Specification and Verification with Spec#

Yannick Welsch

yannick.welsch@gmx.net
Technische Universität Kaiserslautern, Germany

Abstract. This paper describes a specification and verification technique for object-oriented programs with invariants, known as the Boogie approach. The approach is proved to allow sound and modular verification. Topics covered by this paper are object hierarchies, subclassing, and advanced routine specifications. It also includes a discussion about the concrete implementation of these concepts in Spec# .

1 Introduction

The Spec# programming system [1] consists of a programming language that extends C# with specification constructs, a compiler that emits run-time checks for the specification constructs, a static verifier that modularly proves that the run-time checks never fail, an environment with base class library contracts and a Visual Studio extension. We will shortly describe the architecture underlying the Spec# programming system. The main focus of this paper then lies on the specification concepts underlying the front end of the system, and their embedding in the source language.

1.1 Architecture

To understand how the Spec# programming system works, we first present its architecture in Fig. 1.1, referred to as the Boogie pipeline [2]. Input to the pipeline is the Spec# source code written in the Spec# language, superset of C# , with additional specification features like method and class contracts [3], frame conditions, non-null types and enhanced exception categories. The Spec# compiler does additional static type checks in comparison to C# , and emits dynamic checks for the contracts as part of the generated byte code.

The output in form of the CIL (Common Intermediate Language [1]), the executable format of the .NET virtual machine, is then translated into BoogiePL [4], a simple coarsely typed imperative language, along with a logical encoding of the semantics of the source language, in our case Spec# . This additional step of transformation allows to separate source-encoding concepts from actual reasoning. This also leads to a modular architecture which offers the possibility for different front-ends (e.g. for Java Bytecode [5] or Eiffel source code[2]).

[1] formerly called Microsoft Intermediate Language or MSIL
[2] see Ballet at http://se.inf.ethz.ch/people/schoeller/iver.html

BoogiePL can be seen as a high-level front-end to a theorem prover. Before BoogiePL programs are turned into first order verification conditions, additional invariants, which seem obvious to the programmer, are inferred [6]. The verification conditions are then fed to a theorem prover (e.g. Simplify [7]), and tested for validity. The aim is to prove that the dynamic checks always succeed.

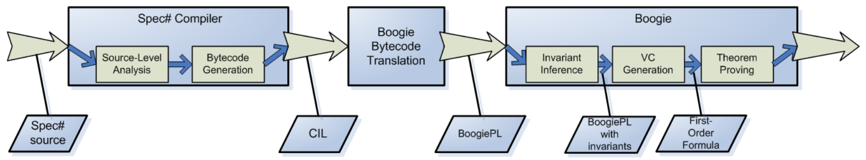

Fig. 1. The architecture of Boogie

In the next section the specification concepts underlying the front end of the system are presented.

2 The Boogie Approach

We give a short motivation for the new approach, then we introduce step by step the different concepts of the approach, where in each increment we enhance the previous definitions.

2.1 Object Invariants and Information Hiding

An object invariant, specified in [8] as

a relation on an object's data that the programmer intends for to hold

is often seen as a shorthand for a postcondition on the constructor and pre- and postcondition on every public method (like in Eiffel[9],[10]). This seems appropriate, but special care must be taken: A method implementation can not always violate the object invariant for the duration of the call, in the presence of reentrance and callbacks. This is illustrated by Listing 1.1, where the call in method `M1` to `N`, resulting in a callback to `M2`, breaks the invariant, although each method, at first sight, doesn't seem to break the invariant.

Most of the techniques reasoning about object invariants developed in the last 20 years, called in [11] as the *classical technique*, have restricted themselves to invariants on fields of primitive values and are thus very restricted in their applicability in practice. A notable exception is [12], using as a basis an ownership type system. The specification technique explained here deals with layered object structures while retaining modularity.

Another issue with expressing invariants as pre- and postconditions is the exposure of the internal representation of a class. A combination of good information hiding and specification completeness has to be found. The solution

```
class T {
  private bool x,y;
  invariant x == y;
  public T()
  {
    x = false; y = false;

  }
  public void M1(U u)
  {

    x = !x;
    u.N(this); //calls M2();
    y = !y;

  }
  public void M2()
  {


    if (x)
      y = x;

  }
}
```

Listing 1.1. Invariant breaking in the presence of callbacks

```
class T {
  private bool x,y;
  invariant x == y;
  public T()
    ensures st = Valid; {
    x = false; y = false;
    pack this;
  }
  public void M1(U u)
    modifies x,y; {
    unpack this;
    x = !x;
    u.N(this); //calls M2();
    y = !y;
    pack this;
  }
  public void M2()
    requires st = Valid;
    modifies y; {
    unpack this;
    if (x)
      y = x;
    unpack this;
  }
}
```

Listing 1.2. Annotated same program

presented here achieves this by introducing a publicly available abstraction of whether or not invariants hold.

