The Omnibus Language and Integrated Verification Approach

Thomas Wilson

Department of Computing Science and Mathematics,
University of Stirling,
Stirling, FK9 4LA.
Scotland

This thesis has been submitted to the
University of Stirling in partial fulfilment
of the requirements for the degree of
Doctor of Philosophy.

August 2007

Declaration

I hereby declare that this thesis has been composed by myself, that the ideas which I do not attribute to others are my own and that the work has not previously been presented for any University degree.

Thomas Wilson

August 2007

Abstract

This thesis describes the Omnibus language and its supporting framework of tools. Omnibus is an object-oriented language which is superficially similar to the Java programming language but uses value semantics for objects and incorporates a behavioural interface specification language. Specifications are defined in terms of a subset of the query functions of the classes for which a frame-condition logic is provided. The language is well suited to the specification of modelling types and can also be used to write implementations. An overview of the language is presented and then specific aspects such as subtleties in the frame-condition logic, the implementation of value semantics and the role of equality are discussed. The challenges of reference semantics are also discussed.

The Omnibus language is supported by an integrated verification tool which provides support for three assertion-based verification approaches: run-time assertion checking, extended static checking and full formal verification. The different approaches provide different balances between rigour and ease of use. The Omnibus tool allows these approaches to be used together in different parts of the same project. Guidelines are presented in order to help users avoid conflicts when using the approaches together. The use of the integrated verification approach to meet two key requirements of safe software component reuse, to have clear descriptions and some form of certification, are discussed along with the specialised facilities provided by the Omnibus tool to manage the distribution of components. The principles of the implementation of the tool are described, focussing on the integrated static verifier module that supports both extended static checking and full formal verification through the use of an intermediate logic.

The different verification approaches are used to detect and correct a range of errors in a case study carried out using the Omnibus language. The case study is of a library system where copies of books, CDs and DVDs are loaned out to members. The implementation consists of 2278 lines of Omnibus code spread over 15 classes. To allow direct comparison of the different assertion-based verification approaches considered, run-time assertion checking, extended static checking and then full formal verification are applied to the application in its entirety. This directly illustrates the different balances between error coverage and ease-of-use which the approaches offer. Finally, the verification policy system is used to allow the approaches to be used together to verify different parts of the application.

Acknowledgements

The biggest thanks must go to my supervisors Savi Maharaj and Bob Clark. They gave me the freedom to explore my own ideas and taught me how to write to an academic level through feedback on draft after draft. I will be forever grateful for this and I really hope that my work has justified the belief they always showed in me.

I must thank all my family for their support. I often discussed ideas with my Dad, who was effectively my third supervisor. My Mum was always there for me when I needed her and my little brother always kept me entertained. I was lucky to have some good friends. I must mention Ali, Jennifer, Lloyd and Sarah. Most of all I want to thank my wonderful girlfriend Donna for all her positivity and encouragement. How on Earth did I manage without her?

I should thank Kirsty Bell who carried out a case study with Omnibus for her honours dissertation. This was quite a difficult task because the tools were still in development but she did excellent work.

There have been a lot of people who have believed in me and encouraged me over the years. Two teachers had a big impact on my life: my high school Maths teacher Mr Tinning and Mr McFarlane who taught me for my final year in primary school. Without their encouragement at crucial stages of my life, I do not believe I would have reached this stage.

I was lucky to get the opportunity to work at a company called CBS Online while I was still at school and the owner Martin Wardman had enough faith in me to put me on the programming team at the age of 17. Those experiences directly motivated this research and setup my career.

I had the opportunity to undertake an exchange to the University of California, Santa Barbara in the third year of my degree and work I did with Tevfik Bultan and Richard Kemmerer helped give me the necessary background to undertake this project.

Thank you to the people I met at conferences and were so encouraging. Kind words from Joe Kiniry, Patrice Chalin, David Cok, Erik Poll and Rustan Leino made a big difference to me.

Finally, thank you to my Ph.D. examiners, Ken Turner and Joe Kiniry, who went through my thesis in great detail. I greatly enjoyed the opportunity to discuss my work with them during my Viva.

for my dad and granddad

Contents

Chapter 1

Introduction

The software development industry is plagued by massive failures [6, 7, 25, 44, 45, 52, 65]. Examples of failed software projects include the crash of the Ariane 5 rocket in 1996, 40 seconds after launch, with a reported cost of $500 million [52, 83]; the abandonment of a $526 million supply-chain management system by J Sainsbury plc in 2004 [25]; and the ongoing saga of the UK's NHS IT system [6]. The U.S. Department of Commerce has estimated the economic cost of faulty software in the U.S. alone at $22.2-$59.5 billion per annum [45]. Other estimates for this figure are as high as $60-$70 billion per annum [25]. People have also lost their lives as a result of software failures. For example, between 1985 and 1987 a software error in a computer-controlled radiation therapy machine led to patients receiving massive overdoses, killing at least three people [65]; a similar mishap between 2000 and 2001 led to another eight fatalities [65].

It is widely acknowledged that this situation is something we should look to address. Edsger Dijkstra had the following comment [38]:

> "The average customer of the computing industry has been served so poorly that he expects his system to crash all the time, and we witness a massive

worldwide distribution of bug-ridden software for which we should be deeply ashamed."

While it is widely accepted that there is no single *silver bullet* [18] to the problems of the software development industry, we can still look for areas of improvement. Increasing the level of software component reuse could help address some of the problems of software development by allowing developers to make more use of existing components rather than implementing their own.

Studies have suggested that on average as much as 85% of a typical new application could be developed by reusing existing software, meaning as little as 15% application-specific code may be required if reuse is fully exploited [85, 92]. Reuse can save time and money since, given a suitable framework, it is quicker and cheaper to reuse an existing component than it is to write your own. If the components are rigorously ensured to be correct then component reuse can also improve reliability.

There is a range of dedicated frameworks for component reuse that are used in practice including JavaBeans, .NET and CORBA. In these frameworks, software components are described using type signature interfaces supplemented by interface documentation (e.g. javadoc, docgen). Typically, no quantifiable assurances of the correctness of the components are provided.

The level of software component reuse is currently considerably below the theoretical threshold. The most cited barrier to reuse in a 2005 NASA survey on software reuse practices in the Earth science community was that components were "difficult to understand/poorly documented" [5].

Another often-cited barrier to reuse is the so-called *not-invented-here syndrome* [15] where software developers favour re-implementation over reuse. This syndrome has acted to limit the reuse of software components. The not-invented-here syndrome, in this context, may be a natural human defence mechanism that programmers develop through experience to protect them from the dangers of reuse. Without any dependable assurances of correctness, developers have to resort to the use of informal information, like the reputation of the vendor that produced the component, to decide whether they can trust the hidden implementation to work. The reality is that reuse without adequate support can lead to results that are worse, not better, than without reuse [52].

1.1 Formal methods and assertion-based verification

Techniques that exploit the mathematical properties of software systems in order to formally reason about their correctness are referred to as *formal methods*. In this thesis we consider the use of formal methods to help address some of the problems within the software development industry.

Formal methods require the production of a *formal specification* which defines precisely the desired behaviour for a software application or component. The specification can then be used as a basis for checking the correctness of an implementation. *Formal verification* is the process of formally ensuring that specifications are internally consistent and satisfied by an implementation. There is a range of different formal verification approaches.

We focus on a class of formal methods called *assertion-based verification*, built around the use of special expressions called *assertions* which are embedded within program code. Assertions define properties that must hold at different points during the execution of software applications. If an assertion does not hold at the point at which it is reached then this corresponds to an error. Assertions can be used to provide a specification for formal verification or as an oracle of correctness for software testing.

In assertion-based verification approaches, assertions are used to specify the interfaces between classes and methods. The assertions define the separate obligations on the callers and implementers of each method. Methods are annotated with special assertion annotations called: pre-conditions, post-conditions and frame conditions. A *pre-condition* defines properties that should be established by the caller of a method before the method is called. A *post-condition* defines properties that should be established by the implementer by the end of the execution of the method, assuming that the pre-condition was satisfied by the caller. Post-conditions are typically supplemented by a *frame condition* which defines some restrictions on the data that the method can change. In addition to describing the individual methods, assertion-based object-oriented verification approaches also support the concept of class invariants. A *class invariant* is an assertion that should hold throughout the lifetime of all instances of that class. More specifically, they should be established by constructor methods and maintained by instance method calls of the object.

Formal methods can be used to assess different aspects of software. For example, when asked to perform a specific action, we may wish to demonstrate that the software: will always terminate, will terminate within some maximum time period, and/or if the software terminates, the state that it terminates in will satisfy the specification. The work described in this thesis is restricted to the demonstration that if a specific action of the considered software terminates that it will do so in a state which satisfies the formal specification. This is referred to as *partial correctness*. Where the software is also shown to always terminate, this is called *total correctness* but that is beyond the scope of this thesis. Likewise, the thesis does not consider techniques to show that software will always terminate within a specified time frame. We also restrict our attention to a specific type of software: sequential object-oriented code. We do not consider concurrency or non object-oriented systems.

1.1.1 Current realities and future promise of formal methods

Formal methods could help address some of the current failings of the software development field. The use of formal methods can reduce the amount of testing that is needed by catching errors earlier in the development process. For example, the use of the SPARK formal method in the implementation of the control system for the Lockheed C130J aircraft (commonly known as “Hercules”) assisted in a reduction of the required testing budget by 80% [24].

Increased levels of reuse of verified components could also help the situation. Two key technical problems that are inhibiting the reuse of software components are:

1. software components need to be properly documented with clear and precise descriptions of what they are intended to do [71], and
2. justification needs to be provided that the hidden implementation of a software component satisfies its descriptions [70].

Formal methods provide the facilities to address these failings [69]. Formal specifications can help provide clear and precise descriptions of what a component should do and formal verification can provide a basis for trusting that a hidden implementation satisfies its interface. Furthermore, economies of scale can justify the additional costs of formal methods.

The formal verification of software has been a long-standing goal of computing science. In recent years, improvements in computer hardware and in areas such as automatic and interactive theorem proving, model-checking, and static analysis have led to a situation where formal verification of industrial-scale software is now a realistic prospect. There have been some notable successes in the development of safety-critical systems [13, 16, 24, 28, 50, 93]. However, formal methods have had limited impact outside the conventional safety-critical domain.

This thesis aims to provide formal methods that can be applied to the wider commercial software development industry. In our work we have attempted to identify areas which were not being focussed on by the leading verification projects. We developed a new language and set of tools to give us complete freedom to explore the area without being too heavily influenced by the handful of leading international projects that are so dominant. We did not fix on a definition for the language early in the project, formally describe it (its semantics and type system), and stick to it. Instead, the language and tools were radically re-developed to incorporate new ideas as the project progressed. The drawback of this approach is that we do not have a succinct formal definition of the language in its current state which we can present. To produce such a definition is a key item of future work.

The intended audience for our work is researchers in the assertion-based verification community and we hope that these researchers can take aspects from our work and incorporate them into their own languages and tools. The project is intended more as a research vehicle to explore alternative ideas than as an approach to be used immediately in large-scale commercial software development.

1.1.2 Assertion-based verification approaches

In this thesis we focus on three assertion-based verification approaches. They are: run-time assertion checking (RAC), extended static checking (ESC) and full formal verification (FFV).

RAC is a dynamic verification approach built around testing, while ESC and FFV are static verification approaches which involve the generation of obligations called *Verification Conditions* (VCs) which must then be proved using a *theorem prover*. There are a*utomatic theorem provers* which decide the truth of the specified correctness properties automatically without involving the user, and *interactive theorem provers* which allow the user to assist in the proof process. The theorem proving problem is undecidable for the types of logics we consider and so automated theorem provers cannot prove all VCs that are valid.

FFV involves the production of a *heavyweight specification* which should provide a comprehensive characterisation of correctness, while RAC and ESC use only *lightweight specifications* which simply describe some properties that should hold. 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.

Run-time assertion checking

The run-time assertion checking approach takes a program with a set of lightweight assertion annotations written in an extension of the expression language and produces an executable implementation incorporating run-time checks of those assertions. The application can then be tested in a traditional manner, and violations of the assertion annotations will be detected and reported.

A big advantage of RAC is that it avoids the theoretical complexities of theorem prover-based verification. The contribution of RAC to the testing process is that the failures encountered during the execution of the program are detected close to source, aiding analysis and correction. The assertion failure messages that are generated are also far more informative than the stack trace error message that may have otherwise revealed the failure when the flaw triggered the generation of an uncaught exception at some later stage in the execution. The assertion annotations can provide a form of test oracle for the testing process.

A limitation of the approach is that its error coverage is closely associated with the coverage of the corresponding test suite. The approach also uses lightweight specifications, so only the behaviour characterised in these assertions will be checked. Of course, it also impacts run-time performance.

The run-time assertion checking approach was popularised by Bertrand Meyer, who designed the Eiffel language [68] around this principle. RAC tools are now also available for most other mainstream programming languages. For example, tools supporting RAC for Java include Jass [11], iContract [58], jContractor [53] and the JML run-time assertion checker [27].

Extended static checking

Extended static checking uses static verification techniques to detect a range of possible run-time errors and violations of lightweight specifications without the need for user interaction. The VCs produced by ESC tools are discharged by a fully automated theorem prover.

Spurious warnings reported by ESC tools (frequently caused by incompleteness in either the lightweight specifications or the theorem prover) can be suppressed by using a range of assumption constructs that allow programmers to trade these spurious warnings for possible unsoundness. A prominent example is the `assume` statement which allows users to provide an assertion which should be taken as true without further justification. There are different techniques for the handling of loops including unravelling them a finite number of times to avoid the need for loop invariants.

ESC is presented to those outside the formal methods field as an extension of the static checking (i.e. type checking, basic flow analysis, etc.) performed by programmers using standard language compilers. ESC tools produce error messages similar to those produced by standard compilers. This makes it easier to market the approach since software developers are more receptive to “type checkers” than formal methods tools. Like RAC, ESC also uses lightweight specifications that are easier to write than their heavyweight equivalents. ESC helps to find errors earlier in the development process than RAC, when they are easier to correct, and does not require the production of a test suite.

A crucial problem of ESC is that the specifications must be sufficient to act as a basis for modular verification. This is illustrated in chapter 5. The specifications must also be adjusted to fit within the capabilities of the associated automated theorem prover. If specifications are used which are too complex, the tool may do something undesirable like report an error where there is none.

The extended static checking approach was developed by a team at DEC who produced the original ESC tools: ESC/Modula-3 [35] and ESC/Java [41]. More recently Kiniry and Cok have produced the ESC/Java2 tool [30].

Full formal verification

Full formal verification, or full functional program verification, is a static verification approach that allows the correctness of an application relative to some specification to be fully mathematically demonstrated, allowing the production of software that is, in principle, completely correct. FFV can utilise either fully automated or interactive theorem provers.

Interactive theorem provers enable the use of more sophisticated specifications since the producer of the specification does not need to remain within the bounds of what can be proved using an automated theorem prover. The interactive proofs can be saved, distributed with the component and then re-run by clients to re-verify the component.

Fully automated theorem provers are far more desirable because typical users frequently have neither the mathematical training nor time to assist in the proof process. Unfortunately as Jacobs *et al* note that full formal verification of languages like Java requires user assistance [51]: “The semantical complexity of languages like Java... means that code verification is still very complicated and requires a substantial amount of user interaction.” However, Crocker [34] believes that by using a cleaner language and utilising advances in theorem proving and new levels of processing power, fully automated FFV is now possible in practice. Unfortunately even if the proof process can be fully automated, FFV is still very costly since it requires the production of heavyweight specifications formalising the full details of each component being verified. This involves extra effort that, to make matters worse, should be performed before the implementation is written, increasing time to market. Finally, the full formal verification of implementations incorporating loops requires the production of a loop invariant assertion for each loop. Determining suitable loop invariants is a non-trivial exercise.

Projects supporting full formal verification include B [2], SPARK [9], PerfectDeveloper [34], KeY [3], LOOP [49], JACK [19], Jive [72] and Krakatoa [66].

1.2 Thesis contributions

This thesis makes contributions in two areas which are discussed in this section. They are:

1. investigation of the use of a new language loosely based on Java, but using value semantics and containing an operational specification language with frame conditions, and
2. investigation of how different assertion-based verification approaches can be used together in an integrated fashion in different parts of a single software application.

1.2.1 The Omnibus language

The thesis presents a new language called Omnibus. It is designed to use similar concepts to Java but with value semantics for objects. Specifications can be provided in terms of the methods and attributes in the interface of the class with a frame condition logic provided.

The contributions in this work are:

a. a presentation of a specification technique based upon type signature interfaces with support for frame conditions, and a discussion of the problems in providing such support,

b. a discussion of techniques to implement an object-oriented programming language with value semantics,

c. a discussion of the concept of equality with its increased importance in value semantics, and presentation of an automatic lightweight definition and patterns for manual definitions, and

d. an illustration of the challenges in reasoning with reference semantics and a discussion of existing techniques.

Two of the most important language design decisions taken in the development of the Omnibus language are the use of value semantics and an operational specification style. Justification for these choices is given in the following sections.

Why value semantics is used

Objects may either be accessed via references or treated as values. *Reference semantics* is where objects are accessed via references. Java is an example of a language which uses reference semantics. The concepts of reference semantics are familiar to most programmers and are very expressive, providing a powerful mechanism for describing programming solutions. However, they are relatively complicated to reason about and harder to formally model. In *value semantics*, objects are accessed directly, not by reference. SPARK is an example of a language that uses value semantics. Value semantics is less expressive and cannot model certain complex design patterns efficiently. However, it is far easier to reason about and formally model.

The Omnibus language uses value semantics for objects and does not provide any support for reference semantics. We chose to concentrate on value semantics because reference semantics is one of the biggest causes of complexity in verification of languages such as Java/JML. The difficulty of verification of languages built on reference semantics makes the techniques less accessible, increasing the need for user interaction in the theorem proving process. Value semantics offered the possibility of verification techniques that are more accessible. Other projects have used value semantics for the same reason. A prominent example is PerfectDeveloper, which uses a language primarily based on value semantics in order to reduce theorem proving difficulty to the extent that fully automated provers can be used. We wanted to explore the use of value semantics as the basis for a language supported by a range of verification approaches.

Value semantics is also of critical importance even for languages primarily built around reference semantics such as Java/JML. To perform formal verification, the classes in a system must be translated into mathematical logic, and mathematics itself uses value semantics. Value semantics classes can be used to describe the classes in an implementation in a manner that can be directly translated into the underlying logic. Special modelling types of this kind, expressed using value semantics, are used widely in Java/JML. A basis for the specifications, like our value semantics logic, is an important component. This is discussed further in the next section.

There are limitations to value semantics and many of the most interesting outstanding research challenges in software verification are concerned with reference semantics, so a thesis which does not consider reference semantics at all would be somewhat unsatisfactory even under these circumstances. As such we do discuss some of the key challenges in the verification of languages that use reference semantics and promising current approaches for addressing these challenges.

Support for reference semantics could be added to the Omnibus language or the techniques for value semantics presented here could be incorporated into an existing language such as JML which focuses on reference semantics. Reference semantics could be used to provide expressiveness and efficiency with value semantics being used for modelling types and implementations of selected classes.

Why we use operational specifications

Another key language design question is what the *base level of the specifications* should be. Of course, classes can be specified in terms of other classes, but this process must stop somewhere. Leading projects such as JML and Spec# use a set of core types as the base level for specifications. These core types are often referred to as *modelling types* [60]. The problem is then how these modelling types are specified. As Leavens, Leino and Müller say [60]:

> "Modelling types can, of course, be specified using other modelling types, but ultimately some modelling types must be specified in some other way."

The first challenge they discuss in their recent paper [60] is to "develop a technique for formally specifying modelling types in a way that is useful for verification". Modelling types can be provided in a library and should provide core mathematical specification apparatus such as sets and sequences. The types should obey value semantics to be compatible with the underlying mathematical theories used for verification.

Their paper describes four candidate solutions to this problem. Their first candidate solution is to use algebraic specification techniques. Such approaches can be used to describe the methods of a class inductively. This approach is similar to the base-function specification technique of Eiffel [68]. Such specifications are described by Müller [76] as operational interface specifications and are frequently used in run-time assertion checkers. This approach is used extensively in JML, where an operational specification style can be achieved through the careful definition of pure classes. However, no support for frame conditions is provided. Ordinarily in JML, `modifies` clauses can be used to describe the state elements that can be changed, from which the tool can deduce what state elements cannot be changed. However, when writing pure classes in JML, the programmer must implement methods to update the state of the class by returning the new object from the method. This is because, to obey value semantics, the object to which the method is being applied cannot be mutated. The `modifies` clause cannot be used in this circumstance: it can only be used where the target of the method call is being changed.

In this thesis we present an operational specification technique with a frame condition logic to make it more suitable for static verification.

The second proposed solution is to provide an algebraic specification through an equational theory. These kinds of specifications are also used for many JML modelling types. However, among other problems, this technique is unsuitable for run-time assertion checking, which is one of our key interests in this thesis.

The final two proposed solutions are to specify the modelling types in the logic of some standard prover, or to define the modelling types as built-in value types with a pre-defined notion of equality. It is not clear to Leavens, Leino, Müller, or us, how helpful the specifications of the modelling types in the logic of a standard prover would be. The clear problem with using built-in model types is that they would not be easily extensible.

1.2.2 The Omnibus integrated verification approach

The thesis describes how the assertion-based verification approaches RAC, ESC and FFV can be used together in an integrated fashion. Verification tools are usually built to support a single verification approach. This thesis discusses the use of these approaches together in a flexible way, enabling an application to be broken down into sections and different verification approaches used in each section.

The contributions in this work are:

a. a presentation of the case for supporting different approaches for different parts of a single software application,
b. a discussion of the challenges in using the different approaches together, and presentation of guidelines to meet them,
c. a presentation of the concept of verification policies that can be used to manage the use of the approaches together,
d. a discussion of how the integrated verification approach can be used together with the documentation generator and verification certificate system to help support safe reuse of software components, and
e. a discussion of the integrated tool support provided by the Omnibus IDE, focussing on the integrated static verifier which provides support for both extended static checking and full formal verification.

The following sections explain why it is desirable to use the different verification approaches together and why additional support is required to support the safe reuse of software components.

Case for using the approaches together

Integration of verification approaches is desirable because the different 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. Software developers also have varying abilities and do not always have the mathematical training to perform interactive theorem proving as is generally required for FFV. Integrated support for these approaches allows developers with different mathematical training, developing code for different purposes, to use verification at a level of rigour that most suits their needs. The extended static checker and run-time assertion checker, together with the unit testing framework, provide push-button techniques that software developers producing non-critical application-specific code can use relatively cheaply. 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.

Why additional specialised support for reusable components is needed

Assertion-based verification provides for two of the basic needs of safe software component reuse: a mechanism for providing clear descriptions and a basis for trusting the hidden implementations. However, assertion-based verification alone is not sufficient to support the safe reuse of software components. Without specialised support, the users of a component will have to inspect the source code of the component in order to view its specification, and will have to re-verify the component's implementation in order to ensure that the implementation is correct. This is not practicable. For the potential advantages of component reuse to be realised, the users must have separate, clear descriptions of how to use the component, and a basis for trusting the implementation without having to manually re-verify it themselves. They should ideally be isolated from the implementation, i.e. the implementation should be hidden. Our specialised support provides additional facilities to allow this to be achieved.

1.2.3 Strengths and weaknesses of Omnibus

The specification mechanisms of the Omnibus language were specifically designed to support the definition of modelling classes and the language seems to be quite effective at expressing these. The language can also be used to implement applications in their entirety but the limitations of value semantics make it awkward to do so.

Our integrated verification approach appears to hold some promise. Our case study illustrates that the different verification approaches that we support do provide different balances between error coverage and ease of use. The verification policy system allows them to be used together in a highly integrated fashion.

1.3 Related work

This section presents some related languages and verification tools.

1.3.1 Related languages

In this section we review the languages used in the state-of-the-art verification projects. We classify them in terms of two dimensions: whether they use reference or value semantics and the bases for the specifications.

Reference or value semantics as default

Many of the leading verification projects are built around existing programming languages. For example, the JML project is built upon Java and the Spec# project is built upon C#. Facilities are added to support the definition of assertion annotations, but the structure of the packages/classes/methods and the use of reference semantics by default is retained. Classes obeying value semantics are supported via the `pure` modifier. The Eiffel language was designed with assertion-based verification in mind, specifically run-time assertion checking. It also supports reference semantics.

Examples of languages that use value semantics are SPARK, B and Perfect. The SPARK project uses a subset of Ada. B and Perfect are languages designed specifically with formal verification in mind.

Different base levels of specifications

In this section we consider the different techniques used for the specification of modelling types by the different state-of-the-art assertion-based verification projects. We follow Müller [76] in using the classifications: declarative and operational specification languages. In declarative specification languages, specifications are defined in terms of a separate mathematical model. In operational specification languages, specifications are defined in terms of Boolean expressions over the type signature interface of the class.

A prominent example of a declarative specification language is Larch. Using Larch, a program-independent specification, referred to as a universal specification, is defined which provides a collection of mathematical constructs for describing the class. A program-dependent specification of the interface of the class can then be given in terms of the constructs provided in the universal specification. The abstract state of the class will be defined in terms of constructs from the universal specification, and the behaviour of the methods will be specified using pre-conditions, frame conditions and post-conditions over this state. Such languages are not well suited to run-time assertion checking, since the specifications are defined over abstract mathematical constructs that are not directly executable.

Eiffel was one of the first languages to incorporate an operational specification language. In Eiffel, a subset of the query methods of each class are selected as base methods and the rest of the methods in the class are defined in terms of them. Eiffel lacks support for frame conditions and so these must be manually encoded in the post-condition if static verification is to be performed.

There are a number of operational specification languages available for Java associated with the many run-time assertion checking tools, e.g. iContract [58]. However, crucially, most of these still lack frame conditions. An operational specification language which does support frame conditions is the Java Interface Specification Language (JISL) [73]. Frame conditions are supported through a mechanism to explicitly mark areas of the heap that should remain unchanged. This approach requires the programmer to explicitly specify what should remain unchanged, in contrast with the more common modifies-list or changes-list approach, where the programmer specifies what can be changed, and from which the frame condition is then deduced.

Newer languages like JML and Spec# use a hybrid of operational and declarative specifications. The main specification style within these languages is declarative in nature, with specifications written over abstract fields holding values of special modelling types. Once sufficient modelling types have been defined, it is easy and natural for users of the language to define their specifications in terms of fields holding values of these models. However, unlike conventional declarative specifications, the modelling types are defined in an operational style using the source language (i.e. Java/C#) rather than some separate, non-executable mathematical logic. The modelling types are defined using classes written to obey value semantics so that they can be directly translated into mathematics to support static verification. In addition, because they are defined using the source language, they can be executed, thus also facilitating run-time assertion checking. The use of the approaches together in this hybrid fashion enables both formal verification and run-time assertion checking.

The hybrid approach was pioneered by the JML project. JML supports the concept of pure classes which obey value semantics. A library of pure classes is provided with the JML distribution which can be used to express specifications for user classes. The library includes common modelling types such as sets and sequences. Programmers can also define their own modelling classes by writing new pure classes. Programmers can then write specifications for their classes in a declarative style using abstract fields of these modelling types. Unfortunately, little specialised support is provided in JML for the specification of the pure classes that are used for modelling types. Update operations for these classes must be written as methods which return new instances, which means that the programmer receives none of the luxuries of a modifies-list. Instead, they must manually specify how all the state elements of the newly created object relate to the state elements of the original object. The advantages of the hybrid approach can still be achieved within Omnibus by using model functions of no parameters in place of fields. This technique is used extensively in the case study in chapter 8.

1.3.2 Related verification tools

This section presents a survey of related projects which use the assertion-based verification approaches discussed in this thesis.

Eiffel

Eiffel [68] is an object-oriented programming language built around the assertion-based Design-by-Contract (DBC) approach. It was originally developed by Bertrand Meyer in 1985. Support for assertions was built directly into the language from its inception. A run-time assertion checking mechanism is used to detect violations of the specifications provided in Eiffel contracts.

While the Eiffel language was designed to accommodate assertion-based verification, it uses reference semantics and so is closer to conventional programming languages like Java and C#, with assertion annotations, than Omnibus. The supporting tools have also been built around the use of run-time assertion checking rather than static verification although we note that there have been moves towards supporting static verification of Eiffel code [90].

Jass and other Java DBC tools

Jass (Java with ASSertions) [11] is one of a number of RAC tools for Java. It supports an Eiffel-style DBC approach for Java. Assertion annotations are provided in special comments within the source of the class, and the supporting tool generates bytecode implementations containing run-time checks of these assertions. Other RAC tools for Java include iContract [58], jContractor [53] and the JML run-time assertion checker [27] which is discussed below.

Excluding the JML run-time assertion checker, these RAC tools are relatively lightweight extensions of the Java language. They generally do not have a full range of specification constructs (such as frame conditions) to support static verification.

ESC/Java

The original ESC/Java tool [41] was developed by a team at DEC who also produced a tool for Modula-3 called ESC/Modula-3 [35]. The tools use a powerful fully automated theorem prover called Simplify [36]. ESC/Modula-3 and ESC/Java take as input programs written in the Modula-3 and Java languages, respectively, with a range of annotations included in special comments. The ESC/Java tool has been relatively well received by those that have used it in industrial case studies [62].

The ESC/Java tool made an important contribution to the assertion-based software verification field. However, it was quite limited in a number of ways. For example, there was limited support for abstraction in the specifications and the frame conditions logic was primitive. These make it unsuitable for large scale systems.

B

B [2] is a mathematical language and set of associated tools supporting automated and interactive verification. B was developed by Jean-Raymond Abrial and aims to allow mathematically-minded people to develop verified software. B has a software development method associated with it, called the *B method*, which is built around specification refinement. B was used by the French transport industry to develop their safety critical control software.

Other than using value semantics, the Omnibus language is a fairly conventional object-oriented programming language. In contrast, the B language is fundamentally different from conventional object-oriented programming languages and requires different development methodologies.

SPARK

SPARK [9] is a language created by Praxis High Integrity Systems and supported by a range of formal verification and conventional analysis tools. The SPARK language is a subset of Ada, including only features deemed necessary for writing reliable software and permitting analysis and proof. Bounded space and time requirements are imposed by the language, prohibiting dynamic storage allocation, recursion and arrays with arbitrary bounds.

As well as supporting assertion-based verification, SPARK provides comprehensive support for a generalisation of data flow analysis called *information flow analysis*. Information flow analysis incorporates the normal data flow analysis checks that variables are initialised before being read along with checks of how values are used to derive other values. The SPARK approach has been used since the early nineties and has been extensively used to develop systems for the military as well as in the aerospace, rail and security areas. It is particularly well-suited to the development of safety critical systems.

The restrictions that make SPARK so well-suited for the development of safety-critical systems limit its applicability to wider software development. For example, the concept of dynamic binding from object-oriented programming is not supported because it would greatly complicate the bounded memory usage assurances.

PerfectDeveloper

PerfectDeveloper [34] is a formal verification tool produced by Escher Technologies, built upon a new language called Perfect and using a fully-automated theorem prover. It aims to allow everyday programmers to perform full formal verification. The Perfect language is specifically designed with verification in mind. It uses value semantics by default to make verification more amenable to automation. Programmers write stylised specifications from which the tool attempts to generate code automatically. PerfectDeveloper uses an in-house fully automated theorem prover. The code generation and automated theorem proving problems are undecidable, in general, but heuristics can be used to help solve the problem for a wide range of cases. The PerfectDeveloper tool was itself implemented in Perfect (apart from the GUI) in a bootstrapping approach. This has provided a useful case study for the approach [33, 34].

The Perfect and Omnibus languages are quire similar. Both are object-oriented languages primarily built on value semantics. However, the modelling types within Perfect are built into the language and so are not as easily extensible as those in the Omnibus language. The PerfectDeveloper verification tool also aims to support fully automated FFV. We chose to support a range of verification approaches of different levels of rigour in order to cope with the undecidability of the problem.

KeY

KeY [3] is a formal verification tool that is being developed by the Universities of Karlsruhe and Koblenz and Chalmers University of Technology in Gothenburg. It is built upon a commercial Computer Aided Software Engineering (CASE) tool called the Together Control Center (TogetherCC) and targeted specifically at the commercial world. As its input, it accepts UML designs with formal specifications given via UML's Object Constraint Language (OCL) [98] and implementations in JavaCard [26], a subset of Java for developing smart card applications. The KeY tool now also accepts JML. Both interactive and fully automated theorem proving are supported through the KeY prover. The interactive theorem prover has a graphical front-end which can be used to visualise the proof process. Crucially, it retains an OCL/Java-style notation so that users can more easily understand the VCs that are presented to them. A number of case studies have been carried out using the KeY tool [3]. The approach has been applied to parts of the Java Collections Framework, used to develop realistic JavaCard applications and used to verify part of a German rail company's control system.

The KeY tool is built upon the assumption that the fully automated verification of commercial applications is an unachievable goal. To account for this, the tool attempts to provide an accessible interactive verification tool which includes a graphical proof tool and presents the VCs to be proved in the source language instead of some other formalism. These aspects could both be incorporated into Omnibus to make its interactive static verification approach more accessible. However, the KeY project does not address the other goals of Omnibus.

Spec#

Spec# [10] is a programming system under development by Microsoft which consists of the object-oriented Spec# programming language, the Spec# compiler and the Boogie static verifier. The Spec# language is a superset of C#, which is part of Microsoft's .NET platform. The Spec# programming system supports both run-time assertion checking and static verification. A combination of dynamic and static checking is used to implement the verification process. Static verification is supported through the Boogie tool. Spec# is still in the relatively early stages of development and has not been applied extensively yet, but it holds great promise because of its backing by Microsoft and integration with the Visual Studio.NET IDE.

JML

The Java Modelling Language (JML) [20, 59] is a Behavioural Interface Specification Language (BISL) which can be used to annotate Java source files. JML was first developed by Gary Leavens with colleagues and students at Iowa State University, but is now a free software project to which many groups are contributing. JML annotations are written within specially tagged comments.

There is a wide range of tool support for the language including a type checker, a documentation generator [87], a run-time assertion checker [27], a unit testing framework, a range of formal verification tools [19, 49, 66, 72] and an extended static checker [30]. These tools have been used to apply JML to a number of problems, many involving JavaCard. One case study [51] investigated the application of a number of JML verification approaches to a realistic smart card applet from the company SchlumbergerSema. The tools allowed a number of previously unknown problems to be detected, and there was particular enthusiasm towards the ESC/Java2 tool which is discussed below. JML has also been used to verify vote-counting software. Due to tight time constraints, the team that undertook this project did not use any of the formal verification tools for JML, instead opting for a combination of run-time assertion checking and extended static checking.

The JML language is different from the Omnibus language in that it is designed to be fully backward compatible with Java. All aspects of Java are retained, including those that are difficult to statically verify. The Omnibus language does not have to work within this hard requirement and so can explore other areas of the language design space. The different tools in the JML community together cover the range of different verification approaches that Omnibus does. However, they have typically been implemented separately and cannot easily be used together directly.

Run-time assertion checking with JML using the JML compiler

The JML compiler (jmlc) [27] accepts JML-annotated programs and compiles them to Java bytecode instrumented with run-time checks of the assertion annotations. This tool is one of the leading run-time assertion checkers. It incorporates facilities normally seen in full formal verification tools but not run-time assertion checkers that are generally more lightweight.

LOOP

LOOP [49] is a full formal verification tool developed at the University of Nijmegen which translates JML-annotated Java code into theories for the semi-automatic PVS theorem prover [81]. This translation is built upon a formal semantics for sequential Java and JML. This gives LOOP the ability to reason about unannotated code like the KeY project. As a consequence of the use of a semi-automatic theorem prover, verification using LOOP requires skilled user assistance but allows more complicated properties to be handled than any fully automated theorem prover.

