Graph[v,u]);
    assume (∀ u: Vertex ::
      ¬(Graph[v,u] < Infinity ∧ oldSP[v] + Graph[v,u] < oldSP[u]) ⟹
        SP[u] == oldSP[u]);
  }
}
```

presents a high-level implementation of Dijkstra's algorithm, abstracting from any memory model and even shortening several initializations and assignments by logical expressions.

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 message:

```
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 later in more detail, split the verification condition into altogether 11 subgoals and pass each of them to Z3, which can discharge all of them except one. The remaining subgoal, 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, only two express properties of *SP-2*, while a third one states something similar to the subgoal above:

$$\bigwedge u.\ G[v\text{-}1,\ u] < \mathit{Infinity} \wedge \mathit{SP}\text{-}1[v\text{-}1] + G[v\text{-}1,\ u] < \mathit{SP}\text{-}1[u] \Longrightarrow \mathit{SP}\text{-}2[u] = \mathit{SP}\text{-}1[v\text{-}1] + G[v\text{-}1,\ u]$$

$$\bigwedge u.\ \neg(G[v\text{-}1,\ u] < \mathit{Infinity} \wedge \mathit{SP}\text{-}1[v\text{-}1] + G[v\text{-}1,\ u] < \mathit{SP}\text{-}1[u]) \Longrightarrow \mathit{SP}\text{-}2[u] = \mathit{SP}\text{-}1[u]$$

$$\mathit{SP}\text{-}1[\mathit{Source}] = 0$$

Based on those three properties, we attempt to prove the subgoal. Consider the following Isar extract:

proof (*ib-split try-z3*)
  case *goal1*
  note *H1* = ⟨$\bigwedge u.\ G[v\text{-}1,\ u] < \mathit{Infinity} \wedge \mathit{SP}\text{-}1[v\text{-}1] + G[v\text{-}1,\ u] < \mathit{SP}\text{-}1[u] \Longrightarrow \mathit{SP}\text{-}2[u] = \mathit{SP}\text{-}1[v\text{-}1] + G[v\text{-}1,\ u]$⟩
  note *H2* = ⟨$\bigwedge u.\ \neg(G[v\text{-}1,\ u] < \mathit{Infinity} \wedge \mathit{SP}\text{-}1[v\text{-}1] + G[v\text{-}1,\ u] < \mathit{SP}\text{-}1[u]) \Longrightarrow \mathit{SP}\text{-}2[u] = \mathit{SP}\text{-}1[u]$⟩
  note *H3* = ⟨$\mathit{SP}\text{-}1[\mathit{Source}] = 0$⟩

```
  show ?case
  proof ib-assert
    show SP-2[Source] = 0
    proof (cases
             G[v-1, Source] < Infinity ∧
             SP-1[v-1] + G[v-1, Source] < SP-1[Source])
      case True
      moreover with H3 have SP-1[v-1] + G[v-1, Source] < 0 by simp
      ultimately have SP-2[Source] < 0 using H1 by simp
oops
```

Here, it becomes obvious what exactly caused the error in Boogie/Z3 before. Besides the contradiction in the proof attempt, computed shortest paths are always non-negative in Dijkstra's algorithm. From this observation, we can infer an additional invariant for the **while** loop of the implementation:

```
invariant (∀ x: Vertex :: SP[x] >= 0);
```

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

4.2 Tracking Program Positions

Relating formulas to locations in the original program is one of the key aspects of HOL-Boogie; this feature results from exploiting the labeling mechanism of Boogie. Since assertions, subsuming also invariants and postconditions, form the crucial parts of verification conditions, they are tagged by labels holding their program position. After producing a verification condition and loading it in HOL-Boogie, the labels then occur at the formulas to be proven, in the way shown along the example of Dijkstra's algorithm before.

4.3 Specific Tactic Support

HOL-Boogie comes with a set of specific tactics to manipulate verification conditions. They allow the user to navigate to assertions, to prune some of them by applying Z3, and to restrict the list of premises associated with assertions. Some of these tactics are already shown in the verification of Dijkstra's algorithm.

Based on the structure of verification conditions generated by Boogie, the central tactic of HOL-Boogie, *ib-split*, 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 Z3 if the argument *try-z3* is given to the tactic *ib-split*. This essentially gives the "debugging flavor" to HOL-Boogie, since Z3 usually discharges all subgoals except those that are incorrect or inconsistent. The method is based on an oracle calling Z3; the communication uses the SMT-LIB format [19]. Due to this standardized format, it is possible to replace Z3 with other SMT solvers, or combine them for better results.

The list of premises of a subgoal can be pruned by the tactic *ib-filter-prems*. It selects all premises potentially necessary to solve the current subgoal, while cutting off all other premises. Note, however, that this tactic, due to its heuristics, may remove too many premises. Therefore, its purpose is only to assist in finding a draft of a proof for a subgoal, especially in the case of a long list of premises.

Finally, the tactic *ib-assert* serves to unwrap a formula of an assertion by cutting off the label.