2.2 Validity

In this first version of the concepts, subclasses are ignored. A special field *state*, abbreviated by **st**, of type $\{Valid, Invalid\}$ is introduced for each object. It is restricted to appear only in routine specifications, not in invariant declarations or in implementations. The idea here is to make sure that an object's invariant holds whenever its state field is set to *Valid*. An object is allocated in the invalid state. We further define $Inv_T(o)$ for an object o of type T as the predicate that holds iff the object invariant declared in T holds for o in that state. For example, in Listing 1.1, we have $Inv_T(o) \equiv$ o.x == o.y

To influence the *st*ate field by the implementation, two new statements, *pack* and *unpack* (Def. 1 & 2) are introduced, validating and unvalidating an object. For an object o of type T:

$$\begin{aligned}\textbf{pack } o \equiv &\\ &\textbf{assert } o \neq \mathit{null} \wedge o.st = \mathit{Invalid};\\ &\textbf{assert } \mathit{Inv}_T(o);\\ &o.st := \mathit{Valid}\end{aligned} \tag{1}$$

$$\begin{aligned}\textbf{unpack } o \equiv &\\ &\textbf{assert } o \neq \mathit{null} \wedge o.st = \mathit{Valid};\\ &o.st := \mathit{Invalid}\end{aligned} \tag{2}$$

The *pack* statement first checks the object invariant *$Inv_T(o)$* and changes *st* from *Invalid* to *Valid*. *unpack* does the opposite and invalidates the state again. Both statements are not idempotent, so packing or unpacking two times the same object leads to an error (e.g. **pack this**; **pack this**;). As final step in this chapter, object invariants are restricted to rely only on fields (of simple type) of *this*, and thus field updates are restricted to invalid objects, as only these can lead into breaking the invariant. The resulting specification is illustrated in Listing 1.2, which makes the implicit knowledge in Listing 1.1 explicit. Now, there are some possibilities for this program to go wrong. If `u.N` requires the parameter to be in a valid state, the precondition of `N` is not fulfilled at the calling site `M1`. The other possibility is that `u.N` does not require the parameter to be in a valid state, resulting in a precondition violation of `M2` at the place the callback occurs (in `N`). Another possibility, as `M2` states as precondition that *st* must be *Valid*, is to *pack* our object of type `T` in `N`, leading to an error because the invariant does not hold. The only possibility for a valid execution of `M1` here is for `N` to reestablish the invariant (by doing changes to `x` or `y`), *pack* the object, make the callback to `M2`, *unpack* the object again, make a change to `x` or `y` such that x == ˜y and return.

2.3 Components

As we want our invariants, contrary to the *classical approach*, to contain relations between objects being part of a same component, a new field modifier **rep** is introduced that identifies the component objects of an object. The following restriction is now imposed: An component object can only be unpacked if the object it belongs to is unpacked first (see Fig. 2).

An object in return can only be packed if its component objects are valid. This is achieved by introducing a new value *Committed* for *state* and further restraining of *pack* and *unpack* (Def. 3 & 4). Let $Comp_T(o)$ denote the set of expressions `o.f` for each rep field `f` in `T`. Then, for any expression o of type T:

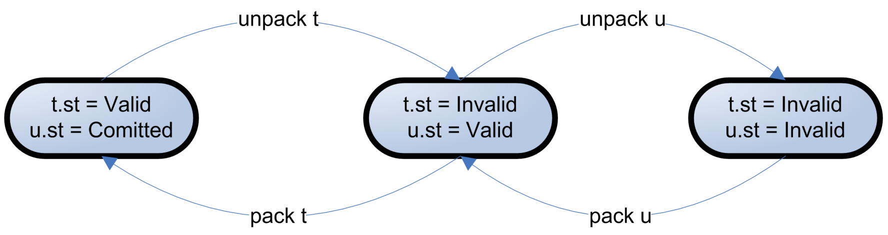

Fig. 2. States of a component consisting of an object t owning an object u. For simplicity, it is assumed that t.u is not null

