# Boogie: A Modular Reusable Verifier for Object-Oriented Programs

Mike Barnett[0], Bor-Yuh Evan Chang[1], Robert DeLine[0], Bart Jacobs[2], and K. Rustan M. Leino[0]

[0] Microsoft Research, Redmond, Washington, USA
{mbarnett,rdeline,leino}@microsoft.com

[1] University of California, Berkeley, California, USA
bec@cs.berkeley.edu

[2] Katholieke Universiteit Leuven, Belgium
bartj@cs.kuleuven.be

**Abstract.** A program verifier is a complex system that uses compiler technology, program semantics, property inference, verification-condition generation, automatic decision procedures, and a user interface. This paper describes the architecture of a state-of-the-art program verifier for object-oriented programs.

## 0 Introduction

A program verifier is built from a number of complex pieces of technology: a source programming language, its usage rules and formal semantics, a logical encoding suitable for automatic reasoning, abstract domains for program analysis and property inference, decision procedures for discharging proof obligations, and a user interface that lets a user understand the results of the verification process. Dealing with these complexities, like other software engineering problems, requires a modular architecture with well established interface boundaries.

In this paper, we describe the architecture of Boogie, a state-of-the-art program verifier for verifying Spec# programs in the object-oriented .NET framework. Internally, Boogie is structured as a pipeline performing a series of transformations from the source program to a verification condition (VC) to an error report (see Fig. 0). The novel aspects of the Boogie architecture include the following:

0. *Design-Time Feedback.* Boogie (together with the Spec# compiler) is integrated with Microsoft Visual Studio to provide design-time feedback in the form of red underlinings that highlight not only syntax and typing errors but also semantic errors like precondition violations.
1. *Distinct Proof Obligation Generation and Verification Phases.* The Boogie pipeline is centered around intermediate representations in BoogiePL [DL05], a language tailored for expressing proof obligations and assumptions (Sec. 3). BoogiePL serves a critical role in separating the generation of proof obligations from the semantic encoding of the source program and the proving

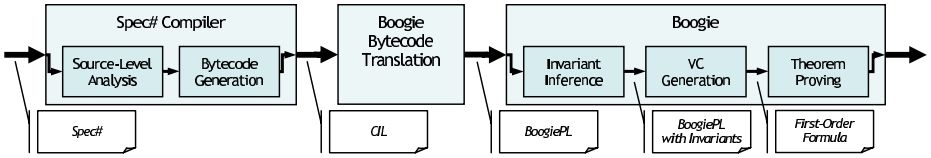



**Fig. 0.** The Boogie pipeline.

of those obligations. This separation has been critical in the simultaneous development of the object-oriented program verification methodology and the core verification technology.

2. *Abstract Interpretation and Verification Condition Generation.* Boogie performs loop-invariant inference using abstract interpretation (Sec. 5) and generates verification conditions to be passed (Sec. 6) to an automatic theorem prover. This combination allows Boogie to utilize both the precision of verification condition generation (that must necessarily be lost in an abstraction) and the inductive invariant inference of abstract interpretation (that simply cannot be obtained with a concrete model).

## 1 Overview

In this introductory section, we give an overview of Boogie's architecture. The rest of the paper provides more details of each architectural component.

**Source Language.** The Spec# language is a superset of C#, adding specification features (i.e., *contracts*) such as pre- and postconditions and object invariants [BLS04]. Spec# prescribes static type checks beyond those prescribed by C# and introduces dynamic checks for the specified contracts. The compiler performs the static type checking and emits the dynamic checks as part of the target code. Boogie makes use of the type properties enforced by the compiler and attempts statically to prove that the dynamic checks will always succeed. Boogie thus checks for error conditions defined by the virtual machine, such as array bounds errors and type cast errors, and error conditions specified by user supplied contracts, such as precondition violations. To ensure soundness of the verification, Boogie additionally checks for error conditions defined by the programming methodology [BDF+04,LM04,BN04,LM05,LM06].

As depicted in Fig. 0, Spec# programs are compiled into CIL, the executable format of the .NET virtual machine. Boogie starts with an abstract syntax tree (AST) for this CIL, which it either gets directly from the compiler or reconstructs from reading a compiled `.dll` or `.exe` file. The latter is the more conventional mode of a program verifier and allows batch processing. The former allows Boogie to run as part of compilation, which enables a clean integration with Microsoft Visual Studio to provide design time feedback. This feedback shows up as red underlinings (fondly known as "red squigglies") in the program text, and the

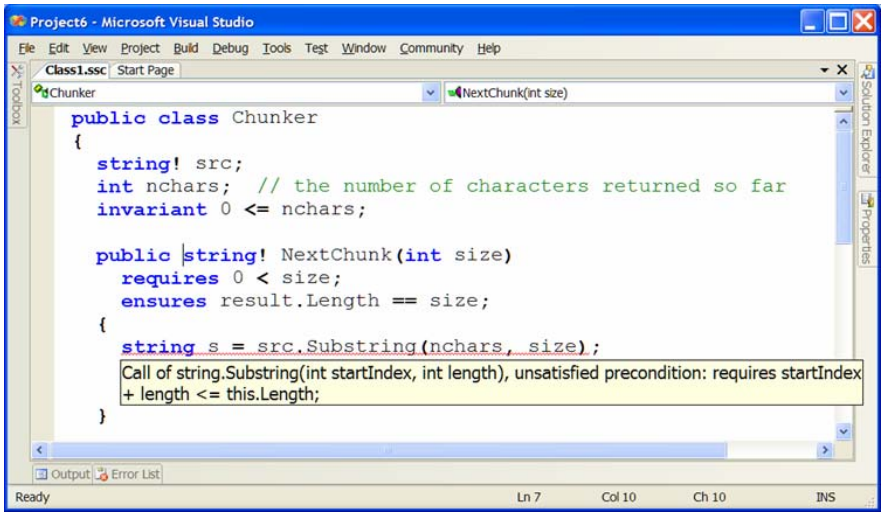



**Fig. 1.** Design time feedback of verification errors within Microsoft Visual Studio 2005. The red squiggly under the call to *Substring* indicates an error. The hover text shows the error to be a precondition violation.

user can get further information by rolling the cursor over these underlinings, which brings up some hover text that explains the problem (as shown in Fig. 1). To our knowledge, Boogie is the first program verifier to provide such interactive design-time feedback.

**Intermediate Language.** The generation of verification conditions from source code involves a great number of verifier design decisions. By staging this process by first translating CIL into BoogiePL, the Boogie architecture separates the concerns of deciding how to encode source language features and their usage rules from the concerns of how to reason about control flow in the program. BoogiePL provides **assert** statements that encode proof obligations stemming from the source program, to be checked by the program verifier, and **assume** statements that encode properties guaranteed by the source language and verification process, available for use in the proof by the program verifier. The architectural layering of verification condition generation via an intermediate program notation was used by ESC/Modula-3 [DLNS98] (cf. [Lei95]), which made use of *guarded commands* whose semantics is given by *weakest preconditions* [Dij76]. This architecture was sharpened by ESC/Java [FLL+02], which defined and staged the translation further [LSS99].

