Department of Computing Science and Mathematics
University of Stirling

Push-Button Tools for Application Developers, Full Formal Verification for Component Vendors

Thomas Wilson, Savi Maharaj, Robert G. Clark

Technical Report CSM-167

ISSN 1460-9673

August 07

Department of Computing Science and Mathematics
University of Stirling

Push-Button Tools for Application Developers, Full Formal Verification for Component Vendors

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@cs.stir.ac.uk, sma@cs.stir.ac.uk, rgc@cs.stir.ac.uk

Technical Report CSM-167

ISSN 1460-9673

August 07

Abstract

Software developers have varying abilities and develop software with differing reliability requirements. Sometimes reliability is critical and the developers have the mathematical capabilities to perform interactive theorem proving but this is not usually the case. We believe that most software developers need easy to use tools such as run-time assertion checkers and extended static checkers that can help them produce more reliable application-specific code cheaply. However, these lightweight approaches are not sufficient to allow the safe reuse of software components. To safely reuse software components we need comprehensive descriptions and assurances of correctness. These requirements can be provided for by full formal verification with the additional costs justified by the economies of scale. Our Omnibus verification tool provides integrated support for all these different types of verification. This paper illustrates these concepts through a sorting implementation.

Keywords: Assertion-based verification, Run-time assertion checking, Extended Static Checking, Formal verification, Software Component Reuse, Tool integration

1 Introduction

The Omnibus system [1,2,3] consists of a superficially Java-like language and IDE tool incorporating a bytecode generator, static verifier and interface documentation generator. The tool supports the integrated use of a range of different assertion-based verification approaches: run-time assertion checking (RAC), extended static checking (ESC) and full formal verification (FFV). Run-time assertion checking takes assertion annotations and converts them into run-time checks. The application should then be tested to uncover assertion failures with the assertion checks helping to detect failures close to source. Extended static checking uses fully-automated theorem proving to statically check the absence of run-time errors and compliance with lightweight assertion annotations. Full formal verification uses a combination of automated and interactive theorem proving to verify the correctness of an application relative to a heavyweight specification.

Our integrated support for these approaches allows developers with different mathematical capabilities, developing code for different purposes, to use verification at a level of rigour that most suits their needs. We make a strong distinction between (i) typical software developers producing a piece of application-specific code and (ii) a component vendor producing a reusable component. We propose push-button techniques that software developers producing non-critical application-specific code can use relatively cheaply. Our extended static checker and run-time assertion checker together with the unit testing framework provide this. Our system also supports full formal verification which developers may choose to use if the reliability of their code is critical and they have the required skills, but this is purely optional. In contrast, component vendors are encouraged to use full formal verification to ensure that future users have a clear description of what the component should do and a strong assurance that it does it. The economies of scale help justify the additional costs. We strongly believe that a developer cannot safely reuse a component from an unknown third-party component vendor without such provisions.

This paper demonstrates the advantages of using the approaches together in an integrated fashion for a concrete example. We start by considering the role of a typical software developer producing an implementation of a sorting algorithm. First we use our run-time assertion checker and unit testing framework to perform comprehensive testing of our implementation. This approach offers an easy-to-use enhancement of conventional testing. The assertion checks allow us to detect deviations from expected behaviour close to the point of failure. The IDE incorporates push-button facilities to automatically invoke the unit tests and monitor assertion failures.

Next, we look at how the extended static checker can provide more general assurances of correctness without the need for test cases. It works by executing implementations with symbolic input values, recording the relationships between these values and using this information to check assertions where they arise. The tool checks that the pre-conditions of all the methods that are called in the implementation are met and that any assertion annotations are respected. We shall see that there are limits to what we can verify using an automated verification tool such as this. We can verify some simple properties of our sorting algorithm such as the ordering of the result and that the sizes of the lists are the same but we cannot verify that the implementation meets the full sorting specification. We must remain conscious of the limits of automated verification when using ESC and use specifications that remain within the bounds of what the tool can check. We refer to this problem of remaining within the capabilities of ESC tools as ESC-compatibility [2].

In the second half of the paper we turn to the role of a component vendor, who is producing a reusable sorting component. We start by considering the use of run-time assertion checking and extended static checking approaches to verify reusable components. Unfortunately we have to make concessions in the expressiveness of the lightweight specifications in order to make them amenable to efficient run-time checking and decidable automated verification. This can lead to specifications that are insufficient to verify uses of that component [2]. The gaps in the error coverage of the approaches can also reduce the dependability of the hidden implementations. If the components are being used by the same people that developed them then the problems that are met can be addressed incrementally, but when the producers and users of the component are different, this form of direct feedback is not practically possible.

Finally, we will look at how full formal verification provides the facilities to produce comprehensive heavyweight specifications and rigorously verify that they are consistent, and met by the implementation. The process requires interactive theorem proving that is relatively difficult and time consuming but we believe it is the responsibility of a component vendor to provide a clear description of what their components should do and justification that it does it. This is needed to certify that the components can be safely reused.