JACK

The JACK (Java Applet Correctness Kit) tool [19] has been developed at the research lab of Gemplus, a smart card manufacturer. It supports the Simplify and PVS theorem provers along with the Atelier B toolkit's automated and semi-automated theorem provers. JACK aims to be usable by normal Java developers and attempts to hide the underlying mathematical concepts from the user. This is achieved by presenting VCs in Java/JML-like notation. JACK is available as a plug-in for the Eclipse IDE.

Jive

Jive [72] is a formal verification tool developed by the Universities of Hagen and Kaiserslauten. It can be used in conjunction with either of the interactive theorem provers, PVS and Isabelle/HOL. Jive originally used its own specification language but has been adapted to work with JML. Unlike most of the tools which automatically generate VCs, in Jive, Hoare rules are either applied manually or via strategies (Java programs that automatically apply several Hoare rules). The Jive tool incorporates a GUI which gives users full visual control over the verification process.

Krakatoa

Krakatoa [66] is another formal verification tool for JML developed at Université Paris-Sud. It generates VCs through an external tool called WhY and then uses the Coq theorem prover (among others) to prove them. The input language for the WhY tool is an ML-like minimal language with very limited support for imperative features like references and exceptions. The translation process centres on a formalisation of the heap.

Extended Static Checking with JML using ESC/Java2

The ESC/Java tool used an assertion annotation language similar to JML, but simpler. Work on the tool halted when the Compaq SRC group was disbanded. The project was re-launched as an Open Source project by Joe Kiniry and David Cok under the name ESC/Java2 [30]. The new tool accepts the JML syntax and has been re-packaged in a more accessible form. The current state of this work is reported in the implementation notes distributed with the tool [31].

Functional Programming Languages

Finally, we consider some functional programming languages which have support for formal verification.

ACL2

ACL2 (A Computational Logic for Applicative Common Lisp) [54] is a functional programming language, based on Common Lisp, incorporating support for formal verification. The associated tool support centres on an automated prover which was developed as a successor to the Boyer-Moore theorem prover. The logic of the prover is an extension of the standard first-order predicate calculus with equality to include recursive function definitions and the concept of mathematical induction. Users of the language can assist the proof process by supplying lemmas to guide the theorem prover in the proof of tricky conjectures. This is particularly important for the handling of inductive assertions.

In being a pure functional programming language, ACL2 is built on value semantics, like Omnibus. However, there is no object-oriented support. The approach of supplying lemmas in order to assist an automated prover can also be used in most other automated verification tools, including Omnibus. The support for recursive function definitions and mathematical induction is a key strength of ACL2 because it enables more sophisticated properties to be specified and verified than a pure first-order logic with equality.

EML

EML (Extended ML) [89] is a framework for the specification and formal development of SML (Standard ML) programs. The EML language is a generalisation of SML which allows unexecutable constructs, that are useful for writing specifications, to be used in addition to the executable constructs of SML. SML can then be thought of as an executable subset of EML. This arrangement allows abstract specifications to be defined using the unexecutable constructs. Implementations can be defined by using only executable constructs. As such, EML can be used to cover the full development process from specification to implementation. An abstract EML specification can be produced initially and then a system of refinement carried out until executable SML is obtained. Proof obligations can be generated to ensure that each refinement step is consistent with the level above.

Unfortunately, there is limited tool support for EML consisting of just basic parsing and type checking. Crucially, no tool support integrated with a theorem prover is available.

RAISE

RAISE (Rigorous Approach to Industrial Software Engineering) [43] is an industry-focussed formal method which has run for over 20 years. It covers the whole development process, from specification, through design to implementation via a step-wise refinement paradigm. The project uses a language called RSL (RAISE Specification Language) which supports many different language styles including functional programming (e.g. Lisp-style) and imperative programming (e.g. Java-style). The process covers most activities associated with software development, including requirements capture, documentation, and project management. An integrated set of tools is provided to support the language and method. These include formal verification tool support but less formal justification approaches are also supported, where *rigorous arguments* can be provided in place of a formal proof. A rigorous argument informally describes how a proof for a property could be constructed.

The RSL language is large and allows the use of many different styles. However, it does not fully support object-oriented programming. There is no concept of dynamic binding and no support for the dynamic instantiation of objects within the language. The tool support also does not cover all aspects of the verification process. The *formal conditions*, needed to prove the refinement is valid, must be generated manually. However, the idea of informal proof justification could be incorporated into other projects, including Omnibus.

Clean/SPARKLE

Clean is a pure functional programming language which has verification support provided by the SPARKLE tool [106]. A key strength of the Clean language is the inclusion of the concept of uniqueness typing, which allows mutation and destructive updates to be used without violating value semantics. This technique works by ensuring that values which can be mutated are clearly identified and the aliasing of these values monitored closely. SPARKLE is an interactive theorem prover for Clean. It can be used to formally verify properties of Clean programs. It has a graphical proof tool with a sophisticated window-based user interface. Verification conditions are presented in a subset of the Clean language so are easily understandable to users. Some automation is possible through a system of proof tactics.

The Clean language is a pure functional programming language and so quite far from the object-oriented Omnibus language. The SPARKLE proof tool is attractive and shares many of the strengths of KeY.

1.4 Structure of the thesis

The remainder of this thesis is structured as follows.

Chapters 2 to 4 are mainly concerned with the Omnibus language. Chapter 2 gives an overview of the language. Chapter 3 discusses some particular aspects of the language in more detail including our frame condition logic, the implementation of value semantics and equality. Chapter 4 outlines some of the challenges of reference semantics.

Chapters 5 to 7 are mainly focussed on the Omnibus integrated verification approach. Chapter 5 illustrates why it is desirable to use the approaches together and how this can be safely achieved through the use of verification policies. Chapter 6 discusses how the framework can allow software components to be reused safely. Chapter 7 describes how our integrated verification tool supports the different approaches.

Chapter 8 presents a case study of the use of our language and verification approach. Chapter 9 presents our conclusions and discusses possible future directions for our work.

Chapter 2

Overview of the Omnibus language

This section presents an overview of the Omnibus language. Specific aspects are discussed further in Chapter 3. Full details of the syntax including an explanation of the style of EBNF used to describe the syntax in this chapter are given in appendix A.

2.1 Fundamental concepts

Omnibus is an object-oriented language designed to be amenable to formal analysis. It is superficially similar to Java, using similar concepts of packages, classes, methods, expressions, statements, accessibility levels, name-declaration binding, etc. but incorporates a behavioural interface specification language and uses value semantics for objects.

An Omnibus project consists of a set of class definitions. Each class can contain declarations of attributes, a range of static and instance methods and test cases. Attributes allow the internal state of a class to be defined. 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. Static functions can be used to define methods which are not applied to any object. The declaration of a method starts with a keyword identifying the type of method.

Constructors and static functions are class methods whereas functions and operations are object methods. Test cases can also be defined within a class. These can be used to describe how object instances of the class should behave for specific concrete values. Omnibus supports `public`, `protected`, and `private` modifiers which change the accessibility level of language constructs in the same way that they do in Java.

In Omnibus, all objects are immutable with the system. New objects are created 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 and has a default definition provided for every object.

Method implementations can be defined using a Java-style implementation language containing an assignment statement, operation call statements, a declaration statement, an `assert` statement, an `if` statement, a `for` loop, a `foreach` loop and a `while` loop. These statements can be used to manipulate local variables and the attributes of the containing class. Loops can be annotated with loop invariant assertions. Omnibus is a block-structured language with static typing, like Java.

Full details of these core elements of the Omnibus language are discussed in more detail in appendix A.

2.2 Behaviour specifications

The aspects of the Omnibus language discussed in appendix A are largely concerned with replicating support for facilities that are present in languages such as Java. In this section we discuss the support for the specification of the behaviour of methods using the Omnibus language.

2.2.1 Principles of Omnibus behaviour specifications

In Omnibus the behaviour of methods can be described using *behaviour specifications* which are constructed in the conventional style from `requires`, `changes` and `ensures` clauses which give pre-conditions, frame conditions and post-conditions, respectively. There can be different levels of specifications for the different accessibility levels in an Omnibus class. In this section we consider specifications at a single level of accessibility and generalise the concept to multiple levels in section 2.2.3.

In Omnibus, as in Java, the current state of an object instance of a class consists of the values of the attributes used to implement the class. We refer to this as the *concrete state*. The behaviour specifications of the methods in the class are described in terms of the attributes and other methods. Methods can be specified in terms of other methods, but this process must stop somewhere with some methods not specified in terms of other methods. The attributes provide such a means of halting this recursive definition process because they themselves are not defined in terms of any other elements. Some methods can be defined in terms of the attributes, then other methods can be defined in terms of them. However, it is good practice for the attributes that are used to implement a class to be declared as private or protected to hide the details of the implementation from users of the class. In such cases, the public behaviour specifications cannot refer to these attributes because they are not publicly accessible. So we again have the problem that some methods must not be specified in terms of other methods but can no longer define these methods in terms of the attributes. What is done instead is that some of the functions in the class can be declared with the special `model` modifier and are not described in terms of the other methods, i.e. they must not have post-conditions. Functions declared with the `model` modifier are referred to as *model functions*. Just as the attributes that are used to implement a class together form a representation of the concrete state of the class, the attributes and model functions that are visible at a particular level are said to represent the *abstract state* of the class. The behaviour of the other methods (the remaining functions along with the constructors and operations) is then defined at that level in terms of them.

This is the same technique as used by Eiffel. In Eiffel the functions that are part of the abstract state are called base functions but are not specifically declared with a separate modifier.

2.2.2 Notes on the specifications of class body elements

In this section we discuss some points about the specifications of different class body elements.

The attributes of an Omnibus class are an implementation detail and should ideally play no part in the specification of a class. There is currently no support for specification-only attributes within Omnibus. In other words, it is not possible to express that an attribute should be part of a public abstract state of the class but not its private concrete state. Public model functions can be coupled with private attributes for that purpose.

Functions can be divided into three categories: static functions, model functions and derived functions. The behaviour of static functions is specified via an `ensures` clause (and a `requires` clause, if required). The behaviour specification of a model function may include a `requires` clause if appropriate but not an `ensures` clause. Instead, the behaviour specifications for the constructors and operations of the class will describe how the values of the model functions change over the lifetime of an object. Invariants (which will be discussed in section 2.3) can be used to express properties of the results of model functions that would usually be given in an `ensures` clause. Derived functions are declared without the `static` or `model` modifiers. Their behaviour must be specified in terms of the attributes and functions via an `ensures` clause (and a `requires` clause, if required).

Constructors are used to create instances of classes. The resulting object should be described in terms of the value of the model functions and attributes via an `ensures` clause. A constructor should not be specified with a `changes` clause since the values of all fields and model functions are undefined before the constructor executes, so all components of the state must be changed by the constructor. This is taken as implicit.

When specifying operations, the `changes` clause should be used to describe which attributes and model functions have their values changed. An `ensures` clause must be used to describe how they are changed.

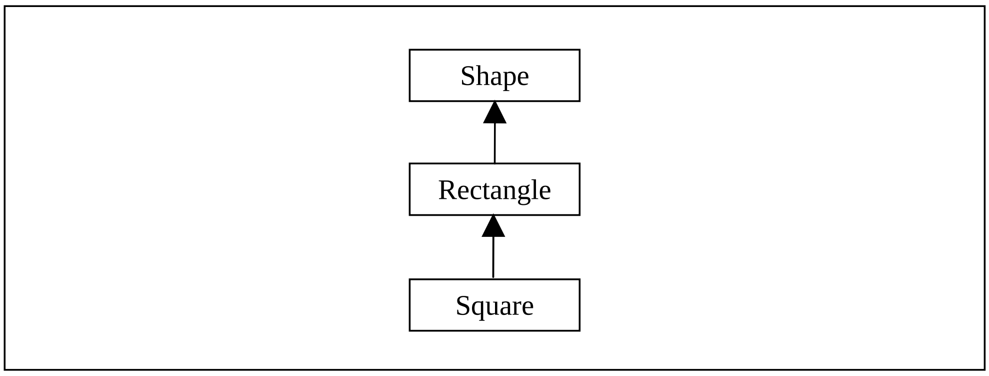
2.2.3 Levels of specification

As is conventional within object-oriented languages that are used for verification, Omnibus requires that, where inheritance is used, the *principle of substitutability* [63] is respected. This principle states that any property which is provable about objects x of type T must be true for objects y of type S where S is a subtype of T [64]. This is essential in order to ensure that the principle of polymorphism from object-oriented programming, the ability to treat instances of a class S as if they were instances of a superclass T, does not introduce the possibility of errors. We will consider the details of how this is checked in section 2.3.2, but the basic principle is to check that the behaviour specification of each of the overridden methods is consistent with the corresponding method specification in the superclass.

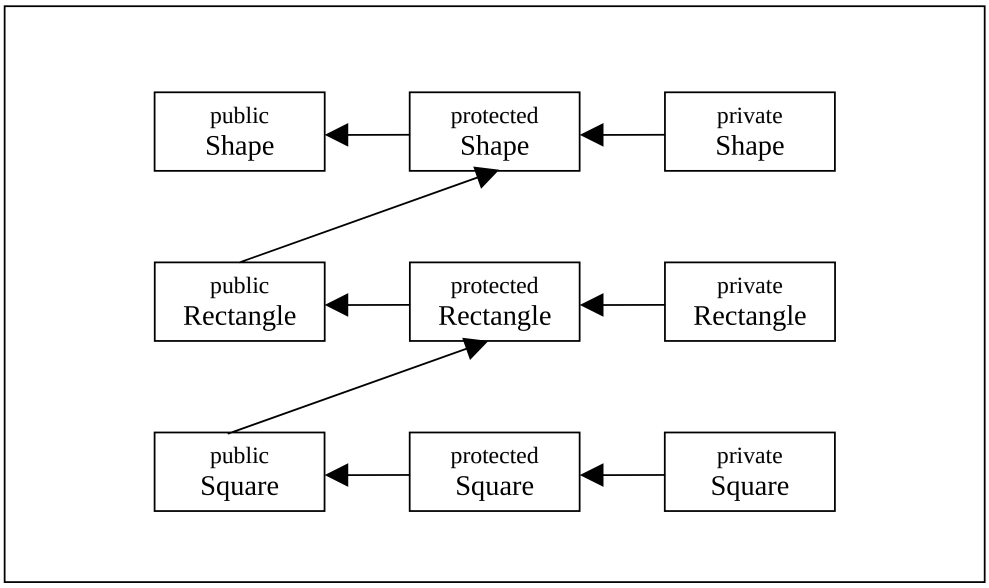
It can also be desirable to have different specifications for a method within a single class for different accessibility levels. For example, we may want a particular method to be a model function at the public level and used as part of the abstract state, but then redefine that function in terms of private concrete fields at the private level.

The principles involved in providing different specifications for a method at different accessibility levels are the same as those for overriding a method from a superclass. In both cases, the specification at the lower level (i.e. the lower accessibility level or the subclass) must be consistent with the specification at the higher level (i.e. the higher accessibility level or the superclass). Both cases also need facilities for handling inheritance of portions of specifications and to ensure that requirements from higher levels of the hierarchy are respected. This has led us to unify the two concepts.

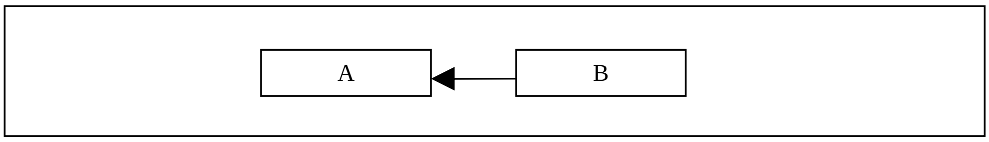
Consider the following inheritance hierarchy where `Rectangle isa Shape` and `Square isa Rectangle`.



The following diagram shows the `Shape` hierarchy with the different levels of the specification for each class shown. The key point is that all directed arrows are treated equivalently regardless of whether they are concerned with inheritance between classes or the different levels of specification within a single class.



To describe how these relationships are handled, we can abstract away from the type of the relationship. All we need to know is that there are two levels of specification in the hierarchy, connected by an arrow.



Such a relationship is then interpreted as follows:

1. Instances of `B` are acceptable wherever an instance of `A` is expected. This makes logical sense since this is part of the definition of inheritance. Similarly, if you have a private level view of an object then you can also interact with it through the public interface.
2. Instances of `A` may or may not also be instances of `B` but, if they are, then they can be cast so that their static types are `B`. Just as if you have a `Shape` it may or may not be castable to a `Rectangle` (it could be a `Circle`, not a `Rectangle` or `Square`), if you have a public `Rectangle` it may or may not be castable to a private `Rectangle` (it could be a `Square`, not a private `Rectangle`).
3. All the requirements (i.e. invariants, constraints and initiallys discussed in section 2.3.1) of `A` are inherited by `B`. Again, this makes sense for both inheritance and moving between views of an object. In both cases the invariants that were assumed to be true in proofs at higher levels must also be true at the current level.
4. Methods are either *introduced*, *inherited* or *overridden*.
   a. Introduced methods: A method is introduced if it is defined in `B` and it is not defined in `A` (either in `A` itself or inherited from a level somewhere above `A`). Constructors are always introduced methods: they cannot be inherited or overridden. Constructors must be shown to establish all the invariants and initiallys, either defined locally in `B` or inherited from `A`. Introduced operations must be shown to maintain the truth of all the local and inherited invariants, and to satisfy all the local and inherited constraints. These laws are discussed in section 2.3.2.
   b. Overridden methods: A method is overridden if it is defined in both `B` and `A` (either in `A` itself or inherited from a level somewhere above `A`). The specification of the

method in B must be compatible with the specification of the method in A. Specifically, the pre-condition can be maintained or weakened, but not strengthened and the post-condition can be maintained or strengthened but not weakened. Overridden operations must be shown to maintain the truth of all the local and inherited invariants, and to satisfy all the local and inherited constraints. These laws are discussed in section 2.3.2.

c. Inherited methods: A method is inherited if it is defined in A (again, either in A itself or inherited from a level above A) and not redefined in B. Through the verification of A, the inherited operation can be assumed to satisfy the invariants and constraints of A. We need only show that it also satisfies the invariants and constraints defined locally in B. These laws are discussed in section 2.3.2.

The situation becomes more complicated when B extends or redefines the abstract state of A, but these complications are the same for both forms that the relationship can take.

2.2.4 Light and full behaviour specifications

There are two syntaxes for behaviour specification supported by Omnibus. These are referred to here as *light behaviour specification* and *full behaviour specification*. Light behaviour specifications provide a single behaviour specification for a method at a single accessibility level, while full behaviour specifications allow multiple behaviour specifications to be provided for different accessibility levels. JML has similar concepts and refers to them as lightweight and heavyweight specifications, but we use the terms lightweight and heavyweight specifications to refer to expressiveness of the specifications.

Lightweight specifications are useful for simple examples where the behaviour of the public derived methods in a class will be described in terms of public model functions. However, in more complicated examples involving inheritance, a class may have a high-level public behaviour definition for clients purely in terms of model functions, a medium-level protected definition exposing some concrete attributes to subclasses, and a low-level implementation entirely in terms of the concrete attributes. Thus, it may be desirable to define some functions as model functions at one accessibility level and derived functions at another. Full behaviour specifications make this possible. Light behaviour specifications are de-sugared, by the verification tool, to full behaviour specifications with a single behaviour level with the same accessibility as the method itself. This is a model if and only if the method itself was declared with the `model` modifier.

For example, the following is an example of light behaviour specifications for the public level of methods from a `Map` class:

```
public model function contains(k:Key):boolean

public model function value(k:Key):Value
  requires contains(k)
```

For convenience, the `requires`, `changes` and `ensures` clauses are written immediately after the method header and provide a specification for the accessibility level which the method is declared with.

The light behaviour specification of the `contains` method is translated by the verification tool into a full behaviour specification with a single public behaviour specification containing the `model` modifier and no `requires`, `changes` or `ensures` clause. The light behaviour specification of the `value` function is translated into a specification with a single public behaviour specification containing the `model` modifier and the specified `requires` clause.

If we were to produce an implementation of this class, we would be required to give a private specification for the public model functions. This would require the use of full behaviour specifications such as the following:

```
public model function contains(k:Key):boolean
  private ensures result = (exists (p:Pair[Key,Value]
                                   where col.contains(p)):
                              p.first() = k)

public model function value(k:Key):Value
```

```
  public requires contains(k)
  private requires contains(k)
           ensures exists (p:Pair[Key,Value]
                                    where col.contains(p)):
                      p.first() = k && result = p.second()
```

Here public and private behaviour specifications are provided for the methods. These are identified by placing the appropriate accessibility modifier before the `requires`, `changes` and `ensures` clauses.

There is also a certain amount of de-sugaring involved when using full behaviour specifications. If the method is declared with the `model` modifier then the behaviour specification with the same accessibility as the method is set to be a `model`. If there is no specification with the same accessibility as the method then one is created with no `requires`, `changes` or `ensures` clause.

In its second definition, the `contains` function is declared with the `model` modifier so the specification with the same accessibility as the method itself is changed to be `model`. There is no public behaviour specification given so one is created. In the `value` function, there is already a public behaviour so the `model` modifier is simply added to it.

2.2.5 Calculation of pre- and post-conditions

The `requires` and `ensures` clauses are used to give the pre- and post-conditions of a method. These clauses consist of a comma-separated list of optionally labelled assertions. These assertions are then conjoined (i.e. logically ANDed). The `changes` clause is used to provide a frame condition and can only be used in behaviour specifications for operations. Frame conditions define a bound on what a method can change. These should contain a comma-separated list of model function names, attribute names and model functions applied to some concrete parameters. The `changes` clause should be used to say what an operation can change, and the `ensures` clause should describe how those things are changed. Together, they give a complete description of the manipulation of an object. Further details of the calculation of the frame condition from `changes` clauses are given in the next section.

2.2.6 Calculation of frame conditions

The Omnibus verification tool translates `changes` clauses into a `nochange` predicate over the initial and final states of the object to which the operation is being applied. This predicate expresses that all elements of the state that are not mentioned in the `changes` clause do not change. Elements in the `changes` list are either model function names, attribute names or model functions applied to some concrete parameters. The calculation of the `nochange` predicate is carried out by considering each of the model functions and attributes in the class and checking whether it is in the `changes` list.

The `nochange` predicate is initialised to the boolean literal `true` and then the algorithm proceeds by conjoining assertions to this value.

For each attribute

```
  attribute att:type
```

If `att` appears in the `changes` list, no changes are made to the `nochange` predicate.

If `att` does not appear in the `changes` list, the following assertion is conjoined to the `nochange` predicate:

```
  this.att = old this.att
```

For each model function

```
  model function func(params):type
    requires func_pre
    ensures func_post
```

where `func` is the name of the function, *`params`* is the list of formal parameters and `func_pre` and `func_post` are the `requires` and `ensures` assertions which are translated into special boolean-returning methods of the class during the static verification process.

If `func` appears in the `changes` list without any parameters, no changes are made to the `nochange` predicate.

If `func` does not appear in the changes list, the following assertion is conjoined to the `nochange` predicate:

```
  forall (params):
    old this.func_pre(params)
    && this.func_pre(params)
  ==>
    old this.func(params) = this.func(params)
```

This expresses that for all parameters that satisfy the pre-conditions of the function for the old and new objects, the corresponding values returned by the function are equal. The syntax used is that of the Omnibus language and is described in appendix A.

If `func` appears in the `changes` list as `func(actparams)`, the following assertion is conjoined to the `nochange` predicate:

```
  forall (params):
    params != actparams
    && old this.func_pre(params)
    && this.func_pre(params)
  ==>
    old this.func(params) = this.func(params)
```

This expresses that for all parameters, other than those specified, that satisfy the pre-conditions of the function for the old and new objects, the corresponding values returned by the function are equal. Note that the assertion `params != actparams` corresponds to a number of not equals assertions for each of the formal parameter/actual parameter pairs.

We now define the post-condition of the operation by conjoining the values of the `ensures` clause with the `nochange` predicate.

2.2.7 Example of calculation of pre- and post-conditions

The following example presents part of a heavyweight specification for a simple Omnibus class for modelling a sequence of elements. This is achieved through the definition of a `Sequence` class with an `Element` template parameter. One of the model functions in this specification, `elementAt`, has a parameter and is defined for values of that parameter between `0` and one less than the `size` of the `Sequence`. The example helps illustrate our handling of the `changes` clause in more detail.

```
spec class Sequence[Element] {
  model function size():integer
  model function elementAt(i:integer):Element
    requires i >= 0 && i < size()
  function isEmpty():boolean
    returns size() = 0
  constructor empty()
    ensures size() = 0
  operation add(e:Element)
    changes size
    ensures size() = old size() + 1,
      elementAt(size()-1) = e
  operation set(i:integer, e:Element)
    requires i >= 0 && i < size()
    changes elementAt(i)
    ensures elementAt(i) = e
  operation addAt(i:integer, e:Element)
    requires i >= 0 && i < size() + 1
    changes size, elementAt
    ensures size() = old size() + 1,
     forall (j:integer := 0 to i – 1):
      elementAt(j) = old elementAt(j),
     elementAt(i) = e,
     forall (j:integer := i+1 to size()-1):
      elementAt(j) = old elementAt(j-1)
}
```

The values of the pre- and post-conditions of these methods are a follows:

The pre- and post-conditions of the `size`, `elementAt`, `isEmpty` and `empty` methods are relatively straightforward since they do not involve any `changes` clauses.

```
function size():integer
pre-condition  := true
post-condition := true

function elementAt(i:integer):Element
pre-condition  := i >= 0 && i < size()
post-condition := true

function isEmpty():boolean
pre-condition  := true
post-condition := result = (size() = 0)

constructor empty()
pre-condition  := true
post-condition := size() = 0
```

The specification of the `add` operation is a bit more complicated. The `changes` clause includes the `size` model function without any parameters and so, as laid out in the previous section, no reference is made to this in the generated frame condition. The `elementAt` model function is not referred to in the `changes` clause and so the frame condition defines that, for all parameters values which satisfy the pre-condition of the function, it should remain unchanged. The `ensures` clause describes that the size increases by one and so the value `size()-1` satisfies the pre-condition of `elementAt` after the operation whereas it did not before the operation (there was no **`old`** `elementAt(size()-1)`, i.e. no **`old`** `elementAt(`**`old`** `size())`).

```
operation add(e:Element)
pre-condition  := true
post-condition :=
size() = old size() + 1
&& elementAt(size()-1) = e
&& (forall (j:integer):
  old this.elementAt_pre(j)
  && this.elementAt_pre(j)
==>
  old this.elementAt(j) = this.elementAt(j)
)
```

In the specification of the `set` operation, the `changes` clause describes that the `elementAt` model function is changed for the value `i`. This leads to the generation of a frame condition that the `size` function is not changed and that the value of the `elementAt` model function is unchanged for all values which are not equal to `i`, and do not satisfy the pre-condition of `elementAt` before and after the operation.

```
operation set(i:integer, e:Element)
pre-condition  := i >= 0 && i < size()
post-condition :=
elementAt(i) = e
&& size() = old size()
&& (forall (j:integer):
  i != j
  && old this.elementAt_pre(j)
  && this.elementAt_pre(j)
==>
  old this.elementAt(j) = this.elementAt(j)
)
```

In the `addAt` operation, the parameter values for which the `elementAt` function is changed cannot be enumerated and so the function is listed without parameters along with the `size` function. This means no frame condition is added.

```
operation addAt(i:integer, e:Element)
pre-condition  := i >= 0 && i < size() + 1
post-condition :=
size() = old size() + 1,
&& (forall (j:integer := 0 to i – 1):
  elementAt(j) = old elementAt(j)
) && elementAt(i) = e
&& (forall (j:integer := i+1 to size()-1):
  elementAt(j) = old elementAt(j-1)
)
```

2.3 Requirements specifications and their verification

In this section we explain the use of requirements and then present the laws for their verification using FFV. For RAC and ESC, checks of the requirements are embedded within the bodies of the methods and so are verified using an extension of the techniques used for implementations which are described in appendices D and E.

2.3.1 Requirements specifications

In the preceding sections we have considered the mechanisms provided within the Omnibus language for the specification of the intended behaviour of methods using `requires`, `changes` and `ensures` clauses. These specifications are used to give a basis for the checking of implementations. However, behaviour specifications can be difficult to write and prone to errors themselves. The standard approach to this problem is to provide a higher level of specification construct for providing assertions that should hold over groups of methods. This simply provides another level of redundancy which can be checked against. The term we use to refer to these higher level specifications are *requirements specifications*.

Three types of requirements are supported by Omnibus: *invariants*, *constraints* and *initiallys*. These should be declared within the body of a class, alongside the methods and attributes. All requirements can be declared with an accessibility modifier.

Requirements specifications like invariants, constraints and initiallys are used in most assertion-based languages like Eiffel and JML. Our use of requirements specifications to verify properties of behaviour specifications is based on Aslan [8]. In contrast, JML does not require that the behaviour specifications themselves satisfy the requirements specifications. Instead, for all verification approaches with JML, the requirements are checked as part of the verification of the implementations. The advantage of this is that there is a single interpretation of requirements specifications for all the approaches. However, we will see in the chapter 8 that our approach can uncover further errors. It also allows properties of the behaviour specifications to be verified before an implementation is produced.

Invariants

Invariants are assertions that should hold over an object at all times. More correctly, invariants should hold over the internal state of an object at all times when it is externally accessible. An object is externally accessible after it is created via a constructor call and after each operation call. The reason we use this looser definition is that we allow an invariant to be temporarily invalidated during the execution of a constructor or operation. However, the invariants must also hold when one local method calls another local method which has the invariants as part of its pre-condition, as discussed in section 3.1. All the constructors of a class should establish the truth of the invariants, and all the operations should maintain their truth.

Constraints

Constraints are assertions that should hold over any two consecutive, externally accessible states of an object. They can refer to methods and attributes in the earlier of the consecutive states using the `old` keyword and to those in the later by omitting this prefix. The only method which the constraint is concerned with is the operation since it is the only method that can update the value of an object.

Initiallys

Initiallys should hold over freshly constructed objects where a freshly constructed object is an object which has been created by a constructor but has not yet been updated via an operation. As such, all constructors should establish the truth of these assertions.

2.3.2 Laws

In the previous section we introduced Omnibus requirements. In this section, we present a set of basic rules for checking that the methods of a class satisfy these requirements. We start by presenting the basic laws for specifications at a specific level in a class and then discuss how the laws can be extended to handle inheritance and specifications at different accessibility levels. These laws are based upon those used in the Aslan [8] system.

Basic laws

Informally, initiallys should be satisfied by the constructors, invariants should be established by constructors and maintained by operations, and constraints should hold across operations.

For convenience, let us assume that the symbols `init`, `inv` and `con` contain the conjunctions of all the initiallys, invariants and constraints, respectively, of the class being considered. In our tool implementation, these are checked separately so that more specific error messages can be generated, but that is not relevant from a logic point of view.

To further simplify the presentation of the following rules, let us also require that a de-sugaring algorithm is applied to all assertions as a preliminary stage. This de-sugaring algorithm consists of two steps: the implicit ‘`this.`’ is added to local methods and attributes, and assertions of the form ‘**`old`** `this.member...`’ are then translated to ‘`old_this.member...`’. We can then model Omnibus operations as mathematical functions from an `old_this` object and some parameters to a `this` object, Omnibus functions as mathematical functions from a `this` object and some parameters to a `result` and Omnibus constructors as mathematical functions from some parameters to a `this` object. We can express the pre- and post-conditions as predicates over these parameters and objects.

Now, consider the generalised description of a constructor given below

```
constructor con(params)
  requires pre
  ensures post
```

The constructor is described using a pre-condition given in the `requires` clause and a post-condition given in the `ensures` clause. We should then check that it establishes the initiallys and invariants of the class. We can express this as follows:

```
Constructor invariant establishment and initially satisfaction
      law:
forall (params, this):
  pre(params) && post(params, this)
==>
  init(this) && inv(this)
```

This states that if the pre-condition holds over the parameters of the constructor, and the post-condition holds over the parameters and the created object, then the initiallys and invariants hold over that object. The *`params`* word is italicised to remind the reader that it should be replaced by the zero or more parameters of the constructor.

Now, consider the generalised description of an operation:

```
operation op(params)
  requires pre
  changes chgs
  ensures ens
```

The pre-condition is given by the `requires` clause. The post-condition is given by the `changes` and `ensures` clauses, with the `changes` clause describing what aspects of the state change and the `ensures` clause describing these changes.

We now define the post-condition of the operation by conjoining the values of the `ensures` clause, `ens`, with the `nochange` predicate:

```
post(old_this, params, this) :=
                                ens(old_this, params, this)
                                && nochange(old_this, this)
```

These pre- and post-conditions should maintain the invariant and satisfy the constraints of the class. We can express this as follows:

```
Operation invariant maintenance and constraint satisfaction
     law:
forall (old_this, params, this):
  inv(old_this) && pre(old_this, params)
  && post(old_this, params, this)
==>
  inv(this) && con(old_this, this)
```

This states that if the invariant holds over the initial object, the pre-condition holds over that object and the parameters, and the post-condition holds over the initial object, the parameters and the resulting object, then the invariants hold over the resulting object and the constraint holds over the two objects.

Inheritance laws

Omnibus allows classes to be declared as subclasses of other classes using the `isa` clause. Such classes can then inherit or override the members of the superclass, and instances of these classes can be used whenever an instance of the superclass is expected. Similar checks are required for consistency between different levels of specification within a single class. For convenience, in the rest of the section we focus on inheritance, but the same laws apply for different levels of specification in a single class.

To ensure that we can safely use the specification of a class in place of the specification of its subclass, we must make sure that the two class definitions are compatible. We must ensure two things: that the requirements of the superclass hold over the class, and that methods which override a superclass method are compatible with the method they override.

Requirements (i.e. the invariants, constraints and initiallys) give a high-level definition of some essential characteristics of a class. If a class is going to be usable in place of its superclass then the requirements of the superclass should hold over the class. Of course, the requirements introduced in the subclass should also hold over the methods inherited from the superclass.

We have already seen that methods which override a method from the superclass should have a specification which is consistent with the specification of the method they override. More specifically, their `requires` clause should not be more restrictive than the one in the superclass. If it were, when the specification from the superclass is used in place of the specification from the subclass, values would be accepted that violated the `requires` clause of the subclass specification. Conversely, the `changes` and `ensures` clauses of the specification of the method in the subclass should imply the `changes` and `ensures` clauses of the overridden method. If this were not the case, then when using the specification from the superclass, the system would conclude consequences of calling the method which were not valid.

For convenience, let us introduce three symbols `super_init`, `super_inv` and `super_con` to represent the conjunctions of all the initiallys, invariants and constraints, respectively, from all the superclasses of the class being considered.

Let us now consider the checks we need to make to ensure these properties. Firstly, constructors should establish the invariants and initiallys from the superclass (and all of its superclasses) as well as the local ones.

```
New Constructor invariant establishment and initially
     satisfaction law:
forall (params, this):
  pre(params) && post(params, this)
==>
  init(this) && inv(this)
  && super_init(this) && super_inv(this)
```

We will now consider operations. There are three types of operations: introduced operations, inherited operations and overridden operations. Introduced operations are operations for which there is no definition in the superclass. Before we added inheritance, all operations were introduced operations. In the presence of inheritance, introduced operations should satisfy the invariants and constraints of the superclasses as well as the local ones.

```
Introduced Operation invariant maintenance and constraint
     satisfaction law:
forall (old_this, params, this):
  inv(old_this) && super_inv(old_this)
  && pre(old_this, params) && post(old_this, params, this)
==>
  inv(this) && con(old_this, this)
  && super_inv(this) && super_con(old_this, this)
```

Note that we can also assume the superclass invariants hold over the starting object.