Verification condition generation involves not just the executable program statements in the source language, but also other declarations of the source program and properties guaranteed by the source language. In the aforementioned ESC tools, the logical encoding of these additional properties, called the *background predicate*, was produced separately from the intermediate program

notation and fed directly to the theorem prover. BoogiePL innovates further by allowing the background predicate to be encoded as part of the intermediate program. That is, BoogiePL includes declarations for mathematical functions and axioms. Consequently, the translation of CIL culminates in a BoogiePL program that encodes the entire proof task—in fact, properties of Spec# and the source program are no longer used after this point in Boogie's pipeline, except when mapping errors back to line numbers in the source text.

Like other languages, BoogiePL can be printed as and parsed from a textual representation. This feature has been the most important vehicle in debugging and experimenting with our semantic encoding of Spec#. For example, it is often convenient to manually perform small changes to the BoogiePL program without having to modify the Spec# compiler and/or the bytecode translator.

This strong interface boundary between the bytecode translation and the rest of Boogie's pipeline also makes it possible for other program verifiers to reuse Boogie's VC generation, simply by encoding their proof obligations as BoogiePL programs. As such, BoogiePL can also be viewed as a high-level front-end to a theorem prover.

**Inferred Properties.** BoogiePL programs are turned into first-order verification conditions, a process which requires loop invariants. While these can come from the source programs, many loop invariants can be "boring" or "obvious" to the programmer, in which case the task of manually supplying these is onerous and having them as part of the source text provides more clutter than insight. Sometimes, the loop invariants are even impossible to express in the source language, as is the case when the invariant needs to refer to variables or functions of the BoogiePL encoding of the source program. Therefore, Boogie includes a framework for abstract interpretation [CC77], which can infer loop invariants of the BoogiePL program. These inferred invariants are inserted as **assume** statements into the loop heads of the BoogiePL program, so that they can be assumed by the VC to hold at the start of each loop iteration.

To modularly combine abstract domains and to support the use of object references that dereference the heap in the source program, Boogie innovates by connecting its abstract domains to a special abstract domain that, essentially, symbolically names locations in the heap, names that then are used by the other abstract domains [CL05].

**Verification Conditions.** After generating loop invariants, Boogie generates verification conditions from the resulting BoogiePL program. There are many logically equivalent ways of expressing the verification conditions, and which way is chosen can have a dramatic impact on the performance of the underlying theorem prover. Boogie performs a series of transformations on the program, essentially producing one snippet of the verification condition from each basic block of the BoogiePL procedure implementation being verified [BL05]. The verification condition is represented as a formula in first-order logic and arithmetic.

```
public class Example {
  int x;
  string! s;
  invariant s.Length >= 12;
  public Example(int y) requires y > 0; { ... }
  public static void M(int n) {
    Example e = new Example(100/n);
    int k = e.s.Length;
    for (int i = 0; i < n; i++) { e.x += i; }
    assert k == e.s.Length;
  }
}
```

**Fig. 2.** An example Spec# class. Its BoogiePL translation is shown in Fig. 3.

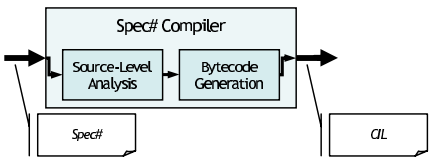
It is then passed to a first-order automated theorem prover to determine the validity of the verification condition (and thus the correctness of the program).

The verification conditions are encoded in such a way as to make it possible to reconstruct from a failed proof an error trace (i.e., an execution path through the procedure leading to a proof obligation that the theorem prover is unable to establish [LMS05]). The bytecode translator performs enough bookkeeping to map the BoogiePL error trace back into a Spec# error trace, much like a compiler performs enough bookkeeping for a source-level debugger to operate from compiled code. Typically, a failed proof indicates an error in the program or some missing condition in a contract, but due to incompleteness in the theorem prover, there is also the possibility of spurious error reports. If the theorem prover runs out of some limited resource, such as the allotted time or space, that event is reported.

**Theorem Prover.** Boogie can generate verification conditions for the off-the-shelf theorem prover Simplify [DNS05], as well as for Zap [BLM05], a set of decision procedures developed at Microsoft Research. At the moment, most of our Boogie experience has been with Simplify, but we expect to shift our use toward Zap. The Boogie architecture makes it fairly easy to retarget the final step of the VC generation to a new theorem prover.

## 2 Spec#

Figure 2 shows a synthetic example program to highlight some of the features of Spec#; a more detailed introduction is found in the Spec# overview paper [BLS04]. The example shows one class, called *Example*, which contains



two fields, an object invariant, a constructor, and a method. The body of the method allocates a new *Example* object. The actual argument to the constructor, $100/n$, contains a potential division-by-zero error. The loop repeatedly increments the $x$ field of the newly allocated object by various amounts. The code also saves the length of the string $e.s$ and later checks, using an assert statement, that it is unchanged by the loop.

Spec# incorporates a non-null type system [FL03]; the type `string!` means the field $s$ can never hold the value `null`. We have found this to be the most common specification in object-oriented programming.

The Spec# compiler generates standard .NET assemblies. A .NET assembly contains bytecode in the form of method bodies within type definitions and *meta-data* for describing extra-runtime features of the types and their members. The meta-data format allows *custom attributes*, which are arbitrary user-defined data that we use to encode specifications. Due to the limitations of the meta-data format, we persist specifications as *serialized ASTs*. We use the same meta-data format to store *out-of-band* specifications. These are Spec# specifications for types and methods that are already defined in third-party assemblies. For instance, the Spec# distribution [Spe06] provides out-of-band contracts for the two most central assemblies in the .NET Base Class Library (BCL), `mscorlib.dll` and `System.dll`. Both the Spec# compiler and Boogie have the ability to weave together an assembly and its out-of-band specification so that it appears as if the contracts were natively present in the original assembly. One of the key benefits is that client code, which generally is heavily dependent on the BCL, receives warnings/errors related to incorrect usage of the library APIs. This feature has been critical in obtaining a usable development system with contracts.

## 3 BoogiePL

BoogiePL [DL05] is an effective intermediate language for verification condition generation of object-oriented programs because it lacks the complexities of a full-featured object-oriented programming language, while also introducing features of the target logic. As a result, it distributes the complexity of verification condition generation over two well-defined phases, each of which is significantly less complex than the whole. Compared with Spec#, BoogiePL retains the following features: procedures (but not methods), mutable variables, and pre- and postconditions. On the other hand, it lacks the following complications: expressions with side effects, a heap with objects, classes and interfaces, call-by-reference parameter passing, and structured control-flow. It introduces the following features for modeling: constants, function symbols, axioms, non-deterministic control-flow, and the notion of "going wrong".