Combining easy-to-use but powerful verification tools with libraries of clearly specified dependable components provides a practical development method that should be amenable and attractive to practising software developers. Furthermore, our framework should allow reusable components from unknown third-party component vendors to be safely used.

Section 2 considers the role of a typical software developer producing an implementation of a sorting algorithm. Section 3 considers the role of a component vendor producing a reusable sorting component. Section 4 gives conclusions and future work.

2 Role A: application developer

First, let us adopt the role of a typical software developer who has been asked to implement a sorting algorithm. We use the term “application developer” in the situation where we are not concerned with the reuse of our code by other developers. Suppose, as an illustrative example, we are required to produce a static function which accepts a `List` of integers and returns the corresponding `List` in ascending order. Our first attempt to implement the insertion sort algorithm is shown below. We assume that we are provided with a `List` class with methods with an obvious interpretation.

```
 1: public class Sorter {
 2:    public static function insertSort(inList:List):List {
 3:            var a:List := inList;
 4:            var j:integer := 1;
 5:            while (j <= a.size()) {
 6:                    var key:integer := a.elementAt(j);
 7:                    var i:integer := j-1;
 8:                    while (i >= 0 && i < a.size()
 9:                                    && a.elementAt(i) >= key) {
10:                            a.set(i+1,a.elementAt(i));
11:                            i := i-1;
12:                    }
13:                    a.set(i-1,key);
14:                    j := j+1;
15:            }
16:            return a;
17:    }
18: }
```

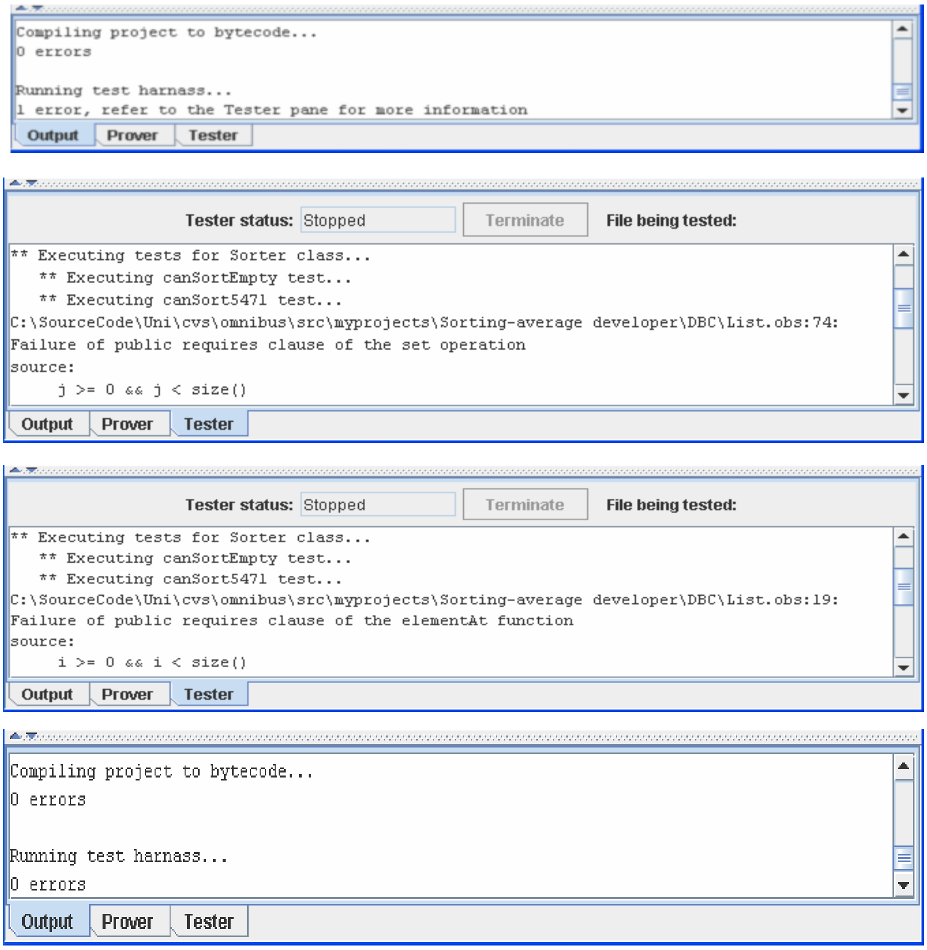
2.1 Run-time assertion checking and unit testing

Of course, we must ensure that the implementation is correct. Let us first look at how the run-time assertion checker can be used. We can start by giving a specification for the sorting method to be verified against. We can use an **`ensures`** clause to provide a post-condition that should hold at the end of the method. A suitable **`ensures`** clause is shown below and could be substituted in place of line 2 of the original code. It consists of two assertions, describing that the returned list should be sorted in ascending numerical order and the result should be a permutation of the input i.e. the number of occurrences of all elements in the two lists should be equal. The method `isPermutationOf` is provided by the `List` class.

```
public static function insertSort(inList:List):List
        ensures "The returned List is sorted":
                        forall (m:integer:= 1 to result.size()-1):
                                result.elementAt(m)
                                        >= result.elementAt(m-1),
                "Returned value is a permutation of the input":
                        result.isPermutationOf(inList) {
```

When the file is compiled to bytecode, run-time checks will be generated for these assertions and those in the other classes in the project. We can now test the application in the conventional manner and assertion failure messages will be generated whenever these assertions are violated. Alternatively, Omnibus provides the facilities to define unit tests within the body of the class to be tested. These can then be automatically invoked by the tool and assertion failures automatically detected.

