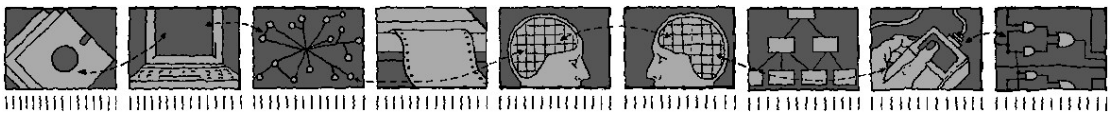
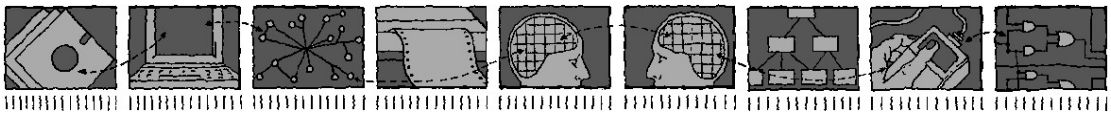


Department of Computing Science and Mathematics
University of Stirling



A Case Study in Integrated Assertion-Based Verification with Omnibus

Thomas Wilson, Savi Maharaj, Robert G. Clark

Technical Report CSM-176

ISSN 1460-9673

January 2009

Department of Computing Science and Mathematics
University of Stirling

A Case Study in Integrated Assertion-Based Verification with Omnibus

Thomas Wilson, Savi Maharaj, Robert G. Clark

Department of Computing Science and Mathematics
University of Stirling
Stirling FK9 4LA, Scotland
Telephone +44-1786-467421, Facsimile +44-1786-464-551

Email [twi,savi,rgc]@cs.stir.ac.uk

Technical Report CSM-176

ISSN 1460-9673

January 2009

Abstract

We present the example of the specification, implementation, and verification of a library system in Omnibus. Three approaches to verification (runtime assertion checking, extended static checking, and full formal verification) are applied to the example, and we compare the ease of use and the error coverage of each approach. We then discuss how the three approaches may be used together within Omnibus in an integrated manner, explain the benefits of this, and show how integration is supported by the Omnibus IDE.

Keywords: static checking, run-time checking, integrated formal methods, object-oriented programming

1. Introduction

In [13-15] we introduced the Omnibus system, which provides a flexible, integrated approach for developing verified, object-oriented software. Omnibus supports three main approaches to verifying assertion-annotated software: Runtime Assertion Checking (RAC), Extended Static Checking (ESC), and Full Formal Verification (FFV). Different verification policies may be applied to different components within a single software project, according to the level of rigor that is needed and the amount of resources available for carrying out verification.

Since 2005 the Omnibus system has undergone further development. The guidelines for combining verification approaches have been refined and tool support for enforcing these guidelines has been greatly improved [14]. In this paper we present a medium-sized case study, carried out using Omnibus. The case study provides a showcase for the range of verification techniques available in Omnibus, and illustrates the advantages and disadvantages of each technique. The case study also demonstrates how different verification techniques may be integrated within a single project.

We begin with an overview of the Omnibus IDE and language. This is followed by the case study, which concerns a library system. We describe the implementation of the library system and then apply the RAC, ESC and FFV approaches, in turn, to verify the project in its entirety. This was done to permit direct comparison of the error coverage of each approach; we do not necessarily recommend that the verification approaches be applied sequentially to real examples in this way. We see that the approaches provide increasing error coverage and require increasingly expressive assertion annotations. We then describe how the approaches can be used together in a more integrated fashion, utilising new tool and language features.

2. An Overview of Omnibus

Omnibus provides three distinct verification approaches: RAC (exemplified in [7]), which uses lightweight assertions which are converted into run-time checks; ESC (exemplified in [8]), which provides automatic static detection of run-time errors and violations of lightweight assertion annotations; and FFV (exemplified in [1]), which consists of proving the correctness of a program with respect to a fully expressive, heavyweight specification.

By providing integrated support for the approaches within a single tool, the Omnibus IDE allows different approaches to be used for different parts of a single project. The choice of approach is managed through a system of Verification Policies. Each policy defines how to manage the verification of a single class. Policies for RAC, ESC, and FFV are given as defaults and further policies may be defined by fine-tuning these; for example, by defining what theorem proving tool to use, how to handle troublesome constructs, or what detail to include in failure reports.

Verification is carried out in a number of different ways, depending on the chosen policy. For RAC, a Java generator translates the program into a Java program with embedded run-time checks. When the program is executed, a run-time check report is generated giving details of the assertion checks that have been executed. For ESC and FFV, the program is subjected to symbolic execution, which generates verification conditions, expressed in a generic logic [15]. These are then translated into the native logic of a theorem-proving tool, which carries out the proofs. For ESC, the prover that is currently used is Simplify [9] which is fully automatic. For FFV, there is a choice between Simplify or the interactive theorem-prover PVS [11]. The proof output links directly back into the IDE.

The Omnibus programming language [15] is superficially like Java, using similar concepts of packages, classes, methods, expressions, statements etc. To this is added a behavioural interface specification language. The language has been designed so as to facilitate verification and ease the annotation burden for developers. A key simplification is the use of value semantics by default for objects.

An Omnibus program consists of a set of *class definitions*, each containing various methods for manipulating instances of the class. Methods may be *constructors*, which create an object; *functions*, which allow objects to be queried without side-effects; or *operations*, which update objects. Constructors are *class methods* whereas functions and operations are *object methods*.

In Omnibus, all objects are immutable with the system creating new objects behind-the-scenes as needed to preserve value semantics. This is hidden from the programmers who are allowed to think in terms of updating objects. There is a single equality operator which represents deep equality.