Inherited operations are operations which are defined in the superclass but have no corresponding definition given in the subclass. These operations are implicitly inherited by the subclass and can be used as if they were identically redefined in the subclass. The invariants and constraints of the subclass and superclass should hold over all operations, including the inherited ones. However, the verification of the superclass has already ensured that the operation satisfies the invariants and constraints of the super classes. We need only check that it also satisfies the local invariants and constraints.

```
Inherited Operation invariant maintenance and constraint
     satisfaction law:
forall (old_this, params, this):
  inv(old_this) && super_inv(old_this)
  && pre(old_this, params) && post(old_this, params, this)
==>
  inv(this) && con(old_this, this)
```

Overridden operations are operations which are defined in both the subclass and superclass. In that case, the method in the subclass is said to override the method in the superclass and when instances of the subclass are used in place of the superclass, the method definition in the subclass will be the one that is actually invoked. Once again, we must ensure that the invariants and constraints of the sub- and super classes hold over the operation. This time, we must also ensure that the specification of the method in the subclass is compatible with the specification in the superclass. Firstly, any values that satisfy the pre-condition in the super class method should satisfy the pre-condition in the subclass.

```
Overridden operation application law:
forall (this, params):
  super_pre(this, params) && inv(this) && super_inv(this)
==>
  pre(this, params)
```

Secondly, any values that satisfy the post-condition of the local method should satisfy the post-condition of the superclass. We also need to ensure that the method in the subclass satisfies the requirements of the subclass and superclass. However, the verification of the superclass will already have ensured that the specification of the superclass method satisfies the requirements of the superclass. Since we have already proved the new specification implies the overridden one, we need only check the local invariants and constraints.

```
Overridden operation refinement, invariant maintenance and
     constraint satisfaction law:
forall (old_this, params, this):
  super_pre(old_this, params) && inv(old_this)
  && super_inv(old_this) && post(old_this, params, this)
==>
  super_post(old_this, params, this)
  && inv(this) && con(this)
```

We also need to check that functions in the subclass that override functions in the superclass are compatible. Again, the values that satisfy the pre-condition of the method in the superclass should satisfy the pre-condition in the subclass, and values that satisfy the post-condition in the subclass should satisfy the post-condition in the superclass.

```
Overridden function application law:
forall (this, params):
  super_pre(this, params) && inv(this) && super_inv(this)
==>
  pre(this, params)

Overridden function refinement law:
forall (this, params, result):
  super_pre(this, params) && inv(this) && super_inv(this)
  && post(this, params, result)
==>
  super_post(this, params, result)
```

The laws we presented in this section are a generalisation of the basic laws that were introduced in the previous section. We can derive the basic laws from them, by using the fact that every class which does not explicitly specify a superclass implicitly has the `omni.lang.Object` superclass, which contains no requirements and no methods or attributes.

Chapter 3

Design issues for the Omnibus language

There are a number of interesting problems that were encountered during the development of the Omnibus language. Firstly, the use of functions instead of fields in specifications presented some challenges. These resulted from the fact that functions can have pre-conditions while classically fields do not. Secondly, we had to develop techniques to implement value semantics for our object-oriented language. This was not trivial because value semantics can be naturally modelled through the storing of object values but the dynamic binding concept from object-oriented programming requires the use of references to access objects. Finally, we had to provide support for the concept of equality. This is particularly important for a language such as ours which is based on value semantics because there is no primitive definition such as equality of references which we can fall back on. We discuss these issues in this chapter.

3.1 Using functions instead of fields for specifications

In section 1.3.1 we considered conventional specification approaches where specifications are either defined in terms of the methods in the interface without frame conditions (the Eiffel base function approach), defined in terms of abstract fields with the base level defined in some other manner, or a hybrid approach where the lowest level specifications are defined in terms of the methods in the interfaces and higher levels are defined using fields. For the reasons set out in section 1.2.1, the Omnibus approach is built around specifications defined in terms of the methods in the interface, with support for frame conditions.

Defining specifications in terms of fields and defining specifications in terms of methods are obviously not entirely unrelated. For example, we can formally model fields using methods without side-effects. We used methods as the main mechanism for expressing Omnibus specifications so that we could use them to express modelling types. However, when using methods instead of fields to represent the abstract state of a class, a number of new problems are encountered.

Fields are nice for formal modelling. We do not need to consider pre-conditions: we can retrieve their value at anytime, regardless of whether an object is in a valid state or not (i.e. whether its invariants are currently satisfied). This is not so for methods. Methods may be specified with pre-conditions which must be taken into consideration. We cannot simply ignore these pre-conditions and formally model the methods as underspecified total functions (i.e. allow calls that do not satisfy the pre-conditions but do not specify what the return values will be in those cases). To check an assertion defined in terms of a method call using run-time assertion checking, we must execute the body of the method; an assertion violation or run-time error may result if the pre-condition is not satisfied.

3.1.1 Checking method pre-conditions in specifications

It is important when supporting run-time assertion checking and static verification that the semantics of these two forms of checking assertions are consistent. However, as Chalin has discussed [22] within the JML project, the interpretation of method calls within specifications differs for run-time assertion checking and static verification. For run-time assertion checking, it is natural to define checks of the pre-conditions of methods in code at the start within the method implementation. Without any specialised additional support, this would mean that the pre-conditions of the methods are checked whenever they are called, whether it is from expressions within assertions or from expressions within the normal program execution. Such a consistent interpretation of method calls is quite elegant: wherever a method call occurs, be it in an assertion annotation or normal piece of program code, its pre-condition should be respected.

However, the JML static verification tools have favoured an approach of formally modelling pure methods (i.e. those without side-effects that can be used in specifications) as underspecified total functions. This means that the pre-conditions of method calls appearing in assertion annotations do not have to be satisfied, although the post-condition of the method can only be used if it is.

Within our system where run-time assertion checking and static verification are used in an integrated fashion, it is not acceptable to have different semantics for method calls in static and dynamic verification. It would be difficult to adopt the underspecification approach for run-time assertion checking. We could turn off pre-condition checks for method calls made from within assertions, but then the execution of the body of the method may result in an assertion failure or a run-time error. This is because the verification of the absence of assertion failures and run-time errors from the method body is dependent on the initial truth of the pre-condition. We would have to introduce additional apparatus to cope with this, perhaps introducing additional special return values for exceptional executions.

It is far easier and more consistent to adopt the standpoint that the pre-conditions of all methods, whether in assertions or general expressions, should be shown to hold whenever a method is called. We can verify this through the generation of *side-conditions*.

3.1.2 Model functions with pre-conditions in changes clauses

In the previous section we discussed the challenge that method pre-conditions pose for the use of method calls in specifications. Method pre-conditions also cause problems for the interpretation of `changes` clauses.

Ignoring the complexities of inheritance for a moment, frame conditions involving fields are relatively straightforward to handle. If only fields are used to model the abstract state of a class then the fields whose values are changed by a method should be listed in the frame condition and those that are not changed should be omitted. The final post-condition of the method is the conjunction of the user-provided post-condition and the calculated frame condition. We can simply add assertions that the values of each field that is not mentioned in the frame condition are the same before and after the method.

Where methods are used to model the abstract state of a class, the frame condition should make reference to these methods and this can complicate matters. To illustrate the problems, consider the following `Stack` example:

```
class Stack[Element] {
  model function size():integer
  spec model function elementAt(i:integer):Element
    requires i >= 0 && i <= size()-1
  ...
}
```

Here we have a portion of the specification of a `Stack` class. We present only the two methods that are used to model the abstract state of the class: `size` and `elementAt`. The `size` method returns an integer representing the number of elements currently being held by the `Stack`. The `elementAt` method accepts an integer parameter between `0` and one less than the `size` of the `Stack`, and returns the element at the corresponding position. A pre-condition is used to formalise that the `elementAt` function is only defined for some index values.

First consider the specification of an `empty` constructor:

```
constructor empty()
  ensures size() = 0
```

Frame conditions are not used in conjunction with constructors since there is no previous state. In the `ensures` clause we should specify the values given to each of the constructs in the abstract state, here the `elementAt` and `size` model functions. We state that the `size` is set to zero but do not explicitly refer to the `elementAt` function. This is because the pre-condition of the `elementAt` function restricts the valid parameters of `elementAt` to be between `0` and the `size()-1`. No parameters satisfy this and so `elementAt` contains no values. This shows how the domain of model functions can be affected by changes to other model functions.

Now let us consider the specification of a `push` operation without a frame condition:

```
operation push(e:Element)
  ensures size() = old size() + 1, elementAt(size()-1) = e
```

The `push` operation increases the `size` by one and assigns the passed value to the last position in the `elementAt` model function. When designing the Omnibus language, we were faced with the question of what should the frame condition of such an operation be. Clearly `size` should be included in the frame condition, but what about `elementAt`? By increasing the `size` by one, an additional parameter value is accessible for `elementAt`. Does this alteration of the acceptable parameters for the method constitute a “change” and so should `elementAt` be listed in the `changes` clause?

A similar problem arises for the `pop` operation which is shown without frame condition below:

```
operation pop()
  requires size() > 0
  ensures size() = old size() – 1
```

Here the size is decreased by one, reducing the valid indices of the `elementAt` function by one. Should `elementAt` be listed in the `changes` clause because its domain has changed? Or should it not be, because the values at the valid indices have not changed?

The Omnibus approach that was outlined in the chapter 2 is that methods must be listed in the `changes` clause if any of the indices which are valid before and after the method are changed. These are really the only values for which it makes any sense to have a frame condition because they are the only values to have both old and new values. All other values are excluded from the calculated frame condition. This approach is consistent with the decision to omit `changes` clauses from constructor specifications. It does, however, appear to be a source of some confusion for users. Using the Omnibus approach the specifications of `push` and `pop` with frame conditions would be as follows:

```
operation push(e:Element)
  changes size
  ensures size() = old size() + 1, elementAt(size()-1) = e

operation pop()
  requires size() > 0
  changes size
  ensures size() = old size() – 1
```

We found that some users were confused by the absence of any reference to the `elementAt` function in the `changes` clause. Values of that function are changed: in the `push` operation the newly accessible last value is assigned to the passed `e` value and in the `pop` operation the previous last value is no longer accessible. However, using the Omnibus approach, `elementAt` is not listed in the `changes` clause since no values of `elementAt` that satisfy the pre-condition of the function before and after the operation are changed.

Before discussing alternatives, recall that we can include model functions in `changes` clauses with specific parameters. For example, we could specify that a `replace` operation changes only the last element in a `Stack`:

```
operation replace(e:Element)
  requires size() > 0
  changes elementAt(size()-1)
  ensures elementAt(size()-1) = e
```

In the Omnibus approach we take the values that satisfy the pre-conditions both before and after the method as the starting point for the frame conditions. The assertions that are calculated from the `changes` clauses and added to the post-condition only refer to these values. Alternatively, we could take the values that satisfy the pre-conditions before or after as the starting point. For example, we could require that all the parameters of the model functions that are accessible after the operation call and are not the same as before the operation be listed in the `changes` clause. This would lead to the following specifications for `push` and `pop`:

```
operation push(e:Element)
  changes size, elementAt(size()-1)
  ensures size() = old size() + 1, elementAt(size()-1) = e

operation pop()
  requires size() > 0
  changes size
  ensures size() = old size() – 1
```

Using this technique we have to list `elementAt(size()-1)` in the `changes` clause of the `push` operation because it is accessible after the method call and its value is not the same as it was before the method. If it was not listed then, using this interpretation of `changes` clauses, an assertion would be added asserting that `elementAt(size()-1) = old elementAt(size()-1)`. However, `size()-1` is not a valid index of **`old`** `elementAt` and so this would be a logical error; the pre-condition of the method call would be violated. In contrast, this approach would not require `elementAt` to be mentioned in the `changes` clause of `pop` because all values of the `elementAt` function that are accessible have the value they had before the operation call.

The mirror of this technique is to require that all the parameters of the model functions that are accessible before the operation call and do not have the same value after the operation be listed in the `changes` clause. This would lead to the following specifications for `push` and `pop`:

```
operation push(e:Element)
  changes size
  ensures size() = old size() + 1, elementAt(size()-1) = e

operation pop()
  requires size() > 0
  changes size, elementAt(old size()-1)
  ensures size() = old size() – 1
```

This time, the newly accessible value of `elementAt` in the `push` operation does not have to be mentioned because it was not accessible before the operation. However, the value that was previously at the end of the `Stack` does have to be mentioned in `pop` because it no longer has the same value as it had before (it no longer has a value).

We could combine these two approaches, requiring that any parameters of the model functions that are accessible either before or after the operation call and do not have the same value at those points should be listed in the `changes` clause. This would lead to the following specifications for `push` and `pop`:

```
operation push(e:Element)
  changes size, elementAt(size()-1)
  ensures size() = old size() + 1, elementAt(size()-1) = e

operation pop()
  requires size() > 0
  changes size, elementAt(old size()-1)
  ensures size() = old size() – 1
```

This may be more desirable for programmers because it is more intuitive despite requiring them to write more code, particularly code that is essentially redundant. However, if we were to adopt this approach then failure to include `elementAt` with the appropriate parameter in the `changes` clause would become a difficult to detect error. Consider what would happen if we used this approach but omitted any mention of `elementAt` from the `changes` clause.

```
operation push(e:Element)
  changes size
  ensures size() = old size() + 1, elementAt(size()-1) = e
```

Since `elementAt` is not listed in the `changes` clause, all indices of the function that are valid either before or after the method are assumed not to change. Hence, `elementAt(size()-1)` will be assumed to have the value it had before the operation which would be **`old`** `elementAt(size()-1)`. In other words, the assertion `elementAt(size()-1) =` **`old`** `elementAt(size()-1)` will be added to the final post-condition. But `size()-1` is not a valid index of the **`old`** `elementAt` function and so this method call will not satisfy its pre-condition. Furthermore, the generated assertion that `elementAt(size()-1)` is ‘unchanged’ will contradict the assertion in the user provided `ensures` clause that it will equal the passed value `e`.

This is highly undesirable since contradictions can cause problems for our tool, like many other verification tools, and lead to it reporting misleading errors. This final approach is the most prone to introducing contradictions in the calculated post-condition with its generated frame condition because it produces the strongest frame condition assertions. Our approach produces the weakest frame condition assertions so that contradictions are minimised.

3.1.3 Vicious circle of logic

In this section we consider the complexities involved with the use of methods instead of fields in defining class invariants.

The vicious circle of logic in JML and Spec#

The classical view of class invariants is that they should hold whenever a class is externally accessible, i.e. in between public method calls. This can be demonstrated by showing that all constructors establish the invariants and then, assuming that the invariants hold before an instance method, showing that they hold after the method. That way, by induction, whenever another class has an instance of the object they can assume that the invariant holds over it. We can naïvely implement this by implicitly conjoining the invariants to the pre-conditions of all the methods of the class and then checking the invariants at the end of the methods.

Normally Java methods cannot be used in JML assertions because they may have side-effects. If we allowed this, then whether or not the assertions were evaluated would affect the behaviour of the program, and testing of a program with assertion checking enabled would not necessarily be valid when assertion-checking was disabled. However, there is no problem in using methods without side-effects in assertions. Methods without side-effects are called *pure methods* in JML and Spec# and are declared with the `pure` modifier. The tools then check that the methods do not change variables other than those local to the method. Pure methods are extremely useful because they allow us to write specifications that do not have to refer to only fields. We would like pure methods to be an extension of normal methods, with the properties of normal methods applying to them along with the additional properties: here that the method cannot have side-effects. Recall that one of these properties is that the class invariants can be assumed at the start of the method and must be re-established at the end of the method.

There is a problem of how to handle assertions which contain constructs such as method calls whose pre-conditions are not satisfied. Often such occurrences indicate limitations in the thinking of the programmer, coupled with logical errors. A run-time checker would normally check such pre-conditions whether they were invoked via an assertion or a standard expression in an implementation. We can adopt a similar approach within static verification: whenever an expression construct such as a method call is used, check its pre-condition separately. Such a check is referred to as a *side-condition*.

We would like to be able to use pure methods in our specifications (in our `requires` clauses, our `ensures` clauses and, crucially, in our invariants) and we would like to generate side-conditions to check their use, but in doing so we hit a problem. Consider the following JML example:

```
public class PosCounter {
  /*@ spec_public */ int c;

  //@ public invariant count() >= 0;

  //@ ensures \result == c;
  public /*@ pure */ int count() {
    return c;
  }

  //@ requires count() > 0;
  public void dec() {
    c--;
  }
}
```

This simple class contains an integer attribute `c`, a pure method `count` for returning the value of `c` and a method `dec` for decreasing `c` by one. An invariant asserts that `count` should be maintained to be non-negative.

Using our naïve interpretation here, we include the invariants as part of the pre-condition of the `count` method. Hence we have (ignoring the need to check the invariant is maintained in this trivial case):

```
  //@ requires count() >= 0;
  //@ ensures \result == c;
  public /*@ pure */ int count() {
    return c;
  }
```

The cyclic reference of the method itself in its pre-condition is already worrying and may well suggest to us that something is wrong, but let us continue regardless and now consider the method `dec`. De-sugaring the method, we again add the invariant to the pre-condition (although in this case it is implied by the original pre-condition and so we omit it) and also add an `assert` statement at the end of the method to check the invariant. This gives us:

```
  //@ requires count() > 0;
  public void dec() {
    c--;
    assert count() >= 0;
  }
```

Now, consider the execution of the `assert` statement. The local method call yields a side-condition that the pre-condition of that method is satisfied. The pre-condition of this method is precisely:

```
  count() >= 0
```

From the `ensures` clause of the `count` method we know that we can re-write occurrences of the `count` method using the value of the field `c`, but before we can use this re-write we have to show that its pre-condition is satisfied. The problem is that methods used to define the invariants cannot have the invariants as part of their pre-conditions (implicitly or explicitly). If they do then the situation degenerates when we have an object whose validity (i.e. whether the invariants hold over it) we do not know. In the example above the object whose validity we are not sure of is the `this` object after a local assignment to one of its attributes. We cannot then check the invariants over this object because these invariants may be described in terms of methods whose values are only defined for valid objects.

The things which previously allowed JML to side-step this problem were its formal modelling of methods as underspecified total functions and its avoidance of side conditions for assertions. Were these to be altered then this issue would have to be addressed.

The developers of the Spec# system ran into this problem because they generate side-conditions for assertions. To address the problem, all invariants are private in accessibility and can only refer to private members. Public methods cannot be used in the invariants because the invariants are part of their pre-conditions.

The vicious circle of logic in Omnibus

Within Omnibus the same question is raised of whether the invariants should be part of the pre-conditions of methods. Only methods without the invariants in their pre-conditions can be used to define the invariant. The problem also arises through the use of local method calls (excluding those with the `old` prefix) in `ensures` clauses of constructors and operations. Any such methods should also not have the invariants as part of their pre-conditions. So should all methods have the invariants as part of their pre-conditions? Should none?

There are three ways that we see of addressing this:

1. Functions and operations might no longer have invariants as part of their pre-conditions. This would mean that all functions and operations could be freely used in invariants. This is the most extreme response and is completely unworkable since it would destroy the verification approaches that are at the heart of Omnibus. It is included here only for completeness.
2. Functions might no longer have invariants as part of their pre-conditions but operations might. This means that any local function can be used in an invariant but local operations cannot (something which is often very useful). A key problem with this is that often functions will not make sense when called with parameters that would ordinarily be ruled out by the invariants implicitly included in their pre-conditions. To counteract this, the programmer would have to repeat portions of the invariants in the pre-conditions of the functions. This is highly undesirable.
3. Model functions might not have the invariants as part of their pre-conditions but derived functions and operations might. This means that the `ensures` clauses of constructors and operations and invariants must be completely defined in terms of

the model functions. This sounds more workable than it is. In reality, invariants will often be far more readily expressible via derived functions than model functions, and the programmer will likely get annoyed if they are not permitted to do so.

4. Model functions and methods declared with a special modifier might not have the invariants as part of their pre-conditions but the other methods might. This means model functions and selected other methods can be used in invariants. This is a good compromise. It is a flexible approach and seems sufficient to verify all existing Omnibus code.

The approach adopted by Omnibus is that model functions and methods declared with a special `helper` modifier, with the same meaning as `helper` in JML, do not have the invariants as part of their pre-conditions but the other methods do. This means model functions and selected other methods can be used in invariants.

The same problem also appears in a number of other places in Omnibus, including `ensures` clauses of operations and constructors and other types of requirements. The general rule is as follows: only model functions and `helper` functions can be applied to `this` in invariants, constraints, initiallys, and `ensures` clauses of operations and constructors. You are free to use any method over **`old`** `this` since this is known to be valid. In `ensures` clauses for non-helper functions you are free to apply any method to `this`, since again you have grounds to assume that it is valid.

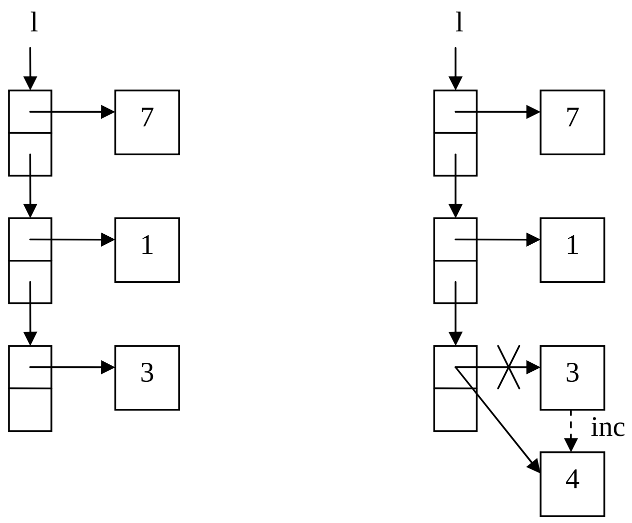
3.2 Implementing value semantics

In this section we discuss the techniques used to implement value semantics for an object-oriented language. References must be used to implement the objects in order to support dynamic binding. To give value semantics, copying is then essential to ensure that objects that are referred to by multiple references are not mutated. However, it is possible to make some optimisations to avoid unnecessary copying.

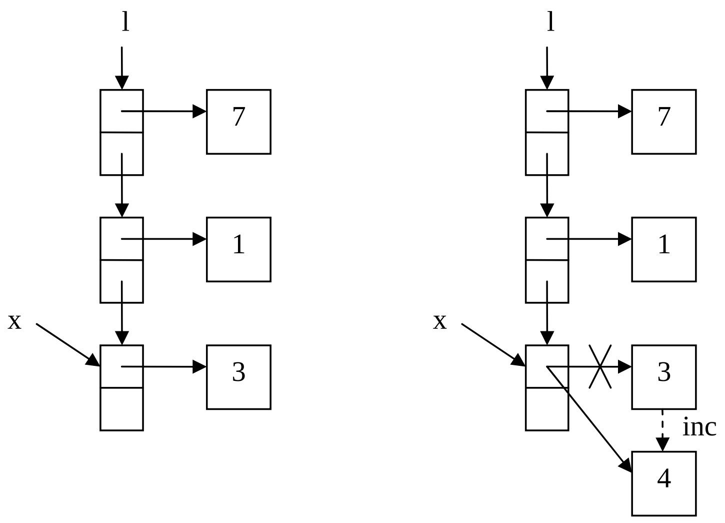
3.2.1 Copy-on-update implementation of value semantics

In value semantics, operations cannot simply be implemented as mutations of objects as is done in Java. If we did this and there were other references to the object then the values of these references would change, violating value semantics. For example, suppose we have a `Counter` object whose abstract state consists of a `count` value and can be increased and decreased by `inc` and `dec` operations to increment and decrement the value of the `Counter`, respectively. Now, if we have only a single reference to the object then we can freely implement the operations as mutations, but if there are other references to the object then we must create a new object so that the value of the other reference is unchanged.

The situation becomes more complicated when we are changing an object contained in a data structure. To illustrate this, consider the following linked list example. Here, we have a classical linked list with each node containing a link to the node's value and a link to the next node in the chain. The linked list is accessed via a reference, `l`, to its first node. Now suppose we want to update the value of the final element in the linked list, which is currently a `Counter` with the value `3`, by applying the `inc` operation. For simplicity let us enforce that this operation must create a new object instead of mutating its original value in case there were any other references to the `Counter` containing the `3`. As is standard we can then change the value pointer of the final node to point to the new `Counter` value. This situation is illustrated diagrammatically below.



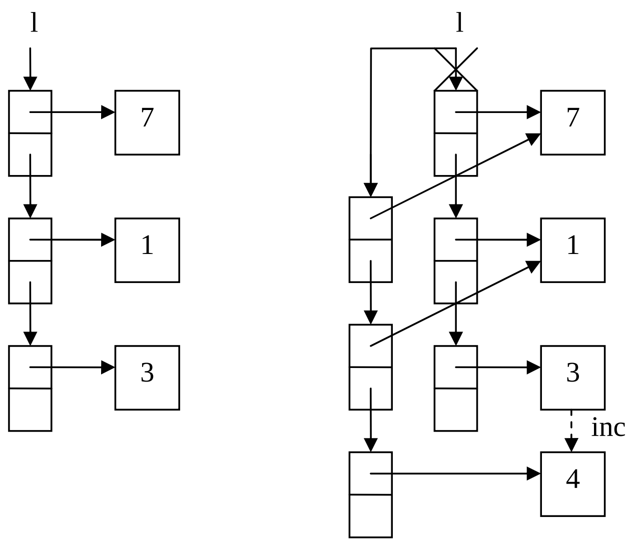
We were careful not to implement our `inc` operation as a mutation in case there were existing references to it, so we may believe that this is an acceptable implementation, compatible with value semantics. However, consider the case shown below where there was an existing reference to the final node.



In this scenario, the value of the node pointed to by `x` had the value of a `Counter` object with count `3` before the operation and the value of a `Counter` object with count `4` after the operation. As this operation was applied to the `List` and not `x`, `x` should not have changed.

The cause of this problem is that the node itself is also an object and so implementing changes to it as mutations can also lead to violations of value semantics. So, if there are other references to the node then we cannot update it via a mutation, and must instead create a new node. Similar reasoning dictates that we must create copies of the intervening nodes in case there are references to these.

As an extreme, conservative solution to this problem we could implement all operations by creating new objects. This would give us the diagram shown below.



This is clearly extremely inefficient. To change the value of a single element in the list we have had to create new nodes for all the elements in the list.

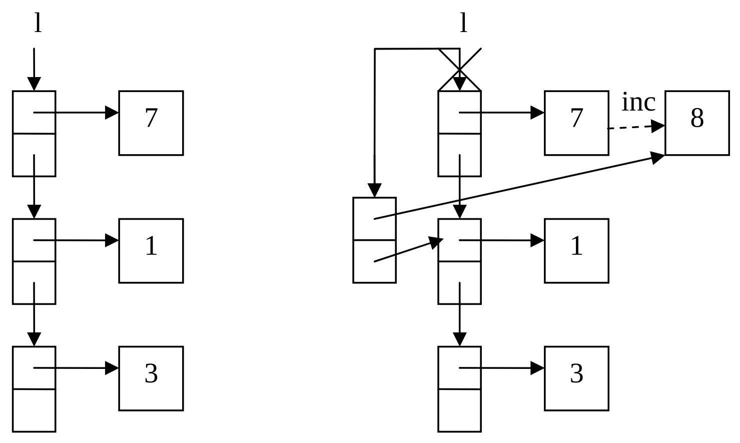
However, there are some reasons why the situation is not quite as bad as this, and techniques that we can use to further improve the situation.

3.2.2 Optimisations of copy-on-update

Not everything is copied

Firstly, we are not creating a new copy of the entire list, i.e. all the nodes and all the object values. All the object values which are unchanged are simply reused in the new list. This can be seen in the final diagram: the first two nodes point to the same object values that their previous incarnations did. In this case, that is not particularly significant because our object values are simply `Counter` objects and will not be considerably larger in terms of memory usage than the nodes themselves. However, in more realistic examples, the object values will likely be considerably larger than the nodes and so creating copies of nodes will be far less costly than creating full copies of the object values themselves would be. To summarise, a new copy of only one of the object values is created, specifically the one that is being changed.

Also, we do not need to create copies of portions of the `List` that are unchanged. In the example we used there were no such portions that were unchanged since we changed the value of the last node and the next pointers of all the nodes preceding it. However, if we had updated the first node in the list instead of the last then we could have reused the remainder of the list unchanged since neither the values nor next node pointers need to be changed.



Balancing techniques can reduce amount of copying

We can view the internal structures of reference-based data structures as a hierarchy. The access point is at the top of this hierarchy with remaining nodes arranged below it in order of the depth of references that must be used to reach the object. For example, for our list, the first node would be at the top of the hierarchy and the last node at the bottom.

Now, in the preceding section we saw that portions of the list that are unchanged can be reused directly without the need for any copying. For such a portion to be directly reusable, its values and its next pointers should not require any changes. We can exploit this by structuring our data structures so that we minimise the depth of the data structure and/or ensure that nodes which are most likely to change are higher up in the hierarchy. There is a limited amount that we can do to achieve these goals in the list example. However, if we generalise the example to a tree data structure then we can use existing techniques such as AVL trees and red-black trees [99] to enforce balancing and hence minimise the depth of the hierarchy. We may also be able to devise a technique to ensure that the nodes that are most likely to be changed are at the top of the hierarchy.

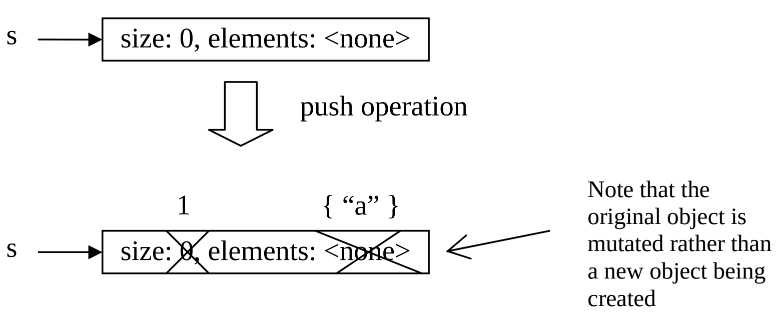
Can use mutation if there is only a single reference to the object to be updated

In our discussion of the list example, we introduced complications by supposing that there were existing references to the nodes in our data structure. In the most extreme case, if there may be existing references to any of the nodes in our data structure then we must create new copies of any that we need to update in any way. It is in this copying process that the inefficiencies arise. In contrast, if we knew that there were no existing references to the nodes in our data structure then we could freely update any of them using mutations, and our originally proposed solution would be perfectly acceptable. We can freely use mutation to implement operations if we know that there are no existing references to the object (other than the one via which we are accessing it). So, a possible solution is to use mutation to update objects with only a single reference to them and create new objects during other operation calls. This approach is known as *uniqueness typing* [84].

The problem is how we can detect how many references there are to an object which needs to be updated. We can do this either statically or dynamically (e.g. reference counting). A static approach would be the more desirable because it would not involve any run-time overheads. We cannot detect all possible optimisations using a static modular approach but we can detect some. For example, consider the following code snippet:

```
var s:Stack[String] := Stack[String].empty();
s.push("a");
```

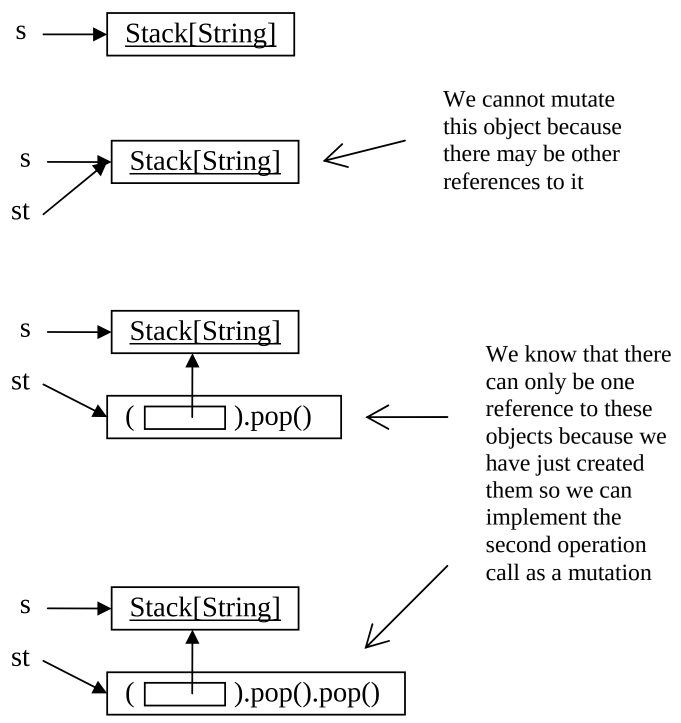
We can statically deduce that at the point when the `push` operation call is encountered, there can only be a single reference to the stack `s`. This is because we have just created the object and so there is no possibility that there is another reference to it. Therefore, we can implement this operation call as a mutation, illustrated by the following diagram.


s
size: 0, elements: <none>
push operation
1
{ "a" }
s
size: ~~0~~, elements: ~~<none>~~
Note that the original object is mutated rather than a new object being created

However, the process falls down somewhat when we hit an interface boundary. For example, suppose that instead of declaring a new `Stack` as a local variable, we have a `Stack` parameter passed to our method.

```
function thirdTop(s:Stack[String]):String {
   var st:Stack[String] := s;
   st.pop();
   st.pop();
   return st.top();
}
```

In this case we cannot make any assumptions about how many references there are to the `Stack` object that is passed into the method and so the first `pop` operation must create a new object. However, we can statically deduce that there is only one reference to the `Stack` object held in `st` by the time we reach the second operation call since this new `Stack` object was only created by the first `pop` operation. Therefore, we can implement the second operation call as a mutation as shown below.


s
Stack[String]
s
st
Stack[String]
We cannot mutate this object because there may be other references to it
s
st
Stack[String]
( ).pop()
We know that there can only be one reference to these objects because we have just created them so we can implement the second operation call as a mutation
s
st
Stack[String]
( ).pop().pop()

The static deduction approach is necessarily incomplete. Perhaps, in a particular case, the `Stack` passed to the `thirdTop` operation will only have a single reference to it and so we could implement the first operation call via a mutation. We cannot detect this case statically in a modular fashion but we could detect it dynamically. This would require a form of reference counting mechanism similar to that used in primitive garbage collection algorithms. Ideally this would be implemented within the VM but could be simulated within the generated code. However, it is not clear that the gains from using mutation in these cases would compensate for the additional run-time overhead required to track the number of references to each object. It is also not clear whether the number of cases that could be dynamically deduced to be appropriate for mutation would be significantly higher than those that could be statically deduced to be as such. This section described possible extensions to the Omnibus verification tool. None of the approaches described are implemented in the current version of the Omnibus verification tool.

3.3 Equality in Omnibus

Equality is a key concept within Omnibus. It is even more important in a system built upon value semantics than it is in one built on reference semantics.

In Java, which is built on reference semantics, there are two concepts of object equality: equality of references and equality of objects. The `==` and `!=` operators allow the programmer to compare the addresses held in two references for equality. If two object references hold the same address then these references point to the same object on the heap. Equality of the objects themselves can be defined using the special `equals` method. This is a normal method which returns the boolean value `true` iff the passed object is equal to `this`. The writer of the class is free to give any implementation for this function although there are some informal requirements that they should follow (it should be commutative, etc.).

In Omnibus, there is only one concept for equality of objects which can be accessed via the `=` and `!=` operators. This operator represents value equality and should evaluate to `true` whenever the two objects to which it is applied are interchangeable without observable differences. This is actually quite a complicated concept and this section will explore the range of support which Omnibus provides for it.

3.3.1 Built-in concept of equality

In Java, if you do not define an `equals` method for your class, a default implementation is inherited from the `java.lang.Object` class. This provides you with some concept of equality of objects without the programmer having to do any work. Omnibus provides similar facilities to automatically define an equality operator even if none is explicitly given.