In Omnibus, unit tests are blocks of code that should create instances of the containing class, manipulate them and then check that they have the expected values. We can write the following simple test cases to check that our sorting implementation can handle the empty `List` and a list containing the elements `5`, `4`, `7` and `1`.



```
Compiling project to bytecode...
0 errors

Running test harnass...
1 error, refer to the Tester pane for more information
```

Tester status: Stopped

File being tested:

```
** Executing tests for Sorter class...
   ** Executing canSortEmpty test...
   ** Executing canSort5471 test...
C:\SourceCode\Uni\cvs\omnibus\src\myprojects\Sorting-average developer\DBC\List.obs:74:
Failure of public requires clause of the set operation
source:
     j >= 0 && j < size()
```

Tester status: Stopped

File being tested:

```
** Executing tests for Sorter class...
   ** Executing canSortEmpty test...
   ** Executing canSort5471 test...
C:\SourceCode\Uni\cvs\omnibus\src\myprojects\Sorting-average developer\DBC\List.obs:19:
Failure of public requires clause of the elementAt function
source:
     i >= 0 && i < size()
```

```
Compiling project to bytecode...
0 errors

Running test harnass...
0 errors
```

Figure 1. From the top, respectively: (a) the run-time assertion checker reporting an error, (b) details of the first error, (c) details of the second error and (d) no errors

```
test canSortEmpty {
        var l1:List := List.empty();
        var l2:List := Sorter.insertSort(l1);
        assert l2.size() = 0;
}

test canSort5471 {
        var l1:List := List.empty().add(5).add(4).add(7).add(1);
        var l2:List := Sorter.insertSort(l1);
        assert l2.size() = 4;
        assert l2.elementAt(0) = 1;
        assert l2.elementAt(1) = 4;
        assert l2.elementAt(2) = 5;
        assert l2.elementAt(3) = 7;
}
```

We can now invoke the run-time assertion checker. This compiles the classes in the application to bytecode containing run-time checks of the assertion annotations, executes the test cases and monitors assertion failures.

The initial results of the assertion checker are shown in Figure 1(a). The assertion checker encountered an error when executing the unit tests. The details of the error can be accessed via the Tester pane of the Omnibus tool. The contents of the Tester pane are shown in Figure 1(b).

The assertion failure message tells us that the pre-condition of the `set` operation call made at line 13 in the original code is not being met during the execution of the `canSort5471` test. On closer examination we see that we should be adding one to `i` rather than subtracting one.

```
13a:                          a.set(i+1,key);
```

Re-running the run-time assertion checker we now get a different error the details of which are shown in Figure 1(c). This time the pre-condition of the first call of `elementAt` within the outer loop is not being met. This is due to a faulty exit condition for the loop which should be changed to:

```
 5a:                  while (j < a.size()) {
```

No more errors are reported when we re-run the run-time assertion checker as shown in Figure 1(d).

It is important to be clear about what this tells us. It tells us that for those particular test cases, the explicit tests in the assert statements and automatically generated run-time assertion checks are met. This means that the implementations are consistent with their specifications, for these test values, and so if the specifications correctly characterize our requirements then, for these values, the implementations are correct. However, it does not tell us about the correctness of the algorithm for other values.

For example, does the algorithm work for lists that are already sorted, for lists of size one, for lists with repeated values? We could write separate tests for those cases but, of course, nothing short of exhaustive testing can tell us about the general correctness of the implementation. To do that we need to use a static approach such as extended static checking.

2.2 Extended Static Checking

Instead of testing the algorithm with assertion checks we can use the extended static checker to verify general properties of the implementation. This process works by executing the algorithm with symbolic input values, recording the relationships between these symbolic values and then converting assertion checks to formulae over these symbols, which are then passed to a fully automated theorem prover. If any of these formulae cannot be proved then a corresponding error message is reported.