Figure 3 shows the translation of the Spec# example given in Fig. 2. While we give details on how this translation is obtained in Sec. 4, we observe some salient features of BoogiePL here. BoogiePL looks somewhat like a high-level assembly language in that the control-flow is unstructured but the notions of statically-scoped locals and procedural abstraction are retained; however, in-

traprocedural control-flow is given by a non-deterministic **goto**. Also, observe that the heap has been made explicit with the global variable *Heap* and similarly the implicit receiver object of the method is now an explicit parameter *this*.

A BoogiePL *program* consists of a *theory* that is used to encode the semantics of the source language, followed by an *imperative part*. We show the abstract syntax for BoogiePL; for punctuation and other concrete details, see [DL05].

$$\mathit{program} ::= \mathit{typedecl}^* \ \mathit{symboldecl}^* \ \mathit{axiom}^* \ \mathit{vardeclstmt}^* \ \mathit{proc}^* \ \mathit{impl}^*$$

We use the meta-level symbols $^*, ^+, ^?$ to indicate a sequence, a nonempty sequence, and an optional syntactic entity, respectively, use | for alternatives, and use $\langle\cdot\rangle$ for grouping.

A theory consists of *type declarations*, *symbol declarations*, and *axioms*.

*typedecl* ::= **type** *typename* ;
*symboldecl* ::= *constdecl* | *functiondecl*
*constdecl* ::= **const** *var* : *type* ;
*type* ::= **bool** | **int** | **ref** | **name** | **any** | *typename* | *arraytype*
*arraytype* ::= [ *type* , *type* ] *type*
*functiondecl* ::= **function** *function* ( *type*$^*$ ) **returns** ( *type* ) ;
*axiom* ::= **axiom** *expr* ;

BoogiePL has types and the type checker enforces that every expression is properly typed. However, all type information is erased during the translation into verification conditions. The reason for having the types is to improve readability by expressing intent and to catch simple errors. However, any expression may be cast to type **any** and thence to any other type; just as types are erased in verification conditions, so are casts. In addition to built-in types like **bool** and **int**, BoogiePL supports user-defined types (*typename*) and arrays (*arraytype*). The types used to index into arrays can be any types, not just integers (we might therefore have called arrays maps). For brevity, we show only 2-dimensional arrays.

BoogiePL's expressions include boolean, reference, and integer literals and arithmetic and first-order logical operators:

*expr* ::= *literal* | *var* | *unop expr* | *expr binop expr* | *expr* [ *expr* , *expr* ]
  | *funapp* | *quant* | **cast** ( *expr* , *type* ) | **old** ( *expr* )
*literal* ::= **false** | **true** | **null** | *integer*
*binop* ::= $\Leftrightarrow$ | $\Rightarrow$ | $\vee$ | $\wedge$ | $<:$ | $\leqslant$ | $<$ | $\neq$ | $=$ | $+$ | $-$ | $*$ | $/$ | $\%$
*unop* ::= $-$ | $\neg$
*funapp* ::= *function* ( *expr*$^*$ )
*quant* ::= ($\forall$ *vardecl*$^*$ *trigger*$^*$ $\bullet$ *expr* ) | ($\exists$ *vardecl*$^*$ *trigger*$^*$ $\bullet$ *expr* )
*vardecl* ::= *var* : *type* $\langle$**where** *expr*$\rangle^?$
*trigger* ::= { *expr*$^+$ }

For use in procedure postconditions and implementations, the expression **old**($E$) refers to the value of $E$ in the procedure's pre-state. The **where** clause in a

```
const System.Object : name;
const Example : name;
axiom Example <: System.Object;
function typeof(obj : ref) returns (class : name);

const allocated : name;
const Example.x : name;
const Example.s : name;

var Heap : [ref, name]any;

function StringLength(s : ref) returns (len : int);

procedure Example..ctor(this : ref, y : int);
  requires ... ∧ y > 0;   modifies Heap;   ensures ...;

procedure Example.M(n : int);
  requires ...;   modifies Heap;   ensures ...;

implementation Example.M(n : int)
{
  var e : ref where e = null ∨ typeof(e) <: Example;
  var k : int, i : int, tmp : int, PreLoopHeap : [ref, name]any;

  Start :
    assert n ≠ 0;
    tmp := 100/n;
    havoc e;
    assume e ≠ null ∧ typeof(e) = Example ∧ Heap[e, allocated] = false;
    Heap[e, allocated] := true;
    call Example..ctor(e, tmp);

    assert e ≠ null;   k := StringLength(cast(Heap[e, Example.s], ref));
    i := 0;
    PreLoopHeap := Heap;
    goto LoopHead;

  LoopHead :
    goto LoopBody, AfterLoop :

  LoopBody :
    assume i < n;
    assert e ≠ null;
    Heap[e, Example.x] := cast(Heap[e, Example.x], int) + i;
    i := i + 1;
    goto LoopHead;

  AfterLoop :
    assume ¬(i < n);
    assert e ≠ null;   assert k = StringLength(cast(Heap[e, Example.s], ref));
    return;
}
```

**Fig. 3.** A simplified version of the BoogiePL resulting from translation of the *Example* class in Fig. 2. The *Length* property of strings is translated specially as a BoogiePL function. The local variable *PreLoopHeap* , which stores a copy of the entire heap, is later used by the invariant inference.

variable declaration postulates a unary constraint on the variable's value (like a type qualifier). Triggers are for use by the underlying theorem prover in deciding how to instantiate universal quantifiers [DNS05].

The imperative part of a BoogiePL program consists of global variable declarations, procedure headers (*procedures*), and procedure implementations (*implementations*).

*vardeclstmt* ::= **var** *vardecl*$^*$ ;
*proc* ::= **procedure** *procname* ( *vardecl*$^*$ ) ⟨**returns** ( *vardecl*$^*$ )⟩$^?$
  ⟨**free**$^?$ **requires** *expr* ;⟩$^*$ ⟨**modifies** *var*$^*$ ;⟩$^*$ ⟨**free**$^?$ **ensures** *expr* ;⟩$^*$
  *implbody*$^?$
*impl* ::= **implementation** *procname* ( *vardecl*$^*$ ) ⟨**returns** ( *vardecl*$^*$ )⟩$^?$
  *implbody*
*block* ::= *label* : *cmd*$^*$ *transfercmd*
*implbody* ::= { *vardeclstmt block*$^+$ }

As a syntactic sugar, a procedure header can have an optional implementation body, which has the same effect as an implementation declaration with the same name. While many languages have named out-parameters and an anonymous return value, BoogiePL simply allows multiple return values; they are all named as out-parameters in the **returns** clause.

An implementation body consists of a sequence of local variable declarations, followed by a sequence of *blocks*. An implementation starts at the block listed first, in a state where the procedure's preconditions hold and where global and local variables and in-parameters have values that satisfy their respective **where** clauses.

A block has a label and a sequence of commands, followed by a control transfer command.