While at this high level the two approaches to this problem appear similar, they are implemented in diverse fashions. In Java, the default equality operator simply compares the references of the two objects; this is enough because the programmer also has the concept of reference equality to work with. However, in Omnibus, the equality operator has only one definition and this must be powerful enough to reason effectively about the equality of many object expressions. As a result, the automatic definition of equality in Omnibus is far more complex.

If no equality operator is defined for a class, the Omnibus system automatically defines one. Leibnitz's rule is used to reason about equality of object instances of the class for static verification. An implementation of structural equality checking is provided for implementations and run-time assertion checking. This gives an overly conservative but useful and safe automatic definition of equality. These concepts are introduced in the remainder of this section.

Leibnitz's rule is simply this: if the same method is applied to two objects that are equal with parameters that are equal then the result will be equal. So, for example, if we have an object `obj` which has a method `func()` then Leibnitz's rule allows us to deduce that `obj.func() = obj.func()`. More interestingly, suppose we have objects `obj1` and `obj2` that we know are equal and `p1` and `p2` that we know are equal. Now if there is a method `calc` in the class of `obj1` and `obj2` that takes two parameters then we can deduce that `obj1.calc(p1,x)` = `obj2.calc(p2,x)`. This is true because the objects are equal and the corresponding parameters are equal.

Leibnitz's rule is of most use when coupled with substitutions. Suppose we know that an `ensures` clause tells us that `modFunc() =` **`old`** `modFunc().op(param)`, and an invariant requires us to check that `modFunc().isGood()`. We can then substitute **`old`** `modFunc().op(param)` for `modFunc()`, allowing us to demonstrate the invariant by showing **`old`** `modFunc().op(param).isGood()`. This facility is used extensively throughout all Omnibus examples which have been produced so far.

Leibnitz's rule gives some very basic rules about equality. These rules are always respected by Omnibus classes (it is not possible to write a method which returns different values for the same parameters – there are no static variables) but obviously do not give the rich definition of equality that will often be required. Programmers will often want objects that are constructed using different sequences of method calls to be considered equal. For example, consider a `Stack` where an element is pushed and then popped. The programmer would probably want these two `Stack` objects to be considered equal, but Leibnitz's rule alone does not permit you to deduce this.

Structural equality is a mechanism for comparing any two objects. Using structural equality, two objects are equal if their dynamic classes are equal (i.e. they are instances of the same class) and all of their corresponding attributes are equal. This is a recursive definition which checks the equality of two objects by checking the equality of all of their subcomponents (i.e. the attributes that define the implementation of the class).

Like Leibnitz's rule, structural equality gives a basic definition for equality. Structural equality is consistent with Leibnitz's rule, since if all the attributes of two objects of the same class are equal then the result of the same method with the same parameters will be the same. We will see later that, like Leibnitz's rule, there are limitations to structural equality. Again, though, in these cases the programmer can provide an implementation for equality.

To illustrate the default definition of equality provided by Omnibus, consider the following `Array` class.

```
class Array[Element] {
  model function length():integer
  model function access(i:integer):Optional[Element]
    requires i >= 0 && i < length()

  invariant length() >= 0

  constructor ofSize(n:integer)
    requires n >= 0
    ensures length() = n,
     forall (j:integer := 0 to n-1):
      access(j).isNil()

  operation assign(k:integer, e:Element)
    requires k >= 0 && k < length()
    changes access(k)
    ensures access(k) = Optional[Element].of(e)

  operation assignOptional(k:integer, e:Optional[Element])
    requires k >= 0 && k < length()
    changes access(k)
    ensures access(k) = e
}
```

Note that no definition for equality is given explicitly, so the default definition will be automatically introduced by the system. Let us now consider some expressions involving this class in order to explore what the default definition gives us. The expressions we will consider will refer to the following variables:

```
p1:Person
arr:Array[Person] := Array[Person].ofSize(3).assign(1, p1)
```

We now know the following facts about model functions of the `arr` object:

```
arr.length() = 3
arr.access(0).isNil()
!arr.access(1).isNil()
arr.access(1) = p1
arr.access(2).isNil()
```

From Leibnitz’s rule we can then deduce the following:

```
  arr.assign(1, p1) = arr.assign(1, p1)
```

This is a straightforward application of Leibnitz’s rule.

However, we cannot deduce the following:

```
  arr = arr.assign(1, p1)
```

Intuitively, this should be true since, on assigning to the element at index value `1`, the value that it currently has should not change the array object. However, Leibnitz’s rule is not enough to deduce this. To enforce this we need to provide a manual definition for equality of `Array`s.

While we cannot deduce the equality of `arr` and `arr.assign(1,p1)`, we can deduce the equality of their model functions:

```
  arr.length() = arr.assign(1, p1).length()
  && (forall (i:integer := 0 to 2):
    arr.access(i) = arr.assign(1, p1).access(i))
```

3.3.2 Manually defining the equality operator

Let us now look at how programmers can provide their own definition for equality. Before looking at a range of techniques for defining equality, let us quickly look at how an equality operator for the `Array` class could be defined.

```
class Array[Element] {
  model function length():integer
  model function access(i:integer):Optional[Element]
    requires i >= 0 && i < length()
  ...
  function equals(a:Array[Element]):boolean
    returns length() = a.length()
     && forall (i:integer := 0 to length()-1):
      access(i) = a.access(i)
}
```

Note how equality is defined via an `equals` method which accepts an instance of the current class. In the following sections we will look at how to define these methods.

Equality over model functions of no parameters

It is relatively straightforward to define an equality operator for a class whose abstract state consists only of model functions of no parameters. Two objects are simply equal if all their model functions evaluate to the same values. Consider the following `Library` class whose abstract state consists of a `Collection` of `Book`s, a `Collection` of `Member`s and a `Collection` of `Loan`s of `Book`s to `Member`s. The `equals` method simply compares the values of these model functions.

```
class Library {
  model function books():Collection[Book]
  model function members():Collection[Member]
  model function loans():Collection[Loan]

  ...

  function equals(l:Library):boolean
    returns books() = l.books()
     && members() = l.members()
     && loans() = l.loans()
}
```

Equality over model functions with parameters of restricted range

The first example considered the simplified case where the model functions of the class had no parameters. We will now consider a slight generalisation of this where the model functions have parameters but they have a restricted range. In such cases, we can simply specify the equality operator by quantifying over a variable in this range. Consider the following example of a `List` class which contains a `size` model function with no parameters and an `elementAt` model function with a single index parameter `i`. This parameter is restricted to range from zero to one less than the `size`. We can then define equality of `List`s by asserting that their `size`s should be equal and that the values of the `elementAt` model function from `0` up to the `size()-1` are equal. In other words, the model functions are equal for all values that satisfy their pre-conditions. This fits our intuition of equality of `List`s: two `List`s should be considered equal if they contain the same elements.

```
class List[Element] {
  function size():integer
  function elementAt(i:integer):Element
    requires i >= 0 && i < size()

  ...

  function equals(l:List[Element]):boolean
    returns size() = l.size()
     && forall (i:integer := 0 to size()-1):
      elementAt(i) = l.elementAt(i)
}
```

Equality over model functions with parameters of restricted range with additional restrictions

In the previous section we considered the case where the only restriction on the parameters of a model function was to limit its range. However, there may be other restrictions within the pre-condition of the model function. In such cases we can simply add these additional restrictions to the quantification over the parameters of the model function in the `equals` method using a `where` clause. Consider the following example of a `YahtzeeGame` class. First, there is a model function containing the number of rolls that have been taken in the current turn. This can range from `0` up to `3`. Next there is a `diceAt` model function used to hold the values of the `Dice` that are currently showing. The method can be called with parameters from `1` up to `5` apart from when there have been no rolls yet (in which case the dice do not have any values yet). These values then change as follows. Initially the number of rolls will be `0`, signifying that the dice have not been rolled yet in the current turn. The dice can then be rolled, at which point the number of rolls is increased to one and values are given to the dice positions `1` up to `5`. The dice can then be rolled up to a limit of 2 further times before a position on the board is selected and the number of rolls is reset to `0`. There is also a range of other model functions that are not of interest to us here. The key part of interest to us is the definition of the equality in relation to these two model functions. The handling of the `rolls` model function is straightforward, and we can handle the `diceAt` model function like the `elementAt` function in the `List` class, adding the additional restriction that the number of rolls should be greater than zero in a `where` clause of the quantification over the `pos` parameter.

```
class YahtzeeGame {
  model function rolls():integer
  model function diceAt(pos:integer):Dice
    requires pos >= 1 && pos <= 5,
     rolls() > 0

  ...

  function equals(g:YahtzeeGame):boolean
    returns rolls() = g.rolls()
     && (forall (pos:integer := 1 to 5 where rolls() > 0):
      diceAt(pos) = g.diceAt(pos))
     && ...
}
```

Equality over model functions with parameters of unrestricted range

So far we have only considered model functions with parameters of restricted ranges. In these cases we can use a quantifier with the quantified variables restricted to that range. The advantage of this kind of quantifier is that it can be evaluated at run-time. If a model function has a parameter without such a restricted range, the situation is much harder to deal with because we cannot simply write an executable specification. In such cases we have little alternative but to write separate specifications and implementations for the method. Consider the following example of a `Collection` class which is described in terms of a `contains` model function. The method takes an `Element` parameter whose values cannot be enumerated over like the integer ranges we encountered earlier. So, we describe the `equals` method using an `ensures` clause containing a universal quantification over `Element`s. Two `Collection` objects are equal if their `contains` functions have the same value for every possible `Element`. Note that we cannot use a `returns` clause here because the unconstrained universal quantifier is not executable. The specification of this class is shown below:

```
class Collection[Element] {
  model function contains(e:Element):boolean

  ...

  function equals(c:Collection[Element]):boolean
    ensures result
     <==> forall (e:Element): contains(e) = c.contains(e)
}
```

Now let us consider how to define the implementation of the class and, specifically, the `equals` method. First, we introduce a `List` attribute and define the `contains` model function in terms of it. So we are using a `List`, which may contain duplicates, to implement our `Collection`, which should not, and ruling duplicates out via an invariant. We can then implement our `equals` function by iterating over the elements in each of the `List`s in turn and checking that the other contains all of their elements. If each `Collection` contains all of the `Element`s in the other, then the `contains` functions must yield the same value for all `Element` values and hence the `Collection`s are equal.

```
public class Collection[Element] {
  private attribute elements:List[Element]
  public model function contains(e:Element):boolean
    private returns elements.contains(e)

  private invariant
              forall (e:Element): elements.countOf(e) <= 1
  ...

  public function equals(c:Collection[Element]):boolean
    ensures result = true
     <==> forall (e:Element): contains(e) = c.contains(e)
  {
    foreach (e:Element in elements) {
     if (!c.contains(e)) {
      return false;
     }
    }
    foreach (e:Element in c.elements) {
     if (!contains(e)) {
      return false;
     }
    }
    return true;
  }
}
```

Coping with irrelevant information in the abstract state

The techniques that we have looked at so far for defining equality break down when the abstract state contains information that is irrelevant. For example, consider the following specification of a `Stack` class:

```
class Stack[Element] {
  model function elements():Array[Element]
  model function size():integer

  invariant size() >= 0
  invariant size() <= elements().length()

  invariant forall (i:integer := 0 to size()-1):
     !elements().access(i).isNil()

  constructor empty(capacity:integer)
    requires capacity >= 0
    ensures size() = 0,
     elements() = Array[Element].ofSize(capacity)

  operation push(e:Element)
    requires size() < elements.length()
    changes size, elements
    ensures size() = old size() + 1,
     elements() = old elements().assign(size()-1,e)

  operation pop()
    requires size() > 0
    changes size
    ensures size() = old size() - 1

  function top():Element
    requires size() > 0
    returns elements().access(size()-1)

  function equals(s:Stack[Element]):boolean
    returns size() = s.size()
     && elements() = s.elements()
}
```

Here the abstract state contains an `Array`, which is used to store the elements in the `Stack`, and an integer to record the number of elements in the `Stack`. An element is deemed to be contained in the `Stack` if it is in one of the first `size()` positions of the `Array`. The elements in the `Array` after the first `size()` are not deemed to be in the `Stack` and their values are irrelevant. The standard definition for equality is given which compares the values of the model functions of the current class and the one passed to the `equals` method.

However, in this example, we cannot simply compare the values of the model functions. If we do this then objects whose values differ only in irrelevant details will not be considered equal. For example, consider we have a `Stack s1` containing the elements `“abc”`, `“def”` and `“ghi”`, i.e.:

```
var s1:Stack[String] :=
   Stack[String].empty(5).push(“abc”).push(“def”).push(“ghi”);
```

From the specification it will have the following values:

```
  s1:Stack[String]
where s1.size() = 3
  && s1.elements().length() = 5
  && s1.elements().access(0) = “abc”
  && s1.elements().access(1) = “def”
  && s1.elements().access(2) = “ghi”
  && s1.elements().access(3).isNil()
  && s1.elements().access(4).isNil()
```

Now let us define a new `Stack s2` formed by taking the `Stack s1`, pushing `“jkl”` onto it and then popping it off i.e.

```
var s2:Stack[String] := s1.push(“jkl”).pop();
```

Intuitively, these two `Stack`s should be considered equal since they contain the same elements in the same order but they would not be equal using our standard definition of equality. This is because the values of the model functions are as follows:

```
  s1:Stack[String], s2:Stack[String]
where s1.size() = 3 && s2.size() = 3
  && s1.elements().access(0) = “abc”
  && s2.elements().access(0) = “abc”
  && s1.elements().access(1) = “def”
  && s2.elements().access(1) = “def”
  && s1.elements().access(2) = “ghi”
  && s2.elements().access(2) = “ghi”
  && s1.elements().access(3).isNil()
  && s2.elements().access(3) = “jkl”
  && s1.elements().access(4).isNil()
  && s2.elements().access(4).isNil()
```

The problem here is that the abstract state contains information that is irrelevant, namely the values in the `elements()` `Array` at indices between `size()` and `capacity()-1`.

There are a number of techniques we can use to cope with this problem. Firstly, we could assert that irrelevant information always has the same value. For example, in this case we could assert that the information in the elements array at indices between `size()` and `capacity()-1` should always be `null`. This can be formalised in the following invariant:

```
private invariant forall(i:integer := size() to capacity()-1):
     elements.access(i).isNil()
```

We would then need to adjust the `pop` operation to set the newly irrelevant value from the `elements` array to be `null`.

```
public operation pop()
  requires size() > 0
  changes size, elements
  ensures size() = old size() – 1,
    elements() = old elements().assignOptional(size()-1,
                                       Optional[Element].nil())
```

This would enable us to use the existing definition of equality to fulfil our intuition.

Alternatively we could adopt a different approach for the specification of the equality operator. We could write a new function to return only the relevant elements of the abstract state and then compare the results of these functions for the two objects. For example, we could define a `relevantElements` function to return the range of values from the elements array that are relevant and then specify our equality operator in terms of it.

```
private function relevantElements():Array[Element]
  returns elements().range(0, size())

function equals(s:Stack[Element]):boolean
  returns size() = s.size()
    && elements().size() = s.elements().size()
    && relevantElements() = s.relevantElements()
```

This section described possible extensions to the Omnibus verification tool. None of the approaches described are implemented in the current version of the Omnibus verification tool.

Chapter 4

Challenges of reference semantics

The modular static verification of object-oriented languages which use reference semantics is extremely complicated. Indeed, until recently, the state-of-the-art approaches had soundness holes for data structures incorporating complex internal structures that utilised references [77]. This issue had been side-stepped by the majority of the verification field working with languages like JML. Recent work [37] has helped provide a sound basis for the modular verification of JML and other OO languages with reference semantics. This chapter demonstrates the problems with the modular static verification of OO languages built upon reference semantics and then discusses existing solutions.

Chapters 2 and 3 discussed the Omnibus language with its support for value semantics. This chapter is concerned with reference semantics and uses examples presented in JML, not Omnibus. The chapter describes existing approaches rather than presenting new ones. The support for value semantics that we have previously described could be combined with the techniques for reference semantics presented here. Either Omnibus could be extended with the techniques presented in this chapter, or the concepts of Omnibus could be incorporated into the systems we describe.

4.1 Problems of modular reasoning about reference semantics

References are used extensively in Java programs. All objects are accessed via references. In this section we will illustrate the problems this causes for modular static verification.

4.1.1 Review of the basics of reference semantics

Consider the following piece of code where we define a Java class `BankAccount`:

```
public class BankAccount {
  private int balance;

  public BankAccount() {
    balance = 0;
  }

  public int getBalance() {
    return balance;
  }

  public void deposit(int amount) {
    balance += amount;
  }

  public void withdraw(int amount) {
    balance -= amount;
  }
}
```

Now consider the following statements that use this class:

```
BankAccount a = new BankAccount();
BankAccount b = a;
```

In the first line we create a new `BankAccount` object and assign it to the local variable `a`. The effect of this is that variable `a` is given the value of a reference to the newly created object. In the second line we assign the value of `a` to a new variable `b`. The effect of this is that the new variable `b` is given the value of the variable `a` which is a reference to the previously created `BankAccount` object. Thus, both variables contain references to the same object. This situation, where multiple references point to the same object, is called *sharing* and is common in OO programs.

Operations to alter the state of objects are implemented in Java using methods which change the values of the attributes of the object. These are called *destructive updates* and they act to mutate the object. If we use a method to change the value of the object via one of the references then the value of the other reference will also have changed. Suppose we now execute the following line of code:

```
a.deposit(100);
```

Naturally, `a.balance` will now hold `100` since we have explicitly applied the `deposit` method to the `a` reference. However, `b.balance` will now also hold the value `100` since `b` points to the same object as `a`.

The dependencies appear relatively straightforward here; we have two objects that clearly point to the same object. However, in modular verification we must consider the different parts of an application separately. When we are passed a reference to an object via a parameter, we do not know what other references there are to it. The passed reference may only be used as a local variable which will not be used further and whose lifetime will end imminently after the method we are considering returns. On the other hand, the passed reference may be used as an attribute of another object or may be used as a local variable but will be manipulated further after our method returns. These later situations can cause problems as we will see in the following sections.

4.1.2 A class with a mutable, reference-based internal state

Consider the following code:

```
public class Person {
  private BankAccount acc;

  public Person() {
    acc = new BankAccount();
  }

  public void setAccount(BankAccount a) {
    acc = a;
  }
  public BankAccount getAccount() {
    return acc;
  }
}
```

This code defines a class `Person` which contains a `BankAccount` attribute along with *setter* and *getter* methods [105]. The constructor creates a new `BankAccount` object and stores a reference to it in the private `acc` attribute. We can use the `getAccount` and `setAccount` methods to query and change this value, respectively. In the `setAccount` method we do not know whether there are other references to the passed `BankAccount` object. Likewise, callers of the `getAccount` method do not know what references there are to the object we return. As it happens, it is part of the internal state of this object but they would not know this from the type signature alone. If we are performing modular verification then the interface is all we can use to reason about other classes.

To clearly illustrate the problems that can arise from this situation, let us introduce an invariant to the `Person` class and attempt to verify it in a modular fashion. Let us alter the `Person` class as follows:

```
public class Person {
  private BankAccount acc;

  //@ private invariant acc != null && acc.getBalance() >= 0;

  public Person() {
    acc = new BankAccount();
  }

  //@ requires a != null && a.getBalance() >= 0;
  public void setAccount(BankAccount a) {
    acc = a;
  }

  //@ ensures \result != null;
  public BankAccount getAccount() {
    return acc;
  }
}
```

We have introduced a series of JML annotations in special comments which start with `//@`. The `requires` annotations provide pre-conditions, the `ensures` annotations provide post-conditions and the `invariant` annotations provide class invariants. A class invariant should (crudely) hold throughout the lifetime of all object instances of the class. More specifically, it should hold after the execution of any of the class's constructor methods and after any subsequent calls of its methods. It may be temporarily invalidated within the body of a method as long as it is re-established by the end of the method. Note that our invariant here has to be declared as private since it refers to a private field. This follows from Meyer's guideline that specifications at a specific accessibility level can only refer to methods and fields declared at that accessibility level or above [68].

The invariant that we have added states that the `acc` attribute should not be `null` and its `balance` should be non-negative. The `requires` clause of the `setAccount` method ensures that any `BankAccount` values that we assign to the local variable satisfy this requirement. Therefore, the invariant is respected. This class is correct JML and follows the principles of JML modular static verification.

4.1.3 The representation exposure problem

Now let us consider some code which uses this class.

```
Person p = new Person();
BankAccount acc = p.getAccount();
acc.withdraw(100);
```

Here we start by constructing a new `Person` object and assigning it to the local variable `p`. We then use the `getAccount` method from the `Person` class to retrieve a reference to the `BankAccount` object of the `Person`. Finally, we use the `withdraw` method to withdraw the value `100` from the account. This code follows the principles of JML static verification since the code respects the specifications of the `Person` and `BankAccount` classes. The `Person` constructor, `getAccount` method in the `Person` class and the `withdraw` method in the `BankAccount` class all have no pre-conditions and so our calls are valid.

However, the invariant of the `Person` object accessed by `p` is violated when we perform the `withdraw` operation on the `acc` variable. This is because the `BankAccount` that `getAccount` returns is used in the internal representation of our `Person` object. The problem is that the `getAccount` method gives mutable access to the internals of our `Person` class. Code outside the class can then use this reference to indirectly alter the value of the `Person` class. Because of the information hiding required to support modular verification they cannot be aware of this dependency because it is not described in the interface of the class.

This problem is called *representation exposure* and arises when a reference to an object used in the internal representation of an object is made accessible to clients of the class. Clients can then manipulate the object directly, bypassing the interface of the class which has been verified to maintain the correctness of any invariants over it.

4.1.4 The argument object dependency problem

We can arrive at the same problem through a slightly different means. Suppose we, again, use the `Person` class with the invariant. This time consider the following code instead:

```
Person p = new Person();
BankAccount a = new BankAccount();
a.deposit(100);
p.setAccount(acc);
a.withdraw(200);
```

Here we start by creating a new `Person` and assigning it to `p`. A default `BankAccount` object will be created by the `Person` constructor and stored in its `acc` attribute. We then create a new `BankAccount` and assign it to a local `a` variable. We then call the `deposit` method to deposit `100` into the `BankAccount`. Next we call the `setAccount` method of the `Person p`, passing the `BankAccount` we created. The pre-condition of this method requires that the passed object is not `null` and has a non-negative `balance` so that the invariant of the `Person` class is respected. Our passed `BankAccount` reference passes both of these criteria. Finally we withdraw `200` from our `BankAccount`. There is apparently nothing wrong with this since there is no pre-condition of the `withdraw` method. Therefore, this code also passes JML modular static verification.

However, again, the invariant of the `Person` object accessed via `p` is violated by the final statement. The `setAccount` method call takes a reference to our `BankAccount` object `a` and stores it in the attribute of the `Person` class. At the end of this method, the invariant is satisfied, just as our verification of the `Person` class ensured. However, when we use the `withdraw` method to remove `200` from the `BankAccount` the balance of the `BankAccount` goes to `-100`. This not only affects `a` but also our `Person` object `p` and leads to a violation of the invariant. Again, this dependency is not described in the interface of the `Person` class and so we do not know about it when verifying this code.

This problem is called *argument object dependency* and arises when a reference is passed into a constructor or method and incorporated into the internal representation of the object. Clients can then use the passed reference to manipulate the object's internal representation directly, bypassing its interface which is supposed to protect the integrity of the data.

4.1.5 Different views of the same problem

These are essentially different mechanisms by which you can reach the same problem. Via both approaches you end up with an object accessible through two reference variables where there is an invariant relating to one of the references that is not known by the other. If the second of these references is then used to mutate the object, the invariant relating to the first reference may be violated in a way that cannot be detected by the modular static verification.

The standard type signature interfaces are not enough to prevent this problem. Neither are standard pre- and post-conditions. Interfaces built out of these specification components alone say nothing about the number of references to the objects that are passed about.

4.2 Sound modular static verification techniques

Until relatively recently, there was no widely-accepted approach to solve this problem [77]. Users and tool developers working with JML had to work around this problem. The problem can be avoided if the programmer is careful about the references they use (e.g. they could clone the `BankAccount` before getting or setting it). However, there was no tool support to check for this which acted to limit the soundness of the verification approach as a whole. These limitations led people such as myself and David Crocker of the PerfectDeveloper project [33] to investigate the use of value semantics (which, in any event, is of fundamental importance even within languages primarily built upon reference semantics).

There are now two leading approaches to this problem. The first approach is to restrict the use of sharing through alias control. This technique allows us to describe the allowed aliasing of an object within the interface of a class. This approach is currently being incorporated into JML [37]. The second approach requires that all the invariants relating to object instances of a specific class are visible wherever those objects can be manipulated and that checks are made at the point of manipulation that no invariants in the system are violated. This work is already supported within JML tools.

4.2.1 Alias control through Universes

Alias control is perhaps the most natural solution to this problem. The fundamental problem is the unrestricted freedom in languages like Java for references to point to any object of the appropriate type. With this fundamentally unstructured and non-modular starting point we cannot easily perform static verification modularly. We cannot restrict the use of aliasing via existing specification languages, such as JML, because there are no constructs to allow us to express such restrictions. We need a new vocabulary which allows us to talk about aliasing so that we can describe suitable restrictions.

A number of alias control mechanisms were developed in the 90s [4, 29, 47] but most require a range of complicated annotations. Muller has pulled together the best aspects of a range of approaches to form his Universe type system [37, 46, 74-76, 78]. This provides a flexible alias control mechanism with relatively lightweight annotation burdens.

In this section we will look at how the Universe type system addresses the problems we saw earlier, and show how it can be used to implement a sophisticated example consisting of a doubly-linked list with iterators supporting deletion (an example taken from [37]).

Basic principles of the Universe type system

The Universe type system arranges the objects in an application into so-called *Universes*. Every object may have some other object declared as its *owner*. All objects which have the same owner are said to be in the same universe. Only the owner of an object can mutate that object. All modifications of the objects in a universe must go through the owner. This allows the system to ensure that the interface of a data structure is not bypassed. A class can then only define invariants over objects that it owns. Since only the considered object can update these objects, it is sufficient to verify that the invariant is maintained by the methods in the considered class.

A key innovation by the Universe type system is that, instead of explicitly stating the owner object of variables, it is generally sufficient to simply describe its relationship to `this`. Typically we require the owner of a field to be either `this` or the owner of `this`. When we declare an object variable or create a new object we must indicate what the owner of the object should be. To indicate that `this` should be the owner, we use the `rep` modifier. Objects with the `rep` modifier are created in the universe owned by `this` and can be viewed as part of its *rep*resentation (hence the modifier used). To indicate that the owner of `this` should be the owner, we use the `peer` modifier. Objects with the `peer` modifier will be created in the same universe as `this` and, hence, can be viewed as *peer*s of `this`. References whose owners are anything else must be declared with the `any` modifier and cannot be mutated.

Class with a mutable, reference-based internal state re-visited

Let us revisit our `Person` class. Our `acc` attribute is part of the internal representation of our `Person` and so we should declare it with the `rep` modifier.

```
public class Person {
  private /*@ rep */ BankAccount acc;

  //@ private invariant acc != null && acc.getBalance() >= 0;

  public Person() {
    acc = new /*@ rep */ BankAccount();
  }

  //@ requires a != null && a.getBalance() >= 0;
  public void setAccount(/*@ rep */ BankAccount a) {
    acc = a;
  }

  //@ ensures \result != null;
  public /*@ rep */ BankAccount getAccount() {
    return acc;
  }
}
```

By adding the `rep` modifier to the declaration of the `acc` attribute we have implicitly stated that the owner of `acc` is `this`. The result is that only this object is allowed to manipulate the value of that object. We can also define an invariant over it because it is owned by `this`.

Prevention of representation exposure

So let us now re-consider the first troublesome block of code. It was as follows:

```
Person p = new Person();
BankAccount acc = p.getAccount();
acc.withdraw(100);
```

The first thing we must do is declare the universes of the variables. We can no longer simply declare a variable `p` of type `Person`, we must also define what universe it is in. Let us assume that this code is in the body of a method of a different class. We can declare each type using either `rep`, `peer` or `any`. We want to mutate the variable `p` so we must declare it using `rep`.

```
/*@ rep */ Person p = new /*@ rep */ Person();
```

We would like to declare the `acc` local variable using `rep` as well so that we can mutate it. In other words, we would like to declare ourselves as its owner. However, the return type of the `getAccount` method is declared with the `rep` modifier which means that the owner of that object is the `Person` object itself. Thus, we cannot assign it to a variable which we are claiming ownership of ourselves. Instead, we can only declare it to be a `any` reference.

```
/*@ any */ BankAccount acc = p.getAccount();
```

Now, since we have a `any` reference, we cannot use the `withdraw` method to update `acc`. Only the owner of `acc` can update it. If the `getBalance` method were declared to be `pure` (i.e. it just calculates a value and does not mutate the object) then we could call it, but we cannot call non-pure methods or assign to its fields.

Prevention of argument object dependency

Let us now re-consider the second block of code:

```
Person p = new Person();
BankAccount a = new BankAccount();
a.deposit(100);
p.setAccount(acc);
a.withdraw(200);
```

Again, we should declare `p` using the `rep` modifier so that we can manipulate it.

```
/*@ rep */ Person p = new /*@ rep */ Person();
```

This time, we can also declare `a` using the `rep` modifier since we are not constrained by a value from elsewhere. Since we are the owners of `a` we can freely execute the next statement which makes a deposit to the account.

```
/*@ rep */ BankAccount a = new /*@ rep */ BankAccount();
a.deposit(100);
```

However, we can no longer pass this `BankAccount` to the `setAccount` method of `p` since the method requires as its parameter a `BankAccount` owned by `p`. Our

`BankAccount a` is owned by the object whose class contains the code being executed, which we have assumed in our premise was not `Person`.

So the universe type system prevents both of these troublesome situations from arising by associating with each object an owner and requiring that only the owner can update the object.

Exploring the restrictions of the Universe type system

Let us now explore a bit more of what can and cannot be done with the Universe type system.

Consider the following code:

```
/*@ rep */ Person p = new /*@ rep */ Person();
/*@ any */ BankAccount a = p.getAccount();
p.setAccount(a);
```

We saw earlier that we could not retrieve the `BankAccount` from a `Person` and then mutate it ourselves. This time, we retrieve the `BankAccount` as a `any` reference and then pass it back to the `Person p` using the `setAccount` method. This is fine since we can tell that the owner of `a` is `p` from the return type of the `getAccount` method and this fulfils the requirement of the `setAccount` parameter.

Consider this piece of code:

```
/*@ rep */ Person p1 = new /*@ rep */ Person();
/*@ rep */ Person p2 = new /*@ rep */ Person();
/*@ any */ BankAccount a1 = p1.getAccount();
p2.setAccount(a1);
```

Here we start by declaring two `Person` objects: `p1` and `p2`. We retrieve the `BankAccount` of `a1` and then attempt to assign it to `p2` using the `setAccount` method. This is not allowable since although `p1` and `p2` are instances of the same class, they are different objects. Universes are structured around objects, not classes. From the return type of `getAccount` we know that the owner of `p1.getAccount()` is `p1`. The parameter of `setAccount` is also declared with `rep` and so the owner of the parameter to `p2.setAccount` must be `p2`, which `p1` is not.

A larger case study: doubly linked lists with iterators supporting deletion

Finally, consider the doubly-linked list with iterators which Müller uses to illustrate the approach [37]. This example is relatively complicated and cannot be handled by many of the other aliasing control mechanisms.

Consider first the `Node` class which has references to the previous and next nodes (that are in the same universe as `this`) and a `any` reference to the value contained at this node named `elem`.

```
class Node {
  public /*@ peer */ Node prev, next;
  public /*@ any */ Object elem;
}
```

Next let us consider the `LinkedList` class. Firstly, it contains references to the first and last nodes. These are declared using `rep` and so are owned by `this` and stored within the universe owned by `this`. This allows only `this` and the peers of `this` to manipulate them. The `iterator` method constructs an `Iterator` as a `peer`. This, together with the `remove` method, is discussed in the next section. We can also define an `equals` method which tests for deep equality. We can accept a `any` reference to the other `LinkedList` (i.e. a `LinkedList` from any other universe) because we only need to inspect it and do not need to manipulate it.

```
public class LinkedList {
  protected /*@ rep */ Node first, last;

  public /*@ peer */ Iterator iterator() {
    return new /*@ peer */ Iterator(this);
  }
  void remove(/*@ rep */ Node np) {
    ...
  }
  public boolean equals(/*@ any */ LinkedList l) {
    /*@ any */ Node f1 = first;
    /*@ any */ Node f2 = l.first;
    while (f1 != last && f2 != l.last && f1.elem == f2.elem) {
     f1 = f1.next;
     f2 = f2.next;
    }
    return f1 == last && f2 == l.last && f1.elem == f2.elem;
  }
  ...
}
```

Now let us consider the `Iterator` class. The `iterator` method of the `LinkedList` constructs one of these and returns it to the clients. The iterator has two attributes: the `LinkedList` to iterate over which is from the same universe (and hence declared with `peer`) and a reference to the current `Node`. We must declare this `Node` using `any` since it is neither owned by `this` (`rep`) or by the owner of `this` (`peer`). In fact, we know that the owner should be the `LinkedList` that we are iterating over and we can formalise this in an invariant referring to the special `owner` field. The `remove` method is of particular interest. We cannot implement the deletion ourselves because we are not the owners of the `Node` and so cannot manipulate it. Only the `LinkedList` containing the nodes can manipulate them. We can, however, define a `delete` method in `LinkedList` and pass our `Node` to it.

```
public class Iterator {
  protected /*@ peer */ LinkedList list;
  protected /*@ any */ Node pos;

  //@ invariant pos.owner == list;

  Iterator(/*@ peer */ LinkedList l) {
    list = l;
    pos = l.first;
  }
  public /*@ any */ Object next() {
    /*@ any */ Object result = pos.elem;
    pos = pos.next;
    return result;
  }
  public void remove() {
    list.remove(pos);
  }
  ...
}
```

Evaluation of the Universe type system

We have seen that the Universe type system is fairly powerful. It was able to rule out the problems that we met earlier but is still able to model interesting examples like doubly-linked lists with iterators that incorporate deletion. Recent work has also shown that it can be applied to industrial examples [46]. Certain adjustments are typically needed in the structuring of the applications, but these changes encourage a more modular and more maintainable architecture and so could be viewed quite favourably.

There are, however, still some common implementation patterns that it does not support [78], e.g. the composite pattern where there is no separate owner through which all manipulations take place [60]. New techniques are needed to extend the work to handle these problems.

Currently there is no support for the transfer of ownership or multiple ownership. Transfer of ownership could perhaps be achieved through the use of cloning like that which is currently used within Omnibus to support value semantics. Handling of multiple ownership may require a hybrid approach incorporating aspects of the techniques described in the next section.

4.2.2 Visibility-based invariants

The second approach is called visibility-based invariants and requires that all the invariants that can be affected by a field assignment are visible in the methods that contain the updates. Whenever we assign to a field of an object we not only check the invariant of the class the code is in but all the other invariants that are visible and could be affected by the changes.

This approach does not work for the `Person-BankAccount` example as it is since the invariant in class `Person` is private and so is not visible from the client code outside the class. To support this verification mechanism we would have to make the invariant public which would require us to make the `acc` attribute public so that we can refer to it in the invariant. We would also have to describe in the specification how this attribute is manipulated so that clients can tell at all times what `BankAccount` is owned by the object. Then any time we mutate a `BankAccount` object we have to prove that either it is not referenced by a `Person` object and hence cannot violate the invariant or that it is referenced by a `Person` object but maintains the invariant. For our example code snippets, this process would raise an error at the final lines where we attempt to mutate the `BankAccount` objects which are referenced by a `Person` object.