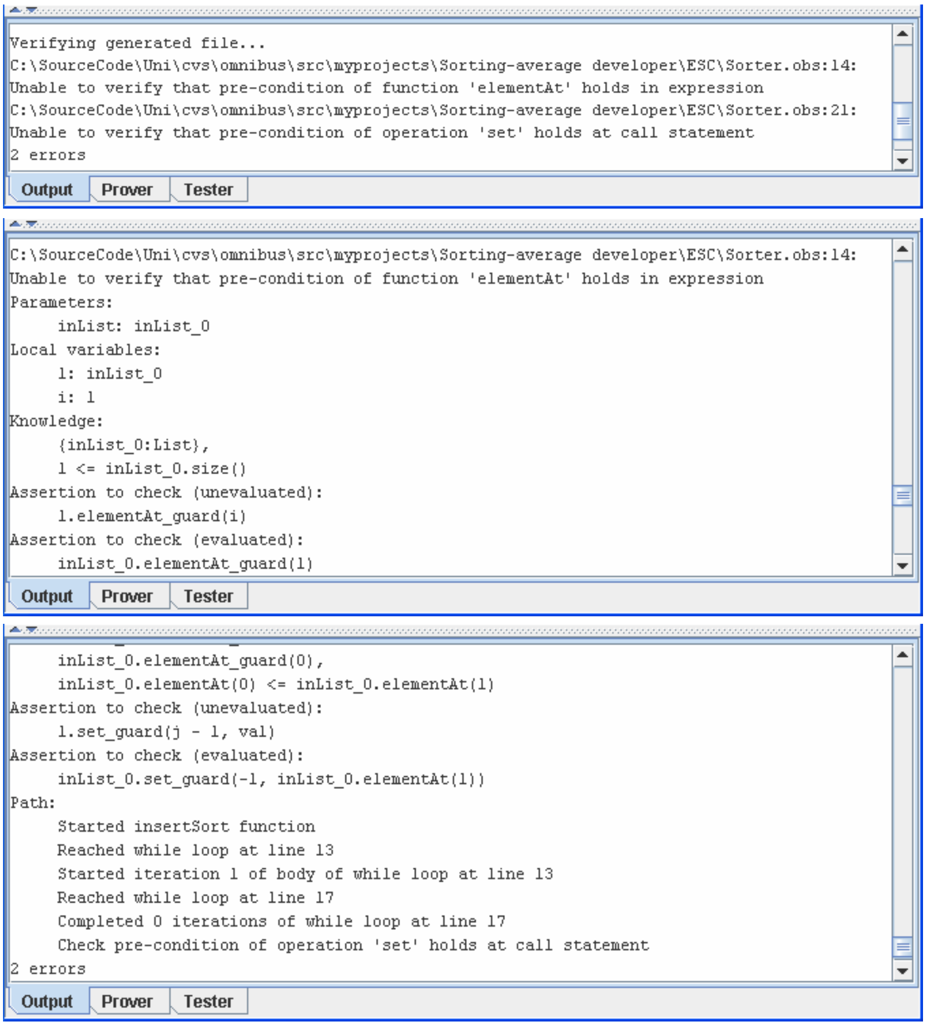
We can take the original unasserted implementation of our sorting algorithm and run it through the extended static checker. We do not need to add assertion annotations or define unit tests like we did when using run-time assertion checking. The extended static checker will simply check that any assertion annotations that we provide, or that are in other classes that we use, are not violated. For example, it will automatically check that the pre-conditions of all the methods we call are met.

Running the extended static checker on the original code yields 2 errors, which are shown in Figure 2(a). The errors that are reported match those we found using the run-time assertion checker. We can get further information on the errors to help us work out the causes of the failures. We can ask for the variable values, assertions that are known to be true, the assertion to be checked and execution path information to be displayed. The full details of the first error are shown in Figure 2(b). We can see from this that during the execution of the method we are trying to retrieve the `inList_0.elementAt(1)` when `inList_0.size()` may equal `1`. The full details of the second error are shown in Figure 2(c). This time it is clear that the index we are passing to the set operation is `-1` when we enter the body of the outer while loop and don't execute the inner loop at all. This reasonably leads us to make the same corrections as we made after the run-time assertion checking.

The tool has checked that there are no crashes, i.e. no assertion failures or conventional run-time errors (e.g. divisions by zero), in all possible executions of the implementation. The key advantage is that the checker does not assume anything about the input List: it could contain repeated elements, it could already be ordered, or it could be empty. We have not verified that the implementation actually sorts the `List`, we have simply verified that it does not crash. To verify that the method sorts the `List` we should add a suitable specification and then re-run the checker. Unlike the run-time checker, we do not need to define any test cases to use the extended static checker.

