Flexible and Configurable Verification Policies with Omnibus

THOMAS WILSON, SAVI MAHARAJ, ROBERT G. CLARK

Department of Computing Science and Mathematics, University of Stirling, Stirling, Scotland

{twi,sma,rgc}@cs.stir.ac.uk

The three main assertion-based verification approaches are: Run-time Assertion Checking (RAC), Extended Static Checking (ESC) and Full Formal Verification (FFV). Each approach offers a different balance between rigour and ease of use, making them appropriate in different situations. Our goal is to explore the use of these approaches together in a flexible way, enabling an application to be broken down into parts with different reliability requirements and different verification approaches used in each part. We explain the benefits of using the approaches together, present a set of guidelines to avoid potential conflicts and give an overview of how the Omnibus IDE provides support for the full range of assertion-based verification approaches within a single tool.

1. Introduction

There are three assertion-based techniques for the integrated specification, implementation and verification of Object-Oriented software: Run-time Assertion Checking (RAC) [12], Extended Static Checking (ESC) [10] and Full Formal Verification (FFV) [4]. RAC and ESC are lightweight approaches which accept programs annotated with lightweight specifications that describe some key properties, but do not attempt to be complete in any sense. In RAC, the lightweight assertion annotations are converted into run-time checks and the application is then tested to uncover assertion failures. The key contribution of RAC to the testing process is the ability to detect errors close to their source, easing analysis and correction. In ESC, the consistency between code and assertion annotations is checked statically allowing errors to be detected without testing. The process depends on the presence of a powerful fully-automated theorem prover such as Simplify [5]. ESC is neither sound nor complete, aiming simply to detect assertion violations and a range of common run-time errors. FFV provides support for traditional, heavyweight assertion-based verification. The approach requires the production of heavyweight specifications (also known as

Behavioural Interface Specifications) for all components being verified and a range of code annotations such as loop invariants. Again, the code is statically analysed and verification is performed with the use of either a fully-automated or interactive theorem prover.

Traditionally, support for these approaches has been developed by separate teams, yielding separate tools often targeting separate languages. The Java Modelling Language (JML) [9] has provided a common language for a range of lightweight and heavyweight assertion-based verification tools but those tools are still developed and used separately. Even comparative case studies from the JML group have used the tools independently rather than in an integrated fashion [8]. Other tools such as Spec# [1] and PerfectDeveloper [4] combine aspects of RAC and FFV but until now there has been no tool supporting the full range of approaches in an integrated fashion.

Such a tool is desirable since the different assertion-based verification approaches provide different balances between rigour and ease of use, matching the different balances between reliability requirements and development resources in different parts of a software development project. The approaches also have neatly complementing strengths and weaknesses. The lightweight RAC and ESC approaches allow reliability to be improved without requiring prohibitive investments in time and effort, but ESC breaks down somewhat in the presence of external components and RAC requires the use of testing to uncover failures. In contrast, FFV is capable of sufficiently describing and statically verifying external components but its costs cannot typically be justified unless the component is going to be reused in different projects or reliability is critical.

We are interested in allowing the approaches to be used together in an integrated fashion within different parts of the same project. We will show, using an example, how using the approaches together is helpful for verification, identify some key challenges in using the approaches together and present sets of guidelines to assist the user in selecting the right approach for each section of the project and avoiding conflicts between the approaches. We will also outline a range of ways in which the approaches can be configured, from simply adjusting the range of checks that are performed to strategies for handling constructs that cannot be converted to efficient run-time checks or cannot be handled by the corresponding theorem prover.

We have developed a new Object-Oriented language called Omnibus [16,18,19] designed to be superficially quite similar to Java but easier to reason about formally. The main simplification made by the language is the use of value semantics by default for objects. The language is supported by the Omnibus Integrated Development Environment (IDE) which includes file and project management facilities, a type checker, a documentation generator, a Java code generator incorporating support for dynamic assertion checks, a unit testing framework and a static verifier which supports interactive and automatic verification.

The examples in this paper are presented in the Omnibus language but the ideas we discuss could be equally well applied to approaches based on other languages such as JML. We also used ESC/Java2 [3] to test Java equivalents of the Omnibus ESC examples.

The Omnibus IDE allows developers to define verification policies and assign different policies to each file in a project. The built-in policies are RAC, ESC and FFV along with a number of variations, but the user can easily customize or extend these to create a verification framework to meet their exact needs. The code generator and static verifier modules then use these settings to guide them.

Section 2 gives a brief overview of the Omnibus language. Section 3 explains why and how the different assertion-based verification approaches can be used together. Section 4 discusses some problems that arise in combining the approaches and gives guidelines to help to avoid these problems. Section 5 presents recommendations on how to choose a verification policy for an application. Section 6 discusses some useful ways in which the Omnibus IDE allows the verification approaches to be configured. Section 7 presents a brief overview of the Verification Policy Manager tool. Section 8 discusses related work and Section 9 presents some conclusions and future work.

2. The Omnibus Language

Omnibus is a new Object-Oriented language designed to be amenable to formal analysis [16]. It is superficially similar to Java, using similar concepts of packages, classes, methods, expressions, statements etc, but incorporates a behavioural interface specification language and uses value semantics for objects.

The use of value semantics by default removes the need for developers to use a range of complex annotations to marshal the complexities of reference semantics.

An Omnibus application consists of a set of *class definitions*. Each class contains a range of methods for manipulating instances of the class. There are three main types of method declaration: *constructors*, *functions* and *operations*. Constructors allow objects to be created, functions allow objects to be queried without side-effects and operations allow objects to be updated. The declaration of a method starts with a keyword identifying the type of method. 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. In deep equality, two objects are equal if they are instances of the same class and all the corresponding elements of their abstract state are (recursively) equal.

Behaviour specifications: The behaviour of methods can be described using behaviour specifications which are constructed from `requires`, `changes` and `ensures` clauses which give pre-conditions, frame conditions and post-conditions, respectively. A subset of the functions in the class is taken to represent the abstract state of the class. These functions are called *model functions*, and are similar in concept to the base functions of Eiffel [12]. They are declared with the `model` modifier and do not have post-conditions. The behaviour of the other methods (the remaining functions along with the constructors and operations) is then defined in terms of them. When specifying operations, the `changes` clause should be used to describe which model functions have their values changed and the `ensures` clause should be used to describe how they are changed.

Method calls can be used within specifications and other assertions but where they are used, the pre-conditions of the called methods must be respected. This will be checked at run-time in RAC, as in all RAC tools, but also by our static ESC and FFV approaches to ensure a consistent handling of these methods. This addresses a key problem raised by Chalin [2] where, in the static verification approaches associated with JML, the pre-conditions of method calls in specifications were not checked.

The Omnibus language supports the use of lightweight behaviour specifications for RAC and ESC and heavyweight behaviour specifications for FFV. There is no formally quantifiable difference between lightweight and heavyweight specifications. The difference is in the mindset of the writer. When writing a heavyweight specification the developer should attempt to describe the interface of the component as completely as possible with the assertion language. This will typically involve the use of quantifiers and recursion. When using lightweight specifications, practical concerns must be kept at the forefront of the developer's thinking in order to keep run-time checking overheads down or theorem proving difficulty within the scope of automated proving. This will require constructs such as quantifiers and recursion to be used sparingly. The completeness of post-conditions and frame-conditions can be compromised but, even in lightweight specifications, the pre-conditions should be described as completely as possible so that it is always possible to identify when a method call is invalid.

Requirements specifications: The requirements of a class are specified using `initially`, `invariant`, and `constraint` assertions. The `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: The implementation of the class is defined in terms of method implementations that manipulate the values of the private *attributes*. Each of the model functions must be described at the private level in terms of the attributes. Loops can be annotated with loop invariant assertions.