Visibility-based invariants are very powerful and can be applied in situations where ownership-based verification is not possible. However, there are some serious problems with the approach [60]. Firstly, it is an intrinsically non-modular approach and involves the sort of complicated reasoning that is characteristic of non-modular static verification. Considerably more proof obligations are generated since invariants have to be checked whenever field assignments take place and not just at the end of the methods in the class that contains the declarations of the fields. The proof obligations may also be very complicated since they must refer to all the objects in the heap. Secondly, the visibility requirement is very strict. For example, if an invariant involving an array is defined then any method that uses an array must be checked to make sure it does not violate the invariant. Thirdly, subtyping causes real problems for the visibility requirement since a subclass may be defined in a separate package where it cannot see a relevant invariant.

The complexity of the visibility-based invariant approach compromises the ability to which it can be automated and, by being non-modular, will not scale well.

Future work on the combination of support for reference semantics with the value semantics techniques of the Omnibus language are discussed in section 9.3.1.

Chapter 5

Combining verification approaches

In this thesis we focus on three assertion-based techniques for the integrated specification, implementation and verification of object-oriented software: Run-time Assertion Checking (RAC) [68], Extended Static Checking (ESC) [41] and Full Formal Verification (FFV) [34]. 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 source, easing analysis and correction. In ESC, the consistency between code and its 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 [36]. ESC is neither sound nor complete, aiming simply to detect assertion violations and a range of common run-time errors. FFV, or full formal verification, provides support for traditional, heavyweight assertion-based verification. The approach requires the production of heavyweight specifications (also known as Behavioural Interface Specifications [59]) 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.

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 different approaches to be used together in an integrated fashion within different parts of the same project.

Traditionally, support for these approaches has been developed by separate teams, yielding separate tools often targeting separate languages. JML 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, until recently, used the tools independently rather than in an integrated fashion [51] although there were suggestions of using the approaches together. In [51], Jacobs et al. proposed using ESC to guide the selective application of FFV. The idea is first to apply ESC to the entire project, verifying much of it, and then to use interactive FFV to attempt to verify the remainder. In [30], Cok and Kiniry discussed how beneficial they thought it would be to have an integration of tools that support JML, but at the time there was no tool support for this.

Other tools have combined static and dynamic checking to some extent. Spec# [10] uses a combination of static and dynamic techniques to verify its assertion annotations. They exploit the fact that constructs that are difficult to statically check are often relatively easy to dynamically check 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 [33] 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.

Our integrated support for ESC and FFV hinges on our generic logic which allows us to support interactive and automated provers. While ESC and FFV tools have been implemented separately, many FFV tools support the use of both automated and interactive provers (e.g. [2, 3, 9, 19]) and some support different interactive theorem provers (e.g. [72]).

Independently of our own work on the integrated use of approaches together [97, 101-103], people in the JML community have been working towards the integration of ESC and interactive verification (which comes within our definition of FFV) [57]. More recently, an architecture of JML4 has been proposed which will allow the integration of RAC, ESC and FFV [23]. The proposal does not include anything similar to our concept of verification policies.

Portions of this chapter have previously been presented elsewhere [102, 103].

5.1 Addressing limitations of ESC with the other approaches

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.

5.1.1 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 fairly well received [62] and appears to hold great promise.

5.1.2 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 [41]. 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. The `Optional` class is used to model possibly null values. This class has two constructors: `nil` for where no value is provided and `of` for where there is a value. We will add annotations later as needed to respond to errors reported by our ESC tool.

```
public class Bag[Element] {
 private attribute arr:Array[Element]
 private attribute size:integer
 private attribute comparer:Comparator[Element]

 public constructor containing
    (input:List[Element],
     comp:Comparator[Element])
 {
  comparer := comp;
  size := input.size();
  arr := Array[Element].ofSize(size);
  for (i := 0 to size - 1) {
    arr.assign(i,
               Optional[Element].of(input.elementAt(i)));
  }
 }

 public operation extractMin(out min:Element)
 {
  var minIndex:integer := 0;
  min := arr.access(0).value();
  for (i := 1 to size-1) {
    if (comparer.compare(arr.access(i).value(),
                              min) = Ordering.before()) {
      minIndex := i;
      min := arr.access(i).value();
    }
  }
  size := size - 1;
  arr.assign(minIndex, arr.access(size));
 }
}
```

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

```
Bag.obs:22: Unable to verify public requires clause of the
     access function declared in omni.lang.Array at line 6
Bag.obs:22: Unable to verify public requires clause of the
     value function declared in omni.lang.Optional at line 6
Bag.obs:24: Unable to verify public requires clause of the
     access function declared in omni.lang.Array at line 6
Bag.obs:24: Unable to verify public requires clause of the
     value function declared in omni.lang.Optional at line 6
Bag.obs:31: 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 24 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 [41].

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).isNil()
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 := Bag.containing(ints,
7:      DefaultIntegerComparator.init());
8:   var sortedInts:List[Integer]
9:      := List[Integer].empty();
10:   while (b.size() > 0) {
11:      var m:integer;
12:      b.extractMin(out m);
13:      sortedInts.add(m);
14:   }
15:   return sortedInts;
16:  }
15: }
```

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 the `containing` and `extractMin` methods 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 size 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 [10]. 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 circumvent 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.

5.1.3 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 static modular checking. So RAC does not 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 have no formal basis within the ESC system and so can be used to circumvent 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 can be 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 [10].

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 [71], and
2. sufficient basis to trust that the implementation satisfies the specification [70].

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 a later section.

5.2 Guidance on how to use 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 class of verification since the application may contain classes that are not verified with any of the approaches; 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; where these occur we treat them as two separate usage links.

5.2.1 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 do not 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 cannot typically 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, will 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. 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.

5.2.2 Guidelines for avoiding conflicts between approaches

We have developed a set of guidelines to help developers to avoid conflicts between the approaches.

1. All pre-conditions of methods should be RAC- and ESC-compatible if they are called from RAC- or ESC-verified classes.
2. All post-conditions of methods should be ESC-compatible if they are called from ESC-verified classes.
3. All assertions should be checked using at least one of the approaches.
4. A class should not directly use a less rigorously verified one where RAC, ESC and FFV offer verification at increasing levels of rigour.

Guidelines 1 and 2 are to do with the compatibility of specifications between approaches. Guideline 3 ensures that when the approaches are used together, no assertion checks are missed. Guideline 4 describes forms of integration that will be troublesome and should be avoided.

Guideline 1: Pre-condition compatibility

Guideline 1 says that the pre-conditions for all methods should be RAC- and ESC-compatible if an RAC- or ESC-verified class calls the method. As was discussed earlier, to ensure RAC- and ESC-compatibility, 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 RAC and ESC.

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. 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 rewriting 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-compatibility when used in conjunction with guideline 2. 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: Post-condition compatibility

Guideline 2 says that the post-conditions for all methods should be ESC-compatible if an ESC-verified class calls the method. Post-conditions are even more prone to ESC-compatibility problems since, in FFV, heavyweight specifications may have been required to sufficiently describe the method.

This problem can be addressed by using `which ensures` clauses to provide lightweight ESC-compatible redundant specifications for non ESC-compatible `ensures` clauses. 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 5.1.3 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 5.1.3 was not ESC-compatible when we attempted to verify the `IntegerSorter` class using ESC. Alternatively we could have defined a test case within the `Bag` class to check its ESC-compatibility. Omnibus tests can have a policy specified in a `policy` clause; if present, this indicates that the specified approach should be used to verify the test. So an FFV-verified `Bag` class we can define a test to be verified using ESC. We refer to tests to be verified with RAC, ESC or FFV as *RAC-targeted*, *ESC-targeted* or *FFV-targeted tests*, respectively.

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. If we do not 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.

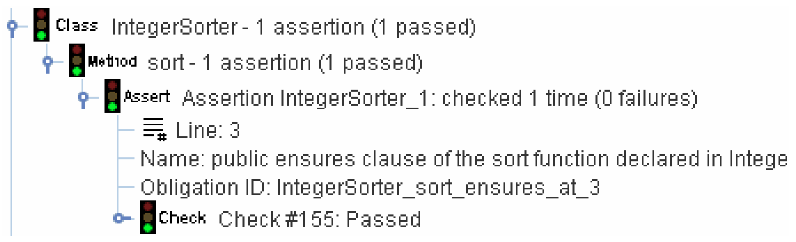
In our experience, the majority of methods can be written to be ESC-compatible and, for those that cannot, redundant specifications are generally easy to write and are useful to verify some desired properties of the specification.

Guideline 3: Assertion coverage

Guideline 3 states that all assertions should be checked using at least one of the approaches. For example, every time a method is called, its pre-condition should be checked and every time an ESC `assume` statement is used it should be checked by another approach to ensure that it is true. The use of different approaches for different classes can make it possible for assertions to be missed because RAC checks assertions in a different way from ESC and FFV. In ESC and FFV, the caller of a method is responsible for checking the pre-condition of the method whereas, in our RAC approach, the called method is responsible for checking its own pre-condition.

The Omnibus IDE provides two tools for tracking the verification performed: the *check viewer* and the *obligation viewer*. The check viewer displays the details of the verification that was performed on a file. This is useful because the errors only describe the things that were checked and failed, but the programmer may be interested in reviewing the things that were checked and passed.

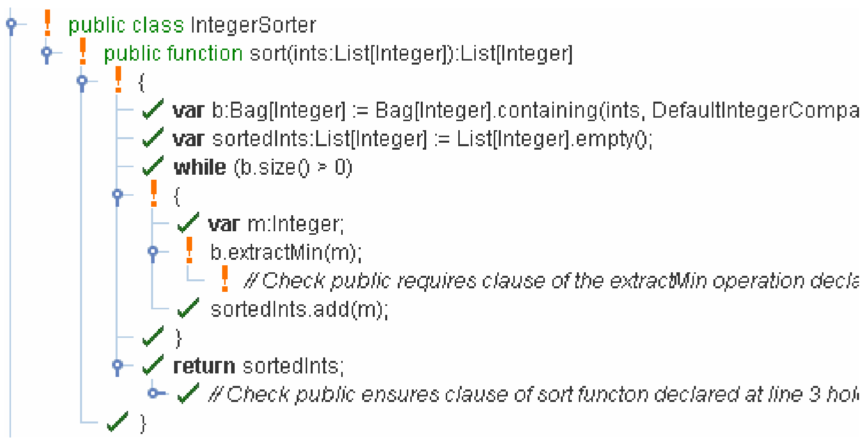
Consider the situation where we use ESC to verify the `Bag` class and RAC to verify the `IntegerSorter` class. Since we use RAC to verify the `IntegerSorter` class, a run-time check will be generated for the `ensures` clause of the `sort` function. This is the only check that is generated for the file and, given an appropriate test case, it is checked so the check viewer (shown below) reports that the verification was successful.


```
Class  IntegerSorter - 1 assertion (1 passed)
  Method  sort - 1 assertion (1 passed)
    Assert  Assertion IntegerSorter_1: checked 1 time (0 failures)
      Line: 3
      Name: public ensures clause of the sort function declared in Intege
      Obligation ID: IntegerSorter_sort_ensures_at_3
      Check  Check #155: Passed
```

No run-time check was generated for the pre-condition of the `extractMin` operation of the `Bag` class called from the `sort` function in the `IntegerSorter` class because in our RAC approach it is the responsibility of the method being called to run-time check its pre-condition. However, the `Bag` class has no checks because it was verified using ESC, not

RAC. So there is a gap in the assertion coverage of the approaches. The assertion to check the `requires` clause of the `extractMin` operation call is not being checked by any of the approaches. The first problem is how we can detect this situation, the second being how we can correct it.

In order to detect the situation, Omnibus supports the generation of *correctness obligations* which give details of all the assertions that need to be checked for each class. In order to ensure assertion coverage, *justification* should be provided for each correctness obligation. Justification can take the form of the execution of a run-time assertion check or the proof of an appropriate VC. The obligation viewer tool displays the correctness obligations for a class together with any corresponding justification. A screenshot of the tool is shown below. The correctness obligations are shown as special comments within a tree display of the source of the class. Ticks indicate that there are no unjustified correctness obligations in a branch. Exclamation marks indicate that there are some unjustified correctness obligations in a branch. Crosses are used to indicate errors (i.e. obligations which have a failed run-time check or failed VC as justification). The screenshot indicates that, when RAC is used for the `IntegerSorter` class and ESC is used for the `Bag` class, there is an unjustified correctness obligation that the `requires` clause of the `extractMin` operation is satisfied. ESC `assume` statements would also appear as unjustified correctness obligations unless they were checked using one of the other approaches.



```
! public class IntegerSorter
  ! public function sort(ints:List[Integer]):List[Integer]
    ! {
      ✓ var b:Bag[Integer] := Bag[Integer].containing(ints, DefaultIntegerCompa
      ✓ var sortedInts:List[Integer] := List[Integer].empty();
      ✓ while (b.size() > 0)
      ! {
        ✓ var m:Integer;
        ! b.extractMin(m);
          ! // Check public requires clause of the extractMin operation decl
        ✓ sortedInts.add(m);
      ✓ }
      ✓ return sortedInts;
        ✓ // Check public ensures clause of sort functon declared at line 3 ho
    ✓ }
```

Correctness obligations are also used to identify special cases which should be verified using different policies, as discussed in section 5.3.2, and as the basis for the certification system discussed in chapter 6.

In order to ensure no pre-condition checks are missed, all classes should, in general, include run-time 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 4: Non-decreasing rigour

Guideline 4 advises that a class should not directly use (i.e. instantiate and call methods of) another class which was verified using a less rigorous verification approach. RAC, ESC and FFV provide verification at increasing levels of rigour. An ESC-verified class should not directly use an unverified class because the unverified class will have no specification that can be used to reason about its method calls. If an ESC-verified class uses an RAC-verified class, the specifications in the RAC-verified class may need to be enhanced so that they include constructs, such as private `ensures` clauses for model functions and `changes` clauses, which are not required for RAC but are essential for the use of ESC. Similarly, FFV-verified classes should not directly use unverified, RAC-verified or ESC-verified classes because the specifications of those classes will not be sufficient.

Guideline 4 states that these combinations of approaches 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 by 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 extreme we will be forced to increase the verification level of the entire application to 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 5.2.3.

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/RAC 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 [97].

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 do not 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 5.2.3) 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.

5.2.3 Recommendations for when to use each approach

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 developers flexibility to make such choices themselves, we have created 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.

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.

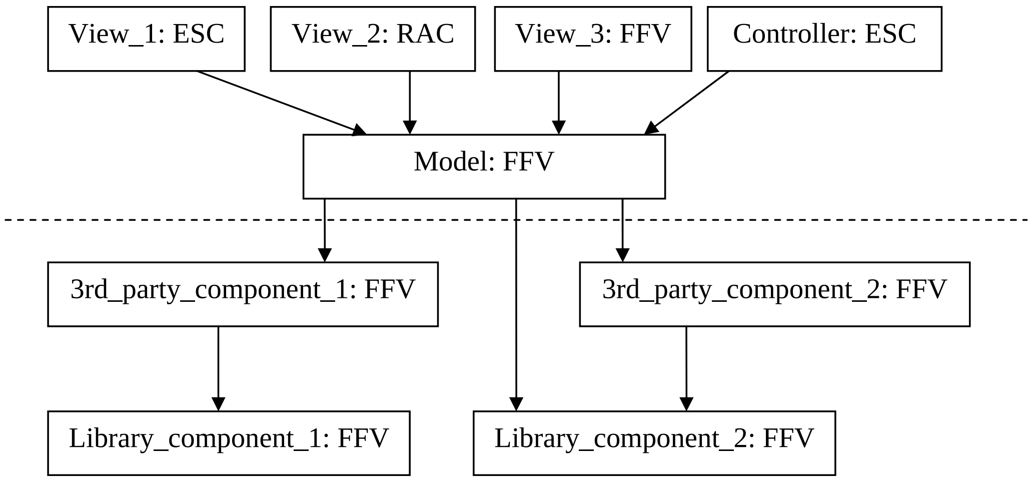
When to use RAC instead of ESC: ESC requires considerable up-front effort in order to provide the tool with enough annotations to perform static modular checking. Even if a system has no bugs, the ESC tool will keep reporting warnings until the developer has provided sufficient assertion annotations.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

Structuring the verification of an application

The decisions about 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 that the interactions between the verification approaches for the classes in the application are focussed around the uses relationships, so it may be useful to produce a diagrammatic 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.



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 [97].

5.3 Verification policies

The Omnibus IDE manages the support for different verification approaches 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 defined as built-in verification policies but the user is also free to combine aspects of the different approaches to create new policies. A single policy can combine the use of run-time checking and static verification and different policies can even be used within a single class.

5.3.1 Configuring the approaches

In this section we will discuss some ways in which the run-time checking and static verification can be configured. Static verification encompasses ESC and FFV. Related work includes the configurable options in tools such as ESC/Java2 [30] and PerfectDeveloper [33]. Full details of the settings for Omnibus verification policies are given in appendix C.

Configuring run-time checking

There are a number of ways in which the run-time 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 run-time checked 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). 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.

Configuring static verification

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, 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. These 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. Our ESC approach also makes a range of compromises in soundness in order to make the approach easier to use. Loops are analysed by unravelling 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, our ESC approach 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 run-time 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. The user 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 [36] and the interactive PVS prover [81]. 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 often cannot.

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 unravelling 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 run-time checking, a number of constructs from the assertion language may cause problems, 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 to 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.

Creating new approaches

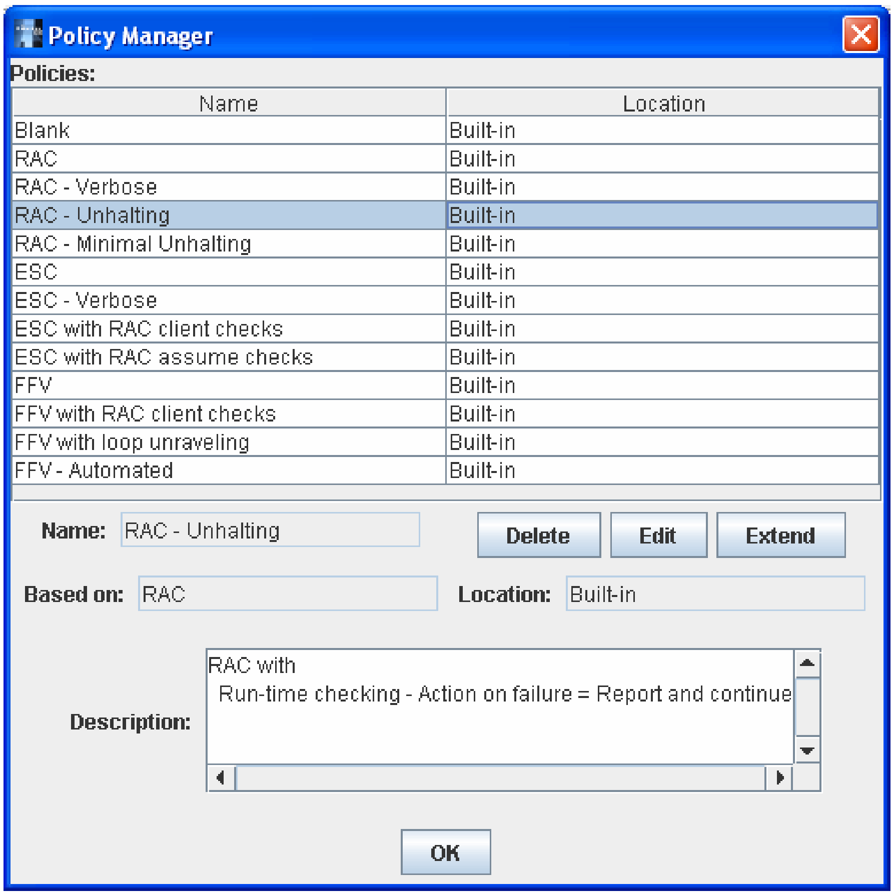
Through this configuration process, we can create new approaches that combine aspects of the traditional approaches. For example, we could develop different configurations for: (i) FFV, but with the loop unravelling technique from ESC to avoid the need for loop invariants; (ii) ESC and FFV with and without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

5.3.2 Verification policy tool support

There are three main tools that allow the developer to use policies within Omnibus: the policy manager, the policy editor and the policy selector. These are discussed in the following sections.

Policy Manager

The Policy Manager is a tool for managing the definitions of verification policies. It displays all the policies currently loaded into the system and provides options to delete them, edit them and extend them to create new policies. A screenshot of the tool is shown below. The upper area displays the verification policy names along with whether they are built-in or saved in the current project. The lower area displays details of the currently selected policy, including the policy it is based on and a description of how it differs from the policy it is based on.



Policies:

Name	Location
Blank	Built-in
RAC	Built-in
RAC - Verbose	Built-in
RAC - Unhalting	Built-in
RAC - Minimal Unhalting	Built-in
ESC	Built-in
ESC - Verbose	Built-in
ESC with RAC client checks	Built-in
ESC with RAC assume checks	Built-in
FFV	Built-in
FFV with RAC client checks	Built-in
FFV with loop unraveling	Built-in
FFV - Automated	Built-in

Name: RAC - Unhalting

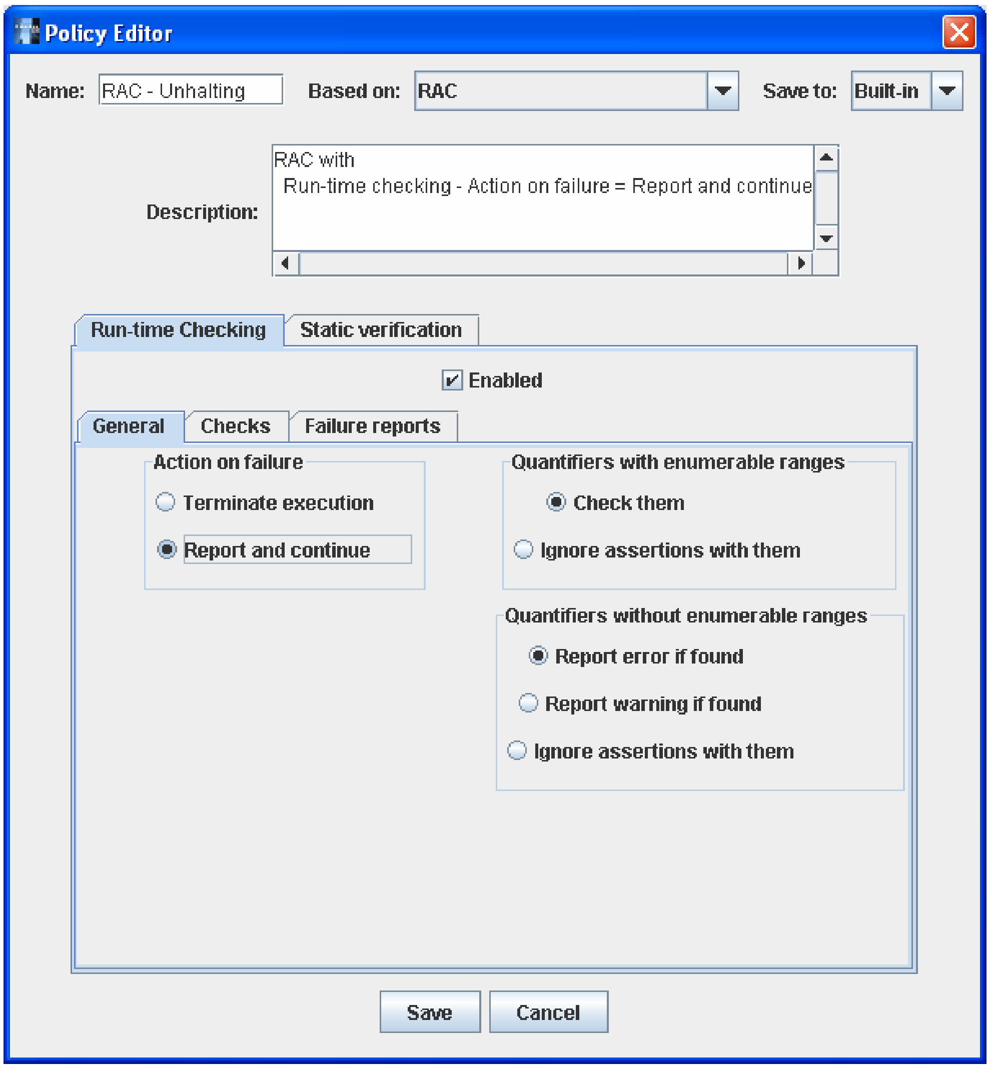
Based on: RAC

Location: Built-in

Description:
RAC with
Run-time checking - Action on failure = Report and continue

Policy Editor

Creation and editing of policies is achieved through the policy editor, a dialogue allowing the developer to completely configure a particular policy. A screenshot of this dialogue is shown below. The upper area allows the name of the policy, the policy it is based on and the location where it is saved to be edited. Immediately below that is a preview of the automatically generated description of the policy describing how it differs from the policy it is based on. In the centre of the dialogue are a series of tabbed panes for specifying the properties of the policy. Information on the configurable properties for policies can be found in appendix C along with details of the property settings for the built-in policies.



Name: RAC - Unhalting
Based on: RAC
Save to: Built-in

Description:
RAC with
Run-time checking - Action on failure = Report and continue

[✓] Enabled

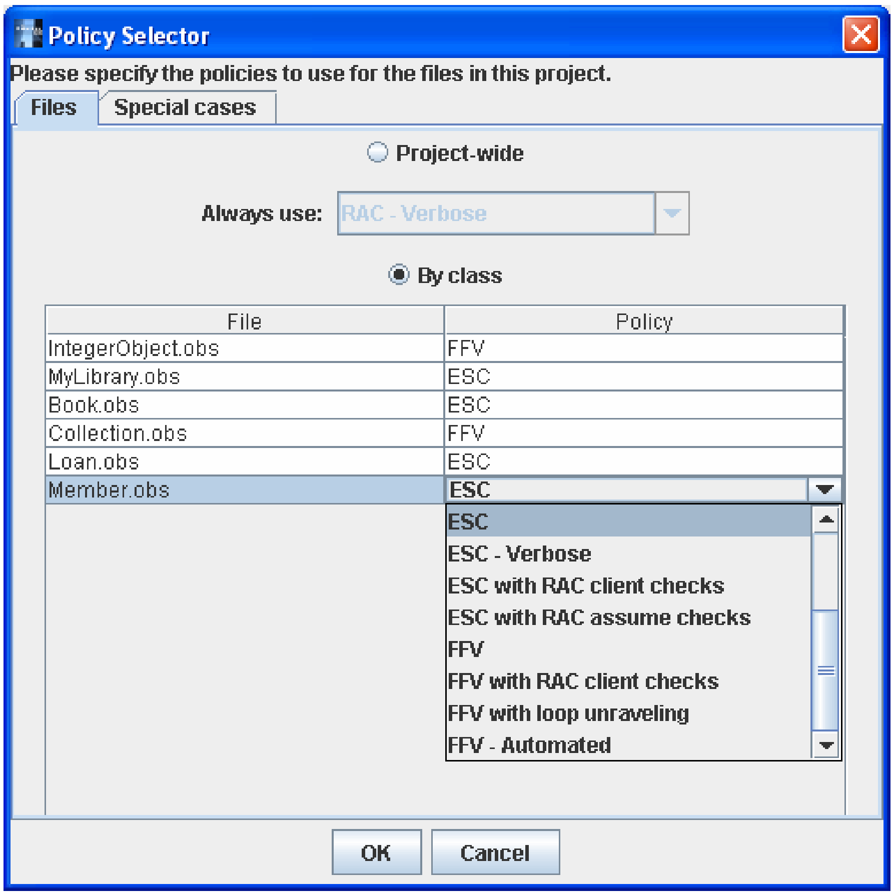
Action on failure
○ Terminate execution
● Report and continue

Quantifiers with enumerable ranges
● Check them
○ Ignore assertions with them

Quantifiers without enumerable ranges
● Report error if found
○ Report warning if found
○ Ignore assertions with them

Policy Selector

Finally, the Policy Selector tool is used to describe how verification policies should be used to verify an Omnibus project. This tool contains two parts, the first of which allows the policies to be used on a file-by-file basis. A screenshot of this part of the tool is shown below. For convenience, the policies to be used for each file can be specified in one of two ways. A single policy can be specified for all files in the project, or different policies can be specified for each file using the provided table.



Please specify the policies to use for the files in this project.

( ) Project-wide

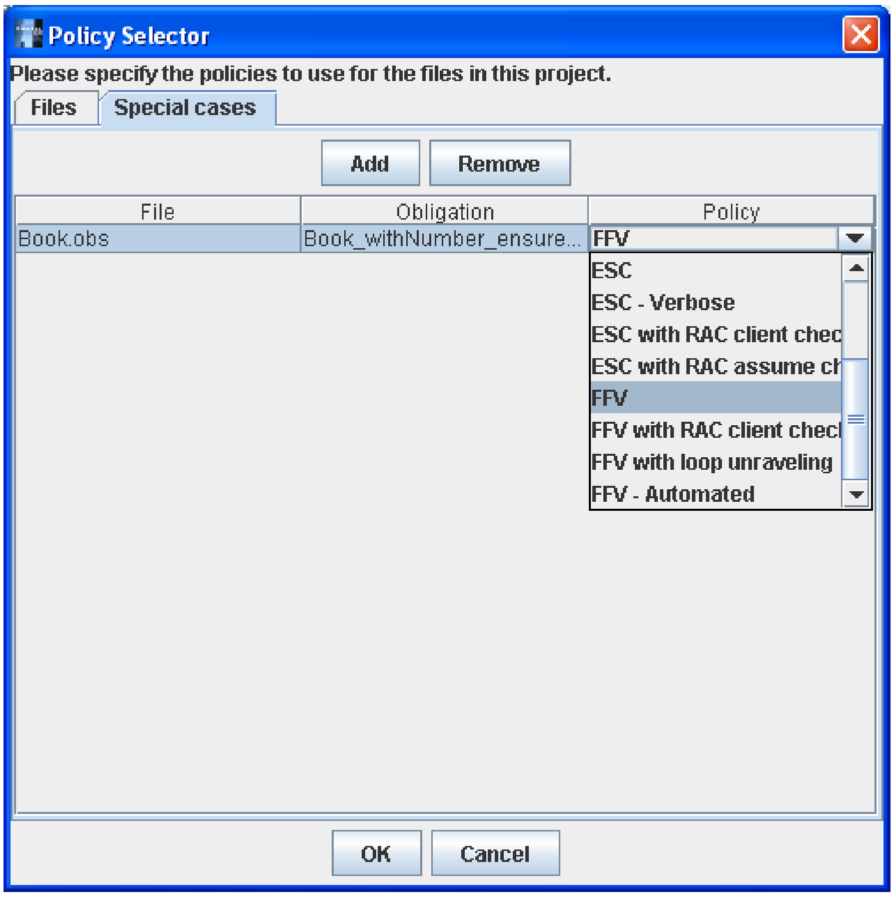
Always use: RAC - Verbose

(●) By class

File	Policy
IntegerObject.obs	FFV
MyLibrary.obs	ESC
Book.obs	ESC
Collection.obs	FFV
Loan.obs	ESC
Member.obs	ESC

ESC
ESC - Verbose
ESC with RAC client checks
ESC with RAC assume checks
FFV
FFV with RAC client checks
FFV with loop unraveling
FFV - Automated

The second part of the Policy Selector allows particular correctness obligations to be identified as special cases and use a different policy from the rest of the file. A screenshot of this part of the tool is shown below. For each special case, the file and ID of the obligation are given and then the policy to be used for the obligation is selected.


Policy Selector

Please specify the policies to use for the files in this project.

File	Obligation	Policy
Book.obs	Book_withNumber_ensure...	FFV

ESC
ESC - Verbose
ESC with RAC client chec
ESC with RAC assume ch
FFV
FFV with RAC client chec
FFV with loop unraveling
FFV - Automated

Section 8.5.2 demonstrates the power of the use of special cases.

Chapter 6

Safe software component reuse

In this chapter we consider the use of the Omnibus tool with its integrated verification approaches for the development of software components in a way that enables them to be reused safely. We assume that the developers of components and the users of the component may be distinct. Crucially, we must implement the component, verify it and then *distribute* it. Once it is distributed, we cannot easily alter it to correct errors. So, as responsible component vendors, we would like to ensure that our component is well documented and free from errors 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 [71], and
2. some form of assurance that it does it [70].

6.1 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). A system of *digital* *signatures* is typically used to provide assurances of correctness. This approach uses encryption techniques to allow components to be digitally signed by an organisation which users can decide whether they trust or not. When the producer of the component is an organisation like Sun this works relatively well since the interfaces of their libraries of components are comprehensively documented and the components contain very few implementation errors. However, the system does not always work as well when applied in general to third-party component vendors. There are no requirements on the level of documentation of reusable component interfaces and so it is often patchy. It is also not clear if a hidden implementation of a component can be trusted to be free from errors. The producers of the component may attempt to get their code digitally signed by a trusted organisation but there are no clear procedures to govern this process and so this causes a signification bottleneck [12]. As an alternative, the producers of the component may simply sign their own code but then the process has contributed no further assurances. As a result, the reuse of software components in this way often introduces problems. To protect themselves against this, many programmers develop “not invented here” syndrome and prefer re-implementation to reuse of existing components when the implementer of the component is not known to be dependable. This has acted to limit the spread of reusable software components.

6.2 Use of the approaches for reusable components

In this section we discuss the use of the different verification approaches we have considered to support the safe reuse of software components.

6.2.1 Developing reusable components with RAC and ESC

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 in 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 we have to make compromises in the expressiveness of the assertion annotations we use in conjunction with these approaches.

Using ESC we may not be able to verify the correctness of some implementations relative to a full, heavyweight specification of correctness and so we may have to use a relatively incomplete specification. At this point the assertion annotations and our conception of the correctness of the component diverge. We can document the additional requirements in interface documentation, but they cannot be included within the assertion annotations because they cannot be verified by the approach. This limits the extent to which the vendor can provide a clear and comprehensive description of what the component should do. Also, concessions have to be 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. This limits the extent to which successful verification can be taken as an assurance of correctness.

Using RAC it is possible to check more expressive heavyweight specifications. 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 do not meet their 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.

6.2.2 Developing reusable components with FFV

The needs of reusable components can be more completely met by the FFV approach. 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. Using RAC and ESC we may have to make concessions in the specifications we use, but we are not forced to do this in FFV because the interactive prover can be used to verify any expressible properties. RAC and ESC also have soundness and completeness loopholes which FFV does not. The error coverage of RAC is limited by the coverage of the test harness and the completeness of the specifications. The error coverage of ESC can be limited by the use of assumption constructs and, again, incompleteness of the specifications. FFV can be used to avoid these restrictions. The interactive proof mechanism allows complicated properties to be verified, allowing programmers to use more sophisticated specifications. 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.

FFV is, however, a costly approach. Our FFV approach does not support loop invariant generation and instead requires the user to devise suitable loop invariants. It can take several attempts to get these right, and often the loop invariants can be more error-prone than the code itself. It can be discouraging when a product of the verification process is harder to debug than the original code. 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.

The costs of FFV will often be beyond acceptable levels unless reliability is critical. However, for reusable components, the economies of scale may help justify this additional effort. For example, consider the extent to which components in the standard Java libraries are reused. It may be considered worthwhile to use FFV to verify these kinds of components so that comprehensive heavyweight specifications can be used to describe the interfaces and the implementations can be rigorously shown to respect them before distribution.

Using FFV for a component also gives clients that use the component maximum freedom in which verification approaches to use themselves. We saw earlier that our guideline 4 rules out certain combinations of verification approaches. If we verify a component using RAC or ESC we cannot verify clients that use the component subject to FFV. If the producer and user of the component are the same people, they could adjust the component to use FFV if they wished to use FFV for clients. However, if the producer and users are different this may not be possible. In that case the clients will have to work around this constraint and only use RAC or ESC for their classes which use the RAC- or ESC-verified component. Consider again the case of components in standard libraries. If we used RAC or ESC for these, they could not be used in projects that are FFV-verified. If we use FFV though, we could use RAC, ESC or FFV to verify classes which use the components.