Behaviour specifications: The behaviour of a method is described using behaviour specifications, expressed as *requires*, *changes* and *ensures* clauses. These state the method's pre-conditions, frame conditions and post-conditions, respectively. A subset of the functions in the class are taken to represent the abstract state of the class; these *model functions* are declared with the `model` modifier and do not have post-conditions. The behaviour of the remaining methods is then specified in terms of the model functions. A method may have different behaviour specifications at different levels of accessibility (public, private, or protected). For example, a function can be declared as a model function at the public level but described as a derived function in terms of the private attributes at the private level.

Requirements specifications: The requirements of a class are specified using *initially*, *invariant*, and *constraint* assertions. Initially assertions should hold over all freshly constructed objects, invariant assertions should hold over objects whenever they are accessible by code in other classes and constraint assertions should hold across any operation calls.

Implementations: Implementations of classes are defined in terms of method implementations that manipulate the values of private *attributes*.

Libraries: Like JML [4], Omnibus hides mathematical abstractions like sequences and sets behind a façade of library classes. The case study includes examples of the use of some of these classes, such as `List` and `Collection`.

3. Informal description of the Library System

The Library contains a set of `Item`s which can be loaned out. Each `Item` has a unique `itemNo` and `title`. There are three concrete types of `Item`: `Book`s, `CD`s and `DVD`s. In this paper we focus on the use of `Book`s. A `Book` has additional attributes: an ISBN, a list of authors (which may not contain duplicates), a publisher and a published date.

The Library can hold multiple copies of each `Item`. Each `Copy` has a unique `copyNo` along with the `itemNo` identifying the item of which it is a copy. Items and copies can be searched, added, deleted or changed. Adding a new item creates a new copy and deleting the last copy of an item deletes the `Item`'s details from the system.

People must register with the library as members before they can loan out a copy of an item. Members can be searched, added, deleted or changed. Members can loan out copies of items. For each `Loan`, the details of the `memberNo`, `copyNo` and start and end dates of the `Loan` are recorded. Loans can be searched, added, deleted or changed via a standard loan/return/renew procedure.

4. Implementation of the Library System

Here we present an implementation with lightweight assertion annotations suitable for verification using RAC. In later sections we will add further annotations to allow the use of ESC and FFV. An initial implementation was developed as part of an investigation into a novice user's experience of using Omnibus [3]. Some of the code (and errors) discussed here derive from this work. The full code for the library system may be viewed at [12].

4.1. Auxiliary classes

Most of the classes in the case study are simple data structures. These include `Book`, `CD`, `Card`, `Copy`, `DVD`, `Item`, `Loan`, `Member` and `Name`. These all follow the same simple structure. Each of them is defined in a file with sections for the abstract state, constructor and update operations.

As an example, consider the `Book` class. The abstract state is represented by four public model functions, which also serve as accessor functions for the private attributes:

```
public class Book isa Item {
  private attribute isbn:ISBN
  private attribute authors:List[Name]
  private attribute publisher:String
  private attribute pubDate:Date

  public model function isbn():ISBN
  { return isbn; }
```

```
public model function authors():List[Name]
{ return authors; }
public model function publisher():String
{ return publisher; }
public model function pubDate():Date
{ return pubDate; }
```

Requirement specifications such as `invariant`s are used to describe restrictions on the abstract state. For example, there should not be any duplicates in the list of authors. The `omni.lang.List` class that is used to model the lists contains a function called `containsDuplicates` which can be used to describe this:

```
public invariant noDuplicatesInAuthors:
  !authors().containsDuplicates()
```

The constructor simply assigns the passed parameters to the corresponding attributes (including attributes inherited from the superclass `Item`).

```
public constructor newBook(iNo:integer, t:String, isbnNo:ISBN,
      auths:List[Name], pub:String, pDate:Date)
{itemNo := iNo;
  title := t;
  isbn := isbnNo;
  authors := auths;
  publisher := pub;
  pubDate := pDate; }
```

To complete the `Book` class, four operations, `setISBN`, `setAuthors`, and `setPubDate`, allow updating of the state elements. These are declared to change the relevant state element. As an example, `setAuthors` is shown below:

```
public operation setAuthors(as:List[Name])
  changes authors
{ authors := as; }
```

Other auxiliary classes are `ISBN`, `ItemStatus`, `MemberStatus`, and `Date`, the details of which are omitted.

4.2. The main Lib class

The bulk of the code is within the `Lib` class. This contains the items, copies, members, cards and loans currently in the library along with methods to manipulate them.

Accessor functions

As in `Book`, the accessor functions for the private attributes are declared as public model functions. This enables them to be referred to in `changes` clauses.

The model functions for the `Lib` class are:

```
public model function items():Collection[Item]
{ return items; }
public model function copies():Collection[Copy]
{ return copies; }
public model function members():Collection[Member]
{ return members; }
public model function cards():Collection[Card]
{ return cards; }
public model function loans():Collection[Loan]
{ return loans; }
```

Formalisation of invariants of Lib class

Seven class invariants were identified for the `Lib` class in the original informal specification of the library [3]. Four invariants state that the identification numbers used for each item, copy, card and member (item number, copy number, card number and member number, respectively) are unique. Two invariants state that the copies and members that are referred to in any loan must be in the library's recorded copies and members, respectively. The final invariant expresses that a single copy cannot be recorded as on loan more than once at any one time.