Libraries: Like JML, Omnibus hides mathematical abstractions like sequences and sets behind a façade of library classes. Users interact with these classes through methods just like any other class, and do not need to learn additional mathematical notation to manipulate them. For example, the arrays, lists and comparators in our main example are all expressed as Omnibus classes.

3. Combining assertion-based verification approaches

In this section we show why it is useful to use the approaches together in an integrated fashion. We start by selecting one approach, ESC, and discuss some of

the reasons why it has been relatively well received. We then illustrate a key flaw in the approach and show how the other approaches help address it.

3.1 Strengths of ESC

The ESC approach provides a push-button technique to statically detect a range of possible run-time errors and violations of lightweight specifications. It provides better error coverage than conventional type checking, and allows problems to be uncovered earlier in the software development cycle than RAC when they are cheaper to correct, without the need to use testing. Furthermore, it requires considerably less effort to use than FFV.

ESC tools feel like type checkers to use, producing type checker-like error messages that developers are generally more receptive to than traditional formal methods. The approach uses lightweight interface specifications that allow design decisions to be documented and warns of inconsistencies between these annotations and the code. While appearing like type checkers to the user, the implementations of ESC tools have more in common with traditional formal verification tools, utilizing fully automated theorem provers behind the scenes. The approach is neither sound (so it can miss errors) nor complete (so it can report spurious warnings) but is relatively easy to use, fast and powerful. ESC has been relatively well received [10] and appears to hold great promise.

3.2 A Problem with ESC

There is, however, a critical problem with ESC: the lightweight specifications developed while using ESC to check a particular class may not be sufficient as a basis for later static modular checking of classes that use this class. In this section we will illustrate this problem by presenting an example, showing how the ESC approach is used to the point where it breaks down, show the facilities ESC provides for coping with this situation and then investigate how RAC and FFV handle the problem.

When applying ESC to a particular class, the process typically starts by taking an unannotated or partially annotated piece of code and using an ESC tool to check for errors. This will usually yield a number of warnings indicating a combination of bugs in the code and incompleteness in the specifications of the current class or a class it uses. The user will then enter into a cycle of correcting code and adding

annotations to address the issues raised until the tool processes the class without warnings.

Consider the following Omnibus example adapted from the classic example presented in the founding paper on ESC/Java [7]. It represents a `Bag` class which is constructed from a `List` of elements of which the minimums can subsequently be removed, in turn, using the `extractMin` operation. Comparisons are carried out using a passed `Comparator` object. An `Array` is used to store the elements, with the first `size` positions in the array `arr` containing the elements currently in the `Bag`. The constructor copies all the elements from the passed `List` into the `Array` and sets `size` to the size of the passed `List`. The `extractMin` operation calculates the index and value of the minimum element and then copies the last element in the `Array` to that index, reduces the size by one and returns the minimum element. We will add annotations later as needed to respond to errors reported by our ESC tool.

```
 1: public class Bag[Element] {
 2:       private attribute arr:Array[Element]
 3:       private attribute size:integer
 4:       private attribute comparer:Comparator[Element]
 5:
 6:       public constructor containing
 7:                         (input:List[Element],
 8:                          comp:Comparator[Element])  {
 9:            comparer := comp;
10:            size := input.size();
11:            arr := Array[Element].ofSize(size);
12:            var i:integer := 0;
13:            foreach (e:Element in input) {
14:                 arr.assign(i, e);
15:                 i := i + 1;
16:            }
17:       }
18:
19:       public operation extractMin(out min:Element) {
20:            var minIndex:integer := 0;
21:            min := arr.access(0);
22:            for (i:integer := 1 to size-1) {
23:                 if (comparer.compare(arr.access(i),
24:                                      min).isBefore()) {
25:                      minIndex := i;
26:                      min := arr.access(i);
27:                 }
28:            }
29:            size := size - 1;
30:            arr.assign(minIndex, arr.access(size));
31:       }
32: }
```

Applying ESC to this example using our Omnibus IDE verification tool yields the following warnings:

```
Bag.obs:21: Unable to verify public requires clause of the
access function declared in omni.lang.Array at line 6
Bag.obs:21: Unable to verify 'arr.access(0)' is not null
Bag.obs:23: Unable to verify public requires clause of the
access function declared in omni.lang.Array at line 6
Bag.obs:23: Unable to verify 'arr.access(i)' is not null
Bag.obs:30: Unable to verify public requires clause of the
access function declared in omni.lang.Array at line 6
```

These warnings expose undocumented design decisions. For example, the first of the errors at line 23 warns that `i` may not be a valid index of the `arr` array. We know from the loop condition that to reach these lines `i` must be less than or equal to `size-1` but there is no known connection between `size` and `arr.length()` and so we cannot tell whether `i < arr.length()`. The reader may argue that the implementation of the constructor together with the implementation of the `extractMin` operation should be sufficient to deduce this but our ESC tool uses modular checking of methods. When reasoning about the `extractMin` operation, it can reason about the other methods in the class using only their specifications. The developers of the ESC/Java tool adopt the same position since modular checking is essential if the approach is to scale [7].

In order to get the Omnibus ESC tool to accept the `Bag` class we need to add the following invariants to formalize the intended relationship between `size` and `arr`, and a pre-condition that the `Bag` must be non-empty before `extractMin` can be used. The invariants must be established by the end of the body of the `containing` constructor, can be assumed at the start of the `extractMin` operation and must be re-established by the end of the body of `extractMin`. To describe the pre-condition of `extractMin` we need to make the size of the `Bag` publicly accessible by introducing a new public function to return it.

```
private invariant size >= 0 && size <= arr.length()
private invariant forall (i:integer := 0 to size-1):
                        arr.access(i) != null
public model function size():integer
    private returns size
public operation extractMin(out min:Element)
    requires size() > 0
```

These alterations permit the code to pass the checks performed by our ESC tool and so the user can move on to another class. Consider the `IntegerSorter` class shown below which uses an instance of the `Bag` class to sort a `List` of integers. It

starts by constructing a `Bag` from the passed `List` and an initially empty `List` named `sortedInts` into which the sorted values will be put. It then repeatedly extracts the minimum from the `Bag`, adds it to the `List` until the `Bag` is empty at which point it returns the `sortedInts`. The function contains an `ensures` clause asserting that the `sortedInts` list is of the same size as the input list.

```
 1: public class IntegerSorter {
 2:      public static function sort
 3:                              (ints:List[Integer])
 4:                                  :List[Integer]
 5:                   ensures result.size() = ints.size() {
 6:           var b:Bag[Integer]
 7:                := Bag[Integer].containing(ints,
 8:                     DefaultIntegerComparator.init());
 9:           var sortedInts:List[Integer]
10:                := List[Integer].empty();
11:           while (b.size() > 0) {
12:                var m:integer;
13:                b.extractMin(out m);
14:                sortedInts.add(m);
15:           }
16:           return sortedInts;
17:      }
18: }
```

Applying ESC to this example yields the following warning:

```
IntegerSorter.obs:16: Unable to verify public ensures
clause of sort function declared at line 5 holds at return
statement
```

The tool is unable to deduce that the size of the returned `List` is equal to the size of the passed `List`. This is because the specification of the `Bag` class does not explain how the `containing` constructor and `extractMin` operation alter the `size` of the `Bag`. Once again, this is a product of the fact that, in the modular checking process, methods can only reason about other methods using their specifications.

We have now hit the problem: the lightweight specification we developed through verification of the `Bag` class is insufficient as a basis for the modular checking of our new class `IntegerSorter`.