*cmd* ::= *passive* | *assign* | *call*
*passive* ::= **assert** *expr* ; | **assume** *expr* ;
*assign* ::= *var* ⟨[*expr* , *expr* ]⟩$^?$ := *expr* ; | **havoc** *var*$^+$ ;
*call* ::= **call** *var*$^*$ := *procname* ( *expr*$^*$ ) ;
*transfercmd* ::= **goto** *label*$^+$ ; | **return** ;

The **assert** and **assume** commands indicate conditions to be checked or used, respectively, in the verification. If the given expression evaluates to **true**, then each of these commands proceeds like a no-op. If the condition evaluates to **false**, the assert command *goes wrong*, which is a terminal failure. For the assume command, if the condition evaluates to **false**, one is freed of all subsequent proof obligations, thus indicating a terminal success. The assume command, which is known as a *partial command* [Nel89] or *miracle* (cf. [BvW98]), is a crucial ingredient when encoding verification problems as programs (cf. [Lei95]). The **havoc** command assigns an arbitrary value to each indicated variable; when present, the variable's **where** clause constrains this value. The **goto** command jumps nondeterministically to one of the indicated blocks. The **return** command ends the

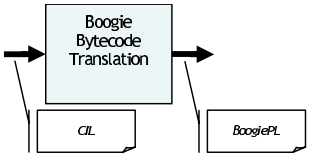
implementation. It goes wrong if the procedure's non-**free** postconditions are not satisfied; otherwise, the procedure implementation terminates successfully.

The **call** command is defined in terms of the specification of the procedure being called. It goes wrong if the procedure's non-**free** preconditions do not hold. Otherwise, the state after the **call** command satisfies the procedure's postconditions and the **where** clauses of the procedure's out-parameters. Specifically, it is not assumed that the procedure implementation that gets executed is one of the implementations declared in the program.

Free pre- and postconditions, like **where** clauses, are used for encoding properties guaranteed by the source language. For the most part, these features are just for convenience, because they encapsulate the many assumptions that would otherwise have to be sprinkled in many places. However, in the VC generation for loops (see Sec. 6), **where** clauses are essential for maintaining enough information about loop targets.

## 4 Translating CIL to BoogiePL Programs

In this section, we describe some key issues and design choices for the bytecode translator. These issues include encoding the heap, allocation, and fields, axiomatizing the Spec# type system, translating call-by-reference parameters, translating methods and method calls, and generating frame conditions.




The bytecode translator first transforms a method body, consisting of CIL instructions, into a *normalized AST*, which is essentially an enriched object model that augments CIL with contract features: non-null type annotations, **assert** and **assume** statements, loop invariants, method contracts, object invariants, and various custom attributes for the programming methodology. In a normalized AST, as in CIL, a method body contains no structured programming constructs such as **if** statements or **while** statements. Rather, a method body contains a sequence of labeled blocks, each of which contains a sequence of statements, some of which may be conditional or unconditional branch statements that specify the label of the target block.

One of the differences between a normalized AST and BoogiePL is that in the former, a statement may contain expressions that may have side effects and that may go wrong (such as method calls). For this reason, Boogie transforms the normalized AST into a *flattened AST*, where the values of expressions are assigned to evaluation stack slots and only evaluation stack slots appear as operands of expressions and statements. A flattened AST is suitable for walking the sequence of statements and generating BoogiePL commands based on the kind of statement, though we do need a dataflow analysis to provide some flow-sensitive contextual information necessary for the translation of some statements.

In addition to inferring the CIL type of each local variable and evaluation stack slot, the analysis attempts to track the following information:

- managed pointers (i.e., reference parameters or arguments for reference parameters or the pointers used when dealing with structs)
- method pointers (which appear when creating a delegate instance)
- type tokens and *System.Type* objects (which appear when reflecting over types)
- booleans (for which CIL code uses integers)

**Encoding the Heap, Allocation, and Fields.** Recall that BoogiePL has no built-in notion of a heap, object allocation, or fields. The translation models the heap as a BoogiePL global two-dimensional array, named *Heap*, that maps an object reference $o$ and a field name $f$ to the current value of $o.f$ (as shown in Fig. 3) [Bur72]. Field names encountered during translation are emitted as unique constants, whose names are qualified by the name of the declaring class.

We use a 2-dimensional heap [PH97] rather than one 1-dimensional "heap" per field (cf. [DLNS98,Lei95]), because the encoding of our modular verification methodology quantifies over field names (see frame conditions below).

Object allocation is modeled by adding an extra boolean field called allocated to each object, which indicates whether the object has been allocated. Allocating an object consists of choosing an object that has not yet been allocated and setting its allocated bit, see Fig. 3 (cf. [HW73,DLNS98,Lei95]). In managed code like Spec#, all objects reachable from the program are allocated. This property is important for proving that newly allocated objects are distinct from previously allocated objects, which is crucial for reasoning about object state updates. Allocatedness information is emitted in the form of procedure preconditions, frame conditions, and loop invariants, as well as axioms that state that objects reachable from allocated objects are allocated (cf. [LN02]). This statement is complicated somewhat by the fact that a path between two objects may pass through one or more `structs`.

Static fields are stored in the heap, just like instance fields. In particular, fields are translated as follows:

| | | |
|---|---|---|
| $o.f$ | translates to | $Heap[o, \mathsf{C\_f}]$ |
| $C.g$ | translates to | $Heap[\mathsf{TypeObject}(C), \mathsf{C\_g}]$ |

where fields $f$ and $g$ are declared by class $C$ and $o$ is an expression of type $C$. A major advantage of storing static fields in the heap as opposed to, for example, separate global variables, is that one frame condition can govern both static and instance fields. It also allows a more uniform treatment of both kinds of fields by the modular verification methodology.

**Axiomatizing the Spec# Type System.** In order to model the semantics of Spec# type tests and typecasts, an axiomatization of the subtype relation on reference types is required. This axiomatization additionally helps in deriving object distinctness results, which reduces the number of inequalities that users need to include in method contracts, object invariants, loop invariants, etc.

It turns out to be a challenge to author a type axiomatization that is queried efficiently by our theorem prover, so we are considering adding a decision procedure specifically for this purpose. Unfortunately, this choice would probably require that BoogiePL be made aware of the Spec# type system.

**Translating Call-By-Reference Parameters.** Both C# and Spec# support call-by-reference argument passing (reference parameters are marked `ref`). A call-by-reference parameter of type $T$ takes as an argument not a value of type $T$ but a pointer to a variable of type $T$. Accesses to the parameter are thus dereferences of the pointer.

BoogiePL does not support call-by-reference parameters directly. Since it does support both in- and out-parameters, we model reference parameters by performing copy-in/copy-out for the purposes of verification. In the Spec# program, when a variable $x$ is passed as an argument to a reference parameter $p$, then in the BoogiePL program the value of $x$ is passed as an argument to an in-parameter. At the start of the body of the callee, the in-parameter is copied into a local variable. Accesses to $p$ in the Spec# program are translated into accesses of the local variable in the BoogiePL program. When the procedure completes, the local variable is copied into an out-parameter and at the call site the out-parameter is copied back into $x$.