6.3 Specialised support for open source components

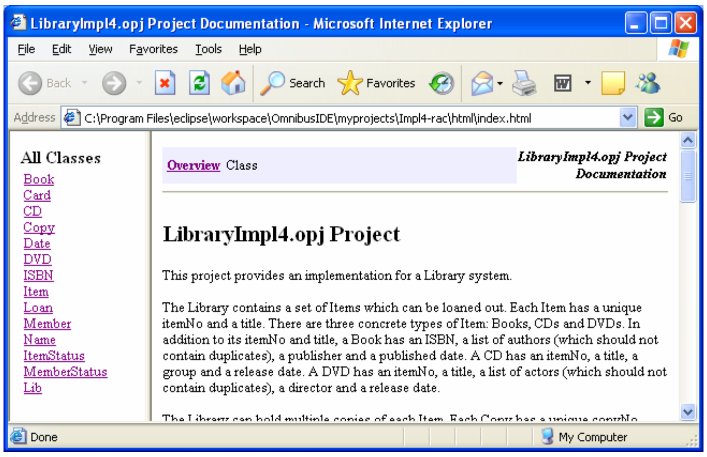
The Omnibus IDE provides a range of special facilities to directly address the needs of safe software component reuse providing that the components have their source code distributed with them. These facilities are discussed in this section. Techniques for components which do not have their source distributed with them are discussed in section 6.4.

6.3.1 Interface documentation

The first requirement for safe software component reuse is a clear, unambiguous description of what the each component should do. The assertion annotations in our RAC, ESC and FFV approaches provide a framework for the unambiguous specification of the intended behaviour of components. It is, however, important that support is provided to extract these specifications from the files so the users do not have to refer to the source files themselves to find these specifications.

There is a standard solution to this problem. It is to provide tools to automatically generate interface documentation from the source files. The JavaDoc tool [96] provides such support for Java, taking the type signature interface and special documentation comments and generating summaries in an HTML form. The JMLdoc [87] tool provides the same facilities, but also includes any JML assertion annotations in the generated documentation. We have followed this approach in the Omnibus system.

The Omnibus IDE provides an option to generate interface documentation for each Omnibus project. This documentation uses the same frame-based layout as JavaDoc. There is an index with links to all the files in the project in the left frame. A summary page for the project is initially displayed in the right frame. The summary includes the description of the project provided in the Project Window of the Project in the IDE and descriptions of the source files taken from their description clauses.


LibraryImpl4.opj Project Documentation - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address C:\Program Files\eclipse\workspace\OmnibusIDE\myprojects\Impl4-rac\html\index.html Go
Done
My Computer

All Classes
Book
Card
CD
Copy
Date
DVD
ISBN
Item
Loan
Member
Name
ItemStatus
MemberStatus
Lib

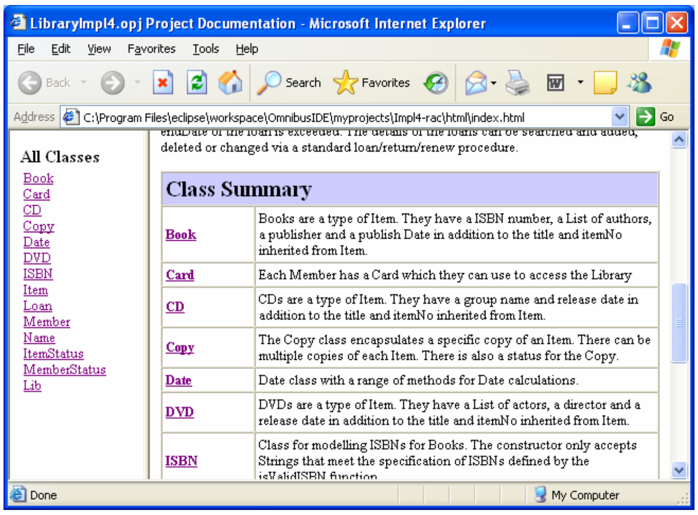
Overview Class

LibraryImpl4.opj Project Documentation

LibraryImpl4.opj Project

This project provides an implementation for a Library system.

The Library contains a set of Items which can be loaned out. Each Item has a unique itemNo and a title. There are three concrete types of Item: Books, CDs and DVDs. In addition to its itemNo and title, a Book has an ISBN, a list of authors (which should not contain duplicates), a publisher and a published date. A CD has an itemNo, a title, a group and a release date. A DVD has an itemNo, a title, a list of actors (which should not contain duplicates), a director and a release date.

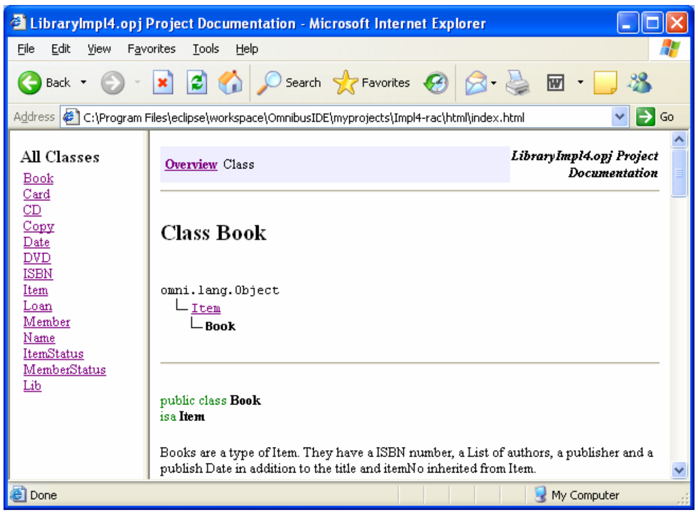

LibraryImpl4.opj Project Documentation - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address C:\Program Files\eclipse\workspace\OmnibusIDE\myprojects\Impl4-rac\html\index.html Go
Done
My Computer

All Classes
Book
Card
CD
Copy
Date
DVD
ISBN
Item
Loan
Member
Name
ItemStatus
MemberStatus
Lib

deleted or changed via a standard loan/return/renew procedure.

Class Summary	
Book	Books are a type of Item. They have a ISBN number, a List of authors, a publisher and a publish Date in addition to the title and itemNo inherited from Item.
Card	Each Member has a Card which they can use to access the Library
CD	CDs are a type of Item. They have a group name and release date in addition to the title and itemNo inherited from Item.
Copy	The Copy class encapsulates a specific copy of an Item. There can be multiple copies of each Item. There is also a status for the Copy.
Date	Date class with a range of methods for Date calculations.
DVD	DVDs are a type of Item. They have a List of actors, a director and a release date in addition to the title and itemNo inherited from Item.
ISBN	Class for modelling ISBNs for Books. The constructor only accepts Strings that meet the specification of ISBNs defined by the

Each class in the project has its own specification page which again follows the standard JavaDoc structure. The specification of each method is included in the file along with details of the requirements specifications and test cases.


LibraryImpl4.opj Project Documentation - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address C:\Program Files\eclipse\workspace\OmnibusIDE\myprojects\Impl4-rac\html\index.html
Go
Done
My Computer

All Classes

Book
Card
CD
Copy
Date
DVD
ISBN
Item
Loan
Member
Name
ItemStatus
MemberStatus
Lib

Overview Class

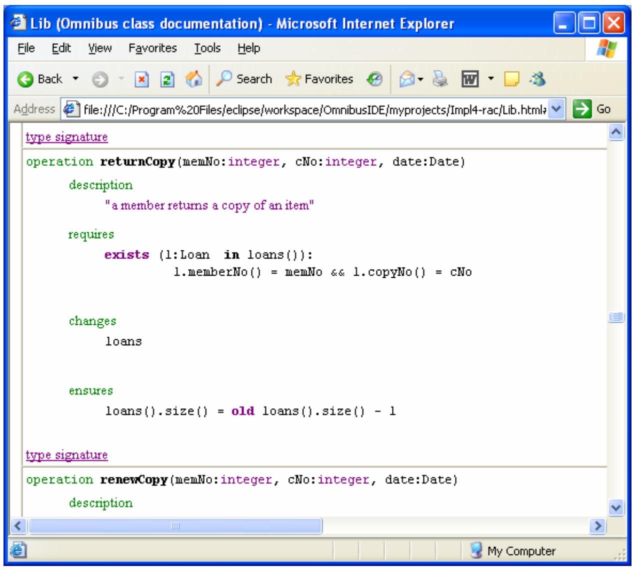
LibraryImpl4.opj Project Documentation

Class Book

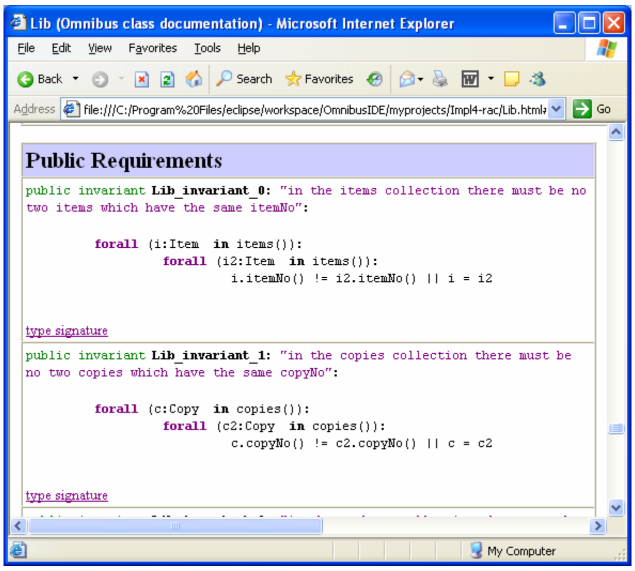
```
omni.lang.Object
  └ Item
     └ Book
```

public class **Book**
isa **Item**

Books are a type of Item. They have a ISBN number, a List of authors, a publisher and a publish Date in addition to the title and itemNo inherited from Item.

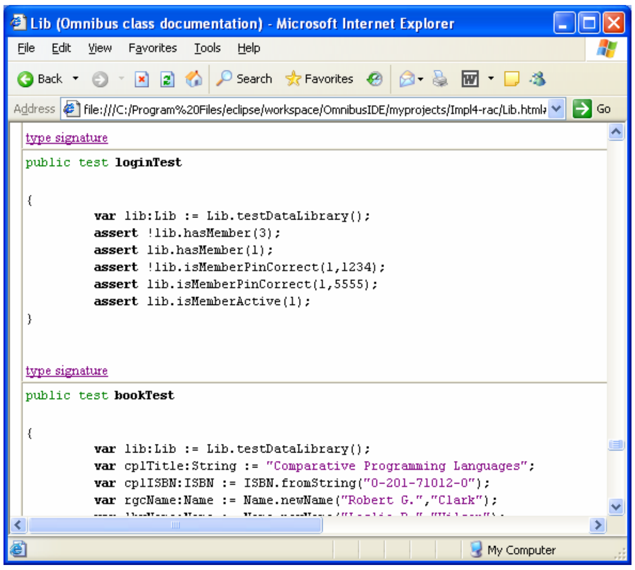

Lib (Omnibus class documentation) - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address file:///C:/Program%20Files/eclipse/workspace/OmnibusIDE/myprojects/Impl4-rac/Lib.html
Go
My Computer

```
type signature
operation returnCopy(memNo:integer, cNo:integer, date:Date)
      description
           "a member returns a copy of an item"

      requires
           exists (l:Loan  in loans()):
                   l.memberNo() = memNo && l.copyNo() = cNo


      changes
           loans


      ensures
           loans().size() = old loans().size() - 1

type signature
operation renewCopy(memNo:integer, cNo:integer, date:Date)
      description
```


Lib (Omnibus class documentation) - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address file:///C:/Program%20Files/eclipse/workspace/OmnibusIDE/myprojects/Impl4-rac/Lib.html
Go
My Computer

Public Requirements

```
public invariant Lib_invariant_0: "in the items collection there must be no
two items which have the same itemNo":

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


type signature
public invariant Lib_invariant_1: "in the copies collection there must be
no two copies which have the same copyNo":

        forall (c:Copy  in copies()):
                forall (c2:Copy  in copies()):
                        c.copyNo() != c2.copyNo() || c = c2


type signature
```


Lib (Omnibus class documentation) - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address file:///C:/Program%20Files/eclipse/workspace/OmnibusIDE/myprojects/Impl4-rac/Lib.html
Go
My Computer

```
type signature
public test loginTest

{
        var lib:Lib := Lib.testDataLibrary();
        assert !lib.hasMember(3);
        assert lib.hasMember(1);
        assert !lib.isMemberPinCorrect(1,1234);
        assert lib.isMemberPinCorrect(1,5555);
        assert lib.isMemberActive(1);
}


type signature
public test bookTest

{
        var lib:Lib := Lib.testDataLibrary();
        var cplTitle:String := "Comparative Programming Languages";
        var cplISBN:ISBN := ISBN.fromString("0-201-71012-0");
        var rgcName:Name := Name.newName("Robert G.","Clark");
```

6.3.2 Verification certificates

The other requirement for supporting safe software component reuse is some sort of basis for trust in the hidden implementation. If the source code of the component is made available to the users of the component then the assertion-based verification processes provide such a basis, but the users should not have to manually re-verify the component themselves to determine the correctness of the component. The Omnibus system uses the concept of *verification certificates* to manage the justification of hidden implementations for open-source components. The Omnibus system does not support closed-source components although there are approaches, which we discuss at the end of the chapter, which can be used to handle these. Each verification certificate provides a summary of the verification that has been applied to the component along with any evidence needed to re-verify the component.

A verification certificate must be user-readable so that users of the component can refer to it and easily find the verification information they need. A summary of the verification details could be incorporated into the standard documentation generator. However, it is essential that a verification certificate can be re-verified in order to guarantee its validity. If the verification information is embedded within the HTML documentation, the HTML could easily be edited to adjust the verification summary to make it more favourable before the component is distributed.

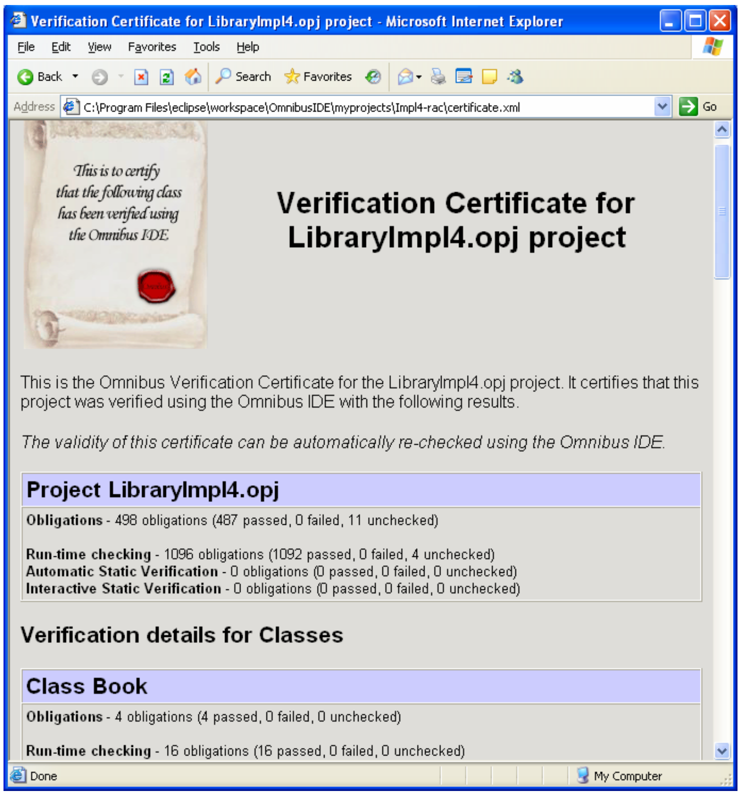
One approach to address this is to support the verification of the HTML documentation (incorporating the verification certificate) by re-verifying the component, re-generating the HTML documentation and then checking the new documentation against the old documentation. The files would have to be textually identical to verify the certificate. However, this approach is very brittle. If, for example, we changed the format of the generated documentation in any way, this process would report that the distributed certificate did not match the re-generated one. We really need the verification certificate to have the summary of the verification details and nothing else. We need to be able to change the layout and add additional explanatory text without breaking the re-verification system.

So there are two requirements for verification certificates: they must be user-readable and they must contain just the raw verification information. These requirements are in apparent conflict but XML can be used to address them both. The raw details of the verification certificate are written to an XML file and then an XSL stylesheet is used to re-format this data into a presentable form.

The format of the XML file is quite straightforward. Each file has a top-level project entry which contains the name of the project, a summary of the verification of the project, and details of each of the classes. Each class entry contains the name of the class and a summary of the verification for the class. The verification details consist of the total number, passed number, failed number and unchecked number of correctness obligations, run-time assertion checks, automated static verification VCs and interactive static verification VCs. An extract of an XML verification certificate is shown below.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="certificate.xsl"?>
<project>
  <name>LibraryImpl4.opj</name>
  <verification>
    <obligations>
     <total>498</total><passed>487</passed>
     <failed>0</failed><unchecked>11</unchecked>
    </obligations>
    <runtime-checks>
     <total>1096</total><passed>1092</passed>
     <failed>0</failed><unchecked>4</unchecked>
    </runtime-checks>
    <automatic-static-verification>
     <total>0</total><passed>0</passed>
     <failed>0</failed><unchecked>0</unchecked>
    </automatic-static-verification>
    <interactive-static-verification>
     <total>0</total><passed>0</passed>
     <failed>0</failed><unchecked>0</unchecked>
    </interactive-static-verification>
  </verification>
  <class>
    <name>Book</name>
    <verification>
     <obligations>
      <total>4</total><passed>4</passed>
      <failed>0</failed><unchecked>0</unchecked>
     </obligations>
     <runtime-checks>
      <total>16</total><passed>16</passed>
      <failed>0</failed><unchecked>0</unchecked>
     </runtime-checks>
     <automatic-static-verification>
      <total>0</total><passed>0</passed>
      <failed>0</failed><unchecked>0</unchecked>
     </automatic-static-verification>
     <interactive-static-verification>
      <total>0</total><passed>0</passed>
      <failed>0</failed><unchecked>0</unchecked>
     </interactive-static-verification>
    </verification>
  </class>
  <class>
    <name>Card</name>
    ...
  </class>
  ...
</project>
```

This file provides just the raw details of the verification process, without additional formatting information. However, it is not very readable. So, it is coupled with an XSL stylesheet which translates the XML into a readable form. When the XML certificate file is opened, the XSL stylesheet is used to format it and the results are as follows:


Verification Certificate for LibraryImpl4.opj project - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address C:\Program Files\eclipse\workspace\OmnibusIDE\myprojects\Impl4-rac\certificate.xml
Go
This is to certify that the following class has been verified using the Omnibus IDE
Done
My Computer

Verification Certificate for LibraryImpl4.opj project

This is the Omnibus Verification Certificate for the LibraryImpl4.opj project. It certifies that this project was verified using the Omnibus IDE with the following results.

The validity of this certificate can be automatically re-checked using the Omnibus IDE.

Project LibraryImpl4.opj

Obligations - 498 obligations (487 passed, 0 failed, 11 unchecked)

Run-time checking - 1096 obligations (1092 passed, 0 failed, 4 unchecked)
Automatic Static Verification - 0 obligations (0 passed, 0 failed, 0 unchecked)
Interactive Static Verification - 0 obligations (0 passed, 0 failed, 0 unchecked)

Verification details for Classes

Class Book

Obligations - 4 obligations (4 passed, 0 failed, 0 unchecked)

Run-time checking - 16 obligations (16 passed, 0 failed, 0 unchecked)

This is a more user-readable form. While being user-readable, the verification certificate itself in uncluttered by presentation details and so can be easily re-generated and checked against the distributed file. The XML file will be the same even if the IDE versions used to originally generate and now re-verify the certificate are different. The XSL presentation of the certificate may be improved between versions of the IDE, but that does not affect the re-verification process.

Certification Levels

It is important to provide some kind of assistance for the interpretation of the verification information in the verification certificate. The users need to know what the different numbers mean and what they should be looking for.

To do this, we could use a Certification Level system. Such a system could have 4 levels: 0 to 3. The appropriate certification level would be determined from the status of the correctness obligations and the soundness compromises made. Level 0 would be awarded if there were any failed correctness obligations. Level 1 would be awarded if there were no failures but some correctness obligations were unchecked. Level 2 would be awarded if all correctness obligations were passed but this either involved run-time checking which is incomplete because of its use of testing or unsound static verification techniques like loop unravelling or assumption constructs. Level 3 would be awarded if all correctness obligations were justified using static verification without use of unsound facilities.

Using this approach, ESC could be used to perform verification to level 3 as long as loop invariants were provided for any loops, and assumption constructs were either not used or separately verified with FFV.

6.3.3 Distribution and use of components

It is also important to provide a convenient mechanism for distributing components. The Omnibus IDE does this, again following the lead of Java, through *jar files*. Users can select an option in the IDE to build a *project jar file*. These special jar files contain the original Omnibus source files, executable bytecode versions of the source files, any PVS proof files (so that PVS proofs can be re-run), documentation files for the project, the verification certificate, and other project details including the verification policies used.

It is easy for other developers to use these jar files: they simply add them as references to their projects and they will be automatically extracted and read by the IDE during the verification of their project. The IDE provides options to display the documentation and certificate for a jar file. When these are selected, the jar file is automatically extracted and the documentation or certificate opened within the IDE. This allows these elements to be accessed by users without requiring them to extract the jar file themselves.

Support for the distribution of reusable software components through component repositories is discussed in section 9.3.4.

6.3.4 Re-verification of components

Omnibus project jar files contain the original Omnibus source files along with a range of derived elements such as executable bytecode versions, documentation files and the verification certificate. Of course, the derived elements could be re-generated from the Omnibus source, but this would require further use of the IDE which may not be convenient. It is more convenient to include copies of these derived elements. However, this introduces the potential for these derived elements to be maliciously or accidentally edited, making them inconsistent with the original Omnibus source and other derived elements. We may wish to re-verify the project jar file to ensure that the derived elements are consistent. We first consider how verification certificates can be checked and then consider the other derived elements.

Checking verification certificates

Users of a component can see the verification details of a component from its verification certificate. This certificate is no more than an XML file containing a summary of the verification details. As such, it would be possible for malicious users to generate a project jar file, extract it, edit the certificate XML file, and repackage it as a jar file. Such a process could be used to present the verification of the component as more favourable than it is in reality. If verification certificates are to be trustable, it is important to be able to combat this.

The Omnibus IDE provides facilities to check the verification certificate of a jar file. The user simply selects the Verify Certificate for Jar File option and specifies the jar file. The tool then opens the project contained in the jar file, re-runs the verification process and checks the verification certificate this produces against the one included in the jar file. This process is carried out automatically by re-running the run-time tests and automated prover and checking the distributed interactive proofs. For the certificate to be valid, it must be identical to the one which is reproduced.

This process will catch two kinds of malicious alterations. First, if the producer of a component manually adjusts the XML certificate, this will be detected because the newly generated certificate will not match the manually edited one. Second, if the producer of a component manually adjusts the original Omnibus source files for the project and this alters the correctness obligations then the newly generated verification certificate will be different and an error will again be reported.

This approach requires the presence of the original Omnibus source files in the jar file. As such, it could be termed an o*pen source reuse approach*. We could adjust the approach to operate without the presence of the original source by password encrypting the jar files. This could allow us to ensure that only the IDE can read and write these jar files. Then we would know that all verification certificates in one of these jar files would have to have been generated by the IDE and so must be consistent with the project. The problem with this approach is that encryption processes are always vulnerable to being deciphered, and forged assurances of correctness may be at least as damaging as the original system without any assurances of correctness.

Checking other derived elements

The verification certificate checking process checks that the derived verification certificate is consistent with the original Omnibus source. It would be desirable to have similar checks for the other derived elements in the jar file: the bytecode implementations and the HTML interface documentation.

It is not as straightforward to check these elements. We could attempt to use the same approach as the verification certificates: to re-generate them and compare with the derived versions, requiring the distributed versions to be identical to re-generated versions. However, unlike the XML verification certificates which have a simple, fixed format, the bytecode implementations and HTML documentation are complicated files with changing formats. For example, we may want to add an optimisation to the bytecode generator in the IDE or improve the layout of the HTML interface documentation. The problem is that the bytecode and HTML generated for a project with versions of the IDE before and after such changes would not be identical and so, using this re-generation approach, a failure would be reported.

One way the problem could be addressed is by recording the versions of the bytecode generator and HTML interface generator used to generate a project jar file and having the re-verifier download these same versions and use them to re-generate the files. The identical file comparison could then be used. This process could be automated by separating and archiving these generator modules and incorporating a facility to automatically download the appropriate versions from some central repository site into the IDE.

6.4 Specialised support for closed source components

The previous section presented techniques which can be used to manage the safe reuse of components which are distributed with their source code. However, the assertion-based verification techniques within the Omnibus system are still relevant for components for which the source code is not (and perhaps cannot be) supplied. There is still the need to clearly describe what the component should do and to provide this in an accessible form. The assertion-based interface specifications and interface documentation generator are still sufficient for this. However, the verification certification system is built around the source code verification techniques of the Omnibus tool and the executables cannot be re-verified via this certificate without the presence of the source code. A fundamentally different approach is needed to address this.

Proof-Carrying Code (PCC) [79, 80] is an approach which can be used to certify that untrusted code (such as components from unknown third-party component vendors) satisfies specific safety rules. Using the approach, a safety proof is produced by the code producer and distributed with the executable. This provides justification that the executable satisfies a set of safety rules. The code consumer is then provided with a proof validator tool which they can use to easily check that the proof provides adequate justification of the safety of the code.

PCC has much in common with the Omnibus approach described in the previous section. PCC, like the Omnibus approach, is based upon verification rather than the concepts of trust and encryption which underpin the digital signatures approach. PCC places the burden of producing the safety proof on the code producer just like the Omnibus approach places the burden of producing the verification certificate on the component vendor.

The crucial difference between the approaches is that PCC safety proofs provide justification that the executable satisfies the safety property while Omnibus verification certificates provide justification that the source code satisfies its specification. By providing justification of the executable, the source code is not needed by the code consumer in order to re-check the code. However, the ideas of PCC have not been used widely in practice yet and one of the biggest outstanding challenges is how the safety proofs are produced. This problem is of a similar order of difficulty to the verification process itself and suffers from many of the same challenges. Indeed, PCC systems are generally built upon existing formal verification techniques.

Recent work in the MOBIUS project [12] aims to support the PCC approach for a variant of Java for the embedded industry. They provide two certification approaches: logic-based verification and type-based verification. Their logic-based verification approach is similar to the Omnibus approach except extended to support PCC. As in the Omnibus approach, the code producer produces an implementation and performs formal verification to demonstrate that the code satisfies its specification and any safety rules (expressed using JML in the MOBIUS project). A special proof transforming compiler would then be used to produce the executables, in place of the standard compiler. This tool takes the source code and source level justification and produces a bytecode program and a safety proof for the bytecode program corresponding to the source level verification. A new annotation language for Java bytecode is being developed in order to support this called the Bytecode Modelling Language

(BML) [21]. The supporting tools are still in development but the approach seems to hold great promise.

Chapter 7

Tool support

The Omnibus IDE provides tool support for the techniques presented in this thesis. It consists of a number of components including a type checker, documentation generator, bytecode generator with support for RAC checks, and an integrated static verifier supporting ESC and FFV. The components of most technical interest are the bytecode generator and the integrated static verifier. The next section discusses the RAC facilities provided via the bytecode generator. The remainder of the chapter is devoted to a discussion of the integrated static verifier.

The implementation of the Omnibus IDE was written using the Java programming language and verified using testing. It was based upon an earlier system called JOOSE (Java-based Object-Oriented Symbolic Executor). We were unable to use verification tools such as JML's run-time assertion checker or ESC/Java2 because our implementation required the use of version 1.5 of the Java SDK which those tools were not compatible with at the time. This was not very desirable but we had to work with the tools that were available when the project begun.

7.1 Run-time assertion checking

This section discusses the run-time assertion checking support within the Omnibus IDE. In order to be executed, Omnibus files are translated to bytecode via an intermediary Java form. Run-time assertion checks are embedded within the generated Java, which is then compiled to bytecode using the standard Java compiler.

Translation of general file structure

The translation of the Omnibus file structure to the Java file structure is relatively straightforward. The package, import and class structures are purposely very similar.

Special Java methods

A range of special methods is generated for each class when it is translated to Java, not defined explicitly in the corresponding Omnibus class.

Each class has an `equals` method which provides an implementation for equality. The default is an automatically generated implementation of structural equality which checks the other object is an instance of the same class and then compares the corresponding attribute values if it is.

A `toString` method is also automatically defined. This returns a `String` containing the full name of the class and the attribute values of the object. This method is used by the run-time assertion checker to report variable values.

Finally, special quantifier functions are defined for each quantifier used in an assertion within the class. To be executable, each quantified variable in the quantifier must be restricted with either a range or `in` clause. If a range clause is used, a `for` loop is used to iterate over the possible values for the quantified variable. If an `in` clause is used, an `omni.lang.Iterator` is retrieved from the collection specified and used to iterate over the values in the collection.

Translation of Omnibus class members

Omnibus attributes map directly to Java fields. All Omnibus attributes are encoded in Java with the `protected` modifier so as to prevent direct modification from client classes, but accessor functions are automatically introduced so the values of the attributes can still be read.

Omnibus functions map to Java methods returning a value of the return type specified in the Omnibus declaration. The type checking phase ensures that the class attributes cannot be changed by a function, so there can be no side-effect on the value of the `this` object.

Omnibus constructors map to Java static methods returning a new object instance of the current class. Omnibus operations map to Java instance methods returning a new object instance of the current class.

Omnibus tests are translated to special Java static methods which cannot be directly called from normal Omnibus code in method implementations. Instead, these methods are called by the IDE to perform testing of the class. A special `test_class` static method is also defined which runs all the tests in the same class. Within the implementation for each test is code to record timing info and catch assertion failures generated during the test.

Translation of method bodies

The body of each Omnibus method when translated to Java has the same basic form. First, for non-static methods, local variables are declared for each of the attributes and initialised to the corresponding attribute values. Within the remainder of the code, it is these local variables that are manipulated, the attribute values of the object on which the method was called remaining unaltered so as to preserve value semantics. Where local method calls are made via an implicit or explicit `this` object, a new object instance of the class is constructed with the current attribute variable values.

Next, checks are written for the pre-condition of the method and any requirements that should hold at the start of the method (e.g. the invariants should hold at the start of each operation). After this, the starting of the method is recorded. This involves the opening of a new frame in the call stack. The Omnibus run-time system maintains its own call stack, independent of the one used by Java, so that line numbers can be reported relative to the Omnibus source, not the generated Java.

The code which is provided as the implementation of the method in the Omnibus source file is then translated to Java. This is discussed below.

If the method is a constructor or an operation, the end of the method is recorded, checks are generated for the `changes` and `ensures` clauses and appropriate requirements, and a new instance of the current class is constructed from the local attribute variables and returned.

Translation of statements

Each statement in the Omnibus assertion language is mapped to a series of Java statements. The Omnibus implementation language is purposely similar to the Java implementation language and so this mapping is quite straightforward. Special additional support that is added includes tracking of the current line numbers in the Omnibus call stack and the appropriate generation of assertion checks.

Generation of assertion checks

Assertion checks are generated for the specifications of methods and assertions within the implementations (e.g. `assert` statements, loop invariants). All the checks have the same form. First, the policy for the obligation is looked up and the tool ensures that we are meant to use run-time assertion checking for the obligation. A check for the assertion is then generated. If it fails, an error message will be reported along with the level of context information specified in the corresponding verification policy. A report of the check will be recorded in the run-time check report file containing details of the source of the assertion and whether it passed or failed. This report file is read by the IDE in order to give details of the assertions that were checked and to identify those that were not checked.

7.2 Integrated static verifier

Previously, support for extended static checking and full formal verification have been provided by separate tools produced by separate teams. There have been relatively few “extended static checkers”, and they have been implemented separately from FFV tools.

However, ESC and FFV are very similar processes. Both are static verification processes which involve the production of theories in the logic of a specific prover. These must then be proved either automatically or interactively to successfully complete the verification. These theories contain a formalisation of core semantic properties of the language, definitions and axioms describing the specifications of the classes in the project, and VCs over these specifications. The differences between the approaches are largely concerned with the handling of particular language features (such as assumption constructs and loops) and the theorem provers used to process the VCs.

Conventional verification tools are closely linked to a specific theorem prover and generated theories in the logic of that prover. However, separate theorem provers can be supported if the static verification module instead produces theories in an intermediary *generic logic* which can then be translated into the logic of a specific prover.

We can use the concept of a generic logic together with a configurable static verifier to support both ESC and FFV within one tool. This is what our integrated static verifier does, and this chapter describes how it is implemented. We discuss the implementation in three parts: the translation into generic logic, the mapping of this generic logic into the logics of two specific provers, and prover-specific interfacing.

7.3 Translation to generic logic

The first step in the Omnibus static verification process is to produce theories in the generic logic. These theories contain axiomatic descriptions of the specifications of the classes in the project and the VCs which must be proved.

7.3.1 Base level for generic logic specifications

When attempting to use a generic logic, we have a similar problem to that encountered in the definition of specifications within the language: what is the base level for the specifications? In this case our specific challenge is: what mathematical apparatus do we use to define the specification of the classes within the generic logic?

Firstly, we must select a core logic and for this we use first-order predicate calculus with equality and function symbols. This choice is heavily influenced by the need to support effective automated verification to enable extended static checking. Such a logic is expressive enough to model interesting properties but not so expressive that it hampers the automation level of the verification. This design choice is compatible with our two target theorem provers: Simplify [36] and PVS [81].

Secondly, it is useful to build some of the core concepts of the language into the generic logic itself. The two main built-in functions are `is` and `valid`. The `is` function is equivalent to the `instanceof` expression in Java. It accepts an object and a class and yields true iff the specified class is a static class of the object. The `valid` function accepts an object and a class, and yields true iff the object also satisfies the invariants of the specified class. These functions are not defined within the generic logic itself and are instead defined separately in the specific logics of the underlying provers as we will see later. In this way, they can be thought of as being built into the generic logic, but they must then be defined for each of the underlying provers. The definition of core concepts in this way is sometimes referred to as the *background predicate* [32].

7.3.2 Structure of theories

The theories in the generic logic simply consist of a series of declarations, axioms and VCs. A declaration is used to introduce a *function symbol* which can be used to model concepts from the Omnibus language using functions in logic. Axioms are used in the generic logic to describe properties of the introduced functions. Finally, VCs over the defined functions are expressed. A range of context information is stored along with the actual formula to check in order to aid error reporting. For example, the following information is stored for each VC:

1. The source of the VC: parent file, parent method, line number
2. The corresponding error message
3. If the VC is from an implementation: the path condition, the path information, local variable and parameter values

7.3.3 Translation of classes into generic logic

The first phase of the translation of a class into generic logic is a formalisation of the type hierarchies. As described in section 2.2.3, a *class level* is introduced for the public and private accessibility levels of each class and the protected level if it has any members. The public class level of a class inherits from the protected class level of its superclass, or the public level if the superclass has no protected class level. The protected and private class levels of a class inherit from the level immediately above them in the class.

For each class level, axioms are generated describing the built-in `is` and `valid` functions for the specific class level. The first axiom describes that if an object is an instance of the current class level that it is also an instance of the parent class level. The second axiom describes the validity for the current class level in terms of the invariants for that level. Special functions are introduced for each of the requirements at the current class level, along with axioms describing their truth in terms of the corresponding assertions with which they are declared.

If the current class is not declared with the `spec` or `native` modifier, axioms are also generated for a special `build` constructor. The `build` constructor is a special function used internally by the static verifier to create an instance of the specified class from a set of attribute values. It accepts as its parameters the values for all the template parameters of the class and the values for all the attributes of the class. An axiom is used to define the special build constructor for the current class level. Further axioms are used to describe attribute and template parameter values for objects constructed using the build constructor.

For example, suppose we have an Omnibus class `Shape`:

```
public class Shape {}
```

The following declarations and axioms would be generated. The syntax used is that of our generic logic. Declarations consist of the `declare` keyword followed by the name of the function, the formal parameters of the function and the return type. Axioms consist of the `axiom` keyword followed by the name of the axiom and then a boolean expression.