In ESC there are two techniques we can use to cope with this eventuality:

1. iteratively increase the detail of the original specification or
2. use assumption constructs.

Iteratively increasing the detail of the original specification: The first and most obvious approach is to return to our `Bag` specification and add to the specification

the details the tool needs to verify the new class. In this case, we simply need to describe how the `containing` constructor and `extractMin` operation change the `size` of the `Bag`. This is made easier by the fact that we have already defined a public `size` function. Hence we can alter the headers of `containing` and `extractMin` in our `Bag` class to be:

```
public constructor containing(input:List[Element],
                              comp:Comparator[Element])
    ensures size() = input.size()

public operation extractMin(out min:Element)
    requires size() > 0
    ensures size() = old size() - 1
```

With these additional annotations, the ESC tool is able to successfully deduce that the sizes of the input and returned lists in the `IntegerSorter.sort` method are equal.

While this has solved the original problem, there is still a host of related problems lying in wait. The problem was triggered by the `ensures` clause of the `IntegerSorter.sort` method which stated that the sizes of the source `List` and the sorted `List` should be equal. Of course, we might want to more completely characterize this method. For example, the entries in the result should be ordered and result should be a permutation of the input. If we wanted to include such things in the `ensures` clause of `IntegerSorter.sort` then we would need to further augment the `Bag` specification to describe how the contents are manipulated and how the values returned relate to them.

This approach assumes that the users of the tool are developing the classes being analysed themselves or at least have access to the original source code in order to deduce and add assertion annotations. However, this may not always be the case. Realistic applications are made up of code written specifically for the application being developed, components reused from built-in libraries and possibly components produced by third-party component vendors. If the specifications of the external components (i.e. the components developed by others) are insufficient as a basis for the modular checking then iteratively increasing the detail of the specification in this way is not an option.

A variation of this technique could be used for external components where specifications are provided “out-of-band” i.e. separately from the component [1]. So whereas the client may not have access to the inner details of the external

component, they do have read/write access to the specification of the component. However, without access to the original code, they have no way of determining the subtleties of exactly how the implementation handles different circumstances and no way of checking their best guesses.

Use of assumption constructs: The other major technique which we can use to get around this problem in ESC is to make use of assumption constructs. One such assumption construct is the `assume` statement which permits the user to give an assertion to be added to the system's current knowledge without further justification. So, for example, when we found that the specification of the Bag class was insufficient to verify that the sizes of the input `List` and sorted `List` are equal, we could simply have added an `assume` statement to say that they should be. Such a statement could be placed just before the `return` statement and would appear as follows:

```
15a: assume sortedInts.size() = ints.size();
```

This would then enable the system to deduce the truth of the `ensures` clause at the end of the method.

The problem with assumption constructs is obvious: they are, as their name implies, assumptions without formal justification within the system. They are unsound and allow the user to circumnavigate the entire checking process. However, they do allow the user to suppress spurious warnings and can be used in conjunction with external components as well as pieces of code written by the developers themselves. Furthermore, while there is no basis for the assumptions within the system, the user can base them upon informal information such as the names of the class and methods, associated interface documentation and domain information. They can then include descriptions of informal justification as comments within the code. We will also see how they can be used in conjunction with RAC.

3.3 Plugging the gaps in ESC with RAC and FFV

Let us now consider how the other approaches can help address this problem.

Incorporating RAC: RAC is another approach that we can use to verify the correctness of an application relative to lightweight specifications. We can take our `Bag` and `IntegerSorter` classes with their specifications, generate an implementation incorporating run-time assertion checks and then test it to detect

failures. The key advantage of RAC in tackling this problem is that while in ESC the specification of a class forms the sole basis for verification of its use in classes that use it, in RAC the implementation can be used to check properties that were not described by the specification. Taking our example, the incompleteness of the lightweight specification of the `Bag` class would not cause a problem for the verification of the `IntegerSorter` class using RAC. This is because although the specification does not explicitly describe how the size changes, the implementation does and that is what is used to check assertions in RAC. Another way of looking at this is that RAC never gives a warning unless there is a run-time error or a violation of an assertion, whereas ESC also reports warnings if the specifications are insufficient to perform modular static checking. So RAC doesn't suffer from this crucial limitation although it has its own limitations (like the need to be associated with a testing strategy in order to detect assertion failures) that prevent it being an ideal replacement.

We have seen how assumption constructs are a useful way to provide additional information about components although they had no formal basis within the ESC system and so could be used to circumnavigate the verification process. However, while the assumption constructs cannot be verified statically, they can be converted into RAC checks. Thus ESC can be used to check everything except the assumption constructs and RAC used to check the assumption constructs. This can be applied to our `IntegerSorter` example, the class being verified by the ESC tool, assuming that the `assume` statement holds, something that can then be verified by testing the application with its generated RAC run-time checks. This technique is used by the Spec# tool [1].

Incorporating FFV: Up until this point, we have considered only lightweight specifications and approaches. However, an obvious solution to the problem of lightweight specifications not providing sufficient information is to write more descriptive heavyweight specifications. By using these together with FFV, we can get around the problem, since a heavyweight specification can be sufficiently informative to provide a basis for static modular checking. However, FFV is too costly to form an ideal complete replacement for ESC.

Let us for the moment restrict our attention to reusable components, i.e. components that have been written by another developer, whose source code we have no access to and whose specification we have read-only access to. There has

been a wide range of work on reusable software components. To safely reuse components from external sources we need two things:

1. specifications that sufficiently describe the interface of the component [11], and
2. sufficient basis to trust that the hidden implementation satisfies the specification [13].

The problem is that ESC does not completely address either of these issues. However, FFV can. The heavyweight specifications produced for FFV provide a way of sufficiently describing the interface of a component and its associated verification approach provides a basis for trusting a hidden implementation. Also, while the costs of FFV cannot typically be justified for entire applications, the economies of scale make it a more attractive proposition for reusable components. So suppose the `Bag` class was developed by a third-party component vendor. They could use FFV to fully specify and verify their component, giving a specification with all the information needed to check the `IntegerSorter` class. An outline of a heavyweight specification of the `Bag` class is given below:

```
public class Bag[Element] {
    private attribute arr:Array[Element]
    private attribute size:integer
    private attribute comparer:Comparator[Element]
    ...
    public model function contents():List[Element]
         private returns arr.range(0,size).toList()

    public model function comparer():Comparator[Element]
         private returns comparer

    public function size():integer
         returns contents().size()

    public constructor containing
                          (input:List[Element],
                           comp:Comparator[Element])
         ensures contents() = input,
                    comparer() = comp
    { ... }

    public operation extractMin(out min:Element)
         requires size() > 0
         changes contents
         ensures old contents().contains(min),
              forall (e:Element in old contents()):
                    comparer()
                      .compare(min, e).isBeforeOrSame(),
              forall (e:Element in old contents()):
                    if e = min then
```

```
                        contents().countOf(e)
                        = old contents().countOf(e)– 1
                  else
                        contents().countOf(e)
                        = old contents().countOf(e)
    { ... }
}
```

If we are verifying the `IntegerSorter` class using FFV then the tool can use this heavyweight specification to reason about the `Bag` class. However, the ESC tool is not able to effectively reason about it because of its use of the recursive `countOf` method. We will look at how we can ESC-check the `IntegerSorter` class using this specification in the next section.

4. Challenges in using the approaches together

There are a number of problems that can arise when using the approaches together. These occur when, in using a particular approach to verify a class, we have to reason about the use of a class that was verified using a different approach. In this section we present some guidelines for avoiding conflicts between the approaches. We have included “unverified” as a classification of verification since the application may contain classes that are not verified with any of the approaches and interactions with such classes may be particularly troublesome.