Translating from IL introduces a snag: accesses to reference parameters appear as pointer dereferences, and the pointers being dereferenced are read from the parameter in some preceding instruction. As a result, it is impossible to tell by looking at the instruction itself which reference parameter is being referred to. This problem is solved by the bytecode translator's dataflow analysis mentioned above.

Using copy-in/copy-out is sound only if there is no aliasing amongst the actual arguments for reference parameters. Therefore, we disallow such aliasing in Spec#. (This is not yet implemented in the compiler, but we intend to impose enough restrictions on actual arguments that a simple syntactic check suffices to forbid such aliasing, cf. [Rey78].)

**Translating Methods and Method Calls.** For each declaration of a method or method override, the bytecode translation generates a BoogiePL procedure. For each method implementation, it also generates a BoogiePL implementation. Having a separate procedure per override permits specification refinement in subclasses.

For translating method calls, we distinguish two cases. When we can determine the exact target of a call (that is, the call is statically bound, such as for a non-virtual method or a base call), it is translated into a call to the associated procedure. When the call is dynamically bound, we translate the call into a call of an additional BoogiePL procedure that we generate for virtual methods. This gives us the flexibility to use a slightly different specification for such calls, as used by our methodology [BDF+04].

**Method Framing.** In BoogiePL, the effect of a procedure is framed by its **modifies** clause. Specifically, a procedure may assign to a global variable only if the variable appears in the procedure's **modifies** clause. As mentioned above, the Spec# heap is modeled in BoogiePL as a global variable (observe the **modifies** clauses in Fig. 3 for *Heap*). Since almost all methods may potentially modify the heap (by creating a new object or assigning to a field), the heap appears in almost every procedure's **modifies** clause. However, this is clearly an overapproximation. Therefore, additional framing information is encoded in the BoogiePL program in the form of an extra postcondition on the procedure. This postcondition is known as the *frame condition*. The precise form of the frame condition depends on the modular verification methodology used; for example, for the original Boogie methodology [BDF+04], each procedure gets a frame condition of the following form:

$$
\begin{array}{l}
(\forall o\colon \mathbf{ref}, f\colon \mathbf{name} \bullet \\
\qquad (o, f) \notin W \ \wedge\ \mathbf{old}(\mathit{Heap}[o, \mathsf{allocated}] \wedge \neg \mathit{Heap}[o, \mathsf{committed}]) \\
\qquad \Rightarrow\ \mathit{Heap}[o, f] = \mathbf{old}(\mathit{Heap}[o, f]))
\end{array}
$$

where $W$ are the locations listed in the Spec# method's `modifies` clause, and $\mathit{Heap}[o, \mathsf{committed}]$ is a special field introduced by the modular verification methodology related to an ownership model [BDF+04]. Essentially, the frame condition says that unless a given location satisfies certain criteria, it is guaranteed not to incur a net modification by the method call.

The frame conditions generated by Boogie's bytecode translation are more complicated than the one we have shown here, but we lack the space to describe them in further detail.

**Loop Framing.** In order to generate verification conditions [BL05] for a loop, such as the following:

$$\mathit{LoopHead}\colon \mathbf{assert}\ I;\quad S;\quad \mathbf{goto}\ \mathit{LoopHead};$$

it must be transformed into acyclic control flow that abstracts the behaviors of the loop (for soundness). Specifically, we transform the loop above into the following sequence of statements:

$$
\begin{array}{l}
x_1^0 := x_1;\ \ldots\ x_n^0 := x_n;\quad \mathbf{assert}\ I; \\
\mathbf{havoc}\ x_1, \ldots, x_n;\quad \mathbf{assume}\ I; \\
S;\quad \mathbf{assert}\ I;\quad \mathbf{assume\ false};
\end{array}
$$

where $S$ is the loop body (which may include commands that jump out of the loop) , $x_1, \ldots, x_n$ are the variables (global or local) updated by $S$, and $x_1^0, \ldots, x_n^0$ are fresh local variables. The predicate $I$ serves as a loop invariant.

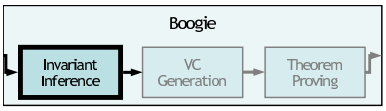
The transformation causes the loop body (as well as code paths that exit the loop) to be verified in all possible states that satisfy the loop invariant. The **assume false**; command indicates that a code path that does not exit the loop

can be considered to reach terminal success at the end of the loop body, provided that the loop invariant has been re-established.

When the loop body updates the heap (which is the typical situation), the heap is **havoc**ed (i.e., assigned an arbitrary value) on entry to a loop. This abstraction results in a sound but gross overapproximation of the set of heap locations that may be modified by loop body executions. Some of the lost precision must necessarily be recovered by inference (see Sec. 5), but we can also emit *loop frame conditions* that increase the precision of the verification and are guaranteed by the verification methodology. For example, one loop frame condition that is always added states that all objects that were allocated on entry to the loop are still allocated at the start of the current iteration:

$$(\forall o\colon \mathbf{ref} \bullet \mathit{Heap}^0[o, \mathit{allocated}] \Rightarrow \mathit{Heap}[o, \mathit{allocated}])$$

## 5 Invariant Inference




Compared with other static analysis techniques, verification-condition generation offers a high degree of precision. However, a well-known issue with producing first-order verification conditions is the need for loop invariants. Loop invariants may be specified by the user in BoogiePL or Spec#, but while some loop invariants are key ideas in the verification of a program and thus useful to incorporate in the source, others—in particular, those that state which heap locations are left untouched by the loop body—are often boring or unintuitive to the programmer. To mitigate the need for user-supplied loop invariants, we use *abstract interpretation* [CC77], which systematically computes overapproximations of sets of reachable states, to infer some loop invariants before generating the verification condition.

Currently, the interaction between the abstract interpretation and the theorem proving is exceedingly simple: the abstract interpreter instruments the input BoogiePL program with the invariants it can infer using the selected abstract domains and passes the instrumented BoogiePL program to the verification-condition generator to produce a formula for the theorem prover. Particularly, there is no feedback from the theorem prover to the abstract interpreter, though we have some evidence that such an interaction may be beneficial [LL05].

In this section, we first discuss the design goals for our abstract interpretation framework used in Boogie and then show how these goals are achieved. Finally, we sketch the kinds of invariants that can be obtained and are particularly important for Boogie to infer.

**A Generic Abstract Interpretation Framework.** Because we would like to use the abstract interpretation in different settings and with varying configurations (e.g., to trade-off precision for efficiency), we heavily modularize the abstract interpretation framework. In particular, when we design the abstract

domains, which capture the kind of invariants we can infer, we want to ignore the following concerns:

0. *Exploration Strategies.* We want to separate out the efficiency concerns in how the (abstract) state space is explored. As is standard, we use a generic fixed point engine.
1. *The Abstract Transition Relation.* The abstract transition relation defines how program statements affect abstract states; that is, given an abstract state at the program point before a given statement, what are the abstract successor states that conservatively approximate the effect of the statement. We want to be agnostic to the input language when designing the abstract domains, and we want to be able to define different abstract transition relations over the same abstract domains easily (e.g., for both intra- and interprocedural analyses). To achieve this goal, we fix some generic operators on abstract domains that can be easily combined to define various abstract transition relations.
2. *Combining Abstract Domains.* It is well-known that a combined analysis can be more precise than the separate sub-analyses working independently. We would like to design and implement logically separate abstract domains independently but obtain the precision of the combined analysis easily. In other words, we want an easy way to construct *reduced product* analyses [CC79].