For example, the invariants that the `itemNos` of the `Items` in the library are unique, that every copy that is on loan is in the `copies` collection and that no copies are on loan more than once are shown below.

```
public invariant itemNosInItemsUnique:
```

```
   forall (i:Item in items()):
    forall (i2:Item in items()):
     i.itemNo() != i2.itemNo() || i = i2
public invariant allCopiesOnLoanInCopies:
   forall (l:Loan in loans()):
    exists (c:Copy in copies()):
     l.copyNo() = c.copyNo()
public invariant noCopiesOnLoanMoreThanOnce:
  forall (l:Loan in loans()):
   forall (l2:Loan in loans()):
    l.copyNo() != l2.copyNo() || l = l2
```

Lib operations

As an example of an operation from the `Lib` class, consider `takeOutCopy`, which is used to loan a copy of an item to a member. This method accepts a member number, a copy number, and a date and takes out a new loan:

```
public operation takeOutCopy(memNo:integer, cNo:integer, date:Date)
  requires exists (m:Member in members()):
      m.memberNo() = memNo,
    exists (c:Copy in copies()):
      c.copyNo() = cNo,
    forall (l:Loan in loans()):
      l.copyNo() != cNo
  changes loans
  ensures loans().size()=old loans().size()+1
{ loans.add(Loan.borrows(memNo, cNo, date,
        calcNewDate(lookupItemFromCopy(cNo)
                    .itemNo(),date))); }
```

The `requires` clause asserts that the given member number and copy number must be known, and that there must not be any existing loan with the given copy number. The `changes` clause states that only the `loans` element of the abstract state may be changed. The `ensures` clause gives a lightweight specification of the behaviour of the method: it simply states that the size of the `loans` collection increases by one. This is sufficient for RAC.

The implementation constructs a new `Loan` object with the appropriate values and adds it to the `loans` collection.

As an example of one of the update methods, consider the `changeMemberName` operation which is used to change the name associated with a given member number:

```
public operation changeMemberName(
       memNo:integer, f:String, s:String)
  requires exists (m:Member in members()):
          m.memberNo() = memNo
  changes members
  ensures members().size() = old members().size(),
    lookupMember(memNo).name() = Name.newName(f, s)
{var memObj:Member:= lookupMember(memNo);
 members.remove(memberObj);
 memberObj.setName(Name.newName(f, s));
 members.add(memberObj);  }
```

The `requires` clause asserts that there must be a `Member` with the given `memberNo`. The `changes` clause states that only the `members` state element is changed. The `ensures` clause gives a lightweight specification of the behaviour of the method, saying that the size of the `members` collection does not change and that the new name is associated with the given member number.

The implementation looks up the member with the given member number, removes that member record, and then replaces it after updating it with the new name.

Lib functions

There are also functions to query aspects of the library. For example, consider the `lookupBookFromISBN` function, which accepts an `ISBN` and returns a `Book` with that `ISBN`:

```
public function lookupBookFromISBN(isbn:ISBN):Book
  requires exists (i:Item in items() where i is Book):
     (i as Book).isbn() = isbn
  ensures result.isbn() = isbn,
     items().contains(result)
{foreach (i:Item in items){
```

```
   if (i is Book){
    var book:Book := i as Book;
    if (book.isbn() = isbn){return book;}
  }}
  unreachable; }
```

The `requires` clause asserts that there must be an `Item` which is a `Book` and has the specified `ISBN`. The `ensures` clause says that the `isbn` of the returned `Book` must equal the passed `ISBN` and that the `Book` must be in the `items` collection.

The implementation iterates through the elements in the `items` collection. If it finds a `Book`, it checks the `isbn` and returns that `Book` if it matches the passed `ISBN`. The `unreachable` statement should never be reached because the `requires` clause ensures that the book will be found.

5. Applying RAC to the Library System

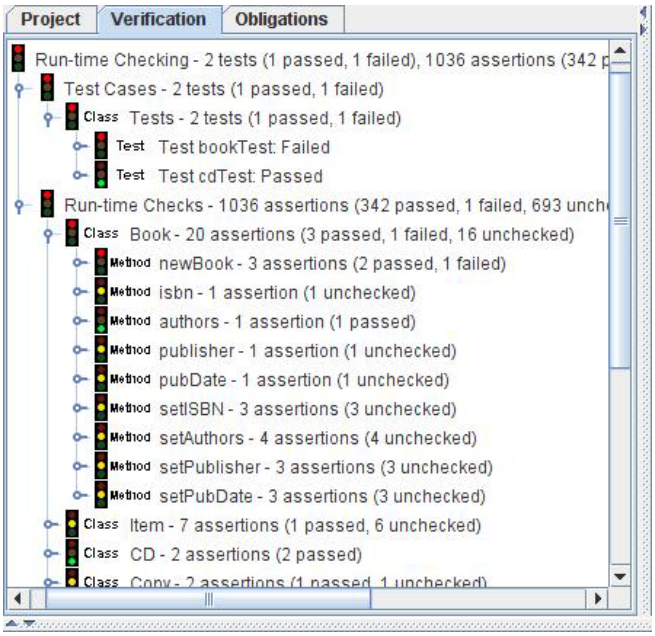
Verification with RAC is built around testing. Tests are used to ensure that the application performs its intended function without violation of any of the assertion annotations.

The Omnibus language provides the facility to define tests and the Omnibus IDE can be used to automatically re-execute them at the push of a button. A Check Viewer tool is used to check the extent that the assertions present in the class were covered by executing the tests.

As an example, a small part of one of the tests produced for the `Lib` class is displayed below. In its complete state, this particular test covers adding, retrieving, updating and changing the loan status of books.

```
test bookTest {
 var lib:Lib := Lib.testDataLibrary();
 // declare new book details
 var t:String:="Comparative Programming Languages";
 var i:ISBN:=ISBN.fromString("0201710120");
 var n:Name:=Name.newName("Robert","Clark");
                           "Wilson");
 var a:List[Name] :=      List[Name].empty().add(n);
 var p:String := "AW";
 var d:Date:=Date.newDate(6,11,2000);
 var plItemNo:integer := lib.getNewItemNo();
 var plCopyNo:integer := lib.getNewCopyNo();
 // add the new book
 lib.acquireBook(t, i, a, p, d);
 // check details of added book
 var plItem:Item := lib.lookupItem(plItemNo);
 assert plItem is Book;
 var plBk:Book := lib.lookupBook(plItemNo);
 assert plItem = plBk;
 // check search functions
 assert lib.lookupItemFromTitle(t) = plBk;
 assert lib.lookupBookFromISBN(i) = plBk; }
```

After a set of tests is executed, the Check Viewer reports which assertions passed (were encountered at least once and evaluated to true all times), failed (evaluated to false at least once) or were unchecked (were not encountered). Below, we show the Check Viewer screen after executing two tests, `bookTest` and `cdTest`. There are some errors present.