Note that we consider the problems from the point of view of a class using another class, where the usage is strictly directed. For example, the `IntegerSorter` class uses the `Bag` class, but the `Bag` class does not use the `IntegerSorter` class, so we only consider the problems caused by the incompatibilities of the two specifications from the point of view of the `IntegerSorter` class. Of course, it may be possible to have bi-directional usage links and where these occur we treat them as two separate usage links.

4.1 RAC- and ESC-compatibility

A key problem with combining the different assertion-based verification approaches is that not all specifications can be converted to efficient run-time checks and handled by the automated provers used by ESC. We refer to these properties as RAC- and ESC-compatibility, respectively.

RAC-compatibility is the more straightforward to define. Certain specification constructs like quantifiers and recursion can cause problems for run-time assertion checkers. We can check quantifiers at run-time if their variables are restricted to enumerable ranges but, even if we do this, it may not be practical for efficiency reasons. Care must also be taken with recursion so as to avoid non-termination (a potential source of unsoundness in our approach) and situations where evaluation of the assertion describing the intended behaviour of the method is as costly as the execution of the method itself. To ensure RAC-compatibility the developer must ensure that all specifications that need to be checked at run-time can be converted into efficient checks. In the next section we will see that we must consider RAC-compatibility even when we are using one of the other approaches to verify a component.

The Omnibus tool is able to check whether assertions are executable by ensuring they don't use certain constructs like quantifiers without enumerable ranges. The efficiency aspect is trickier. Test harnesses can be used to help assess the efficiency but what is acceptable will be dependent on the context. Run-time overheads may prevent the use of assertion checks in certain final products, but they may still be useful in pre-release testing.

ESC-compatibility is more complex to define. Firstly, certain constructs like recursion typically cannot be handled by automated provers and so are not ESC-compatible. However, automated provers can run into problems even when using specifications that do not use such features. Although quantifiers can usually be handled, complicated combinations will often defeat them. Every prover will have things that it can handle well and, as a consequence of undecidability of the problem, must also have things that it handles poorly. A possible approach is to define ESC-compatibility relative to the specific theorem proving capabilities of the ESC tool being used. To determine whether a specification is ESC-compatible for a particular tool, we must experiment with that tool, and the Omnibus IDE supports the definition of test harnesses which can be used to carry out this experimentation. A problem with this definition of ESC-compatibility is that it is implementation dependent, which will become more of an issue as the community moves towards interchangeable provers.

4.2 Guidelines for avoiding conflicts when using the approaches together

We have developed a set of guidelines to help developers avoid conflicts between the approaches. The automated checking of these guidelines is discussed in section 7.

1. All pre-conditions should be expressed in terms that are compatible with all of the approaches. Incompatible pre-conditions should be rewritten using new functions. Test harnesses can be used to help check RAC- and ESC-compatibility.
2. All classes should include run-time checks of their pre-conditions unless it is absolutely clear that unverified or RAC-verified code will never directly use them.
3. All heavyweight specifications that cannot be handled by ESC should provide lightweight ESC-compatible substitutes via redundant specifications. ESC-compatibility should be checked using test harnesses.
4. The system should be structured so that ESC-verified classes never have to directly reason about unverified classes and FFV-verified classes never have to directly reason about unverified, RAC-verified or ESC-verified classes.

Guideline 1 says that all pre-conditions should be written to be compatible with all of the approaches. As was discussed earlier, to establish this we must ensure that they are RAC- and ESC-compatible. To do this the developer should attempt to write the pre-conditions in a suitable form e.g. avoiding the use of recursion and providing enumerable ranges for quantified variables wherever possible, allowing them to be converted into run-time checks. However, to sufficiently describe the pre-condition of a method, it may be necessary to use assertions that cannot be handled by these approaches.

As a somewhat artificial example, consider the following extract of a `Set` class verified using FFV. The public specifications are described in terms of the `contains` and `size` public model functions which are implemented using a private `List` attribute named `contents`. We ensure that `contents` contains no duplicates by using an invariant so that we can calculate the `size` of the `Set` by taking the `size` of the `contents List` and do not need to remove any duplicates

first. We define a `unionOfDisjoints` operation to calculate the union of the set with another set. The pre-condition of the `unionOfDisjoints` operation is expressed in terms of the `contains` model function using a quantifier without an enumerable range and thus the system cannot automatically generate an appropriate run-time check to guard this method. We may wish to write a quantifier to iterate over the `contents` attribute that is used to implement the `contains` model function but we cannot refer to this private attribute in the public specification of `unionOfDisjoints`.

```
public class Set[Element] {
    private attribute contents:List[Element]

    private invariant "No duplicates in contents":
         !(exists (i:integer := 0 to contents.size()-1,
                   j:integer := 0 to contents.size()-1):
              i != j
              && contents.elementAt(i)
                        = contents.elementAt(j))

    public model function contains(e:Element):boolean
         private returns contents.contains(e)

    public model function size():integer
         private returns contents.size()

    ...

    public operation unionOfDisjoints(s2:Set[Element])
         requires "Sets must be disjoint":
                   forall (e:Element):
                    !(this.contains(e) && s2.contains(e))
         changes contains
         ensures
          contains(e) = old contains(e) || s2.contains(e),
          size() = old size() + s2.size()
}
```

We can combat this problem by re-writing pre-conditions involving unconstrained quantifiers using separate new functions that are described using a method with the troublesome assertion in its post-condition and an implementation defining how to implement the check. This solves the problem of RAC-compatibility since the dynamic checks will use the implementation of the new function to check the pre-condition whereas the static approach can use the post-condition of the new function described using the unexecutable assertion. What we have exploited here is that RAC only requires that the pre-conditions and the implementations of called classes can be executed.

For example, in the `Set` class we could introduce an `isDisjointTo` function and define the pre-condition of the `unionOfDisjoints` operation in terms of it. The system will then be able to convert the pre-condition of the `unionOfDisjoints` operation into a run-time check, using the implementation of the `isDisjointTo` function. Run-time checks do not need to be generated for the post-conditions in the `Set` class since the verification of the `Set` class using FFV will prove these, provided that the pre-conditions hold at the start of the method (which the run-time checks ensure). The FFV verification of the `Set` class can still reason about the pre-condition of the `unionOfDisjoints` operation in terms of the unconstrained quantifier which now appears in the post-condition of the `isDisjointTo` function. Furthermore, the FFV verification of the `Set` class will require the developer to prove that the `isDisjointTo` function satisfies its post-condition, ensuring that the developer has implemented the check properly.

```
public operation unionOfDisjoints(s2:Set[Element])
      requires "Sets must be disjoint":
                  this.isDisjointTo(s2)
      changes contains
      ensures
       contains(e) = old contains(e) || s2.contains(e),
       size() = old size() + s2.size()
{ ... }

public function isDisjointTo(s2:Set[Element]):boolean
      ensures result <==>
                        (forall (e:Element):
                             !(this.contains(e)
                             && s2.contains(e)))
{
      // Check all elements in 'this' are not in 's2'
      foreach (e:Element in contents) {
            if (s2.contains(e)) {
                  return false;
            }
      }
      // Check all elements in 's2' are not in 'this'
      foreach (e:Element in s2.contents) {
            if (contains(e)) {
                  return false;
            }
      }
      return true;
}
```

This approach can also be used to ensure ESC-compatiblity when used in conjunction with guideline 3. A new function can be introduced to move a non ESC-compatible assertion from the pre-condition of the original method to the

post-condition of a new method where redundant specifications can be used to provide a partial, ESC-compatible description of the method. Additional assumptions about the value of the method can then be easily formulated in the client code as appropriate.