**Expressions.** For goals 1 and 2 above, we fix a common language of expressions for communicating with or among abstract domains. In implementation, this language is defined using a Spec# interface hierarchy, which is implemented by the BoogiePL AST classes. This setup makes it easy to use the abstract domains with other tools, as one simply needs to implement the abstract interpretation framework interfaces with the AST classes for the language of interest (instead of writing translation routines).

Expressions are simply variables, $\lambda$-expressions (for variable binding), and function symbols applied to expressions:

| | | |
|---|---|---|
| expressions | `Expr` | $e, p \ ::= \ x \mid \lambda x.\, e \mid f(e^*)$ |
| variables | `Var` | $x, y, ...$ |
| function symbols | `FunSym` | $f$ |

A *constraint* is any boolean-valued expression, and the set of function symbols include the usual operators from first-order logic: $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, $\forall$, and $\exists$.

**Abstract Domains.** An abstract domain must implement the signature shown in Fig. 4. Each abstract domain defines a type `Elt`, which represent the elements of the domain. The concretization function $\gamma$ yields the predicate that corresponds to the given element. (In the literature, the concrete domain is usually phrased in terms of sets of machine states, rather than state predicates.) When analyzing a program, we do not need to evaluate the corresponding abstraction

function $\alpha$, so we have omitted it from the signature. As usual, we require a partial ordering on domain elements, a greatest element, and a least element, which are given by $\sqsubseteq$, $\top$, and $\bot$, respectively; $\top$ is required to correspond to `true` and $\bot$ to `false`. We also need join $\sqcup$ and widen $\triangledown$ upper bound operators for the fixed point engine to handle control-flow join points: $\sqcup$ is usually the least upper bound operator and $\triangledown$ must have the stabilizing property.

```
type Elt
  val γ          : Elt → Expr
  val ⊤          : Elt
  val ⊥          : Elt
  val ⊑          : Elt × Elt → bool
  val ⊔          : Elt × Elt → Elt
  val ▽          : Elt × Elt → Elt
  val Constrain  : Elt × Expr → Elt
  val Eliminate  : Elt × Var → Elt
  val Rename     : Elt × Var × Var → Elt
```

**Fig. 4.** Abstract domains.

The final three operations provide a generic interface for implementing abstract transition relations. The `Constrain` operation adds (i.e., conjoins) a constraint to an element, `Eliminate` existentially quantifies (i.e., projects out) a variable, and `Rename` renames a free variable. For example, we might define the abstract transition relation on an assignment statement as follows:

$$\downarrow A$$
$$\boxed{x := e}$$
$$\downarrow \texttt{Rename}(\texttt{Eliminate}(\texttt{Constrain}(A, x' = e), x), x', x) \quad \text{(for a fresh variable } x')$$

which says if $A$ is the state before the assignment, the successor state is obtained by constraining $A$ with $x' = e$ for a fresh variable $x'$, then eliminating the old variable $x$, and finally renaming $x'$ to $x$.

**Base Domains.** Typically, the elements of an abstract domain can be viewed as constraints of a particular form on a set of variables (that is, on independent coordinates). For example, the polyhedra abstract domain [CH78] can represent linear-arithmetic constraints like $x + y \leqslant z$. We have implementations for a number of such standard abstract domains, which we call *base domains*. For inferring linear-arithmetic constraints, we have an implementation of the polyhedra domain [CH78], though we do not currently have implementations of any cheaper numerical domains (e.g., intervals or octagons [Min01]). We also have various basic abstract domains for constant propagation and dynamic type analysis. Finally, we have an important base domain that tracks what parts of the heap are preserved across updates (the *heap succession domain*) [CL05].

**Combining Abstract Domains.** Often, the constraints of interest involve function and relation symbols that are not all supported by any single abstract domain. For example, a constraint of possible interest in the analysis of a Spec# program is $\mathsf{sel}(H, o, \mathsf{x}) + k \leqslant \mathsf{length}(a)$ where $H$ denotes the current

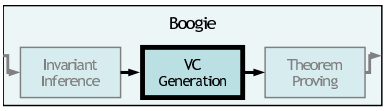
heap, $\mathsf{sel}(H, o, \mathsf{x})$ represents the value of the x field of an object $o$ in the heap $H$ (written $o.x$ in Spec# and written $H[o, \mathsf{x}]$ is our BoogiePL encoding), and $\mathsf{length}(a)$ gives the length of an array $a$. A constraint like this cannot be represented directly in the polyhedra domain because the polyhedra domain does not support the functions sel and length. Rather than building support for these functions into polyhedra, our framework includes a coordinating abstract domain that hides such *alien expressions* from base domains like polyhedra. This coordinating abstract domain, called the *congruence-closure domain*, is parameterized by the various base domains and tracks equalities and performs congruence-closure in order transparently to extend the base domains to work with alien expressions (i.e., expressions it does not understand and should be treated as uninterpreted) [CL05]. Such a modularization is particularly important. For example, including the heap succession domain as a base domain enables other base domains to obtain more precise properties of fields that depend on knowing that parts of the heap are preserved (without requiring the other base domains to know anything about heap updates).

**Inferred Invariants.** To get an idea of the kinds of invariants inferred by Boogie, consider again the example shown in Fig. 3. Using a standard polyhedra base domain (or in fact an interval domain would suffice) with the congruence-closure domain, we get the loop invariant $0 \leqslant i$ that gives the range of $i$. Using the heap succession domain with the congruence-closure domain, we infer the following important frame condition loop invariant (here shown in a simplified form):

$$(\forall\, o\colon \mathbf{ref}, f\colon \mathbf{name} \bullet \\ (o \neq \mathit{this} \vee f \neq \mathsf{Example.x}) \Rightarrow \mathit{Heap}[o, f] = \mathit{PreLoopHeap}[o, f])$$

This loop invariant is not only "boring" to the user but cannot even be specified at the source-level, since it quantifies over all objects and field names; however, it is necessary to be able to verify the assertion at the end of the method body in Fig. 2. The inferred loop invariants are inserted into the BoogiePL program in Fig. 3 as assume statements at the beginning of block $\mathit{LoopHead}$.

## 6 Verification Condition Generation




A BoogiePL program encodes sets of program traces and proof obligations in those traces. Verification condition generation turns those proof obligations into first-order formulas. As already described, BoogiePL is intentionally *not* a structured programming language. That is, a BoogiePL program is a somewhat high-level way of specifying a control-flow graph whose nodes are *basic blocks*. Since our verification conditions are computed using the standard weakest-precondition calculus [Dij76,Nel89], we had to develop a method for computing first-order weakest preconditions of an unstructured program.