$$\begin{aligned}
&\textbf{pack } o \ \equiv \\
&\quad \textbf{assert } o \neq \mathit{null} \wedge o.st = \mathit{Invalid}; \\
&\quad \textbf{assert } \mathit{Inv}_T(o); \\
&\quad \textbf{foreach } p \in \mathit{Comp}_T(o)\{\textbf{assert } p = \mathit{null} \vee p.st = \mathit{Valid};\} \\
&\quad \textbf{foreach } p \in \mathit{Comp}_T(o)\{\textbf{if } (p \neq \mathit{null})\{p.st := \mathit{Committed};\}\} \\
&\quad o.st := \mathit{Valid}
\end{aligned} \tag{3}$$

$$\begin{aligned}
&\textbf{unpack } o \ \equiv \\
&\quad \textbf{assert } o \neq \mathit{null} \wedge o.st = \mathit{Valid}; \\
&\quad o.st := \mathit{Invalid} \\
&\quad \textbf{foreach } p \in \mathit{Comp}_T(o)\{\textbf{if } (p \neq \mathit{null})\{p.st := \mathit{Valid};\}\}
\end{aligned} \tag{4}$$

What has changed to the previous definitions of *pack* and *unpack* is that component objects are taken into consideration. Then the invariant 5 holds in every reachable state of the program: For any class T

$$\begin{aligned}
&\forall o : T \ \bullet \ o.st = \mathit{Invalid} \vee \\
&\qquad (\mathit{Inv}_T(o) \wedge (\forall p \in \mathit{Comp}_T(o) \bullet p = \mathit{null} \vee p.st = \mathit{Committed}))
\end{aligned} \tag{5}$$

This can be proved by induction over the structure of program statements resulting from the definitions of *pack* and *unpack*.

In contrast to other methodologies (see [11],[12]) , no restrictions have been made on copying object references or on allowing multiple references to a component object. This methodology also enables ownership transfer, meaning an object can be owned by different owners over time. This transfer can occur if the object is not in the *Committed* state, e.g. not being owned.

2.4 Subclasses

The methodology presented also handles subclasses. A class frame is defined by a class and the set of fields and invariants defined in that class. If we allow subclassing, an object can now have multiple class frames, and be valid or invalid for a certain class frame as invariants can be specified at different levels of the class hierarchy. To represent the subset of valid class frames, a special field *inv* is introduced, whose value is the most derived class whose class frame is valid for the object. By this definition, we allow a class frame only to be valid when the class frames higher in the class hierarchy are valid too. Another special field *comitted* of type *boolean*, indicating whether the object is committed, is introduced. This results from the problem of encoding the information of whether an object is in the `Comitted` state or not. The old ***state*** field is thus superfluous but we have to ensure that *committed* is *true* only if *inv* is equal to the dynamic type of that object. Packing and unpacking can now be done for each class frame by pack o as T and unpack o from T (see Def. 6 & 7), where T stands for the set of classframes: For an object o of type T with S being an immediate superclass of T:

```
pack o as T ≡                                                              (6)
        assert $o \neq null \wedge o.inv = S$;
        assert $Inv_T(o)$;
        foreach $p \in Comp_T(o)\{$
        assert $p = null \vee (p.inv = \mathbf{type}(p) \wedge \neg p.committed);\}$
        foreach $p \in Comp_T(o)\{$if $(p \neq null)\{p.committed := true;\}\}$
        $o.inv := T$
```

```
unpack o from T ≡                                                          (7)
        assert $o \neq null \wedge o.inv = T \wedge \neg o.committed$;
        $o.inv := S$
        foreach $p \in Comp_T(o)\{$if $(p \neq null)\{p.committed := false;\}\}$
```

These definitions combine the concepts of component objects and subclassing. This means, when packing an object with a set of classframes `T`, we only put the component object belonging to this set into the committed state. It is first required that the invariant holds for the smaller set of classframes `s`. To put the component objects into the committed state, it is also required that the invariant holds for the dynamic type of the component object (denoted by **type**(p)). This is the intuitive meaning, as we want our component object to be completely packed.

From these definitions, program invariants (see Def. 8) guaranteeing the soundness of the approach can be derived:

$$\begin{aligned}
&\forall o, T \bullet o.committed \Rightarrow o.inv = \mathbf{type}(o)\\
&\forall o, T \bullet o.inv <: T \Rightarrow Inv_T(o)\\
&\forall o, T \bullet o.inv <: T \Rightarrow (\forall p \in Comp_T(o) \bullet p = null \vee p.committed)\\
&\forall o, T, o', T', q \bullet (\forall p \in Comp_T(o), p' \in Comp_{T'}(o') \bullet\\
&\qquad o.inv <: T \wedge o'.inv <: T' \wedge q.committed \wedge p = p' = q \Rightarrow o = o' \wedge T = T')
\end{aligned} \tag{8}$$

where quantifications over references range over non-null objects, and $<:$ *denoting the reflexive and transitive subclass relation.*

It can be proved that the statements, which can extend the range of the quantifications, namely *pack*, *unpack*, object creation and field updates, do not break the invariants. The detailed proofs can be found in [8]. With the newly introduced concepts two new innovations are introduced in the next chapter.