To cover all the assertion checks in the Library system, we defined 7 tests, totalling 240 lines of code (excluding comments). Each of these exercises a particular aspect of the system. To get all of the assertions passed, we also needed to correct some errors, described in the next section.

5.1. Errors detected

As one would expect, the RAC process found some errors in the initial implementation. The most common error was missing `old` operators in `ensures` clauses of operations. The `old` prefix is needed to refer to the value of a state element before an operation is carried out. Novice specifiers often forget to use it. For example, consider the original specification of the `disposeOfCopy` operation:

```
public operation disposeOfCopy(cNo:integer)
  requires exists (c:Copy in copies()):
    c.copyNo() = cNo
  changes copies, items
  ensures copies().size() = old copies().size() - 1,
   if (isOnlyOneCopy(
         lookupCopy(cNo).itemNo())) then
    items().size() = old items().size() - 1
   else
    items() = old items()
{ ... }
```

This operation should remove the `Copy` with the specified number and, if it is the last `Copy`, delete the associated `Item`. RAC checking this code yields the following error. (The full error message also gives many other details.)

```
Lib.obs:248: Failure of public requires clause of the lookupCopy function
       declared in Lib at line 780
```

The error arises from the use of the function `lookupCopy` in the `ensures` cause. The `lookupCopy` function is used to search the `copies` collection to find the `Copy` with a given copy number; it requires that that `Copy` is present. The specifier wanted to say that if there was only one copy of the associated `Item` before the operation, then the size of the `items` collection is reduced by one; otherwise it stays the same. However, the call of `lookupCopy`, without an `old` prefix, refers to the library *after* the operation, when the copy has already been removed. This violates the `requires` clause of `lookupCopy`. There is a similar mistake with the call to the function `isOnlyOneCopy`.

There were also a number of genuine mistakes detected by the RAC process. A good example is from the `acquireBook` operation, an extract of which is shown:

```
public operation acquireBook(title:String, isbn:ISBN, authors:List[Name],
       pub:String, pubDate:Date)
  changes items, copies
  ensures
  if (forall (i:Item in old items()
                          where i is Book):
       ((i as Book).title() != title
```

```
      && (i as Book).authors() !=  authors))
 then ...
 else ...
{var countOfCopies:integer := 0;
 foreach (i:Item in items){
  if (i is Book){
   var b:Book := i as Book;
   if(b.title()=title&&b.authors()=authors){
    countOfCopies := countOfCopies + 1;
 }}}
 ... }
```

The intention of this operation is to check for an existing `Book` with the specified title and authors, if there is one, add a new `Copy` of it and if there is not then add a new `Item` and `Copy`. On invoking the RAC checker, we get the following error. Here line 172 is the last line of `acquireBook` and line 1312 is a point in the `bookTest` test where a second `Book` authored by 'Robert Clark' is added.

```
Lib.obs:172: Failure of public ensures clause of the acquireBook operation
       declared in Lib at line 145
Call stack:
 at Lib.acquireBook(Lib:172)
 at Lib.test_bookTest(Lib:1312)
```

The `ensures` clause expresses that both the title *and* the authors list are different from those of all existing books. The implementation iterates through the `items` collection and counts the number of `Book`s with the given title and authors. If this is zero, it assumes the quantification in the specification is satisfied. However, the quantifier should contain a disjunction to describe the intended behaviour.

This error was only picked up because of the comprehensiveness of the full `bookTest` test case. The inconsistency is only exposed when a new `Book` is added with the same title but different authors or same authors but different title as an existing book. It would have been easy to have missed this scenario in our test case, in which circumstance the error would have gone undetected by the RAC process.

We have seen that RAC allowed us to identify a range of errors in the original Library system. However, as the next sections will show, there are other errors which the RAC approach was not able to detect.

6. Applying ESC to the Library System

Next we consider the use of the ESC approach to check the library project. Our starting point is the Omnibus implementation with lightweight specifications used in the previous section. We will see that ESC allows us to uncover errors that were not detected by RAC.

The Omnibus ESC tool is straightforward to use: the programmer simply selects ESC in the Policy Selector and then clicks 'Verify Project'. The tool generates appropriate logical theories and then runs the Simplify prover.

6.1. New errors detected

ESC uncovers several errors and omissions which were not detected by RAC.

In the specifications produced for the RAC approach, we did not give private specifications for the public model functions as these were not needed. For example, the `Item` class included the following public model functions:

```
public model function itemNo():integer
{ return itemNo; }
public model function title():String
{ return title; }
```

The `Item` class contains an operation called `setTitle` which is declared with a `changes` clause expressing that only the `title` model function should be changed, i.e. `itemNo` should not be affected.

```
public operation setTitle(t:String)
 changes title
{ title := t; }
```

On attempting to ESC-check this file we get the following error, where line 20 is the `changes` clause of `setTitle`. ESC is unable to prove that `setTitle` does not change `itemNo`:

```
Item.obs:23: Unable to verify the implementation of the setTitle operation
       respects the public changes clause at line 20
```

The problem is that the ESC tool is unable to deduce the relationship between the `itemNo()` function and the `itemNo` attribute because there is no specification of this relationship. ESC uses

modular checking and can only reason about other methods using their specifications. The relationship between the model function and the attribute is clear from the implementation of the function, but the tool does not refer to this from within the `setTitle` operation.