Guideline 2 says that all classes should, in general, include dynamic checks of their pre-conditions because, while calls made from statically verified code should satisfy the pre-conditions, calls from RAC-verified or unverified code may not. Hence, ESC- and FFV-verified classes that were verified under the assumption that their pre-conditions are always respected, may have that assumption broken. This may lead to a run-time error or assertion failure being generated within the execution of the statically verified class, even though the cause of the error was the silent violation of the pre-condition by the calling class, and not a fault in the implementation of the statically verified class. To guard against this we must ensure that any methods called by unverified or RAC-verified classes must have their pre-conditions checked at run-time so that invalid calls of these methods can be identified and reported correctly to the user. If, however, the class being considered is for use only within this system and all the classes that use the class have been statically checked then, for efficiency reasons, the checks can be omitted.

Guideline 3 says that non ESC-compatible heavyweight `ensures` clauses should have lightweight ESC-compatible redundant specifications. Redundant specifications are generally used to express properties that should follow from the standard specification. Our idea is that if the post-condition of a method in a FFV-verified class cannot be handled by ESC, then a redundant specification is provided giving a lightweight specification that should follow from the original specification but does not have to be complete in any sense. For example, while the heavyweight `Bag` specification in Section 3.3 provides all the information necessary to verify our `IntegerSorter` class, its use of the recursive `countOf` method means the ESC tool cannot effectively deduce that the size of the `Bag` is decreased by one in the `extractMin` method. We can provide this information in a form accessible to the ESC tool via a lightweight redundant specification like the one shown below.

```
public operation extractMin(out min:Element)
    requires size() > 0
    changes contents
```

```
ensures ...
which ensures size() = old size() - 1
```

In this case, we discovered that the heavyweight `Bag` specification from section 3.3 was not ESC-compatible when we attempted to verify the `IntegerSorter` class using ESC. Alternatively we could have written special test harnesses within the `Bag` class to check its ESC-compatibility.

Now, if we are using FFV to verify the use of a class that was verified using FFV then we would use its original heavyweight specification and, if we are using ESC to verify the use of a class that was verified using FFV, we would use its lightweight redundant specification. Of course, the lightweight redundant specifications suffer from the same incompleteness problems as any other lightweight specifications. Normally, when using ESC, the lightweight specification and the code is all we have and if we don't have access to the code then we have to use assumption constructs based on informal domain knowledge or interface documentation. However, in this situation the developer can refer to the heavyweight specification in order to determine if assumption constructs are valid. These justifications could be informally documented in comments within the code or formally verified with FFV.

In our experience, the majority of methods can be written to be ESC-compatible and, for those that cannot, redundant specifications are generally easy to write and are useful to verify some desired properties of the specification.

Guideline 4 describes some combinations of approaches that should be avoided. If the entire application has been developed by a single person then it is always possible to structure the application so that these combinations are avoided. We can achieve this by either increasing the level of verification used for the client class or decreasing the level of verification for the supplier class. For example, it is not allowable to use FFV to verify the `IntegerSorter` along with ESC for the `Bag` class but we can either increase the verification of the `Bag` class to FFV or decrease the verification of the `IntegerSorter` class to ESC. The situation may be complicated by the relationships with other classes in the system. For example, there may be another class in the system that uses `Bag` and is also verified using FFV in which case we should probably increase the verification of `Bag` to FFV rather than decrease the verification of `IntegerSorter` to ESC. In the extremes we will be forced to increase the verification level of the entire application to be

FFV or decrease it to RAC/ESC but, in general, it will be possible to have intermediate levels of verification. We will discuss this further in section 5.2.

While we can always follow guideline 4 if we have developed all the classes in the application ourselves, applications will usually make some use of external components either from built-in libraries or component vendors. These pose a problem since we cannot change their level of verification to fit the verification levels of the other classes in our application. Let us suppose that the `Bag` class is an external component. If it was verified using ESC then we are constrained to use ESC/RAC to verify the `IntegerSorter` class. However, if it was verified using FFV then, assuming guidelines 1, 2 and 3 are followed, we are free to use any one of RAC, ESC or FFV to verify our `IntegerSorter` class. We view this as a sufficient justification to favour the use of FFV for reusable components wherever possible [17].

The following table enumerates the allowable calling relationships between classes verified using the different approaches along with the corresponding constraints. This table is read as follows. Suppose, for example, that we wish to work out whether, when verifying the `IntegerSorter` class using ESC, we can use a version of the `Bag` class verified using FFV. To do this, we look up the ESC row and FFV column and find we can, providing that non ESC-compatible post-conditions have lightweight redundant specification substitutes.

Approach	**Can use classes verified using**			
	None	**RAC**	**ESC**	**FFV**
Unverified	Y	Y	Y[1]	Y[1]
RAC	Y	Y	Y[1]	Y[1]
ESC	N[2]	Y[3,4]	Y[4]	Y[5]
FFV	N[6]	N[6]	N[6]	Y

Constraints:

1. Providing that pre-conditions are dynamically checked and don't contain non RAC-compatible constructs like unconstrained quantifiers.
2. Unless all calls of the methods of this class are guarded by appropriate assume and assert statements. This will be cumbersome unless the number of calls is small.

3. Providing that the specifications are ESC-compatible.
4. Using assumption constructs or, if the class is not an external component, iterative strengthening of specifications to handle incompleteness problems.
5. Providing non ESC-compatible post-conditions have lightweight redundant specification substitutes.
6. Unless the FFV approach is weakened to allow assumptions (as discussed in section 6) and all calls of the methods of this class are guarded by appropriate assume and assert statements. This will be cumbersome unless the number of calls is small.

5. Recommendations for when to use the different approaches

The preceding section described the ways in which it is possible to use the approaches together. It described which combinations are invalid and the constraints that the combinations of the other approaches must satisfy in order to be valid. However, for realistic systems, there will still be multiple ways in which the system could be divided up into sections where different approaches are used. While there are no definite rules on how to select a particular verification approach for each section, and part of the point of our work is to give the developer flexibility to make such choices themselves, we have developed some general guidelines to assist the developer. Developers must also be conscious of the consequences of their choices of verification approaches for the different classes in their application and we also discuss this.

5.1 General guidelines

When to use ESC: Our opinion is that ESC, together with dynamic checking of `assume` statements, is the best approach for the majority of classes. We recommend the use of ESC unless the class being verified falls into one of the categories described below.

When to use RAC instead of ESC: ESC requires considerable up-front effort in order to provide the tool with enough annotations to perform static modular checking. Even if a system has no bugs, the ESC tool will keep reporting warnings until the developer has provided sufficient assertion annotations. If this

up-front investment is not practically possible then RAC can be used instead. In RAC, the verification effort is carried out in the traditional testing phase.

When to use FFV instead of ESC: There are two main situations where a developer might prefer FFV to ESC: (1) to verify critical classes and (2) to verify reusable components. If the cost of an error in the class is very high then the developer may decide that the additional error coverage provided by FFV, albeit at a considerable cost, make it the most desirable approach. If RAC or ESC is used to verify a reusable component then the specifications will likely be lightweight and relatively incomplete. This will hamper clients using ESC to verify classes that use the component. There may be properties of the component that are needed to verify the client's class that are not expressed in the lightweight specification. Assumption constructs will have to be used to address this because the source of an external component is not available to be iteratively strengthened. This will involve extra work and complicates the use of the components. As a result, if RAC or ESC is used to verify a reusable component then it may be better to dynamically check usage of the component using RAC rather than using ESC since clients using RAC do not need to provide additional assumption constructs when using components with lightweight specifications.