2.5 Routine Specifications

Routine specifications describe what is expected of the caller at the time of the call and what is expected of the implementation at the time of return. We first describe a partial solution to the information hiding versus specification completeness problem. Having a layered program, as denoted by our *rep* keyword, we allow the routine to modify layers of private state without explicitly mentioning this state in the modifies clause. The proposed policy lets a routine modify the state of any object that is committed at the time the routine is called (but of course *unpack* must be used in the implementation). An additional idea is the use of expressions of the form $E.\{T\}$ in modifies clauses, denoting all fields of object E declared in class T and its superclasses.

The second innovation comes from the problem that stating in a precondition the exact value desired for an object's *inv* field seems incompatible with dynamically dispatched methods. The innovation here is to introduce a new expression, **1**, with the meaning of $\mathbf{type}(this)$ for the caller, and meaning T for an implementation of the method given in class T. Thus, upon calling a method with precondition $inv = \mathbf{1}$, a caller invoking the method on an object o knows that o is entirely valid, without having to know the dynamic type of o. We thus provide a way for the specification of a dynamically dispatched method to talk about the entire-validity of an object without forcing implementations to reason about possible subclasses.

2.6 Summary

As a brief summary, we have seen that the presented methodology permits object invariants to depend on fields declared in superclasses and on fields of transitively owned objects. However, as we require these special field to be annotated with the modifier *rep*, there is a static limit on the number of owned objects an owning object can have. Another drawback is that we have not considered concurrency, and that the components are accessible only through fields. For more advanced language features as exceptions and final methods, this methodology has to be adapted further. However, the soundness of the current methodology allows a programmer to make strong assumptions about program correctness.

3 Application to Spec#

The aforementioned specification technique leads to rather long specifications, so syntactic sugar and good defaults have to be found in order for it to be applicable in practice. The Spec# language specifies among others the following constructs. The construct **expose**(o) {...} with *o* of static type *T* models our conceptual **pack** o as T ; ... **unpack** o from T;. and *rep* is modeled by the [Rep] modifier. For an expression *o* of reference type *T*, o.IsConsistent is the corresponding to $Inv_T(o)$.

3.1 An Example

We give as an example the specification of an Integer Stack class in Listing 1.3. Our stack is implemented with the help of an array `elems` which stores the values pushed onto the stack and and integer storing the current size of the stack. The `elems` array is marked with the [Rep] modifier, as we want the `IntStack` class to own the `elems`. Spec# also allows to specify non-null types , as for the `elems` field which never points to the null value, which is indicated by the exclamation mark at the type definition. `size` is marked by the [SpecPublic] modifier[3] which makes the field usable in our contracts. Our invariant for the `IntStack` class then specifies that the size must be non-negative and smaller or equal to the capacity of the `IntStack`, which corresponds to the length of the array. The `IntStack` constructor takes as argument the capacity for which the `IntStack` will be initialized and states as precondition that the capacity must be strictly positive. As we want to use the capacity in our contract specifications, we introduce a so-called getter method Cap. The constructor is marked by the [NotDelayed] modifier as we have to initialize the non-null `elems` field before calling the super-constructor by **base**().

The methods to push or pop an integer from the stack are then specified and implemented. We specify that the size increases respectively decreases with each call, and that there has to be sufficient capacity in case of a push and the stack must not be empty in case of a pop action. We can also specify which fields are to be modified by the implementation using the `modifies` clause.