To fix this problem, we must add a private specification to the public model functions stating the relationship between the public model functions and the private attributes:

```
public model function itemNo():integer
 private ensures result = itemNo
{ return itemNo; }
public model function title():String
 private ensures result = title
{ return title; }
```

These missing specifications are not really a mistake in the original code, more an additional annotation burden for using ESC. We note that it was not clear from the error message that this was the problem.

Missing requires clauses

When producing lightweight specifications, it is acceptable to specify as little as the programmer wishes in the `ensures` clauses. However, the `requires` clauses should completely describe the assumptions which a method makes about the state in which it is called. A method can only reasonably expect that the caller respects its `requires` clause and so if the `requires` clause of a method is met and then an assertion failure occurs during the execution of the method, that is a mistake in the called method.

Consider the `Book` class as described in the previous section and RAC-checked. It contains an invariant stating that there are no duplicates in the list of authors of a book. It also contains a constructor, `newBook`, shown in Section 4.1. If we try to ESC-check this class we get the following error:

```
Book.obs:51: Unable to verify the implementation of the newBook constructor
      maintains the public invariant at line 35
```

The problem is that the `author` list that is passed into `newBook` is not constrained to not contain duplicates. The ESC tool correctly picks up this omission. To fix the problem, we add the following requires clause to `newBook`:

```
 requires !auths.containsDuplicates()
```

Our RAC checks did not detect this missing `requires` clause because our test cases were written to test correct execution for expected values, not graceful failure for invalid inputs. Let us consider what happens if we now add a test case to check the rejection of the addition of a `Book` with duplicate authors:

```
test bookTest {
 ...
 var n:Name:=Name.newName("Robert","Clark");
                          "Wilson");
 var a:List[Name] :=      List[Name].empty().add(n).add(n);
 ...
 lib.acquireBook(t, i, a, p, d);
 ... }
```

Without the `requires` clause in `newBook`, we get the following error when RAC-checking the test case:

```
Book.obs:36: Failure of public invariant declared in Book at line 24 at end
      of newBook constructor
```

So, we get a failure report for the invariant at the end of the `newBook` constructor. Since this is not a `requires` clause failure, we would expect it to indicate an error by the implementer of the `newBook`. However, it is not an error in the implementation, it is the omission of a suitable `requires` clause. Here, ESC is much more accurate than RAC at pinpointing the cause of the error.

Truisms in ensures clauses

A subtle mistake was made in the entry of the `changeFineDue` operation in the `Member` class. The operation was initially defined as follows:

```
public operation changeFineDue(f:integer)
 changes fineDue
 ensures fineDue() = fineDue()
{ fineDue := f; }
```

There is clearly an error in the `ensures` clause, which states vacuously that `fineDue()` is equal to itself. Instead, the specifier meant to say that `fineDue()` is equal to the parameter `f`. Unfortunately, this error is not picked up by RAC since `fineDue() = fineDue()` always evaluates to true so no assertion failure is ever triggered.

This error also escapes detection in the ESC-checking of the `Member` class, again because the assertion evaluates to true when it is checked at the end of the implementation of `changeFineDue`. However, the error is picked up by ESC when we check the `Lib` class, which includes a `changeMemberFine` operation. The lightweight specification of this operation asserts that the `fineDue` for a specified member should be set to a passed value. This is implemented via a call to `changeFineDue`. As ESC uses modular reasoning, it uses the *specification* of `changeFineDue` to reason about how the member's fine is changed. The specification does not contain the information required for this, so an error is reported.

This mistake was not picked up by RAC because RAC interprets other methods by executing their implementations, not by reasoning about their specifications. This property of RAC makes it excellent for working with external components with limited specifications, but here it is a hindrance and prevents the detection of a genuine error.

6.2. Limitations of the ESC tool

The ESC tool is not able to verify the full `Lib` and `Date` classes. If verification is attempted, the prover hangs. The specification of the `Lib` class is too sophisticated to be ESC-checked. The problem with the `Date` class is that it was not completely specified.

6.3. Conclusions about ESC

ESC was able to detect a range of mistakes that RAC did not detect. This is, firstly, because ESC considers the execution of the methods for arbitrary symbolic values, not a specific set of inputs covered by associated test cases. In particular, this helped identify errors in the handling of invalid input values which required additional `requires` clauses. Secondly, when verifying a method, ESC reasons about other methods using their specifications and this can help us find errors in the specifications. However, ESC was not able to verify the entire project because some of the specifications were too complex or incomplete.

7. Applying FFV to the Library System

We now consider the full formal verification of the library project. We will see that FFV requires more comprehensive specifications, and therefore uncovers issues that RAC and ESC did not detect.

There is a key difference between ESC and FFV concerning the way that verification is carried out for constructors and operations. ESC checks only that the implementation satisfies the behaviour specification and requirements specification. FFV, however, generates additional proof obligations that the behaviour specification guarantees the requirements, independently of the implementation. Because of this, FFV requires more expressive, heavyweight specifications of behaviour, instead of the lightweight specifications that are sufficient for ESC (and also for RAC).

FFV is performed using either an automated or interactive theorem prover.

FFV with an automated prover

After corrections, it is possible to verify the specifications of the auxiliary classes (i.e. all the classes apart from `Lib`) with the automated Simplify prover. This involves the proof of 19 VCs which takes 2 seconds.

Verification of the `Lib` class with the automated prover is less successful. The first 7 VCs, corresponding to proving the invariants for a constructor which creates an empty library, are proved in a total of just over 4 seconds; but then the prover hangs whilst trying to verify the main operations. This is not an unexpected result and is consistent with previous attempts to use Simplify for FFV projects [9].