Our method, which we have described in detail elsewhere [BL05], produces one VC for every BoogiePL procedure implementation. It starts by transforming the implementation into some loop-free BoogiePL code that over-approximates the loops in the original. It then performs a *single assignment* transformation (cf. [CFR+91,FS01]), resulting in only *passive* code, that is, code without state changes. Finally, to encode the unstructured nature of the control flow, we introduce for every block $A$ a boolean variable $A_{ok}$, defined to be *true* if every execution starting from $A$ is correct (i.e., does not go wrong). For a block:

$A$: *PassiveCommands*; **goto** $B, C$;

the *block equation* that defines $A_{ok}$ is:

$$A_{ok} \Leftarrow \mathfrak{wp}(\mathit{PassiveCommands},\ B_{ok} \wedge C_{ok})$$

where $\mathfrak{wp}$ computes the weakest precondition of *PassiveCommands* with respect to the postcondition $B_{ok} \wedge C_{ok}$. The VC is then:

$$\mathit{Axioms} \wedge \mathit{BlockEqs} \Rightarrow \mathit{Start}_{ok}$$

where *Axioms* is the conjunction of the axioms in the BoogiePL program, *BlockEqs* is the conjunction of block equations (which thus encode the semantics of the passive BoogiePL code), and *Start* is the implementation's start block.

While translating the BoogiePL program into a verification condition, the back-end phase builds up a table mapping labeled subformulas to BoogiePL program elements. The back-end phase uses this table to translate the label output from the theorem prover into an error message in terms of the BoogiePL program [LMS05]. Similarly, the front-end bytecode translation creates a table mapping BoogiePL program elements to Spec# program elements. This table allows it to take an error message on a BoogiePL program and generate an error message on the original Spec# program in terms that the programmer can understand.

Currently, Boogie produces verification conditions for the Simplify theorem prover [DNS05], which has a successful history of being used in program verifiers, starting with ESC/Modula-3 [DLNS98] for which it was first developed. A crucial ingredient in this success is the generation of VCs that allow the theorem prover to find concise proofs, essentially. Simplify uses what it calls the *goal property heuristic* [DNS05], which thrives on the kind of formulas generated by ESC/Modula-3 and, especially, by ESC/Java [FS01,Lei05]. However, Boogie's VCs represent the program's control flow in a different (and more compact) way than was used in ESC/Java, which sometimes causes Simplify to be slower than we would like. In addition, we have found that the formulas we use to axiomatize the Spec# class and interface subtypes are unexpectedly causing the prover to spend too much time in unfruitful ways. Finally, our heavy use of quantifiers in expressing certain changes in the heap sometimes causes Simplify to run out of steam (in particular, it reaches its "matching depth" [DNS05]). We are looking forward to addressing these issues in the further use and development of Zap [BLM05], but at present, Zap is not able to match the utility of Simplify.

## 7 Related Work

The body of previous work in program verification is enormous. In this section, we mention some of the tools that are most closely related to Boogie.

Three static program verifiers for the object-oriented language Java are LOOP, Jive, and KeY. LOOP [BJ01,JP03] takes Java plus contracts written in the Java Modeling Language (JML) [LBR99,LBR03]. It uses the interactive theorem prover PVS [ORR+96], for which it generates proof obligations that look like Hoare triples [Hoa69,JP01]. It also provides some automation by a weakest-precondition tactic [Jac04]. Jive [MMPH97] also uses a Hoare-like logic [PH97] and its custom-built interactive theorem prover operates at the level of Hoare triples (as opposed to first-order VCs generated from the programs). The KeY tool [ABB+05] offers several specification notations, including JML and dynamic logic, and targets several proof engines. The main differences between these three tools and Boogie are that they address a more limited subset of the source language and that they are not automatic.

Program verification technology has also been used in tools that find some program errors without promising to find all errors. These include the Extended Static Checkers for Modula-3 and Java [DLNS98,FLL+02,Lei00,KC04], JACK [BRL03], Krakatoa [MPMU04], and Cadeuces [FM04]. The automation in these tools rivals that of Boogie, and they all support the Simplify [DNS05] theorem prover. In addition, JACK supports PVS and the interactive prover of the Atelier B toolkit. Like Boogie, Krakatoa and Cadeuces generate verification conditions via an intermediate language, called Why [Fil03]. Though developed independently, Why and BoogiePL are more similar than they are different. The Why tool currently supports six different theorem provers, both interactive and automatic, but does not support property inference like Boogie's.

A number of programming languages have built-in specifications or were designed with verification in mind. Among these are Gypsy [AGB+77], Euclid [LHL+81], APP [Ros95], and others mentioned in the Spec# overview paper [BLS04]. The languages SPARK Ada [Bar03], B [Abr96], ACL2 [KMM00], Perfect Developer [Esc06], and C0 [LPP05] include verifiers with interactive theorem provers. The Eiffel language [Mey92] is well known for its pioneering combination of object-orientation and dynamically checked contracts, but it does not yet offer static verification.

## 8 Conclusion

In summary, Boogie is an automatic program verifier for modern object-oriented programs. Its architecture helps tame the complexity of the program verification task. Providing design-time feedback, Boogie moves the program verifier closer to the developer, while still hiding the theorem prover and other verification machinery from the developer. Designed around an intermediate programming notation, BoogiePL, it separates the semantic encoding of the source program from the analysis of this encoding. Since Boogie can also read BoogiePL programs

directly, it offers the possibility for others to write program verifiers by encoding their proof obligations in BoogiePL.

We have applied Boogie to a growing number of small (300–1500 lines) programs, and we are applying Boogie to parts of its own implementation (which is written in Spec#). We also also supporting an experiment in using Boogie on production code. This experience constantly demands support for more programming idioms, more targeted default specifications, better explanations of error messages (especially those having to do with violations of the ownership-based alias-confinement regime), and higher performance.

Boogie can be run as part of compilation, where the compiler provides its in-memory data structures to Boogie for verification. The verification results of Boogie could be used by the compiler's code optimizer to produce better performing code (cf. [Van94,FKR+00]). However, this is not part of the current Boogie architecture. The prospects of including this feedback in the architecture seem promising, but also contains some research questions such as how and to what degree to rely on specifications of code that may not have been verified.

The Spec# compiler produces both metadata and compiled code from Spec# contracts like preconditions. Lately, we have considered the possibility of using only stylized patterns of compiled code. Under such a design, Boogie would reconstruct the contracts from the stylized patterns of CIL instructions, and special method stubs would have to be created to support contracts on abstract methods. The advantage of such a design would be to make it possible to write contracts in .NET languages without contract features, by manually coding the stylized precondition checks. Boogie could then be applied to other .NET languages, too.