Now that we have specified and implemented the integer stack, we demonstrate how the static checker can be used to find errors in our program. Assume we have written a program using the integer stack like in Listing 1.4. We can

[3] precisely a Spec# annotation

```
using System;
using Microsoft.Contracts;

class IntStack {
  [Rep] int[]! elems;
  [SpecPublic] int size;
  invariant 0 <= size &&
    size <= elems.Length;

  public int Cap {
    get {
      return elems.Length; }
  }

  [NotDelayed] public
  IntStack(int cap)
    requires cap > 0;
    ensures size == 0;
    ensures Cap == cap;
  {
    size = 0;
    elems = new int[cap];
    base();
  }

  public void Push(int e)
    requires size < Cap;
    modifies this.*;
    ensures size == old(size)+1;
    ensures Cap == old(Cap);
  {
    expose (this) {
      elems[size++] = e;
    }
  }

  public int Pop()
    requires 0 < size;
    modifies this.*;
    ensures size == old(size)-1;
    ensures Cap == old(Cap);
  {
    expose (this) {
      return elems[--size];
    }
  }
}
```

Listing 1.3. Integer stack implementation in Spec# , inspired by [13]

introduce assert statements which the static checker then tries to verify. For example we want to check that the capacity remains unchanged after pushing an element on the stack. This can be proved by the statical checker, as we specified for Pop() that Cap == old(Cap). However we have pushed two elements onto the stack and tried to pop three elements off.

```
public class Program
{
  static void Main()
  {
    IntStack s = new IntStack(2);
    s.Push(100);
    assert s.Cap == 2;
    s.Push(200);
    s.Pop();
    s.Pop();
    s.Pop(); //error
  }
}
```

Listing 1.4. Using the IntStack

The static checker finds this bug and correctly reports the warning `Call of IntStack.Pop(), unsatisfied precondition: 0 < size`. Even if not everything can be verified statically, we can be sure that at runtime, due to the compiler emitted runtime checks, our program will not enter a non specified state unnoticed, e.g. break an invariant.

4 Conclusion

After a short introduction to the Spec# programming system, a solution to the *classical approach* has been presented, where objects don't even have to be protected from aliasing. The presented approach makes it clear to the programmer at what program points an object invariant can be relied upon. While field updates are restricted, there are no restrictions on read operations (in comparison to read-only references in [14]). The only restriction imposed is that each object must have at most one valid owner at any time, which is enforced at the time of *pack* operations.

In the second part of the paper, we have had a look at the constructs the Spec# language defines. Some more constructs not presented have been introduced to ease the use of the presented specification technique combined with other concepts. As we haved used the Spec# programming tools, it was not always clear why certain more advanced examples would statically verify and others not. To get a good feeling for what can be done with the current tools, we point to [13][4]. However, one has to note that sometimes even small changes to the specifications can change the outcome of the static verification completely. As writing specifications in a way that they are statically verifiable seems to be hard at times, the programmer should not rely on them.

The runtime checks are currently the most important feature which can be put into use in a test environment, where they can help the programmer to detect the program states, which were never intended for to be reached by the program. Another shortcoming of the current approach is the lack of documentation for the more advanced specification statements. In order to use the specification technique successfully for larger examples, we had problems to grasp the exact semantics of different statements, as the defaults on different program constructs were not obvious to us.

[4] Textbook Examples Verified Using Spec# available at http://www.rosemarymonahan.com/specsharp/

References

1. 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.: Post Conference Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille. Volume 3362 of LNCS., Springer-Verlag (2005) 49–69
2. 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. Volume 4111 of Lecture Notes in Computer Science., Springer (2005) 364–387
3. Meyer, B.: Design by contract. In Mandrioli, D., Meyer, B., eds.: Advances in object-oriented software engineering. Prentice Hall (1992)
4. DeLine, R., Leino, K.R.M.: Boogiepl: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70 (2005)
5. Lehner, H., Müller, P.: Formal Translation of Bytecode into BoogiePL. In Huisman, M., Spoto, F., eds.: Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE). Electronic Notes in Theoretical Computer Science (2007)
6. Chang, B.Y.E., Leino, K.R.M.: Inferring object invariants: Extended abstract. Electr. Notes Theor. Comput. Sci **131** (2005) 63–74
7. Detlefs, D., Nelson, G., Saxe, J.: Simplify: A theorem prover for program checking (2003)
8. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology **3**(6) (2004)
9. Meyer, B.: Object-Oriented Software Construction. Prentice Hall (1988)
10. Meyer, B.: Eiffel: The Language. Prentice Hall (1992)
11. Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular invariants for layered object structures. Science of Computer Programming **62**(3) (2006) 253–286
12. Müller, P.: Modular Specification and Verification of Object-Oriented Programs. Volume 2262 of Lecture Notes in Computer Science. Springer-Verlag (2002)
13. Leino, K.R.M., Monahan, R.: Automatic verification of textbook programs. Manuscript KRML 175, 13 May 2007, DRAFT. (2007)
14. Müller, P., Poetzsch-Heffter, A.: Universes: A type system for alias and dependency control. Technical Report 279–1, Fernuniversität Hagen (2001)