FFV with an interactive prover

To fully verify the `Lib` class we must use an interactive prover. When this option is chosen, the IDE generates PVS files containing the specification of the class and the libraries it uses, the proof obligations, and default proof attempts. These files are then opened separately in PVS and the proof attempts are manually developed into full proofs. The Omnibus IDE consults the PVS proof files to keep track of which proof obligations have been satisfied.

As an example, consider the verification that the `registerMember` operation preserves the invariants. The `registerMember` specification is given below. It is quite straightforward, simply constructing a new `Member` with the passed attributes and adding it to the `members` collection.

```
public operation registerMember(name:Name, ad1:String, ad2:String,
       pCode:String, telNo:integer, isAdult:boolean, isStaff:boolean)
 changes members
 ensures members() = old members().add(
Member.newMember(old getNewMemberNo(), ...))
```

The `memberNo` for the new `Member` is calculated using the `getNewMemberNo` function. The specification of this function says that it returns an integer which is not the `memberNo` of any existing `Member` in the `members` collection.

```
public function getNewMemberNo():integer
 ensures !(exists (m:Member in members()):
         m.memberNo() = result)
```

Now let us consider the verification of the invariants. We assume that the invariants hold before the operation and must demonstrate that they are preserved. The `changes` clause says that only the public `members` model function is changed, so any invariant which does not refer to `members` will be trivially maintained. This covers 5 of the 7 invariants from the `Lib` class. The default proof attempts generated by the IDE are almost sufficient to complete these proofs, with only a few extra steps needed.

Next, we have an invariant that says that every `memberNo` referred to in a loan corresponds to an existing member. This is also fairly easy to demonstrate as we have not removed any members from the `members` collection

Finally, we prove the invariant that there are no two members with the same `memberNo`:

```
 public invariant memberNosInMembersUnique:
  forall (m:Member in members()):
   forall (m2:Member in members()):
    m.memberNo() != m2.memberNo() || m = m2
```

This is more complicated to prove, and relies on the specification of `getNewMemberNo`. The proof consists of 104 steps and involves several side-conditions. The details are described in [15].

7.1. New errors detected

FFV found no errors in the `registerMember` operation, but other parts of the project were found to contain errors or omissions that were missed by the other approaches.

Missing ensures clauses

As part of the ESC process, we had to add a `requires` clause to the `newBook` constructor to ensure the invariant is satisfied (Section 6.1). We did not have to provide an `ensures` clause since the invariant was checked at the end of the implementation of the method. For FFV, however, we must show that the specification alone is sufficient to verify the invariant. If we attempt to verify `Book` using automated FFV, we get the following error:

```
Book.obs:13: Unable to verify the public behaviour of the newBook
    constructor satisfies the public invariant at line 7
```

This is because there is no `ensures` clause to describe how the value of the public `authors` model function relates to the passed `auths` lists at the end of the method. We must add an `ensures` clause to describe this.

```
 ensures itemNo() = iNo, title() = t,
  isbn() = isbnNo, authors() = auths,
  publisher() = pub, pubDate() = pDate
```

Ensures clauses not strong enough

After applying RAC and ESC, we have already made a number of corrections to the `acquireBook` operation. However, there is still a remaining error in the heavyweight specification produced by the novice user. This specification with all the existing corrections is shown below.

```
public operation acquireBook(title:String, isbn:ISBN, authors:List[Name],
       pub:String, pubDate:Date)
 requires !authors.containsDuplicates()
 changes items, copies
 ensures if ... then // if it is a new book
    items() = old items().add(...)
    && copies() = old copies().add(...)
  else
    copies() = old copies().add(...)
```

This specification is not sufficient to verify the invariant that the `itemNo`s in the `items` collection are unique.

```
public invariant itemNosInItemsUnique:
```

```
    forall (i:Item in items()):
     forall (i2:Item in items()):
       i.itemNo() != i2.itemNo() || i = i2
```

The problem is that in the case where there is an existing entry for the `Book`, the new value of `items()` is not described. It cannot be assumed to be unchanged because it is mentioned in the `changes` clause to allow it to be changed when a new `Book` needs to be created. The intention was that, when a new `Book` is not to be added, the contents of `items` should not be changed, but this is not documented in the `ensures` clause. Like many specification errors, this is probably the result of the programmer thinking in imperative terms about the declarative specification.

The problem is solved by changing the else part of the assertion to say explicitly that `items` does not change:

```
  else
    copies() = old copies().add(...)
    && items() = old items()
```

This problem was not picked up by RAC because RAC does not test the completeness of `ensures` clauses. It was not picked up by ESC because, when verifying the `Lib` class, the invariant was checked at the end of the implementation and not from the behaviour specification. The tool would have identified the omission if we had ESC-checked a class that uses the `Lib` class and needed to assert some property about `items` after adding a new copy of a `Book`.

7.2. Conclusions about FFV

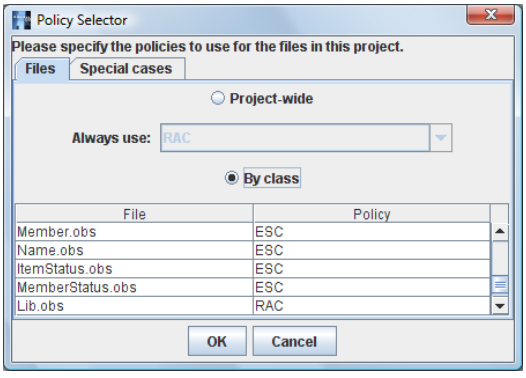
FFV allows the specifications of the system to be improved still further. Richer specifications can be used since the user can guide the PVS prover through proofs that are beyond the automated Simplify prover. However, the costs are much higher. Our experience is that manually carrying out proofs for FFV requires considerable time and mathematical ingenuity and would be beyond many users.