Unfortunately, we cannot use the full sorting specification from the run-time checking example. This is because the method `isPermutationOf`, which it uses, is specified via a recursive method, and our extended static checker cannot effectively reason about recursive specifications. We can, however, check that the returned `List` is sorted, as we did in the first of the assertions in the specification used by the run-time assertion checker, and that the sizes of the input and output `Lists` are equal. A suitable **ensures** clause is shown below and, again, could be substituted in place of line 2 of the original code.

```
public static function insertSort(inList:List):List
      ensures "The returned List is sorted":
                     forall (m:integer := 1 to result.size()-1):
                              result.elementAt(m)
                                      >= result.elementAt(m-1),
               "The size of the sorted List is the same as the size of the
input List":
                     result.size() = inList.size() {
```



```
Verifying generated file...
C:\SourceCode\Uni\cvs\omnibus\src\myprojects\Sorting-average developer\ESC\Sorter.obs:14:
Unable to verify that pre-condition of function 'elementAt' holds in expression
C:\SourceCode\Uni\cvs\omnibus\src\myprojects\Sorting-average developer\ESC\Sorter.obs:21:
Unable to verify that pre-condition of operation 'set' holds at call statement
2 errors
```

```
C:\SourceCode\Uni\cvs\omnibus\src\myprojects\Sorting-average developer\ESC\Sorter.obs:14:
Unable to verify that pre-condition of function 'elementAt' holds in expression
Parameters:
     inList: inList_0
Local variables:
     l: inList_0
     i: 1
Knowledge:
     {inList_0:List},
     1 <= inList_0.size()
Assertion to check (unevaluated):
     l.elementAt_guard(i)
Assertion to check (evaluated):
     inList_0.elementAt_guard(1)
```

```
     inList_0.elementAt_guard(0),
     inList_0.elementAt(0) <= inList_0.elementAt(1)
Assertion to check (unevaluated):
     l.set_guard(j - 1, val)
Assertion to check (evaluated):
     inList_0.set_guard(-1, inList_0.elementAt(1))
Path:
     Started insertSort function
     Reached while loop at line 13
     Started iteration 1 of body of while loop at line 13
     Reached while loop at line 17
     Completed 0 iterations of while loop at line 17
     Check pre-condition of operation 'set' holds at call statement
2 errors
```

Figure 2. From the top, respectively: (a) extended static checker reporting two errors, (b) details of the first error, (c) details of the second error

This adjustment of the specification we use is a practical illustration of the theoretical limitations of automated verification. We had to adjust our specification according to the capabilities of the ESC tool. In section 3 we will see how such adjustments can cause problems for reusable components.

Again, we should be careful to query exactly what has been verified. Loops are a problem for extended static checkers. Their evaluation would generally involve the consideration of an unbounded number of cases: one for each of the number of times that the loop could have been executed. Our extended static checker simply considers a finite number of iterations of each loop. By default we consider the cases where each loop is executed 0 or 1 times in separate branches. So the tool has verified that in all possible executions of the implementation where each loop is executed either zero or one times that there are no crashes and the `ensures` clauses hold at the end of the method.

2.3 Taking ESC further

If we want to prove properties about the implementation where the loops are executed any number of times then we can supply a loop invariant for each loop which the tool can use as a basis for an inductive proof of correctness. However, these assertions are particularly difficult to devise and often the inductive proofs are too difficult for the automated prover we use. In this case a proof of the ordering of the result using loop invariants is beyond the capabilities of our ESC tool even if suitable loop invariants are provided. However, by adding selected loop invariants describing how the size of the lists change and how the values of the local variables relate to the sizes, we can verify the absence of run-time errors for executions with an unbounded number of iterations of each loop. Suitable loop invariants for the first and second loops, respectively, are shown below:

```
alters a, j
maintains j >= 1, j <= a.size(), a.size() = inList.size()

alters a, i
maintains i >= -1, i < a.size()-1, a.size() = inList.size()
```

We must also surround the outer loop by an `if` statement checking that the size of the list `a` is greater than zero since otherwise the invariant `j <= a.size()` will not hold initially. If we want to verify that the result of the method is sorted or that it is a permutation of the input then we would need to provide more comprehensive loop invariants but we cannot verify those using our ESC tool.

2.4 RAC and ESC as complementary approaches

Run-time checking and ESC are both push-button approaches to verification, and they have complementary strengths. ESC can guarantee correctness for a general class of value, but users are forced to compromise on the expressiveness of the specifications. Run-time checking allows the use more expressive specifications, but its downside is that correctness is only guaranteed for specific test values. The Omnibus tool allows easy integration of the two approaches.

3 Role B: component vendor

Let us now consider the role of a component vendor. We assume that the developers of components and the users of the component may be separate. Crucially, we must implement the component, verify it and then ship it. Once it is shipped, we cannot easily alter it to correct errors. So, as responsible component vendors, we would like to ensure that our component is not faulty before we distribute it.

There has been much work on reusable software components. The work of Meyer has greatly influenced us. To allow software components to be reused safely Meyer states we require two things: (1) a clear, unambiguous description of what the each component should do [4] and (2) some form of assurance that it does it [5].

3.1 State-of-practice for reusable components

Currently, in practice, software components are described using type signature interfaces supplemented by interface documentation (e.g. javadoc, docgen) and no quantifiable assurances of correctness are provided. As a result, developers have to use their intuition to fill in the gaps of the descriptions of what components should do and use informal information such as the reputation of the vendor that produced the component to decide whether they can trust it will work. When the producer of the component is an organisation such as Sun this works relatively well since they comprehensively document their interfaces and you can generally trust them to implement things correctly. However, the system is seriously flawed when applied to unknown third-party component vendors. There are no requirements on the level of documentation which is often patchy and it is not clear if we should trust an implementation of a component from one of these vendors to be correct. As a result, the reuse of software components using this framework often introduces problems, sometimes even catastrophic failure. To protect themselves against this, many developers develop “not invented here” syndrome and prefer re-implementation to reuse when the implementer of the component is not known to be dependable. This has acted to limit the spread of reusable software components.