In developing Boogie's abstract interpretation framework, we found on numerous occasions the need to determine whether or not a given predicate holds. This functionality is readily available in the theorem prover, so we have wished that the abstract interpreter and the theorem prover would be more closely related. Indeed, there is already overlap between these two components. For example, both deal with linear arithmetic, both deal with uninterpreted function symbols, and both deal with the heap. Unlike the abstract interpreter, the theorem prover supports quantifiers and therefore provides a simple way to extend its reasoning to special domains; and unlike the theorem prover, the abstract interpreter computes fixpoints, rather than just answering boolean queries. We see the combination of these two components as a possible improvement in the Boogie architecture and as an exciting and important research area.

*Acknowledgments* This work would not have been possible without the efforts of the rest of the Spec# team: Manuel Fähndrich, Wolfram Schulte, and Herman Venter. We are especially grateful for the persistence and patience that Herman Venter has shown as he pioneers the use of Boogie in production code. We thank Peter Müller and Arnd Poetzsch-Heffter for performing case studies and diagnosing bugs in the system, and Francesco Logozzo for writing part of the abstract interpretation code. We are indebted to the Spec# user community and also to the anonymous reviewers.

## References


[ABB+05] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4(1):32–54, February 2005.

[Abr96] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[AGB+77] Allen L. Ambler, Donald I. Good, James C. Browne, Wilhelm F. Burger, Richard M. Cohen, Charles G. Hoch, and Robert E. Wells. GYPSY: A language for specification and implementation of verifiable programs. *SIGPLAN Notices*, 12(3):1–10, March 1977.

[Bar03] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, 2003.

[BDF+04] Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[BJ01] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer, 2001.

[BL05] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 82–87, 2005.

[BLM05] Thomas Ball, Shuvendu Lahiri, and Madanlal Musuvathi. Zap: Automated theorem proving for software analysis. Technical Report MSR-TR-2005-137, Microsoft Research, October 2005.

[BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–60. Springer, 2004.

[BN04] Mike Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In Dexter Kozen and Carron Shankland, editors, *Mathematics of Program Construction (MPC)*, volume 3125 of *Lecture Notes in Computer Science*, pages 54–84. Springer, 2004.

[BRL03] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: a developer-oriented approach. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods, International Symposium of Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439. Springer, September 2003.

[Bur72] Rod M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252, January 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Sixth ACM Symposium on Principles of Programming Languages (POPL)*, pages 269–282, January 1979.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Fifth ACM Symposium on Principles of Programming Languages (POPL)*, pages 84–96, January 1978.

[CL05] Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In Radhia Cousot, editor, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3385 of *Lecture Notes in Computer Science*, pages 147–163. Springer, 2005.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[DL05] Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, March 2005.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.

[Esc06] Escher Technologies. Perfect Developer. http://eschertech.com/, 2006.

[Fil03] Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *The Journal of Functional Programming*, 13(4):709–745, July 2003.

[FKR+00] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: An Optimizing Compiler For Java. *Software—Practice and Experience*, 30(3):199–232, 2000.

[FL03] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In Ron Crocker and Guy L. Steele Jr., editors, *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 302–312. ACM, 2003.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI)*, pages 234–245, 2002.

[FM04] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In Jim Davies, Wolfram Schulte, and Michael Barnett, editors, *Formal Engineering Methods (ICFEM)*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29. Springer, 2004.

[FS01] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 193–205. ACM, January 2001.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, October 1969.

[HW73] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. *Acta Informatica*, 2(4):335–355, 1973.

[Jac04] Bart Jacobs. Weakest pre-condition reasoning for Java programs with JML annotations. *Journal of Logic and Algebraic Programming*, 58(1–2):61–88, January–March 2004.

[JP01] Bart Jacobs and Erik Poll. A logic for the Java Modeling Language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE)*, volume 2029 of *Lecture Notes in Computer Science*, pages 284–299. Springer, 2001.

[JP03] Bart Jacobs and Erik Poll. Java program verification at Nijmegen: Developments and perspective. In *Software Security—Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003*, pages 134–153, November 2003.

[KC04] Joseph R. Kiniry and David R. Cok. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2, including a case study involving the use of the tool to verify portions of an Internet voting tally system. In *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS)*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2004.

[KMM00] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, June 2000.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[LBR03] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06u, Iowa State University, Department of Computer Science, April 2003.

[Lei95] K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, CalTech, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[Lei00] K. Rustan M. Leino. Extended static checking: A ten-year perspective. In Reinhard Wilhelm, editor, *Informatics—10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*. Springer, 2000.

[Lei05] K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, March 2005.

[LHL+81] Butler W. Lampson, James J. Horning, Ralph L. London, James G. Mitchell, and Gerald J. Popek. Report on the programming language Euclid. Technical Report CSL-81-12, Xerox PARC, October 1981. An earlier version of this report appeared as volume 12, number 2 in *SIGPLAN Notices*. ACM, February 1977.

[LL05] K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. In Kwangkeun Yi, editor, *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 3780 of *Lecture Notes in Computer Science*, pages 119–134. Springer, 2005.

[LM04] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In Martin Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.

[LM05] K. Rustan M. Leino and Peter Müller. Modular verification of static class invariants. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *Symposium on Formal Methods Europe (FM)*, volume 3582 of *Lecture Notes in Computer Science*, pages 26–42. Springer, 2005.

[LM06] K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In Peter Sestoft, editor, *European Symposium on Programming (ESOP)*, volume 3924 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2006.

[LMS05] K. Rustan M. Leino, Todd Millstein, and James B. Saxe. Generating error traces from verification-condition counterexamples. *Science of Computer Programming*, 55(1–3):209–226, March 2005.

[LN02] K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, September 2002.

[LPP05] Dirk Leinenbach, Wolfgang Paul, and Elena Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In Bernhard K. Aichernig and Bernhard Beckert, editors, *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005)*, pages 2–12. IEEE Computer Society, September 2005.

[LSS99] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. In *Formal Techniques for Java Programs*, Technical Report 251. Fernuniversität Hagen, May 1999. Also available as Technical Note 1999-002, Compaq Systems Research Center.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, 1992.

[Min01] Antoine Miné. The octagon abstract domain. In *Working Conference on Reverse Engineering (WCRE)*, pages 310–319, 2001.

[MMPH97] Peter Müller, Jörg Meyer, and Arnd Poetzsch-Heffter. Programming and interface specification language of JIVE—specification and design rationale. Technical Report 223, Fernuniversität Hagen, 1997.

[MPMU04] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, January–March 2004.

[Nel89] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, October 1989.

[ORR+96] Sam Owre, S. Rajan, John M. Rushby, Natarajan Shankar, and Mandayam K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer, 1996.

[PH97] Arnd Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitationsschrift, Technische Universität München, 1997.

[Rey78] John C. Reynolds. Syntactic control of interference. In *Fifth ACM Symposium on Principles of Programming Languages (POPL)*, pages 39–46, January 1978.

[Ros95] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.

[Spe06] Spec# homepage. http://research.microsoft.com/specsharp, 2006.

[Van94] Mark T. Vandevoorde. *Exploiting Specifications to Improve Program Performance*. PhD thesis, Massachusetts Institute of Technology, February 1994. Available as Technical Report MIT/LCS/TR-598.