8. Integrated Verification

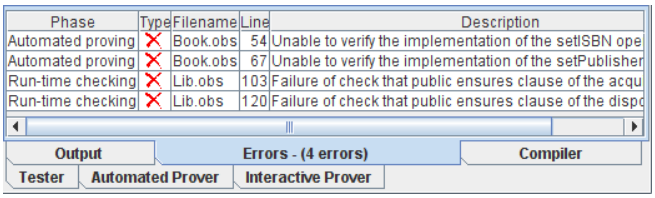
In the preceding sections we considered the use of the different approaches, in turn, to verify the entirety of the library project. In this section we consider how we can use the different verification approaches offered by Omnibus in a more integrated fashion. The question of how to combine the approaches safely is discussed in detail in [13-15] which present a number of guidelines for developers to follow.

8.1. Using different policies for different files

The simplest way to combine the use of the different approaches is to select different verification policies for different classes in the project. For example, ESC was very good at verifying the auxiliary classes in the library project but it could not handle the richer specifications in the `Lib` class or the incompletely specified `Date` class. The Policy Selector permits different policies to be specified for different files. So we could select the RAC policy for the `Lib` and `Date` classes and ESC for the others, as shown below.



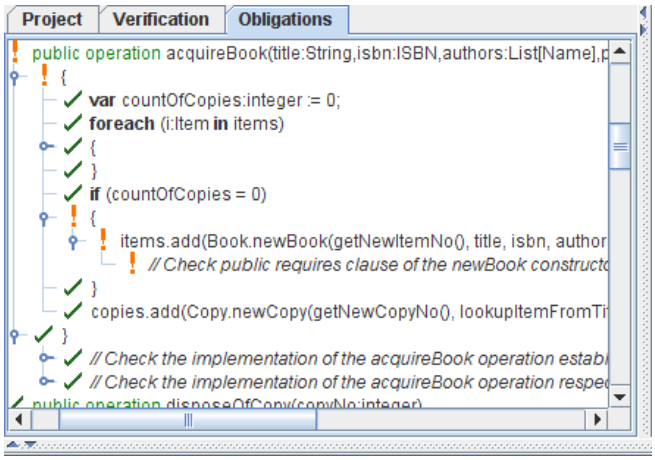
The integrated verification process can be started via the Verify Project option. If the versions of the files before corrections are used then the errors will be detected and displayed together in the Errors tab, as shown below.



If we use the corrected versions, no errors will be reported and the check viewer will report that all run-time assertion checks in the `Date` and `Lib` classes were covered and all the automated VCs for the other classes were proved.

However, combining ESC and RAC in this way violates the guidelines which we previously proposed [13], because ESC policy does not involve the generation of run-time pre-condition checks. The tests used to RAC check the assertions in the `Lib` class use instances of the other classes which were statically verified and have no run-time pre-condition checks. This means the tests may be silently violating the pre-conditions of the other classes, invalidating their verification, which was based on the assumption that the pre-conditions of their methods are respected by calls.

Our Obligation Viewer tool (below) allows this to be detected. For example, the verification of the `acquireBook` operation in the `Lib` class will lead to the Obligation Viewer reporting an unjustified obligation for the check that the pre-condition of the `newBook` constructor is respected when the new `Book` object is created.



To address this problem, our guidelines prescribe that run-time checks must be generated for the pre-conditions of the `Book` class, because it is used in a RAC-verified class. We can do this by changing the verification policy for the `Book` class to be 'ESC with RAC client checks'. This policy includes run-time checks of the pre-conditions as well as statically verifying the implementation. When this change is made, no unjustified correctness obligations are reported.

8.2. Fully integrating the verification approaches

Omnibus provides a number of features to support higher levels of integration between approaches. Here we discuss two of these features: special cases and targeted tests.

The verification policies to be used when verifying a project are usually specified on a project-wide or per class basis, as described in section 8.1. However, targeted tests and special cases allow different verification policies to be selected for specific test cases or individual obligations, respectively. A targeted test is a test declared with a policy clause which names a verification policy to be used to verify the test. Special cases are defined via a Special Cases tab in the Policy Selector by supplying the name of an obligation and the corresponding verification policy to use for that obligation.

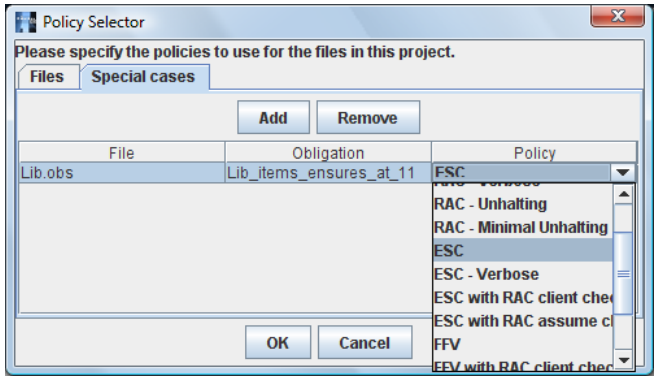
This section describes how these facilities can be used to fulfil various purposes. More examples are given in [15].

Using static verification special cases to omit obligations that are not RAC-compatible

Certain obligations in a RAC-checked project may be too costly to evaluate, either because of their complexity or the frequency with which they are encountered. As an example, consider the `items` model function of the `Lib` class. We may wish to add a private `ensures` clause for this function in order to allow classes that use it to be ESC-checked.

However, in the test cases used to cover all the assertions in the project, the private `ensures` clause of the `items` function is executed 1,531 times, which is costly in execution time and not a particularly smart way to verify the assertion.