3.2 Developing reusable components with run-time assertion checking and extended static checking

Run-time assertion checking and extended static checking help the situation somewhat. The assertion annotations on which they are based provide a structured framework which can be used to provide unambiguous documentation. The associated verification methodologies also give some basis on which to found trust of the hidden implementation. If ESC has been used to verify that the given assertion annotations constituting the interface of the component are met by the implementation then the user of the component can have some reasonable confidence in its correctness. Similarly if RAC is used in conjunction with a specified test harness (which should also be included in the interface of the component) then we can have some confidence that the hidden implementation meets the specification of the component.

Unfortunately ESC and RAC provide only limited solutions. We have to make compromises in the expressiveness of the assertion annotations we use in conjunction with these approaches.

Using ESC we could not verify the correctness of the implementation relative to the full specification for sorting and so we had to use a relatively incomplete specification. At this point the assertion annotation and our conception of the correctness of the algorithm diverge. We can document the additional requirements in interface documentation but they are not included within the assertion annotations because they cannot be verified by the approach. This limits the extent to which we can provide a clear and comprehensive description of what the component should do. We also saw that concessions are made in the soundness of the verification approach (e.g. considering loops by unravelling them a finite number of times) in order to make it possible to use an automated theorem prover.

Using RAC we saw that it was possible to check the assertion annotations of the full sorting specification from Section 2. The main problem with the run-time checking of the assertions is that the more expressive the assertion annotations are, the more time it will take to execute the run-time checks. This problem can be extreme. If full specifications are used then the evaluation of the run-time assertion checks will often take at least as long as the execution of the implementation itself (because for full specifications both the things that are changed and the things that are not changed must be checked, whereas implementations only describe what changes are made). This may be acceptable during the testing process performed by the implementer of the component but retaining the full run-time checks after the component is distributed will often compromise the efficiency of the component too much.

To combat this we can either:

1. disable the checks of the supplier obligations (e.g. the post-conditions) before distribution or
2. adjust the assertions to make them cheaper to check and then retain the run-time checks.

Disabling the checks of the supplier obligations is fine if the test harnesses that were used to check the implementation were sufficient to uncover all possible implementation errors but a dangerous situation arises if they are not. In such a case the user of the component may make a call of the component, satisfying their obligations but exposing a scenario not covered by the test harness, where the component's implementation is not correct and violates its assertion annotations. Crucially, if the assertion checks in the component's implementation are disabled then an assertion failure will not be automatically triggered and the component will simply, silently return a value violating its own assertion annotations. Such problems could be hard to track down since the user will, rightly, initially assume that the implementations of the components meet their specifications. In order to uncover the error they will have to consider the possibility that each of the components fails to meet its obligations. This is highly undesirable and seriously compromises the basis for trusting the hidden implementation.

If we take the alternative approach and adjust the assertion annotations to make them cheaper to check (e.g. limiting the use of quantifiers) then we will compromise our ability to provide comprehensive descriptions of what the component should do, just as is the case with ESC. In this case concessions are made in the completeness of the verification in order to make it practical to check the assertion annotations at run-time.

3.3 Developing reusable components with full formal verification

For a comprehensive guarantee of correctness for software components, we must turn to full formal verification. The Omnibus tool supports this approach by translating from the Omnibus language into the language of the PVS theorem prover. A process of symbolic execution is used to generate verification conditions for implementations. Heavyweight specifications can be used to provide comprehensive, unambiguous descriptions of what the component should do and the formal verification mechanism can give a rigorous assurance that the implementation does this.

This approach can be used by a component vendor to produce a fully verified, reusable `Sorter` component. To see how this process works, we shall look at a part of the verification of this class. Let us attempt to verify that the result returned by the `insertSort` method is ordered, for any number of iterations of each loop. We must first define suitable loop invariants. The loop invariants used in Section 2 do not describe how the ordering of the list changes and so cannot be used for this purpose. Suitable loop invariants are shown below. The first, outer loop ensures that the first `j` elements of the list are in sorted order, where `j` increases by one with each iteration of the loop. The second, inner loop ensures that the elements below `i` are ordered and those from `i+2` upwards are ordered and greater than `key`.

```
alters a, j
maintains j >= 1, j <= a.size(), a.size() = inList.size(),
              forall (o:integer := 1 to j-1):
                     a.elementAt(o) >= a.elementAt(o-1)


alters a, i
maintains i >= -1, i < a.size()-1, a.size() = inList.size(),
              forall (o:integer := 1 to i):
```

```
          a.elementAt(o) >= a.elementAt(o-1),
      forall (p:integer := i+2 to j):
          a.elementAt(p) >= key,
      forall (q:integer := i+3 to j):
          a.elementAt(q) >= a.elementAt(q-1)
```

We can now use the interactive formal verifier. This translates the specifications of the classes in the project into the logic of the PVS prover and then generates a number of obligations over these specifications, which we must prove in order to demonstrate the correctness of our implementation. Default proof attempts are automatically generated by the tool, which allow many obligations to be automatically verified.