When to use none of the approaches: We strongly recommend that at the very least developers use lightweight specifications to document key design decisions and RAC to report when they are violated. This involves modest additional effort but can greatly assist in debugging and provides a springboard to the use of static approaches.

5.2 Structuring the verification of an application

The decisions on what verification approaches to use for each class in an application cannot be taken in isolation. The verification approach we use to verify a class will have repercussions for the classes that use it. The main hard requirement is that we cannot use FFV to verify a class unless we have used FFV to verify all the classes that it uses. We should also look to avoid using unverified classes in ESC verified classes. So, when we use anything other than FFV to verify a class, we are restricting the verification of the classes that use it to not be FFV and when we do not use any of the approaches we make it difficult to use ESC. Thus it is clear, the interactions between the verification approaches for the

classes in the application are focussed around the *uses* relationships and so it may be useful to produce a diagramatic overview of an application and its *uses* relationships. We can do this by breaking the application into horizontal *levels* where the classes in each level may be used by those in levels above but not by those below. The bottom level will be the built-in libraries which necessarily do not *use* any classes from the application or components from third-party component vendors. Above this will be levels for third-party components which may *use* the built-in libraries but not the classes in the application. Similar levels should be identified within the application itself. For example, in a Model-View-Controller (MVC) application, the model classes would be below the views and controllers. Figure 1 shows an example of how the verification of an MVC application might be structured. This kind of structuring mechanism is also consistent with Rushby's concept of a "safety kernel" [15]. The `Model` in our MVC diagram could be interpreted as a "safety kernel" because of its rigorous verification with FFV.

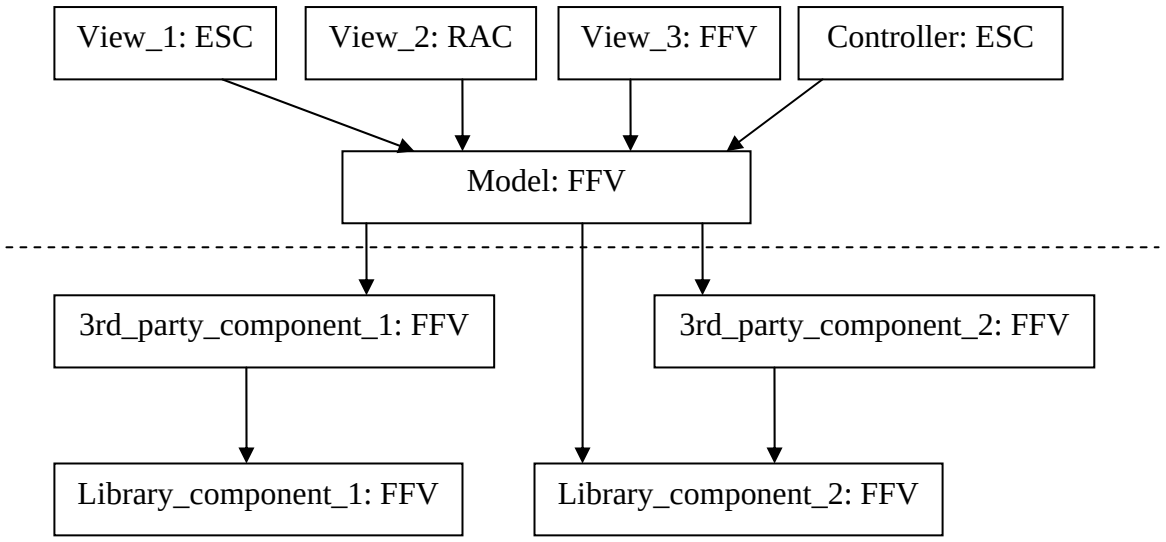


Figure 1: structuring the verification of an MVC application

As was discussed earlier, a software developer has no control over which verification approach was used for library components and third-party components (shown below the dashed line in the figure). The limits this imposes on the other classes are clearly visible in the diagram above. For example, if `3rd_party_component_1` was verified with RAC instead of FFV, then the

developer would not be permitted to use FFV for verifying the `Model` and `View_3`. In order to provide the developer with as much flexibility as possible in configuring the verification of their application, we therefore recommend that FFV should be used for all third-party components and library components [17].

6. Configuring assertion-based verification approaches

In this section we will discuss some ways in which the dynamic and static checking can be configured. Dynamic checking corresponds to RAC while static checking encompasses ESC and FFV.

6.1 Configuring dynamic checking

There are a number of ways in which the dynamic checking process, whereby assertion annotations are converted into run-time checks, can be customized. The most basic customization is the range of checks to perform. The developer may decide that they only want to dynamically check certain types of assertions. The case discussed earlier where only `assume` statements are checked dynamically is a good example. Another example would be reducing the number of checks in order to decrease the time overhead.

It may also be useful to customize the information contained in the assertion failure reports. There is a trade-off between the level of information in the failure messages and run-time efficiency of the compiled executable. By default, the user may want to include just basic information like a description of the failure, the relevant source code and a stack trace. However, if the information in these messages is not sufficient to pinpoint the exact circumstances of a failure then they may prefer to generate more comprehensive error reports including current parameter, attribute and local variable values at the point of failure and detailed information on the execution path (e.g. how many times was each loop executed).

There are a number of constructs in the Omnibus assertion language which are not always convertible into efficient run-time checks (e.g. quantifiers) and others that are never convertible into run-time checks (e.g. iterate assertions which are beyond the scope of this paper). Some quantifier expressions can be converted into run-time checks, e.g. those using only variable declarations associated with a

range of integer values, but those without such restrictions cannot be. Even those that can be converted into run-time checks will likely be expensive to check. The developer may want the system to either ignore (i.e. treat as ‘true’) or prohibit (i.e. generate a type checking error if found) quantifiers that cannot be dynamically checked and may want to ignore or translate quantifiers that can be dynamically checked.

6.2 Configuring static checking

Omnibus provides support for both the ESC and FFV static verification approaches. These are distinct approaches with different aims but the underlying processes they use are strikingly similar. They are both used in conjunction with a theorem prover and start by translating the specifications of the classes and methods used in the application into the logic of the corresponding prover and then generate a range of Verification Conditions (VCs) over these specifications which should be valid if the program is free of the class of errors being checked.

The most immediately apparent difference is that ESC uses relatively lightweight specifications whereas FFV requires relatively heavyweight specifications. ESC also makes a range of compromises in soundness in order to make the approach easier to use. Loops are analysed by unraveling them a finite number of times, instead of requiring loop invariants and proving them inductively. Assumption constructs are permitted to help tackle the incompleteness of the lightweight specifications and automated prover. ESC always uses an automated prover whereas FFV can use either automated or interactive provers. Finally, ESC is a code-centric approach where all verification involves analysis of code, whereas in FFV it is possible to analyse the consistency of specifications independent of any implementation.

As with dynamic checking, the most basic customization is the range of checks that should be performed. The user may want to statically check only certain kinds of assertions. They may also want to verify requirements such as invariants in different ways. Either the behaviour specifications should imply them (our FFV approach) or the implementation should (the ESC approach).

The next most fundamental way in which static checking can be configured is in the choice of theorem prover. Currently our system supports the use of the fully automated Simplify prover [5] and the interactive PVS prover [14]. Which prover

is appropriate is dependent on the skills of the available users and the complexity of the required proof. The selection of the prover has repercussions for other customizations, mainly the handling of certain heavyweight constructs that the PVS prover can handle but which Simplify can't.