In response, we may deem that the assertion is not RAC-compatible [14,15] because it cannot be efficiently run-time checked, and remove it. When using RAC, we must be conscious of the limitations of the verification approach and adjust our specifications to what can be efficiently run-time checked. Of course, then there would be no specification of how the public `items` function relates to the private `items` attribute. This could prevent us from using ESC to verify code that uses instances of the class.



A better option would be to use the Policy Selector (above) to define the verification of the private `ensures` clause of the `items` function as a special case and use ESC to verify it, instead of RAC. The method is easy to statically verify, and this approach enables us to retain the specification while still ensuring that it is respected.

This technique can also be used to statically check obligations that cannot be translated into run-time checks by our tool. For example, obligations involving quantifiers that are not restricted to enumerable ranges could be statically verified as special cases.

Using RAC or FFV special cases to check obligations that are not ESC-compatible

There are limits to what can be proved with automated provers. For example, the automated Simplify prover used by Omnibus is poor at handling recursion, arithmetic and complex quantified expressions. Within ESC we could combat this by adjusting the specifications to remain within the bounds of what the tool can do or by using assumption constructs [13]. Alternatively, we could define the troublesome VCs as special cases and use FFV with interactive theorem proving or RAC with associated tests to justify them. For RAC special cases, additional test cases must be provided in order to execute the special case RAC assertion to give justification that it is valid. By using a RAC targeted test, we can define a test to cover the run-time assertion check, without the burden of also having to ESC-check the test.

ESC-compatibility of RAC- and FFV-checked classes

Specifications developed for RAC and FFV are often not amenable to ESC-checking. For RAC, the problem is generally that the `ensures` clauses are not sufficiently expressive. For FFV, they are generally too expressive. We can use ESC targeted tests to detect these insufficiencies. The ESC-compatibility of the `Book` class introduced in section 4.1 could be checked using an ESC-targeted unit test. This test should use the methods of the `Book` class to create and manipulate a `Book` object. It should be declared with a policy clause specifying that ESC should be used. ESC will then be used to verify the test case, ensuring that the specification is suitable to verify properties of the class.

9. Related Work

JML [4] is another language with tools supporting RAC, ESC and FFV. The tools contain facilities which address many of the limitations of the respective approaches. For example, the JML RAC tool [7] supports the automated generation of unit tests [6]. The ESC/Java2 tool [8] provides feedback on the limits of the automated approach so that users can differentiate between errors and limits of the tool [10]. The KeY tool [1] provides a visual interactive proof environment, making interactive verification more accessible and less costly. These are all separate tools, but it has been proposed [5] to provide built-in support for integration in the next generation of tools for JML.

10. Conclusions and Future Work

Our application of RAC, ESC and FFV, in turn, to the library case study demonstrated that the approaches offered increasing error coverage in return for increasing user burdens such as the need for

more comprehensive specifications and, for FFV, user assistance in the proof process. Our verification policy system enables different approaches to be used for different classes within a single project. When doing this, gaps in the coverage of the assertion checking can arise but our Obligation Viewer tool helps detect these. Our tool also offers the possibility of even greater levels of integration between the different approaches through special cases and targeted tests. This gives great flexibility and means that flaws in the main approach used for a class can be addressed through selective use of other approaches.

Current work on Omnibus is focused on the re-implementation of the Omnibus tools within Omnibus itself. We are developing a formal semantics for the language and using it as a specification for the new type checker, interpreter and static verifier. This work will give us a formal semantics for the language, a rigorously verified toolset and a substantial case study of the Omnibus approach. In the longer term, we propose a number of extensions to Omnibus: disciplined support for reference semantics in the programming language; more libraries and case studies; and further verification options, such as model checking.

References

1. W. Ahrendt, T. Baar, B. Beckert, et al. The KeY tool. Software and Systems Modeling, vol. 4, no. 1, 2005.
2. J. Barnes, J.G. Barnes, *High Integrity Software: The SPARK Approach to Safety and Security*, Addison-Wesley, 2003.
3. K.A. Bell. Developing Reliable Software Using Omnibus. BSc (Hons) thesis, University of Stirling, UK, 2005.
4. L. Burdy, Y. Cheon, D. Cok, et al. An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer, 7(3), 2005.
5. P. Chalin, P.R. James, G. Karabotsos. The Architecture of JML4, a Proposed Integrated Verification Environment for JML. ENCS-CSE-TR 2007-006, Concordia University, 2007.
6. Y. Cheon, M.Y. Kim, A. Perumandla. A Complete Automation of Unit Testing for Java Programs. Software Engineering Research and Experience (SERP '05), 2005.
7. Y. Cheon, G.T. Leavens. A runtime assertion checker for the Java Modeling Language. *SERP 2002*, 2002.
8. D. R. Cok, J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. *CASSIS 2004*, 2004
9. D. Detlefs, G. Nelson, J.B. Saxe. Simplify: A theorem prover for program checking, Tech. Rep. HPL-2003-148, HP Labs, 2003.
10. J.R. Kiniry, A.E. Morkan, B. Denby. Soundness and Completeness Warnings in ESC/Java2. Workshop on Specification and Verification of Component-Based Systems, 2006.
11. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, M.K. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. *CAV 1996*, LNCS 1102, 1996.
12. T. Wilson. Omnibus site. http://www.cs.stir.ac.uk/omnibus/
13. T. Wilson, S. Maharaj, R.G. Clark. Omnibus Verification Policies. *SEFM 2005*, IEEE Computer Society 2005.
14. T. Wilson, S. Maharaj, R.G. Clark. Flexible and Configurable Verification Policies with Omnibus. Software and Systems Modeling, Springer, 2007
15. T. Wilson. The Omnibus Language and Integrated Verification Approach. PhD thesis, University of Stirling, UK, 2007.