A total of 44 proof obligations are generated for this class. Of these the default proof attempts are sufficient to prove 31. We must then assist the tool in the verification of the remaining obligations. Most of these were fairly straightforward but a few required a certain amount of mathematical ingenuity. In all, it took an experienced PVS user about 10 hours to discharge all the proof obligations.

As is often the case, we found that one of the most difficult steps in this approach was devising suitable loop invariants. It took several attempts to get them right and we often found more errors in the loop invariants that in the code itself. It is discouraging when a product of the verification process is harder to debug than the original code itself. Uncovering errors using interactive verification is also particularly costly. It may not be clear whether an obligation should be provable or not. The user may believe that it should be but reach an unprovable sub-goal and be unsure whether they made a mistake in their proof attempt or if the original obligation was invalid. Even if they decide that the original obligation was faulty, a suitable correction may not be readily apparent.

Full formal verification is therefore a costly approach; however, the payoff for carrying it out is that you gain great confidence in the implementation. To successfully perform the verification the user must mathematically justify the correctness of the implementation. The process forces you to think very deeply about your implementation and how it works, helping to ensure that it is precisely what is needed. This can make it useful where reliability is of critical importance. The interactive proof mechanism allows complicated properties to be verified, allowing programmers to use more sophisticated specifications, and the resulting proofs can be re-run by any user to independently check the proof. This provides a powerful basis to support the safe reuse of software components. In future work, we intend to develop this into a scheme for generating and checking correctness certificates for components.

4 Related Work

4.1 Integrating different verification approaches

Our tool provides integrated support for RAC, ESC and FFV. We have previously discussed why such integrated support is desirable and how it can be managed [2]. JML [6] is another project that supports these different approaches. It is built around an assertion-annotation language for Java which is supported by a range of separate tools developed by separate teams and applied separately rather than together in an integrated fashion like they are within Omnibus. JML tools include a run-time assertion checker [7], the ESC/Java2 extended static checker [8] and a number of other static verification tools including LOOP [9], Jive [10], KeY [11] and Krakatoa [12]. The RAC and ESC/FFV tools are inconsistent in their handling of certain aspects of JML [13], whereas all the tools provided by Omnibus use a consistent interpretation of the semantics of assertions.

Many other tools have combined the use of different verification approaches in some way. Spec# [14] combines the use of automated static checking and run-time assertion checking. Many other verification tools provide the facilities to automatically generate run-time assertion checks of their preconditions to ensure that the assumptions on which the static verification is based are not violated.

4.2 Making verification more accessible to software developers

It is clear that automated verification tools are more desirable than interactive verification tools, as long as their performance is satisfactory. However, it does not appear to be possible to perform full formal verification of programs written in languages such as Java using automated tools. The question is then, how can we enable typical software developers, without specialized mathematical skills, to perform verification?

In Section 2 we discussed how push-button techniques such as run-time assertion checking and extended static checking can support the automated verification of code with lightweight assertion annotations. The assertion annotations to be checked are adjusted so that the checks can be efficiently executed in the case of RAC and verified using an automated prover in the case of ESC. There is a vast array of RAC tools available including the JML run-time assertion checker [7], Eiffel [15] and Jass [16]. The JML run-time assertion checker contains a unit testing framework and supports the automatic generation of unit tests. ESC/Java2 [8] is the leading ESC tool.

An alternative approach is to make interactive verification more accessible to developers. In Omnibus, interactive verification is performed using the PVS theorem prover [17]. To perform full formal verification the developers must be familiar with PVS and perform proofs by issuing low-level commands to the prover. Tools such as KeY [11] make interactive verification more accessible to typical software developers by presenting the verification conditions to be proved in the source language (instead of the language of a theorem prover) and providing point-and-click proof environments. Such facilities would also be of great use to component vendors in our system, making it easier for them to fulfil their obligation to perform full formal verification of reusable components.

The need for loop invariants is another obstacle to the use of verification tools. The Spec# tool can automatically deduce loop invariants. PerfectDeveloper [18] avoids the need for many loop invariants by automatically generating implementations with loops from specifications.

The complexities of reference semantics in languages like Java greatly increase theorem proving difficulty. By restricting the language we can make them more amenable to automated verification. Examples of languages containing such restrictions are Perfect [18] and our own Omnibus language, both of which are built on value semantics. The PerfectDeveloper tool exploits the restrictions of the Perfect language in order to perform full formal verification using their fully automated prover. We have found that even using a language based on value semantics, full formal verification often requires interactive theorem proving. The Perfect language is more mature than our Omnibus language and includes facilities for reference semantics and static data which Omnibus currently lacks. PerfectDeveloper, however, does not support the same range of verification approaches as our tool.

There seems to be an increasing interest within the JML community in restricting the source language in order to make verification easier and hence automated verification more practicable. Examples include Muller's Universe type system [19] and read-only references [20]. We note that Cok from the ESC/Java2 project has also highlighted this area [21].