The other major configurable options are concerned with the soundness of the process. Firstly, users can specify whether loops are required to have loop invariants provided. If the approach is to be sound then it must. In this case, a type checking error will be generated if a loop without an invariant is found. Otherwise, if a loop invariant is not provided, the loop will be analysed by unraveling it up to a finite number of times that can be configured by the user (defaulting to 1). Secondly, users can specify whether assumption constructs are permitted. If the approach is to be sound it should not permit these.

As with dynamic checking, a number of constructs from the assertion language may cause problems and so it may be useful to specify special handling for them. Recursion can cause the Simplify prover to either enter into an infinite loop or crash and can be difficult to manage even in the PVS prover. As such, the user may want to disallow its use in assertions, use redundant specifications in its place or ignore it.

6.3 Creating new approaches

Through this configuration process, we can create new approaches that combine aspects of the traditional approaches. For example, we could develop different configurations for: (i) FFV, but with the loop unraveling technique from ESC to avoid the need for loop invariants; (ii) ESC and FFV with and without dynamic pre-condition checks; (iii) ESC with and without `assume` statements dynamically checked; (iv) RAC with different levels of failure reports.

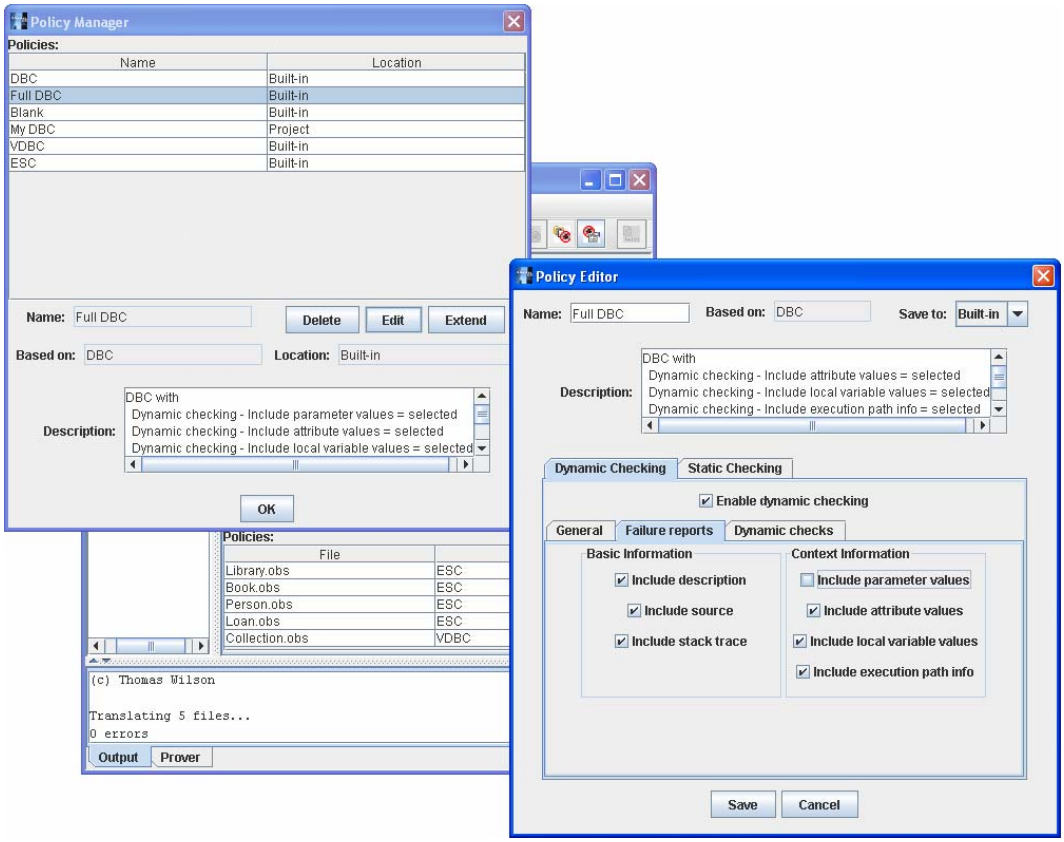
7. Verification Policy Manager

The Omnibus Integrated Development Environment (IDE) supports static verification using fully-automated and interactive theorem provers and the generation of bytecode implementations with assertion checks. Thus, it can be used to support RAC, ESC and FFV.

The tool manages these facilities through the concept of verification policies. A verification policy defines precisely how to manage the verification of the

assertion-annotations in a class. This consists of whether to use dynamic and/or static checking, what checks to generate, the theorem prover to use and various other configuration options. RAC, ESC and FFV are the initial verification policies but the user is also free to combine aspects of the different approaches to create new policies. The policies form a hierarchy with each policy being based on another policy with possibly some changes.

There are three main tools that allow the developer to use policies within Omnibus: the Policy Manager, the Policy Editor and the Policy Selector. The Policy Manager provides a high-level tool for managing verification policies. It displays all the policies currently loaded into the system and provides options to delete them, extend them to create new policies and edit them. Creation and editing of policies is achieved through the Policy Editor, a dialog allowing the developer to completely configure a particular policy. Finally, when a project is opened, a window is loaded allowing, among other things, the verification policy to use when verifying each file in the project to be specified. A screenshot showing these tools is shown.



We are also developing Policy Analyser and Policy Integration Checker tools to assist in the proper definition and use of verification policies. The purpose of our Policy Analyser tool is to ensure that the policies which users define describe the

verification process they desire. It will do this by checking how a policy uses static and dynamic checks together and displaying a summary advising of things such as: the soundness of the policy, how it interacts with other policies and its dependence on testing for the verification of the assertion annotations.

For example, a policy defining the conventional 'ESC' approach would be deemed unsound because of its allowance of assumption constructs and the use of the loop unraveling technique, the user would be warned that because the pre-conditions of methods are not dynamically checked that they may be violated by callers, and no dependence on testing would be reported. If the policy was adjusted to require loop invariants for all loops and dynamically check assumption constructs and pre-conditions then the policy would be deemed sound, but with a dependence on testing to check the assumption constructs.

We are also developing a Policy Integration Checker tool, which will check the structuring of the verification of a project for compliance with the guidelines of Section 4.3. Guidelines 2 and 4 are concerned with the policy settings of a project and the uses relationships between the classes in the project. The policy settings for each file in a project can be retrieved from the Policy Selector and the uses relationships can be calculated during the type checking of the project. Compliance with guideline 2 within a project can be checked by ensuring that policies used for statically verified files have the 'Generate run-time pre-condition checks' setting enabled unless they are only called from other classes in the project that are statically verified. Compliance with guideline 4 can be similarly checked by examining the uses relationships and policy settings.

The above approach checks the guidelines in a non-modular fashion within the closed environment of the enclosing project. However, the classes in a project may be reused in other projects in a way which violates guideline 2 and, in our system, the burden is on the supplier of the component to build in those checks. Run-time pre-condition checks should be generated if a class may be reused within some other project by classes that are RAC-verified or unverified, even if there are no such classes that use it in the current project. To widen the checking to include checks for reuse across projects it would be useful to have some mechanism for differentiating classes that are intended for reuse by other projects from classes not intended to be reused. The system could then check for

compliance with guideline 2 in the presence of reuse by checking that all classes intended for reuse also have their pre-conditions checked at run-time.

Guideline 4 does not suffer from the same problem since, as described in section 5.2, in that guideline the burden is on the user of an external component to structure the verification of their classes around the verification approach used for the component.