4.4 Structured Proofs and Isabelle Proof Support

Without using Z3 from inside HOL-Boogie, many subgoals of a verification condition can already by proven by tools included in Isabelle. In simple academic experiments, the built-in simplifier is already able to solve some subgoals. A more substantial help, however, comes from *sledgehammer*. When applied to a subgoal, it uses external first-order provers to identify necessary facts which are then combined into a proof, usually by passing the facts to a resolution-based built-in prover. Since the amount of facts given as axioms in BoogiePL as well as the number of premises for an assertion can easily grow to an unmanageable size, sledgehammer is of an invaluable help. Usually, around 50% of all subgoals generated from a verification condition can be shown by this method.

5 Scenario II: Interactive Verification of C-Programs

Verifying C programs in HOL-Boogie seems to be a straightforward extension to the previous section. The C front-end of VCC compiles a C program like the following example (computing a maximal unsigned byte for an array whose size is bounded by 2^{40}):

```
#include "vcc.h"
  ...
static UINT8 maximum(__inout_ecount(len) UINT8 arr[], UINT64 len)
  requires(0 < len ∧ len < (1UI64 << 40))
  ensures (∀(UINT64 i; i<len ⟹ arr[i]≤result))
{
  UINT8 max;
  UINT64 p;

  max = 0;
  for (p = 0; p < len; p++)
    invariant(p ≤ len)
    invariant(∀(UINT64 i; i < p ⟹ arr[i] ≤ max))
  {
    if (arr[p] > max) { max = arr[p]; }
  }
  assert(p == len);
  return max;
}
```

into a BoogiePL-program. This 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 (∀ i: bv64 :: $_inrange.u8(i) ⟹ $clt.u8(i, p) ⟹
  $cle.u4($ld.u1($mem, $add.ptr(arr, i, 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);
```

The primitives of the CVM provide operations for:

1. **dereference, load and store in memory**: $ld.u1, $ld.u2, $ld.u4, $ld.u8, $st.u1, $st.u2, $st.u4, $st.u8,... The index indicates the length of the bitvector in bytes. These operations take the padding conventions of the little-endian x86 architecture into account.
2. **bitvector computations**: e.g. cle.u8, clt.u8, mul.u8, etc, ...
3. **pointer arithmetic**: e.g. $add.ptr, $sub.ptr, $base, $offset, ...
4. **memory regions (= pointer sets)**: e.g. $region, $contains, $overlap, ...
5. **memory operations**: e.g. malloc, free, memcopy, ...
6. **framing conditions**: $only_region_changed_or_new(X, mem, mem') expresses that memory mem in the state and memory mem' in its successor state remain unchanged for all pointers not in X, ...
7. **typed ghost memory**: $gmem and its infrastructure.

Ghost memory is a separate memory, which is updated in a way that does not affect the program control flow, where syntactic restrictions guarantee that information never flows from ghost states to concrete program states. Thus, ghost state and any code using it can be eliminated when the program is compiled. It is used in particular to specify the concept of a $valid reference or the $size of an array into which all references are $valid memory. Conceptually, it is a map from references to records with arbitrarily many fields with possibly different types.

When compiling the axioms referring to ghost state, a problem arises: while the typing discipline of BoogiePL is simply many-sorted first-order in most cases, there is a non-standard built-in type construct <x>T (used here for a type name that stands for *field names*) that requires special treatment. There are several axioms that quantify over ghost memory which has the BoogiePL array type [$gid,<x>name]x. We interpret this type by functions of type $gid \Rightarrow \alpha\ name \Rightarrow \alpha$ (where *gid* is the type of ghost references for which an injection from standard memory references exist). For each ghost field, such as $size, the axiomatization also defines a *field tag constant*:

```
const unique $_size : <bv64>name;
```

which we convert into a constant declaration $\$\_size :: bv64\ name$. Thus, so far, this concept can be safely embedded into Hindley-Milner style polymorphism. However, there are axioms with quantifications over *name* (intended to mean: "over all fields") such as in:

```
axiom(∀
r:$region,n:name,oldgmem:[$gid,<x>name]x,newgmem:[$gid,<x>name]x,
          oldmem: $memory, newmem:$memory ::
            . . . n . . .
```

We interpret a BoogiePL axiom of this form as an axiom scheme and create for each field tag constant an instance for it.

The compiled BoogiePL code of the above C program can still be loaded within 36 seconds (on standard hardware) into HOL-Boogie. Its proof is fairly straightforward but profits substantially from the tactic firing Z3.

6 Scenario III: Verification of Background Theories

At present, the axiomatization of the CVM —called Prelude Version 7.0— consists of about 750 axioms (where a certain number of axioms were not made explicit 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* memory/machine 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 [5].

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. 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 model is 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 were 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 were already expressed at the type level, for example:

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

Arithmetic operations for signed and unsigned integers were defined over *bv*32, 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* (omitted here) is defined on bitvectors directly representing the usual intuition "division by two". 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 $memory = \{x :: ref \Rightarrow Bitvector.\ True\}$
types $state = (ref \Rightarrow bool) \times memory$

The pivotal concept of a valid reference, for example, is defined as:

$$\begin{aligned}&\textbf{constdefs}\ valid :: [state,\ ptr,\ int] \Rightarrow bool\\ &\qquad valid\ \sigma\ p\ l \equiv (fst\ \sigma)\ (base\ p) \wedge 0 \leq (offset\ p) + 1 \wedge\\ &\qquad\qquad offset\ p * 8 < length\ (lkup\ (snd\ \sigma)\ (base\ p))\end{aligned}$$

Definitions for *malloc* and *free* are straightforward.

We implemented a little 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.

7 Conclusion

We have presented a novel HOL-based proof environment, called HOL-Boogie, that is integrated into a verification tool chain for imperative programs, in particular C (and C#; not supported yet). Key issues of the 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. a mechanism to generate background theories from consistent, conservative models in HOL,
4. the integration of the target prover in order to discharge as many subgoals as possible, and
5. mechanisms to track attributes in order to exchange meta-information between tools.

7.1 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* [9], Boogie by *Why* [8], and Z3 by the default prover ERGO, and one gets (nearly) the architecture of the Why/Caduceus system [10]. However, its interactive prover-back-end cannot be used to decompose verification conditions and send the "splinters" to the target prover (ERGO is not integrated into Coq), it offers no mechanism for tracking back unsatisfiable subgoals to the source, and it offers little specific tactic support for this application scenario. With respect to the C front-end and the underlying CVM, VCC is more detailed since it leverages features such as byte-wise access into unions. Moreover, support for fine-grain concurrency is actively under development [5].

There is a quite substantial body on programming language embeddings into HOL, be it *shallow* [20,18] or *deep* [12]. In particular, Leinenbach [12] provides a small-step semantics for a language C0, which has been used for system level verification, and Schirmer [20] derives a (shallow-ish) Hoare-Logic from this semantics and formally developed a verification condition generator. C0 assumes a typed memory model (although bitvectors and conversions to standard datatypes could easily be integrated). However, the size of the supported language fragment, many complications in the semantic representation, and the degree of automatic proof support have limited its use in case studies substantially. In contrast to VCC and the Hypervisor Verification Project [5], the idea is to adapt the code to be verified instead of trying to live with the existing code and adapt the tool chain.

7.2 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 of the source in this scenario.
2. **Verified Current CVM Model**. 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 consistency on a more recent model. From our experience, this is a substantial task (several man-months), but routine.
3. **More Automated Proof Support in Isabelle**. Currently, there is not enough automated proof support for bitvectors and for the logical reasoning required to discharge formulas related to memory-framing and updates in the C model.

References

1. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
2. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: PASTE 2005, pp. 82–87. ACM Press, New York (2005)
3. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
4. Basin, D., Kuruma, H., Miyazaki, K., Takaragi, K., Wolff, B.: Verifying a signature architecture: A comparative case study. Formal Aspects of Computing 19(1), 63–91 (2007)
5. Cohen, E., Hillebrand, M., Leinenbach, D., der Rieden, T.I., Moskal, M., Paul, W., Santen, T., Schirmer, N., Schulte, W., Tobies, S., Wolff, B.: The Microsoft Hypervisor Verification Project (manuscript in preparation) (2008)
6. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
7. DeLine, R., Leino, K.R.M.: BoogiePL: A typed procedural language for checking object-oriented programs. Tech. Rep. 2005-70, Microsoft Research (2005)
8. Filliâtre, J.-C.: Why: A multi-language multi-prover verification condition generator. Tech. Rep. 1366, LRI, Université Paris Sud (2003)
9. Filliâtre, J.-C., Marché, C.: Multi-prover verification of C programs. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 15–29. Springer, Heidelberg (2004)
10. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
11. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI 2002, pp. 234–245. ACM Press, New York (2002)
12. Leinenbach, D., Paul, W., Petrova, E.: Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In: SEFM 2005, pp. 2–12. IEEE Computer Society Press, Los Alamitos (2005)
13. Leino, K.R.M., Millstein, T., Saxe, J.B.: Generating error traces from verification-condition counterexamples. Science of Computer Programming 55(1-3), 209–226 (2005)
14. Leino, K.R.M., Saxe, J.B., Stata, R.: Checking Java programs via guarded commands. In: FTfJP 1999, Tech. Rep. 251. Fernuniversität Hagen (1999)
15. Morgan, C.: The specification statement. ACM TOPLAS 10(3), 403–419 (1988)
16. Nelson, G.: A generalization of Dijkstra's calculus. ACM TOPLAS 11(4), 517–561 (1989)
17. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
18. Norrish, M.: C formalised in HOL. Ph.D. thesis, Computer Laboratory, University of Cambridge (1998)
19. Ranise, S., Tinelli, C.: The SMT-LIB standard: Version 1.2. Tech. rep., Dept. of Comp. Sci., The University of Iowa (2006), `http://www.smt-lib.org`

20. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. Ph.D. thesis, Technische Universität München (2006)
21. Schulte, W., Xia, S., Smans, J., Piessens, F.: A glimpse of a verifying C compiler (extended abstract). In: C/C++ Verification Workshop (2007)
22. Wenzel, M., Wolff, B.: Building Formal Method Tools in the Isabelle/Isar Framework. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 351–366. Springer, Heidelberg (2007)