```
axiom Shape_is_ax: forall (this:Shape):
                     is(this, omni.lang.Object)

axiom priv_Shape_is_ax: forall (this:priv_Shape):
                          is(this, Shape)

axiom Shape_valid_ax: forall (this:Shape):
                          valid(this, Shape) = true

axiom priv_Shape_valid_ax: forall (this:priv_Shape):
                              valid(this, priv_Shape) = true

declare Shape_build():Shape

axiom Shape_build_ax: is(Shape_build(), priv_Shape)
```

For an Omnibus class `Set` with a template parameter:

```
public class Set[Element] {}
```

The following axioms would be generated:

```
axiom Set_is_ax: forall (this:Set):
                   is(this,omni.lang.Object)

axiom priv_Set_is_ax: forall (this:priv_Set):
                        is(this,Set)

axiom Set_valid_ax: forall (this:Set):
                     valid(this,Set) = true

axiom priv_Set_valid_ax: forall (this:priv_Set):
                            valid(this,priv_Set) = true

declare Set_build(ElementClass:Class):priv_Set
```

```
axiom Set_build_ax: forall (ElementClass:Class):
                       is(Set_build(ElementClass),priv_Set)

declare Set_ElementClass(ElementClass:Class):Class

axiom Set_ElementClass_ax: forall (ElementClass:Class):
             Set_ElementClass(Set_build(ElementClass))
             = ElementClass
```

7.3.4 Translation of methods into generic logic

The inheritance system for methods is built around *method levels* just as the system for classes is built around *class levels*. Separate method levels are defined for each class and accessibility level for which a specification is given, as described in section 2.2. A particular method level inherits from the method level immediately above it (if there is one). This will be either at a higher accessibility level in the current class or at the lowest accessibility level above private in the corresponding definition in a superclass.

Four special functions are introduced in the generic logic for each method level: the so-called *pre-condition function*, *post-condition function*, *guard function* and *method function*.

The pre- and post-condition functions provide convenient ways to refer to the pre- and post-conditions of the method, respectively. The pre-condition function is a predicate over the input parameters of the method. Its axiom describes that the predicate is true when the specified pre-condition of the method is met. The post-condition function is a predicate over the input parameters and result of the method. Its axiom describes that the predicate is true when the specified post-condition of the method is met.

Unlike the pre- and post-condition functions, the guard and method functions are defined only once for the root of the method level hierarchy, but axioms are provided for all the method levels. The guard function is used to express the pre-condition of a method in a manner compatible with dynamic binding. Because of dynamic binding, it is not possible to define precisely what the pre-condition of a method is for this purpose since the same method may have different pre-conditions at different method levels. So, we introduce a special guard function along with axioms describing when that guard is true for each specific level only. This is achieved using an implication rather than an equality in the axiom. If we had used equality, there would have been a possible contradiction if the pre-condition was weakened since for the additional values allowed by the subclass, the superclass would claim the pre-condition was false whereas the subclass would claim it was true.

The method function is the most important of the functions for a method. It models to the original method within the logic. It accepts the input parameters of the method and produces the appropriate result. It is only defined for the root of the method level hierarchy to permit dynamic binding. The axioms describing the method at each method level are formed by substituting the result variable (either `result` or `this`) in the post-condition, with an instance of the method applied to the input parameters.

For example, suppose we have the following `Shape` class with an `area` method:

```
public abstract class Shape {
  public abstract function area():integer
    ensures result >= 0
}
```

The method does not define a pre-condition and so no pre-condition or guard functions are generated. The method does not override a method in another class, so a new function symbol is defined for the method.

```
declare Shape_area_pub_post(this:Shape,result:integer):bool

axiom Shape_area_pub_post_ax: forall (this:Shape,
                                      result:integer):
        Shape_area_pub_post(this,result) = result >= 0

declare Shape_area(this:Shape):integer

axiom Shape_area_pub_ax: forall (this:Shape):
                            Shape_area(this) >= 0
```

Now consider a `Rectangle` class which extends the `Shape` class and overrides the `area` function.

```
public class Rectangle isa Shape {
  private attribute width:integer
  private attribute height:integer

  public model function width():integer
    private ensures result = width
  {  ...  }
  public model function height():integer
    private ensures result = height
  { ... }

  public constructor ofSize(w:integer, h:integer)
    requires w >= 0 && h >= 0
    ensures width() = w, height() = h
  { ... }

  public function area():integer
    ensures result = width() * height()
  { ... }
}
```

The `Rectangle` class has public `width` and `height` model functions which are described at the private level in terms of private `width` and `height` attributes. The public `ofSize` constructor can be used to create instances of the `Rectangle` and the public `area` function overrides the definition in class `Shape`.

Firstly, functions are defined for the private attributes:

```
declare Rectangle_width_att(this:Rectangle):integer

declare Rectangle_height_att(this:Rectangle):integer
```

Now consider the `width` function. It has public and private behaviour specifications. At the public level it is declared to be a model function while at the private level its behaviour is described in terms of the `width` attribute. This leads to the following axioms:

```
declare Rectangle_width(this:Rectangle):integer

declare Rectangle_width_priv_post(this:Rectangle,
                                  result:integer):bool

axiom Rectangle_width_priv_post_ax:
          forall (this:Rectangle,result:integer):bool
              Rectangle_width_priv_post(this,result) =
                          (result = Rectangle_width_att(this))

declare Rectangle_width(this:Rectangle):integer

axiom Rectangle_width_priv_ax: forall (this:Rectangle):
            Rectangle_width(this) = Rectangle_width_att(this)
```

The `height` function is defined similarly.

The `ofSize` constructor has a pre-condition, so pre-condition and guard functions are defined.

```
declare Rectangle_ofSize_pub_pre(w:integer,h:integer):bool

axiom Rectangle_ofSize_pub_pre_ax:
          forall (w:integer,h:integer):
              w >= 0 && h >= 0

declare Rectangle_ofSize_guard(w:integer,h:integer):bool

axiom Rectangle_ofSize_pub_guard_ax:
          forall (w:integer,h:integer):
            w >= 0 && h >= 0 ==> Rectangle_ofSize_guard(w,h)

declare Rectangle_ofSize_pub_post(w:integer,h:integer,
                                  this:Rectangle):bool

axiom Rectangle_ofSize_pub_post_ax:
          forall (w:integer,h:integer,this:Rectangle):
            Rectangle_ofSize_guard(w,h) ==>
                Rectangle_ofSize_pub_post(w,h,this) =
                    (Rectangle_width(this) = w
                        && Rectangle_height(this) = h)

declare Rectangle_ofSize(w:integer,h:integer):Rectangle

axiom Rectangle_ofSize_pub_ax:
          forall (w:integer,h:integer):
            Rectangle_ofSize_guard(w,h) ==>
                Rectangle_width(Rectangle_ofSize(w,h)) = w
              && Rectangle_height(Rectangle_ofSize(w,h)) = h
```

The definition of the `area` function in the `Rectangle` class overrides the definition in the `Shape` class. For this reason, a new function symbol is not introduced for the method and, instead, the method axiom refers to the root function definition `Shape_area`.

```
declare Rectangle_area_pub_post(this:Rectangle,
                                result:integer):bool

axiom Rectangle_area_pub_post_ax:
          forall (this:Rectangle,result:integer):
                Rectangle_area_pub_post(this,result) =
                      result = Rectangle_width(this)
                                * Rectangle_height(this)

axiom Rectangle_area_pub_ax: forall (this:Rectangle):
            Shape_area(this) = Rectangle_width(this)
                                * Rectangle_height(this)
```

7.3.5 Generation of VCs

Once the specifications of the classes in the project have been translated into logic, the verification process can begin. Verification is performed via the generation of a range of VCs which must then be discharged using a theorem prover. The VCs are also expressed in the generic logic so that they can be translated into the logic of the appropriate theorem prover.

There are two types of VCs: specification consistency VCs and implementation VCs.

Specification consistency VCs

In Omnibus, requirements specifications can be verified in one of two ways. They can either be proved to follow from the behaviour specifications of the methods in the class or from the implementations. For FFV, the requirements should follow from the behaviour specifications. This provides a powerful mechanism for verifying the correctness of the behaviour specifications. The laws described in section 2.3.2 can be used to check this.

The previous sections have described the process whereby the behaviour and requirements specifications are translated into the generic logic. Specifically, we have a special function for each requirement and special functions for the pre- and post-conditions of each method. We can make direct use of these functions in the definition of our specification consistency VCs.

To illustrate this, consider the following `PosCounter` specification.

```
public spec class PosCounter {
  model function value():integer

  initially value() = 0
  invariant value() >= 0
  constraint value() != old value()

  constructor zero()
    ensures value() = 0

  operation inc()
    changes value
    ensures value() = old value() + 1

  operation dec()
    requires value() > 0
    changes value
    ensures value() = old value() - 1
}
```

This simple class has an abstract state consisting of a single integer value. The initially, invariant and constraint requirement specifications state that the value should be initialised to `0` by all constructors, must always be greater than or equal to `0` and should be changed by all operations, respectively. There is a single constructor and operations to increment and decrement the value. The corresponding specification consistency VCs are as follows:

```
vc vc_1: forall (this:PosCounter):
          PosCounter_zero_pub_post(this)
          ==> PosCounter_invariant_0(this)

vc vc_2: forall (this:PosCounter):
          PosCounter_zero_pub_post(this)
          ==> PosCounter_initially_0(this)

vc vc_3: forall (old_this:v_PosCounter, this:PosCounter):
            PosCounter_inc_pub_post(old_this,this)
            ==> PosCounter_invariant_0(this)

vc vc_4: forall (old_this:v_PosCounter, this:PosCounter):
            PosCounter_inc_pub_post(old_this,this)
            ==> PosCounter_constraint_0(old_this,this)

vc vc_5: forall (old_this:v_PosCounter, this:PosCounter):
            PosCounter_dec_pub_pre(old_this)
              && PosCounter_dec_pub_post(old_this,this)
            ==> PosCounter_invariant_0(this)

vc vc_6: forall (old_this:v_PosCounter, this:PosCounter):
            PosCounter_dec_pub_pre(old_this)
              && PosCounter_dec_pub_post(old_this,this)
            ==> PosCounter_constraint_0(old_this,this)
```

VCs 1 and 2 check that the behaviour specification of the `zero` constructor satisfies the invariant and initially. VCs 3 and 4 check that the `inc` operation satisfies the invariant and constraint. VCs 5 and 6 check that the `dec` operation satisfies the invariant and constraint.

Verification Conditions are also generated to ensure that the use of inheritance is consistent with the principle of substitutability. To illustrate the generation of generic logic VCs for this, consider the following example:

```
spec class CurrentAccount {
  model function balance():integer

  constructor open()
    ensures balance() = 0

  operation withdraw(amount:integer)
    requires amount >= 0
    changes balance
    ensures balance() = old balance() - amount

  operation deposit(amount:integer)
    requires amount >= 0
    changes balance
    ensures balance() = old balance() + amount
}
```

This class represents a simple bank account with `withdraw` and `deposit` operations. There are no restrictions on the `balance`: users of the class may withdraw more than they deposit. Now suppose we define a `SavingsAccount` class as a subclass of this class.

```
spec class SavingsAccount isa CurrentAccount {
  invariant balance() >= 0

  constructor open()
    ensures balance() = 0

  operation withdraw(amount:integer)
    requires amount >= 0, balance() >= amount
     // error: strengthens pre-condition
    changes balance
    ensures balance() = old balance() - amount
}
```

The corresponding generic logic VCs for the `SavingsAccount` class are as follows:

```
vc vc_1: forall (this:SavingsAccount):
             SavingsAccount_open_pub_post(this)
             ==> SavingsAccount_invariant_0(this)

vc vc_2: forall (old_this:v_SavingsAccount, amount:integer,
                   this:SavingsAccount):
             CurrentAccount_withdraw_pub_pre(old_this,amount)
               && SavingsAccount_withdraw_pub_post(old_this,
                                                    amount,this)
             ==> SavingsAccount_invariant_0(this)

vc vc_3: forall (this:v_SavingsAccount, amount:integer):
             CurrentAccount_withdraw_pub_pre(this,amount)
             ==> SavingsAccount_withdraw_pub_pre(this,amount)

vc vc_4: forall (old_this:v_SavingsAccount, amount:integer,
                   this:SavingsAccount):
             CurrentAccount_withdraw_pub_pre(old_this,amount)
```

```
              && SavingsAccount_withdraw_pub_post(old_this,
                                                 amount,this)
            ==> CurrentAccount_withdraw_pub_post(old_this,
                                                 amount,this)

vc vc_5: forall (old_this:v_SavingsAccount, amount:integer,
                     this:SavingsAccount):
            SavingsAccount_deposit_pub_pre(old_this,amount)
              && SavingsAccount_deposit_pub_post(old_this,
                                                 amount,this)
            ==> SavingsAccount_invariant_0(this)
```

VC 1 checks that the `open` constructor of the `SavingsAccount` class satisfies the invariant that the `balance` should be non-negative. VC 2 checks that the `withdraw` operation maintains the invariant. Note that the pre-condition from the overridden method is used since that is all the caller can be assured to have established. VCs 3 and 4 check that the specification of the `withdraw` method provided in the `SavingsAccount` class is consistent with the specification it overrides in the `CurrentAccount` class. Finally, VC 5 checks that the `deposit` method inherited from the `CurrentAccount` class satisfies the newly defined invariant. The reader may note that VC 3 is not valid: the pre-condition of the `withdraw` operation as defined in `CurrentAccount` does not imply the pre-condition of the `withdraw` operation as defined in `SavingsAccount`.

Implementation VCs

Implementations are verified using a symbolic execution process [55, 56]. This section includes the details of the symbolic execution rules for the Omnibus language. These provide a description of the semantics of the language. A worked example of the symbolic execution process is given in appendix D.

Symbolic execution is a process for the symbolic analysis of implementations. It is used within the Omnibus project to support static verification. A component which performs the symbolic execution process is called a symbolic executor.

The basic idea behind symbolic execution is to generalise the normal execution rules for a language to support algebraic symbols. Assumptions can be made about the initial validity of these symbols. Then the implementation can be executed with these values, with any additional assumptions recorded as the execution progresses. Any assertions that are met should be provable from the assumptions of the specific path. This approach allows general properties of the correctness of implementations to be verified as it is not tied to any specific concrete test values.

During the standard execution of a method, the system must keep track of the next statement to be executed (the program counter) and the current values for each variable in the program. The symbolic executor is a generalisation of this. In the normal execution of a method, only one path through the implementation is considered for each invocation. In symbolic execution, all paths are considered. If there is a branching point then the different branches are all considered in parallel under their own specific assumptions. The state of the symbolic executor reflects this. Instead of a single mapping of variables to values, there is one for each path. The symbolic executor allows symbolic values (i.e. expressions containing unevaluated symbols) as well as conventional concrete values (i.e. values evaluated to literal values without symbols) to be assigned to variables. Each path also has a Path Condition which is an expression used to keep track of the assumptions that are known to be valid for that specific path. The state information is arranged into a hierarchy of scope levels which can be opened and closed to be consistent with scope level system within the Omnibus language.

Symbols are a crucial part of the symbolic execution approach. A symbol is simply an arbitrary constant value of a specific type whose precise value is not known. They allow the verification of a system to be modularised via the interfaces of the different modules. For example, suppose we wish to verify the correctness of a method with declared pre- and post-conditions. We are interested in its correctness for all values, not just specific test values. We know from the static typing information that the parameters of the method have specific known types but we do not know their specific values. Therefore, we can assign symbol constants of the specified type to these parameters. We may assume that the pre-condition holds over the symbolic values, provided that we separately verify that all calls of this method satisfy it. We can then execute the implementation using the symbolic values, and verify at the end of the implementation that the post-condition is met.

We will often find that we need to verify the correctness of an assertion during the symbolic execution of an implementation. We stated earlier that any encountered assertions should follow from the assumptions that are known to be true for the specific path being considered. This can be checked by ensuring that the specific path condition logically implies the assertion to be proved. The formulae to check this is referred to as a *Verification Condition* (VC).

Symbolic execution primitives

A procedure must be defined for the handling of each construct in the language by the symbolic executor. For each statement, a symbolic execution rule must be defined to describe how the statement is executed. Likewise, procedures must be defined for the verification of each class and method. Finally, the process of evaluating each expression and checking of the pre-conditions within a method must be defined.

These symbolic execution procedures are defined in terms of a set of primitive statements for manipulating the state of the symbolic executor.

Variable manipulation primitives

declare *varName*, *type*	The **declare** primitive declares a new variable of the specified type. The typing information is used in the production of a suitable symbol.
init *varName*	The **init** primitive initialises a new variable with an undefined value.
reassign *varName*	The **reassign** primitive assigns a fresh symbolic value to the specified variable. No information about the variable's value is known other than its validity. This is analogous to **havoc** statements in Boogie.
reassignExcept *varNames*	The **reassignExcept** primitive assigns fresh symbolic values

	to all the variables which can be assigned to, apart from those in the passed list. No information about the new variable values are known other than their validity.

Knowledge manipulation primitives

assume *exp*	The **assume** primitive conjoins an expression to the Path Condition of the current path. The expression is evaluated first.
check *exp*	The **check** primitive generates a VC to check that the specified assertion is satisfied. The generated VC checks that current path condition logically implies the specified assertion. The expression is evaluated first.
clearKnowledge	The **clearKnowledge** primitive clears the Path Condition for the current path. In does this by assigning the Path Condition to the boolean literal `true`.
deny *exp*	The **deny** primitive is similar to the **check** primitive except that the desired truth of the corresponding VC is false, not true. The expression is evaluated first.
localAssume *exp*	The **localAssume** primitive adds an assumption that is included within the Path Condition but it is removed when the current scope level is closed. The expression is evaluated first.

Scope level primitives

openLevel	The **openLevel** primitive opens a new temporary scope level where variables can be added which will be subsequently removed when the scope level is closed.
closeLevel	The **closeLevel** primitive closes the current scope level, removing any local variables.

Branching primitives

branch	The **branch** primitive allows the symbolic execution procedure for a

	construct to be defined in separate cases.
terminate	The **terminate** primitive indicates that termination should halt at the specified point and execution should not continue into subsequent statements.

Expression primitives

value(*exp*)	The **value** function is used to evaluate a particular expression. It yields the evaluated equivalent of the expression. This function is defined recursively for each expression construct as outlined below.
checkPres *exp*	The **checkPres** primitive is used to check that the pre-conditions of all the sub-expressions within a specified expression are met. This is defined recursively for each expression construct as outlined below.

Statement primitives

execute *stmt*	The **execute** primitive is used to symbolically execute a specific statement. This is defined recursively for each statement as outlined below.

Constants

For convenience, we introduce a range of special constants to refer to elements from the class or method definition. The *classAttributes*, *classInvariants* and *classTemplateParameters* constants refer to the attributes, invariants and template parameters of the current class, respectively. The *methodPrecondition* and *methodPostcondition* constants refer to the pre- and post-conditions of the current method, respectively.

Symbolic execution procedures

This section gives the symbolic execution procedures for each language construct in terms of the primitives defined in the previous section.

Classes and class members

package *packPath*; **uses** *usesList*;	**load** ‘top level package variables’ **load** ‘imported + local class variables’

class *name*[*tempParams*] **isa** *suprCls* {	**load** ‘local + inherited methods’
members	**declare** “this”, *name*
}	**declare** *tempParams*
	declare *classAttributes*
	process *members*
function *name*(*params*):*type*	**openLevel**
behaviour	**clearKnowledge**
{	**declare** *params*
code;	**reassign** *classTemplateParameters*
}	**reassign** *classAttributes*
	reassign *params*
	assume *methodPrecondition*
	assume *classInvariants*
	execute *code*
	closeLevel
constructor *name*(*params*)	**openLevel**
behaviour	**clearKnowledge**
{	**declare** *params*
code;	**reassign** *classTemplateParameters*
}	**init** *classAttributes*
	reassign *params*
	assume *methodPrecondition*
	execute *code*
	closeLevel
operation *name*(*params*)	**openLevel**
behaviour	**clearKnowledge**
{	**declare** *params*

code; }	**reassign** *classTemplateParameters* **reassign** *classAttributes* **reassign** *params* **assume** *methodPrecondition* **assume** *classInvariants* **execute** *code* **check** *methodPostCondition* **closeLevel**

test *name* {
 	
code;
 	
}
	openLevel
	
	clearKnowledge
	
	execute *code*
	
	closeLevel

Statements

assert *exp*;	**checkPres** *exp* **check** *exp*
target := *exp*;	**checkPres** *exp* **assign** *target*, *exp*
assume *exp*;	**checkPres** *exp* **assume** *exp*
var *varName*:*type*;	**checkPres** *type* **declare** *varName*, *type* **init** *varName*
var *varName*:*type* := *exp*;	**checkPres** *type* **checkPres** *exp* **declare** *varName*, *type* **assign** *varName*, *exp*
deny *exp*;	**checkPres** *exp*

	deny *exp*	

if (*cond*) {
    trueCode;
}
	checkPres *cond*
	branch 1
	assume *cond*
	execute *trueCode*
		
		branch 2
		assume !*cond*

if (*cond*) {
    trueCode;
} **else** {
    elseCode;
}
	checkPres *cond*
	branch 1
	assume *cond*
	execute *trueCode*
		
		branch 2
		assume !*cond*
		execute *elseCode*

To symbolically execute an if statement, the system splits the statement up into branches for each of the paths through the statement. Firstly, it considers the case where the condition of the if is true. In this case, it assumes the condition of the if, executes the statements in the if and then continues execution after the if. Then, for each of the elseif clauses, it assumes the condition of the elseif along with the negation of the preceding conditions, executes the statements in the elseif and then continues execution after the if. Finally, it assumes the negation of the if and elseif conditions, if an else clause is provided, then it executes the statements in it and continues execution after the if.		

while (*cond*)
    alters *alts*
    maintains *maint*
{
    code;
}
	checkPres *maint*
	check *maint*
	reassignExcept *alts*
	assume *maint*
	checkPres *cond*
	branch 1
	assume !*cond*
		
		branch 2
		assume *cond*
		execute *code*
		check *maint*

	terminate
There are two techniques for symbolically executing loops: invariant-based proofs and loop unravelling. A loop invariant can be provided via `alters` and `maintains` clauses. If a loop invariant is provided, symbolic execution is used to prove that the invariant holds initially and that it is maintained by an arbitrary execution of the loop body. This involves the following steps: 1. Checking the loop invariant holds initially: a. The maintains expression is checked, ensuring that the loop invariant holds initially. 2. Checking the loop invariant is maintained by an arbitrary iteration of the loop: a. All variables in the alters clause are assigned fresh symbolic values. b. The knowledge is adjusted to hold only the `maintains` clause and the loop condition. c. The body of the while loop is executed. d. The `maintains` clause is checked and 3. Checking the remainder of the method: a. All variables in the `alters` clause are assigned fresh symbolic values. b. The knowledge is adjusted to hold only the `maintains` clause and the negation of the loop condition.	

while (*cond*) {
code;
}
	checkPres *cond*
	branch 1 branch 2
	assume !*cond* **assume** *cond*
	execute *code*

	assume !*cond*
Verification using loop invariants is the only sound way of proving code involving loops. However, it can be extremely difficult to devise suitable loop invariants. As such, the developer may prefer to use the loop unravelling technique which checks some finite number of iterations of the loop. Loop unravelling will be used if no loop invariant is specified and loop unravelling is enabled. Unravelling the loop 0 times involves assuming the negation of the loop condition and then executing the remainder of the method. Unravelling the loop once involves assuming the loop condition, executing the body of the loop once, assuming the negation of the loop condition and then executing the remainder of the method. And so on.	
return *exp*;	**checkPres** *exp* **declare** "result", *methodReturnType* **assign** "result", *exp* **check** *methodPostcondition* **terminate**
The return statement supports abrupt termination, i.e. all paths executing a return statement terminate with that statement. The `ensures` and `changes` clauses should also be checked along with requirements if the developer selected to check requirements hold over the implementations. Before these checks are made, the value to be returned is assigned to the special `result` variable.	
unreachable;	**check** false **terminate**
Like the return statement, the unreachable statement also terminates execution paths. When it is encountered during symbolic execution, the system attempts to prove the truth of the assertion `false`. The only way that this can be proved is if there is a contradiction in the current path condition signifying that there is a logical flaw in the assumptions taken in selecting the current execution path.	

Expression evaluation and pre-condition checking

Standard binary operators

The following rules apply for the standard binary operators: addition (‘+’), subtraction (‘-’), multiplication (‘*’), equality (‘=’, ‘!=’), inequality (‘<’, ‘<=’, ‘>’, ‘>=’), static typing (‘is’, ‘as’) and logical equivalence (‘<==>’).

left ***op*** *right*	**checkPres** *left* **checkPres** *right*	value(*left*) ***op*** value(*right*)

Division operators

The following rules apply for the division (‘/’) and modulus (‘%’) operators.

left ***op*** *right*	**checkPres** *left* **checkPres** *right* **check** *right* != 0	value(*left*) ***op*** value(*right*)

Short-circuited logical operators

The following rules apply for the disjunction (‘||’) operator.

left ***op*** *right*	**checkPres** *left* **openLevel** **localAssume** ! *left* **checkPres** *right* **closeLevel**	value(*left*) ***op*** value(*right*)

The following rules apply for the conjunction (‘&&’) and implication (‘==>’) operators.

left ***op*** *right*	**checkPres** *left* **openLevel** **localAssume** *left* **checkPres** *right* **closeLevel**	value(*left*) ***op*** value(*right*)

The following rules apply for the **if..then..else..** operator.

if *cond* **then** *trueExp*	**checkPres** *cond*	**if** value(*cond*)

else *falseExp*	**openLevel** **localAssume** *cond* **checkPres** *trueExp* **closeLevel** **openLevel** **localAssume** ! *cond* **checkPres** *falseExp* **closeLevel**	**then** value(*trueExp*) **else** value(*falseExp*)

Method calls

The following rules apply for method calls in expressions.

object.*methodName*(*params*)	**checkPres** *object* **check** *object*.*methodName*_guard(*params*)

value(*object*).*methodName*(values(*params*))

Variables

name		Returns corresponding variable value (may be a package, class, template parameter type, “this” or a normal variable value)

7.4 Mapping generic logic to specific provers

The verification process starts with the generation of the generic logic definitions, axioms and VCs. Once this has been completed, the generic logic is translated into the logic of the appropriate prover.

The Omnibus static verifier currently supports two theorem provers: Simplify and PVS. For each prover, we must define how to formalise the background, how to express definitions, axioms and VCs, and how to translate all elements of the expression language.

Translation of generic logic into Simplify

Simplify [36] is a fully automated theorem prover developed specifically with ESC in mind. Simplify's input language is an untyped first-order logic with function symbols and equality, and a LISP-style syntax. It includes a theory of arithmetic defining function symbols `+`, `-`, `*` and relation symbols `>`, `<`, `>=`, `<=` with the usual meaning. While the theory is untyped, it distinguishes between propositional values and terms. An important consequence of this is that propositional operators such as `IMPLIES` and `<` cannot be used as terms.

Definition of background in Simplify: The definition of the background `is` and `valid` functions in Simplify is relatively straightforward. Simplify does not need these to be declared in the background before they can be referred to in the generated axioms.

The main purpose of the background axioms in Simplify is to define term-space equivalents of the predicate symbols. Simplify provides built-in functions for equality, implication, negation, inequality, conjunction, disjunction, etc. but they are defined as predicates. As such, they cannot be used as sub-expressions. We follow Cok's solution to this problem [32]: we define term-space equivalents for each predicate and use these in our expressions. The effect of this is that everything is lifted into the term-space and the distinction can then be ignored. We will see later that this does, however, complicate the handling of quantifiers.

The first step in this process is to define term symbols for the logical values `true` and `false`. This can be done as follows in terms of a single special symbol `|@true|`. The following logic is included within the axiom file passed to Simplify on start-up.

```
(EQ true |@true|)
(NEQ false |@true|)
```

Let us now, for example, consider the conjunction operator. Simplify has a built-in predicate AND which takes two predicates and evaluates to true iff both are true. To illustrate the problem, we could not use a call of a boolean method as one of the operands of an AND predicate since methods are defined using function symbols that are part of the term-space. However, we can introduce a new term-space operator && which is equal to the special term true iff both its operands are equal to true, defined using the AND predicate. The axiom formalising this is:

```
(FORALL (x y)
  (PATS (&& x y))
  (IFF
    (EQ (&& x y) true)
    (AND (EQ x true) (EQ y true))
  ))
```

A special predicate, PRED, is defined at the start of each Simplify logic file. PRED is a predicate which accepts a boolean term value and is equivalent to a predicate asking whether the value is equal to the true symbol. This allows boolean expressions in the term space to be lowered to the predicate space.

```
(DEFPRED (PRED x) (EQ x true))
```

These axioms and definitions are the only things that we need to define in the Simplify background in order to translate Omnibus classes and methods into Simplify.

Translation of expressions into Simplify: A mapping must be defined for each expression in the Omnibus expression language to a corresponding expression in Simplify. The framework established makes this relatively easy for the basic operators. For example, we saw how a special term-space && operator is defined for conjunction. So to translate an Omnibus conjunction into Simplify we write "(&&", translate the two operands of the conjunction and then write ")".

The translation of method calls is also relatively straightforward. We simply write a call of the special function used to define the method.

The main challenge is in the handling of quantifiers. Quantifier predicates can be defined in Simplify but, as was explained earlier, in the translation of Omnibus into Simplify all expressions are lifted into the term-space. So, we can translate a quantifier to Simplify if it is used as the top-level expression in an axiom or VC but not as a sub-expression, say, within a post-condition assertion. To combat this, term-space skolem functions are introduced, and axioms are added describing them in the same way that term-space equivalents of the built-in logical operators were defined in the background.

For example, consider the following specification of a `Set` class. The post-condition contains a quantification.

```
public spec class Set[Element] {
  public model function contains(e:Element):boolean

  public constructor empty()
    ensures forall (e:Element):
                !contains(e)
}
```

Consider the translation of the following axiom for the post-condition.

```
declare Set_empty_pub_post(ElementClass:Class,this:Set):bool

axiom Set_empty_pub_post_ax: forall (ElementClass:Class,
                                       this:Set):
         Set_empty_pub_post(ElementClass,this)
         = (forall (e:Element): !Set_contains(this, e))
```

Within the post-condition axiom, the forall quantification is the right operand of an equality operator. When the equality operator is translated to Simplify, it takes two term parameters and so we cannot use the built-in `FORALL` predicate. The universal quantification is left in its unevaluated Omnibus form, surrounded by double angle brackets, in the following piece of Simplify.

```
 (BG_PUSH
  (FORALL (ElementClass this)
    (PATS (Set_empty_pub_post ElementClass this) )
    (PRED
     (==>
      (is this Set)
      (=
        (Set_empty_pub_post ElementClass this)
```

```
         << forall (e:Element): !contains(e) >>
       )
      )
     )
   )
)
```

Note how the type information of the `this` variable is formalised within the body of the quantifier expression using an implication preceding the original quantified expression.

The solution we adopt for the expression of the quantification is to introduce a special boolean term skolem function for each quantifier and define its meaning using a separate axiom at the start of the Simplify file. The post-condition axiom then becomes:

```
 (BG_PUSH
  (FORALL (ElementClass this)
    (PATS (Set_empty_pub_post ElementClass this) )
    (PRED
     (==>
      (is this Set)
      (=
        (Set_empty_pub_post ElementClass this)
        (forall_0 ElementClass this)
      )
     )
    )
  )
)
```

With the `forall_0` skolem function defined as follows:

```
(BG_PUSH (FORALL (ElementClass this)
  (PATS (forall_0 ElementClass this))
  (IFF
    (EQ (forall_0 ElementClass this) true)
    (FORALL (e)
     (PRED
      (==>
        (is this Set)
        (! (Set_contains this e) )
      )
     )
    )
  )
))
```

Translation of declarations, axioms and VCs into Simplify: The axioms and VCs in the generic logic are written to a Simplify logic file ready to be processed by the prover. The declarations do not need to be translated since Simplify does not require the declaration of function symbols. Axioms can be defined in this file using the BG_PUSH instruction. PRED must be used within BG_PUSH to convert the term-space expression back to the predicate-space which BG_PUSH requires. If the top-level expression is a quantifier, then the use of PRED and a skolem function may be avoided by directly using the built-in quantifier predicates and using PRED within its body.

VCs can be defined by simply including the corresponding expression in the logic file. Again, PRED may need to be used to lower the term-space expression to the predicate space.

As an example of the translation of an axiom from the generic logic into Simplify, consider the following axiom defining the invariant of the Rectangle class we saw earlier.

```
declare Rectangle_invariant_0(this:Rectangle):bool

axiom Rectangle_invariant_0_ax: forall (this:Rectangle):
            Rectangle_invariant_0(this) =
                           Rectangle_width(this) >= 0
                             && Rectangle_height(this) >= 0
```

This is translated into the following Simplify axiom.

```
(BG_PUSH
  (FORALL (this)
    (PATS (Rectangle_invariant_0 this) )
    (PRED
     (==>
      (is this Rectangle)
      (=
        (Rectangle_invariant_0 this)
        (&&
          (GE (Rectangle_width this) 0)
          (GE (Rectangle_height this) 0)
        )
      )
     )
    )
  )
)
```

Note the use of the && and GE operators which are the term-space equivalents of the built-in AND and >= predicates.

As an example of the translation of a VC, consider this VC from the PosCounter class we saw earlier.