The other main challenge in checking the guidelines is the need for an assessment of the RAC- and ESC-compatibility of assertions in the checking of guidelines 1 and 3. Guideline 1 states that all pre-conditions should be RAC- and ESC-compatible and guideline 3 states that post-conditions should have lightweight substitutes if they are not ESC-compatible. As we discussed in section 4.1, RAC- and ESC-compatibility are difficult to define in terms of simple, checkable rules. We can, however, provide limited automated checking by ensuring the absence of certain constructs, like recursion and quantifiers without enumerated ranges, which are not RAC-/ESC-compatible. Developers can check RAC-/ESC-compatibility more rigorously through the independent use of test harnesses to check the efficiency of RAC-checks and the ability of the ESC-prover to verify properties of the specifications. However, the concepts are unavoidably context-dependent and, using our definition, impossible to assess statically in a modular fashion, taking account of all possible usage contexts.

8. Related Work

Earlier publications from the JML group hinted at using the approaches together. In [8], Jacobs et al. proposed using ESC to guide the selective application of FFV. The idea is to first apply ESC to the entire project verifying much of it, and then to use interactive FFV to attempt to verify the remainder. In [3], Cok and Kiniry discussed how beneficial they thought it would be to have an integration of tools that support JML, but as of yet there is no tool support for this.

Other tools have combined static and dynamic checking to some extent. Spec# [1] uses a combination of static and dynamic techniques to verify its assertion annotations. They exploit the fact that constructs that are difficult to check statically are often relatively easy to check dynamically and vice versa. Their approach, however, uses a combination of static and dynamic checking to support their single form of verification whereas we offer a range of verification

approaches of varying rigour. Other tools like PerfectDeveloper [4] include provisions to generate RAC pre-condition checks to ensure that the assumptions on which the proofs of correctness are made are not violated by calls made from unverified code.

9. Conclusions and future work

We started this paper by presenting the case for using the approaches together. We selected ESC, which we consider to be the most promising approach, as our starting point. It provides better coverage than conventional type checking, allows errors to be detected earlier in the software development lifecycle than RAC and requires considerably less effort to use than FFV. We then demonstrated a situation where ESC breaks down and showed how the use of RAC and FFV can assist in that situation. Next we highlighted some key challenges in using the approaches together and put forward a set of guidelines to help users of our tool avoid them. These guidelines incorporated an explanation of how pre-conditions containing constructs such as unconstrained quantifiers can be dynamically checked and the presentation of a technique we have developed to use redundant specifications to provide lightweight substitutes for specifications containing non ESC-compatible constructs. We then described a number of useful ways in which the approaches themselves could be configured. In the final sections we outlined the support our tool offers for the use of these approaches together and explained that currently ours is the only system that provides such tool support.

The project is continuing on a number of fronts. Work on the language is currently focusing on the addition of support for reference semantics in a structured manner that is amenable to modular verification. Our existing language is built around value semantics. We have found that this is extremely good for the specification of modelling types (as is used in JML) but can lead to poor modularity in larger applications, requiring that objects are organised in trees. Reference semantics can support better modularity but requires some form of ownership mechanism (e.g. [6]) in order to be verifiable in a modular manner. Unfortunately ownership mechanisms work by imposing a structure like that from value semantics and so are vulnerable to the same structuring problems. While noting recent advances in this area [6], we see the problem of developing a

modular but flexible ownership mechansim for reference semantics as an important open research challenge.

Work on the IDE is currently focused on the Verification Tracker tool which can be used to view the details and status of the verification of a project. The tool tracks the testing of the generated run-time checks and gives details of the generated VCs together with information on whether they have been automatically or interactively proved. We are also investigating the use of different theorem provers.

We have developed a number of small and medium-sized case studies using Omnibus. The next step for the ideas presented in this paper is to develop larger case studies to explore how they operate in a commercial setting.

When using different verification approaches together it is important to ensure that the approaches use consistent interpretations of the language. This has been an issue in the JML community where tools have been developed by different groups who have made different choices about certain semantic issues [2]. All the Omnibus tools are developed by our group and we have made every effort to ensure the consistency of the interpretation of the language for our static and dynamic verification tools. This consistency has not, however, been formally demonstrated. Similarly, we have not yet formally demonstrated the soundness of our guidelines.

A theme in our work is the provision of a spectrum of tools to support verification at different levels of rigour, depending on the reliability requirements of the project and the skills of the developers. We believe most software developers need push-button verification tools, while component vendors should use full formal verification to fully describe their component and provide certification of the correctness of the hidden implementation. This idea is discussed in more detail in [17].

We are also working on a range of facilities to support software component reuse. These include support for the generation of comprehensive interface documentation from specifications, a framework for the certification of components and integrated support for the location and distribution of components.

Acknowledgements: We are grateful for the valuable feedback from the anonymous reviewers and the attendees and organisers of the SEFM 2005 conference.

References

1. M. Barnett, K.R.M. Leino, W. Schulte – "The Spec# programming system: An overview", Proceedings of CASSIS 2004, Springer LNCS 3362, 2004.
2. P Chalin – "Logical Foundations of Program Assertions: What do Practitioners Want?", Proceedings of SEFM 2005, IEEE Computer Society, 2004.
3. D.R. Cok, J.R. Kiniry – "ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2", NIII technical report R0413, 2004.
4. D. Crocker – "Safe Object-Oriented Software: the Verified Design-by-Contract paradigm", Procs. of the 12th Safety-Critical Systems Symposium, Springer-Verlag, 2004.
5. D. Detlefs, G. Nelson, J.B. Saxe – "Simplify: A theorem prover for program checking", Technical Report HPL-2003-148, HP Labs, 2003.
6. W. Dietl, P. Müller – "Universes: Lightweight ownership for JML", Journal of Object Technology (JOT), 4(8), 2005.
7. C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, R.Stata – "Extended static checking for Java", Proceedings of PLDI 2002, ACM SIGPLAN Notices 37(5), 2002.
8. B. Jacobs, C. Marché, N. Rauch – "Formal verification of a commercial smart card applet with multiple tools", Proceedings of AMAST 2004, Springer LNCS 3116, 2004.
9. G.T. Leavens, A.L. Baker, C. Ruby – "Preliminary Design of JML: A Behavioral Interface Specification Language for Java", Dept. of Computer Science, Iowa State University, TR #98-06p, 2003.
10. K.R.M. Leino – "Extended Static Checking: A Ten-Year Perspective", Informatics—10 Years Back, 10 Years Ahead, Springer LNCS 2000, 2001.
11. B. Meyer – "Contracts for components", Software Development Magazine, July 2000.
12. B. Meyer – "Eiffel: The Language", ISBN 0132479257, Prentice Hall, 2000.
13. B. Meyer – "On to components", IEEE Computer, January 1999.
14. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, M.K. Srivas – "PVS: Combining Specification, Proof Checking, and Model Checking", Proceedings of CAV 1996, Springer LNCS 1102, 1996.
15. J. Rushby – "Kernels for safety? ", Safe and Secure Computing Systems, Blackwell. Scientific Publications, 1989.
16. T. Wilson – Omnibus home page. Available at http://www.cs.stir.ac.uk/omnibus/, 2007.
17. T. Wilson, S. Maharaj, R.G. Clark – "Push-button tools for software developers, full formal verification for component vendors", Technical Report CSM-167, Dept. of Computing Science and Mathematics, University of Stirling, 2006
18. T. Wilson, S. Maharaj, R.G. Clark – "*Omnibus: a clean language and supporting tool for integrating different assertion-based verification techniques* ", Proceedings of the Workshop on Rigorous Engineering of Fault Tolerant Systems (REFT), 2005
19. T. Wilson, S. Maharaj, R.G. Clark - "Omnibus Verification Policies: a flexible, configurable approach to assertion-based software verification", Proceedings of SEFM 2005, IEEE Computer Society, 2005.