5 Conclusions and future work

Using our approach, software developers can write code, documenting design decisions in lightweight assertion annotations and check that their implementation satisfies these using push-button verification tools. They can use fully specified and verified reusable software components to help build their applications. Component vendors, on the other hand, write heavyweight specifications and use full formal verification to check that their implementations satisfy these specifications. This will often involve interactive theorem proving and so they will have to have suitably skilled developers. However, if reusable components from unknown third-party component vendors are to be trusted then something like this is essential.

Our lightweight verification tools allow software developers to uncover more problems than conventional testing. The tools are powerful yet easy to use. The main problems are concerned with scalability particularly for the extended static checking.

By providing a framework to specify and certify software components we can allow them to be reused safely. Our hope is that this will encourage developers to increase their reuse of existing software components. Increased reuse can potentially (1) save development and verification time (since the required functionality doesn't have to be manually implemented and debugged) and (2) increase reliability (since the reusable components have been more rigorously verified than can be achieved with the push-button tools).

There are a range of challenges to be overcome before these potential advantages can be fully realized. One of the major challenges with using different assertion-based verification approaches is how the approaches interact. For example, suppose we are verifying a class using ESC which uses a component that was verified using FFV. The ESC tool may not be able to effectively reason about the heavyweight specification of the component e.g. it may use a recursive specification which our ESC tool can't handle. We have previously discussed this problem in [2] and proposed the use of redundant lightweight specifications as a partial solution. This is, however, a complicated problem and we are continuing our research on it.

To support our framework for reusable components we are developing support for component repositories. These will allow components to be easily distributed together with assertion-based interface documentation and certification information. Facilities to search for and retrieve components from these repositories will be incorporated within the Omnibus IDE tool.

References

[1] Wilson, T. Omnibus home page, http://www.cs.stir.ac.uk/omnibus/, December 2006.

[2] Wilson, T., Maharaj, S. and R.G. Clark. Omnibus Verification Policies: A flexible, configurable approach to assertion-based software verification. *Proc. Software Engineering and Formal Methods 2005*, IEEE Computer Society Press, 2005.

[3] Wilson, T., Maharaj, S. and Clark, R.G. Omnibus: a clean language and supporting tool for integrating different assertion-based verification techniques. *Proc. Workshop on Rigorous Engineering of Fault-Tolerant Systems*, Newcastle, 2005.

[4] Meyer , B. Contracts for components. *Software Development magazine*, July 2000.

[5] Meyer, B. On to components. *IEEE Computer* 32(1), 1999.

[6] Burdy, L., Cheon, Y. et al. An Overview of JML Tools and Applications. *Proc. Workshop on Formal Methods for Industrial Critical Systems*, ENTCS vol. 80, 2003.

[7] Cheon, Y., Leavens, G. A Runtime Assertion Checker for the Java Modeling Language (JML). *Proc. Software Engineering Research Practice 2002*, CSREA Press, 2002.

[8] Cok, D.R. and Kiniry, J.R. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2. *Proc. Construction and Analysis of Safe, Secure and Interoperable Smart Devices: International Workshop*, CASSIS 2004, LNCS 3362, 2004.

[9] Jacobs, B. and Poll, E. A Logic for the Java Modeling Language JML, *Proc. Fundamental Approaches to Software Engineering (FASE'2001)*, LNCS 2029, 2001.

[10] Meyer, J. and Poetzsch-Heffter, A. An architecture for interactive program provers. *Proc. Tools and Algorithms for the Construction and Analysis of Systems 2000*, LNCS 1785, 2000.

[11] Ahrendt, W., Baar, T. et al. The KeY Tool. *Software and Systems Modelling* 4(1), Springer-Verlag, 2005.

[12] Marché, C., Paulin-Mohring, C. and Urbain, X. The Krakatoa Tool for Certification of Java/JavaCard Programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1-2), 2004.

[13] Chalin, P. Logical Foundations of Program Assertions: What do Practitioners want? *Proc. Software Engineering and Formal Methods 2005*, IEEE Computer Society Press, 2005.

[14] Barnett, M., Leino, K.R.M. and Schulte, W. The Spec# programming system: An overview. *Proc. CASSIS 2004*, LNCS 3362, 2004

[15] Meyer, B. *Eiffel: The Language*, ISBN 0132479257, Prentice Hall, 2000.

[16] Bartetzko, D., Fischer, C. et al. Jass - Java with Assertions. *Proc. Workshop on Runtime Verification (RV'01)*, ENTCS vol. 55, 2001.

[17] Owre, S., Rajan, S. et al. PVS: Combining Specification, Proof Checking, and Model Checking. *Proc. CAV 1996*, LNCS 1102, 1996.

[18] Crocker, D. Making Formal Methods popular through Automated Verification. *Proc. International Joint Conference on Automated Reasoning 2001*, Springer LNAI 2083, 2001.

[19] Dietl, W. and Muller, P. Universes: Lightweight Ownership in JML. *Journal of Object Technology*, 4(8), 2005.

[20] Birka, A. and Ernst, M.D. A practical type system for reference immutability. *Proc. ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages and Applications*, ACM, 2004.

[21] Cok, D. Reasoning with specifications containing method calls and model fields. *Journal of Object Technology*, 4(8), 2004