```
vc vc_6: forall (old_this:v_PosCounter, this:PosCounter):
             PosCounter_dec_pub_pre(old_this)
               && PosCounter_dec_pub_post(old_this,this)
             ==> PosCounter_constraint_0(old_this,this)
```

This is translated into the following Simplify:

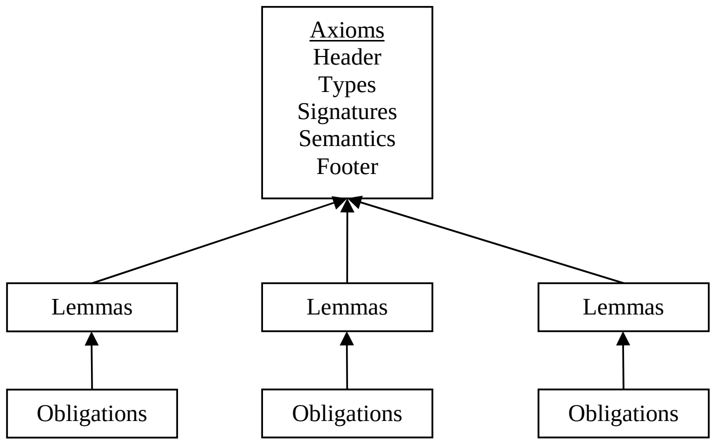
```
(FORALL (old_this this)
  (PRED
    (==>
     (&&
      (&&
        (is old_this PosCounter)
        (valid old_this PosCounter)
      )
      (is this PosCounter)
     )
     (==>
      (&&
        (PosCounter_dec_pub_pre old_this)
        (PosCounter_dec_pub_post old_this this)
      )
      (PosCounter_constraint_0 old_this this)
     )
    )
  )
)
```

The key aspect here is the use of the built-in `valid` function. This comes from the fact that the type of `old_this` in the axiom is `v_PosCounter`, i.e. a valid instance of `PosCounter` and thus satisfying its invariants as well as being an instance of `PosCounter`.

Translation of generic logic into PVS

PVS (Prototype Verification System) [81] is a sophisticated theorem prover developed by SRI International. It is available for Linux and Solaris systems and is executed as a plug-in for the emacs editor. The PVS system contains a specification language, a parser, a type checker, a prover, specification libraries and various associated tools. The specification language is based upon typed higher-order logic. PVS specifications are made up of theories which may contain, among other things, collections of type and constant definitions, axioms and conjectures. PVS types can be defined from the base types (Booleans, integers, etc.) using functions, records and a range of other standard formal mechanisms. PVS also allows the definition of additional, uninterpreted base types and predicate sub-types (a subset of individuals from another type which satisfy some predicate). The inclusion of predicate sub-types means that type checking is undecidable and may lead to the production of proof obligations called Type Correctness Conditions (TCCs) which must be proved in order to show that a theory is well-typed. In PVS, all functions are total, but partial functions can be modelled using predicate subtypes to restrict the domain of the input parameters. There is a range of built-in proof tactics for semi-automating proofs. For example the `(grind)` tactic applies rewrite rules, simplifies using decision procedures, and carries out heuristic quantifier instantiation.

PVS has a declare-before-use requirement within its theories. This can cause a problem in the formalisation of languages such as Omnibus which have no such requirements themselves. In Omnibus, the ordering of methods within a class is irrelevant: the specification of any method can refer to any other method. To address this problem we followed the approach taken by the LOOP project [49] of splitting the generated PVS theories into levels. At the top level we have the formalisation of the background in PVS. Then we have the definition of the special PVS types for each class. This is followed by the type signature declarations of the functions defined by the generic logic declarations, declarations of any required symbol constants, and then the generic logic axioms. Finally, there are conjecture files for each class to be verified along with lemma files for defining useful intermediary lemmas for use in the proof of multiple VCs.



Definition of background in PVS: The definition of the background starts with the definition of the basic concepts of objects and classes in PVS. There is a non-empty set of Classes and a set of Objects. For each class we will introduce a `Class` constant and subtype of `Object` for the object instances of that class. We then also declare the built-in `is` and `valid` predicates which each accept an object and class. For convenience we also introduce `InstanceOf` and `v_InstanceOf` predicate subtypes. These accept a `Class` constant and yield a type for the object instances and valid object instances of that class, respectively.

```
Class: TYPE+

ObjectClass: Class
Object: TYPE

is: [Object,Class -> bool]
valid: [Object,Class -> bool]

InstanceOf(c:Class): TYPE = {this:Object | is(this,c)}
v_InstanceOf(c:Class): TYPE = {this:InstanceOf(c) |
     valid(this,c)}
```

The above definitions constitute the application-independent PVS prelude for Omnibus in its entirety. These definitions establish a framework within PVS which is sufficient to describe Omnibus classes and methods. No further definitions are needed, e.g. there is no formalisation of a heap.

Let us now consider the generation of the type information for each class. The system is simplest for classes without template parameters. Consider the `Shape` class:

```
public class Shape {}
```

Special class constants and types for valid and invalid objects are defined for each class level. Here, there are no protected members and the class is not declared to be `spec` or `native` so there are two class levels: public and private. The object types are declared as predicate subtypes of their corresponding super class levels. The public `Shape` class level (‘Shape’ without an accessibility prefix) inherits from the public `omni.lang.Object` class level, and the private `Shape` class level inherits from the public `Shape` class level.

```
ShapeClass: Class
Shape: TYPE = {this:omni_lang_Object | is(this,ShapeClass)}
v_Shape: TYPE = {this:Shape | valid(this,ShapeClass)}

priv_ShapeClass: Class
priv_Shape: TYPE = {this:Shape | is(this,priv_ShapeClass)}
v_priv_Shape: TYPE = {this:priv_Shape |
     valid(this,priv_ShapeClass)}
```

The situation is a bit more complicated when it comes to classes with template parameters. Consider a parameterised `Set` class.

```
public class Set[Element] {}
```

The first stage is the same as for the `Shape` class, defining `SetClass`, `Set`, `v_Set`, `priv_SetClass`, `priv_Set` and `v_priv_Set` elements in the same way. A special function is also defined for the template parameter. This function maps the `Set` object to the actual template parameter value for the `Set`. Predicate subtypes are then defined for `Set` with a specific template parameter. The predicate restrictions assert that the value of the template parameter type is equal to the type passed.

```
Set_ElementClass: [Set -> Class]

SetOf(ElementClass:Class): TYPE = { this:Set |
     Set_ElementClass(this) = ElementClass}
v_SetOf(ElementClass:Class): TYPE = { this:v_Set |
     Set_ElementClass(this) = ElementClass}
```

```
priv_SetOf(ElementClass:Class): TYPE = { this:priv_Set |
     Set_ElementClass(this) = ElementClass}
v_priv_SetOf(ElementClass:Class): TYPE = { this:v_priv_Set |
     Set_ElementClass(this) = ElementClass}
```

Translation of expressions into PVS: The translation of Omnibus expressions to PVS is more straightforward than the translation into Simplify. There is no hard distinction between terms and predicates, so the built-in operators and quantifiers can be used directly.

For example, reconsider the declaration and axiom for the post-condition of the `Set` class discussed earlier.

```
declare Set_empty_pub_post(ElementClass:Class,this:Set):bool

axiom Set_empty_pub_post_ax: forall (ElementClass:Class,
                                    this:Set):
        Set_empty_pub_post(ElementClass,this)
        = (forall (e:Element): !Set_contains(this, e))
```

The translation of this declaration and axiom into PVS is relatively straightforward. First there is the declaration of the type signature of the post-condition function in the `Signatures` section. Then there is the axiom describing the function. The built-in `FORALL` quantifier is used directly within the expression. This was possible in Simplify because of the separate term and predicate spaces. Note how the `v_InstanceOf` predicate subtype is used in the translation of the quantification over `Element`.

```
Set_empty_pub_post(ElementClass:Class, this:Set):bool

Set_empty_pub_post_ax: AXIOM
  (FORALL (ElementClass:Class), (this:Set):
     (Set_empty_pub_post(ElementClass, this) = (FORALL
     (e:v_InstanceOf(ElementClass)):
        (NOT Set_contains(this, e))
     ))
  )
```

Translation of declarations, axioms and VCs into PVS: Declarations, axioms and VCs are again relatively straightforward to formalise in PVS. Declarations map naturally to PVS function declarations and the axioms of the generic logic map naturally to PVS axioms. The function declarations are added to the Signatures section. The VCs in the generic logic map to PVS conjectures.

As an example of the translation of generic logic declarations and axioms to PVS, consider the `area` function in the `Rectangle` class we saw earlier. Crucially, this method overrides the definition given in the `Shape` class. There are two axioms for this method, one for the post-condition function and one for method function itself. There is no declaration for the method function, it just provides an axiom for reasoning about the corresponding root method function which it overrides.

```
declare Rectangle_area_pub_post(this:Rectangle,
                                result:integer):bool

axiom Rectangle_area_pub_post_ax:
          forall (this:Rectangle,result:integer):
                Rectangle_area_pub_post(this,result) =
                      result = Rectangle_width(this)
                              * Rectangle_height(this)

axiom Rectangle_area_pub_ax: forall (this:Rectangle):
            Shape_area(this) = Rectangle_width(this)
                              * Rectangle_height(this)
```

In the generated PVS, the post-condition function is declared in the Signatures section. No declaration is needed for the method axiom. The post-condition and method function axioms are fairly direct translations of the generic logic axioms.

```
Rectangle_area_pub_post(this:v_Rectangle, result:int):bool

Rectangle_area_pub_post_ax: AXIOM
  (FORALL (this:v_Rectangle), (result:int):
     (Rectangle_area_pub_post(this, result) = (result =
     (Rectangle_width(this) * Rectangle_height(this))))
  )
AUTO_REWRITE+ Rectangle_area_pub_post_ax

Rectangle_area_pub_ax: AXIOM
  (FORALL (this:v_Rectangle):
     (Shape_area(this) = (Rectangle_width(this) *
     Rectangle_height(this)))
  )
AUTO_REWRITE+ Rectangle_area_pub_ax
```

Again let us consider the translation of the VC to verify that the `dec` operation of the `PosCounter` class satisfies the constraint. The VC in the generic logic was:

```
vc vc_6: forall (old_this:v_PosCounter, this:PosCounter):
             PosCounter_dec_pub_pre(old_this)
               && PosCounter_dec_pub_post(old_this,this)
             ==> PosCounter_constraint_0(old_this,this)
```

This is translated into the following PVS:

```
vc_6: CONJECTURE
  (FORALL (old_this:v_PosCounter), (this:PosCounter):
     ((PosCounter_dec_pub_pre(old_this) AND
     PosCounter_dec_pub_post(old_this, this)) IMPLIES
     PosCounter_constraint_0(old_this, this))
  )
```

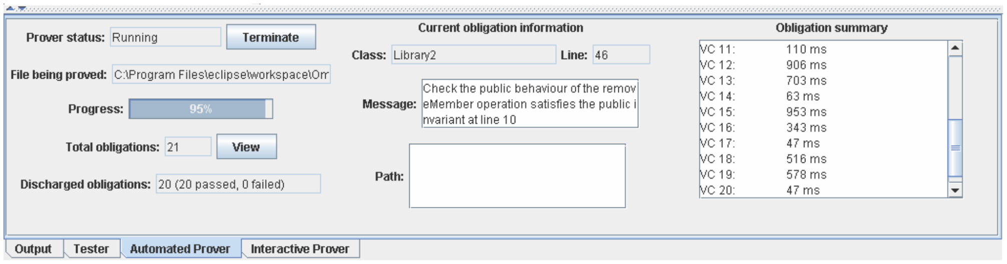
The PVS version of this VC is more concise than the Simplify version. This is because the typing information of the quantified variables is expressed through the typing, whereas it must be explicitly described within the quantified expression in the Simplify version.

7.5 Prover-specific interaction

The Omnibus IDE provides a range of support for managing the process of proving the generated VCs.

7.5.1 Simplify prover support

The automated verification process starts with the generation of generic logic axioms and VCs and the translation of these into Simplify. The generated file is then passed to the Simplify prover for processing. The prover is automatically launched by the Omnibus IDE and its output monitored. This process is carried out in a new thread so that the user can carry out other tasks while the prover is running. The automated prover pane (shown below) allows the user to track the process of the prover and gives them the option to terminate it.



Prover status: Running

File being proved: C:\Program Files\eclipse\workspace\Om

Progress: 95%

Total obligations: 21

Discharged obligations: 20 (20 passed, 0 failed)

Current obligation information

Class: Library2 Line: 46

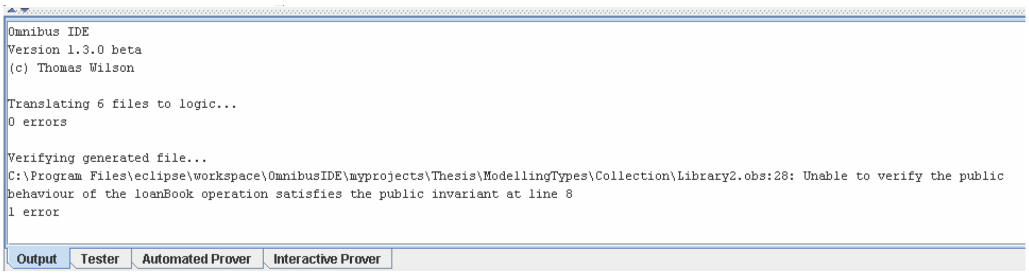
Message: Check the public behaviour of the removeMember operation satisfies the public invariant at line 10

Path:

Obligation summary

```
VC 11:    110 ms
VC 12:    906 ms
VC 13:    703 ms
VC 14:    63 ms
VC 15:    953 ms
VC 16:    343 ms
VC 17:    47 ms
VC 18:    516 ms
VC 19:    578 ms
VC 20:    47 ms
```

When the prover has finished, the IDE checks that each VC has the expected validity. If a VC has the wrong validity then the corresponding error message stored with that VC is displayed. An example error message is displayed below.



```
Omnibus IDE
Version 1.3.0 beta
(c) Thomas Wilson

Translating 6 files to logic...
0 errors

Verifying generated file...
C:\Program Files\eclipse\workspace\OmnibusIDE\myprojects\Thesis\ModellingTypes\Collection\Library2.obs:28: Unable to verify the public
behaviour of the loanBook operation satisfies the public invariant at line 8
1 error
```

The full details of the proof process are recorded and can be viewed through the check viewer.

7.5.2 PVS prover support

The interactive verification process starts with the generation of a set of PVS theories from the generic logic definitions, axioms and VCs. Each Omnibus project has a setting for where the PVS files should be generated. The PVS prover should then be independently used by the developer to prove the conjectures in the generated files. Lemma files are automatically generated and can be used to support reuse within proofs. Proof files are used by PVS to record proof attempts. The IDE generates default proofs for each VC. These simply consist of the `(grind)` tactic. However, together with the auto-rewrites, this is sufficient to prove many simple VCs.

The IDE tool is able to track the progress of the user-made proofs in PVS by parsing the proof files. This information on the validity of the PVS VCs can be viewed via the check viewer in the same way that it is used for the automated verifier.

Chapter 8

Case study

This chapter discusses a case study carried out using Omnibus. The initial version of the project was written by a student as part of their Honours dissertation and this was used as a basis for a case study in verification with Omnibus. Our aim was to gain feedback on the ease with which the Omnibus system could be applied by a member of our target audience, someone without extensive formal methods training. The student produced an implementation and a heavyweight specification of correctness. This gave us an example with real errors generated by another person to which we could apply the Omnibus process. We derived lightweight specifications from the heavyweight specifications and then carried out the verification process and corrected errors. The case study does not provide industrial-scale justification of the Omnibus approach but does allow the ideas it contains to be explored in some more detail.

We start the discussion of the verification of the case study by presenting an implementation with assertion annotations and using the Omnibus tool to RAC-check it. This process uncovers some errors which we correct. We then attempt to ESC-check the RAC-corrected implementation. This uncovers further errors which we, again, correct. Next we consider the FFV approach and verify that a heavyweight behaviour specification, with corrections that are equivalent to those made after RAC and ESC checking, meets its requirements. This uncovers still further errors which we correct. Finally, we explore the benefits of greater levels of integration between the approaches using verification policies.

We are not recommending that the verification approaches be applied to real examples in this way. We perform verification using each of the approaches on the same example in order to permit direct comparison of their error coverage.

The assertion annotations are adapted from those produced by the student and, unsurprisingly, contain errors. Often it was possible for us to spot these errors without the use of the verification tools. However, we purposely retained these errors in our adapted versions so that we could test the error detection facilities of the Omnibus verification tool.

8.1 Process used in the development of the Library system

This section describes the process that was used to develop the Library system. The initial version of the Library case study was produced following a process devised for use with the Omnibus FFV approach. First, a specification was produced and checked, and then a corresponding implementation produced and checked. This process consists of the following steps, taken from section 3.1 of the student's dissertation [14]:

1. Produce a heavyweight specification
    a. Elicit requirements and model the problem. This initial work must be done outside the Omnibus language, perhaps using UML use case diagrams.
    b. Outline the public type signature interface of the class, perhaps using a variation of UML class diagrams. Skeleton class definitions can then be described in Omnibus. Methods must be divided up into constructors, functions and operations.
    c. Select a subset of the functions of the class that together represent the abstract state of object instances of the class and declare these as model functions. If the functions in the class are not sufficient to do this then additional specification-only functions should be introduced to enable this

to be done. Define the remaining functions in terms of the model functions.

   d. Write requirements over the model functions using initiallys, invariants and constraints. What should hold over the model functions initially? What should hold over them at all times? How should they be allowed to change?
   e. Describe the constructors and operations in terms of the model functions using requires, changes and ensures clauses. Care should be taken to use the changes clauses correctly.
   f. Verify that the behaviour specifications satisfy the requirements.

2. Produce an implementation for the specification
   a. Devise private attributes to implement the model functions and hence provide a concrete representation for the state of each class.
   b. Write any additional requirements over the attributes using private initiallys, invariants and constraints.
   c. Describe the model functions in terms of the attributes using private ensures clauses.
   d. Write code for all the methods in the class. The specifications should be used to guide the developer in the production of the code.
   e. Verify that the implementation satisfies the behaviour.

The student derived the requirements of her Library system from a range of formal specification textbooks. She produced an informal description of the requirements together with a series of UML use case diagrams. The design of the system then began through the use of UML class diagrams. This class diagram was then translated into Omnibus type signature interfaces for the classes in the system.

Heavyweight specifications were then produced for each of the classes. First the functions were split into model functions and derived functions and their behaviour specified. Then high-level requirements were defined over these, using invariants. Specifications could then be provided for the constructors and operations for each class. The next stage in the process outlined above was to verify that the behaviour specifications satisfied the requirements specifications. The author of this thesis attempted to verify properties of the specification using the Omnibus static verifier and PVS theorem prover. The errors which were found are reported in section 3.1.4.5 of the student's dissertation [14] and described later in this chapter. Errors are uncovered by this process when an unprovable VC is discovered. During the proof process, the user has to think quite deeply about the specification and often has a clear idea of where the specification is erroneous by the time the proof is abandoned. For example, the user may have expected to have a piece of knowledge available to them but found that it was not because it had not been provided by a suitable requires clause.

The student then produced an implementation of the heavyweight specification. Since the heavyweight specification was very detailed, this stage was relatively straightforward. A private attribute was introduced for each model function and the model functions defined in terms of them using private ensures clauses. Implementations were then provided for the other methods in the classes of the system. The implementation of each method was derived from the heavyweight specification of its behaviour as described in the ensures clause of the method. The student used a number of coding patterns developed by the author of this thesis in order to convert assertions involving quantifiers into Omnibus code. Finally, the author of the thesis used the Omnibus static verifier and PVS theorem prover to check that the implementations of the selected methods satisfied their behaviour specifications. Again, errors are uncovered by the discovery of unprovable VCs.

This process produced a heavyweight specification for the system along with a corresponding implementation. This was ideal for use with the FFV approach. In order to enable the RAC and ESC approaches to be used, the author of this thesis adapted the heavyweight specification produced by the student to make it amenable to RAC and ESC. This involved a reduction in the detail provided in the ensures clauses of the behaviour specifications, e.g. only specifying how the sizes of collections should be changed rather than precisely specifying what elements the resulting collection should hold.

The use of the RAC approach requires the development of test cases to exercise the implementations suitably to ensure that all the assertions checks are executed. Tests should be defined for each usage scenario, e.g. adding/updating/deleting a book or loaning/returning an item. The crucial thing is that the tests for the class should involve the execution of all the assertion checks in all of the methods in the class. This may require that there are multiple invocations for a single method in the tests to cover the different paths through that method. The obligation viewer can be used to determine which assertion checks are being covered and which are not. If an assertion check is not being covered then the test cases should be adjusted to check it. The process for the definition of a test case is as follows:

1. Declare and instantiate variable to hold object instance of the class containing the test, i.e. the class to be tested.
2. For each operation within the particular usage scenario:
    a. Apply the operation to the instance of the object to be tested. Declare variables for each of the parameters of the operation, instantiate them with appropriate values, and then pass these variables when the operation is called.
    b. Check the values of any relevant derived functions through `assert` statements preceded by declarations of variables for any values needed to be passed as parameters to the functions

Errors are detected by the RAC approach via run-time assertion failure messages. These indicate the precise assertion check which has failed along with the relevant context information such as the parameters of the call and the current attribute values. These can be used by the user to determine the logical cause of the failure.

The rest of this chapter describes the system as if it was developed first for use with RAC and then adapted for use with ESC and FFV. We feel that this order of presentation is more natural. The RAC and ESC Omnibus tools were not used as part of the original case study by the student because they had not been fully implemented at the time. If they had been, we would have applied the tools in this order.

8.2 Informal description

The Library contains a set of `Item`s which can be loaned out. Each `Item` has a unique `itemNo` and a `title`. There are three concrete types of `Item`: `Book`s, `CD`s and `DVD`s. In addition to its `itemNo` and `title`, a `Book` has an ISBN, a list of authors (which should not contain duplicates), a publisher and a published date. A CD has an `itemNo`, a `title`, a group (i.e. the band or artist name) and a release date. A DVD has an `itemNo`, a `title`, a list of actors (which should not contain duplicates), a director and a release date.

The Library can hold multiple copies of each `Item`. Each `Copy` has a unique `copyNo` along with the `itemNo` identifying the item of which it is a copy. Each copy also has a status which is either ‘available’ or ‘lost’. Items and copies can be searched, added, deleted or changed. The addition of a new item automatically creates a new copy and the deletion of the last copy of an item automatically deletes the `Item`’s details from the system.

People must register with the library as members before they can loan out a copy of an item. There is a `Member` record in the library for each member. This contains a unique `memberNo`, their name, their address, their telephone number, whether they are an adult, whether they are a member of staff, their current status (either active, inactive or banned), the amount of any fines they owe, and the PIN for use when logging into the system. Each `Member` has associated with them a `Card` with a unique `cardNo` and the `memberNo`. Members can be searched, added, deleted or changed.

Members of the library can loan out copies of items in the library. For each `Loan`, the details of the `memberNo`, `copyNo` and start and end dates of the `Loan` are recorded. A member can choose to renew or return their copy at any time. Fines may be made if the `endDate` of the loan is exceeded. The details of the loans can be searched, added, deleted or changed via a standard loan/return/renew procedure.

The system is described in more detail in section 3.1 of [14].

8.3 Run-time assertion checking

In this section, we will consider the production of an implementation with lightweight assertion annotations and its verification using run-time assertion checking with associated test cases.

The full implementation consists of 15 classes with a total of 2278 lines of Omnibus code (including lightweight assertion annotations and test cases). There are 1034 separate assertion checks generated. The remainder of this section discusses interesting aspects of the example.

8.3.1 Auxiliary classes

Most of the classes in the Library case study are simple data structures that are used by the main `Lib` class.

Simple data structure classes

The following classes all follow the same simple structure: `Book`, `CD`, `Card`, `Copy`, `DVD`, `Item`, `Loan`, `Member` and `Name`. The files which define them contain a number of sections: the abstract state, a constructor and update operations.

As an example, consider the `Book` class. The public accessor functions should be declared to be model functions so we can use them in `changes` clauses. They can be implemented by returning the corresponding attribute value. We could include private specifications describing the methods in terms of the private attributes at the private level but, for run-time assertion checking, this is not required.

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

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

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

  public model function pubDate():Date {
    return pubDate;
  }
```

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

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

The constructor is implemented by assigning the passed parameters to the corresponding attribute.

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

Finally, the operations to support the updating of the newly introduced state elements are added. These are declared to change the relevant state element.

```
  public operation setISBN(isbnNo:ISBN)
    changes isbn
  {
    isbn := isbnNo;
  }

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

  public operation setPublisher(pub:String)
    changes publisher
  {
    publisher := pub;
  }

  public operation setPubDate(pDate:Date)
    changes pubDate
  {
    pubDate := pDate;
  }
}
```

Predicate restricted classes

The `ISBN` class consists of a single state element, a `String`, whose range of values is restricted to be within some valid range. The methods of the class guard assignments to the internal value with checks of the validity assertion. This provides a means of ensuring that all values that can be encapsulated by such a class satisfy its validity requirement.

The definition of the class starts with the declaration of the private attribute and public model function for the value.

```
public class ISBN {
  private attribute value:String

  public model function value():String {
    return value;
  }
```

The validity requirement is defined using a static boolean function, here called `isValidISBN`. The invariant is then defined using this function.

```
  public static function isValidISBN(s:String):boolean
    ensures result = (s.length() = 13 &&
     (forall (i:integer := 0 to s.length()-1):
      Character.isDigit(s.charAt(i))
      || s.charAt(i) = '-'
      || s.charAt(i) = ' '))
  {
    if (s.length() != 13) {
     return false;
    }
    for (j := 0 to s.length()-1) {
     if (!Character.isDigit(s.charAt(j))
        && s.charAt(j) != '-'
        && s.charAt(j) != ' ') {
      return false;
     }
    }
    return true;
  }

  public invariant ISBN.isValidISBN(value())
```

Finally, the constructor accepts a value for the internal value and checks in a `requires` clause that the value is valid.

```
  public constructor fromString(s:String)
    requires ISBN.isValidISBN(s)
    ensures value() = s
  {
    value := s;
  }
}
```

Enumeration classes

The `ItemStatus` and `MemberStatus` classes are similar to the `ISBN` class in that they guard a single value to ensure that it is valid as defined by some invariant. However, these classes have a small number of known possible values. Each of the possible values is allocated an integer value and an invariant ensures that the value of the state element is always one of these valid values. Constructors are then defined for each value, constructing an object with the corresponding value. For convenience, functions can also be defined to easily check which of the values the object has.

```
public class ItemStatus {
  private attribute value:integer

  public model function value():integer {
    return value;
  }

  public invariant value() >= 0 && value() <= 1

  public constructor lost()
    ensures value() = 0
  {
    value := 0;
  }

  public constructor available()
    ensures value() = 1
  {
    value := 1;
  }
  public function isLost():boolean
    ensures result = true <==> value() = 0
  {
    if (value = 0){
     return true;
    } else {
     return false;
    }
  }

  public function isAvailable():boolean
    ensures result = true <==> value() = 1
  {
    if (value = 1){
     return true;
    } else {
     return false;
    }
  }
}
```

Miscellaneous classes

Finally, there is the `Date` class which does not fit into any of the other categories. It has an abstract state consisting of `day`, `month` and `year` elements. There is a constructor for creating a new `Date` object. There are static functions for querying properties of the date and local functions for carrying out date calculations.

```
public class Date {
  private attribute day:integer
  private attribute month:integer
  private attribute year:integer

  public model function day():integer {
    return day;
  }

  public model function month():integer {
    return month;
  }

  public model function year():integer {
    return year;
  }

  public invariant day() <= 31
  public invariant month() <= 12

  public constructor newDate(d:integer, m:integer, y:integer)
  {
    day := d;
    month := m;
    year := y;
  }

  public function nDaysLater(n:integer):Date
  { ... }

  public static function isLeapYear(year:integer):boolean
    ensures result = (year % 4 = 0)
  { ... }

  public static function isNewYear(day:integer,
     month:integer):boolean
    ensures result = ((day > 31) && (month = 12))
  { ... }

  public static function isNewMonthAfterThirty(day:integer,
     month:integer):boolean
    ensures result = (((month = 4) || (month = 6) || (month =
     9) || (month = 11))
      && day > 30)
  { ... }

  public static function isNewMonthAfterThirtyOne
                     (day:integer, month:integer):boolean
    ensures result = (((month = 1) || (month = 3) || (month =
     5) || (month = 7)
      || (month = 8) || (month = 10)) && day > 31)
  { ... }

  public function calcDaysLate(d:Date):integer
    requires d.laterThan(this)
  { ... }

  public function laterThan(d:Date):boolean
    ensures result =  !(year() < d.year()
     || year() = d.year() && month() < d.month()
     || year() = d.year() && month() = d.month() && day() <
     d.day())
  { ... }
}
```

8.3.2 Central Lib class

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

Accessor functions

Again, the accessor functions for the private attributes should be declared as public model functions to enable them to be used in `changes` clauses.

The model functions for the `Lib` class are:

```
  public model function items():Collection[Item] {
    return items;
  }

  public model function copies():Collection[Copy] {
    return copies;
  }

  public model function members():Collection[Member] {
    return members;
  }

  public model function cards():Collection[Card] {
    return cards;
  }

  public model function loans():Collection[Loan] {
    return loans;
  }
```

Formalisation of invariants of Lib class

Seven high-level correctness properties are laid out in the undergraduate student's informal specification [14]. Four properties state that the identification numbers used for each item, copy, card and member (item number, copy number, card number and member number, respectively) are unique. Two properties state that the copies and members that are referred to in any loan must be in the library's recorded copies and members, respectively. The final property expresses that a single copy cannot be recorded as on loan more than once at any one time.

These correctness properties can be expressed as invariants within the `Lib` class. To define the uniqueness of the identification numbers we can use nested universal quantifiers. For example, the invariant that the `itemNos` of the `Item`s in the `items` collection are unique is:

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

We can use the following as the invariant that every copy that is on loan is in the `copies` collection with a similar invariant for the `memberNo` for each loan.

```
  public invariant allCopiesOnLoanInCopies:
     forall (l:Loan in loans()):
      exists (c:Copy in copies()):
        l.copyNo() = c.copyNo()
```

For the invariant that no copies are on loan more than once we can assert that if two loans have the same `copyNo`, they must be equal.

```
  public invariant noCopiesOnLoanMoreThanOnce:
    forall (l:Loan in loans()):
     forall (l2:Loan in loans()):
      l.copyNo() != l2.copyNo() || l = l2
```

Lib operations

As an example of the definition of an operation from the `Lib` class, consider the `takeOutCopy` operation which is used to take a copy of an item from the library and loan it out to a member. This method accepts a member number (`memNo`), a copy number (`cNo`) and a date (`date`) and takes out a new loan.

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

The `requires` clause asserts that there must be a `Member` in `members` with `memNo` as its `memberNo`, that there must be a `Copy` in `copies` with `cNo` as its `copyNo` and that there should not be any `Loan` in `loans` which has `cNo` as its `copyNo`. These assertions help assure that after the new `Loan` is added with the passed details that the invariants are maintained. The `changes` clause specifies only the `loans` element (referring to the public model function) and so the other state elements (the other public model functions) should not be changed. A lightweight specification of the behaviour of the method is given in the `ensures` clause. This simply states that the size of the `loans` collection increases by one. Of course, this is an incomplete characterisation of the behaviour of the method but it is sufficient as a simple assertion to run-time check for this method.

The implementation constructs a new `Loan` object with `memNo`, `cNo`, the passed date as the start date of the loan and an end date calculated using the `calcNewDate` function. The `lookupItemFromCopy` function is used to retrieve the `itemNo` of the copy to pass to this function.

As an example of one of the update methods, consider the `changeMemberName` operation which is used to change the name associated with a member of the library. This accepts a `memberNo`, a first and last name and updates the name of the `Member` with this `memberNo` to hold this value.

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

The `requires` clause of the operation asserts that there should be a `Member` with the specified `memberNo`. This is required to ensure that the `lookupMember` function will always succeed. The `changes` clause specifies that only the `members` state element is changed. The `ensures` clause gives a lightweight specification of the behaviour of the method. It asserts that the size of the members collection should not have changed and that, after the method, the member with the specified memberNo should have the passed name.

The implementation starts by using the lookupMember function to retrieve the Member object with the specified memberNo. It removes this Member from the members collection, updates it using the setName operation and then re-adds it to members. As a result, the old Member with this memberNo should have been renamed and all other Members should have been unchanged.

Lib functions

There are also functions to query aspects of the library. For example, consider the lookupBookFromISBN function which is an example of one of the lookup functions. This function accepts an ISBN and returns a Book with that ISBN from books.

```
public function lookupBookFromISBN(isbn:ISBN):Book
  requires exists (i:Item in items() where i is Book):
    (i as Book).isbn() = isbn
  ensures result.isbn() = isbn,
   items().contains(result)
{
  foreach (i:Item in items){
   if (i is Book){
    var book:Book := i as Book;
    if (book.isbn() = isbn){
      return book;
    }
   }
  }
  unreachable;
}
```

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

The implementation iterates through the elements in the items collection. If it finds a Book, it checks the isbn and returns that Book if it matches the passed ISBN. The unreachable statement should never be reached because, from the requires clause, the return statement should be reached for one of the iterations through the loop.

The memberLoans function is an example of one of the search functions. This accepts a memberNo and returns a collection containing all the Items for which a Copy is on loan to that Member.

```
  public function memberLoans(memberNo:integer)
                                             :Collection[Item]
    ensures forall (i:Item in result):
      exists (l:Loan in loans()):
        l.memberNo() = memberNo
        && lookupItemFromCopy(l.copyNo()).itemNo()
                    = i.itemNo()
  {
    var memberLoansCol:Collection[Item]
                              :=Collection[Item].empty();
    foreach (l:Loan in loans){
     if (l.memberNo() = memberNo){
      memberLoansCol.add(lookupItemFromCopy(l.copyNo()));
     }
    }
    return memberLoansCol;
  }
```

There is no requires clause. If there is no Member with the specified memberNo, an empty collection will be returned. The ensures clause gives a lightweight specification of the behaviour of the function. It asserts that for all the Items in the returned collection, there should be a loan with the passed memberNo and a Copy of the Item.

The implementation creates an empty collection and then iterates through the Loan objects in loans, adding Items which are on loan to the specified Member. The collection is then returned at the end of the function.

8.3.3 Testing

Verification with run-time assertion checking is built around the use of testing. Tests must be used to ensure that the application performs its intended function without violation of any of the assertion annotations.

In Omnibus, we can test the application in one of three ways: defining a command-line application, using a separate GUI or defining test cases.

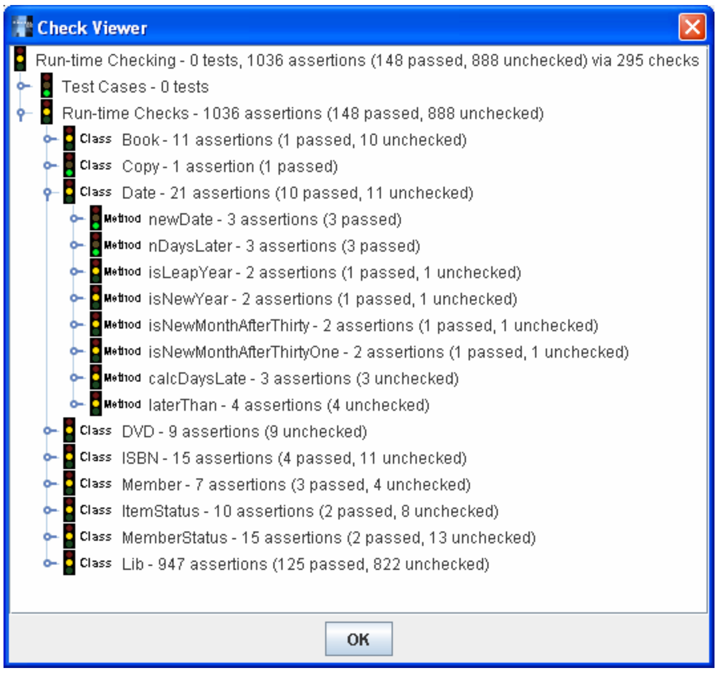
Command-line applications

Omnibus provides the facilities to define command-line applications. We can define a command-line application to test out the `Lib` class and then execute it. Application start points are defined by defining a subclass of the `omni.app.Application` class. The following application adds a new CD and loans it out to the existing member named Joe.

```
uses omni.app

public class Main isa Application {
  public constructor init() {}

  public operation execute(args:List[String],
                           var env:Environment) {
    env.println("Starting Library...");
    var lib:Lib := Lib.testDataLibrary();
    printState(var env, lib);
    env.println("Adding Eels Souljacker CD...");
    var sjCopyNo:integer := lib.getNewCopyNo();
    lib.acquireCD("Souljacker", "Eels",
                  Date.newDate(24,9,2001));
    printState(var env, lib);
    env.println("Loaning Eels Souljacker CD to Joe...");
    var joeMemNo:integer := 1;
    lib.takeOutCopy(joeMemNo, sjCopyNo,
                    Date.newDate(16, 5, 2007));
    printState(var env, lib);
  }

  private operation printState(var env:Environment,
                               lib:Lib)
  {
    env.println("Members:");
    foreach (m:Member in lib.members()) {
     env.println(" "+m.memberNo()+":"+m.name().fName()
                 +" "+m.name().sName());
    }
    env.println("Items:");
    foreach (i:Item in lib.items()) {
     env.println(" "+i.itemNo()+":"+i.title());
    }
    env.println("Copies:");
    foreach (c:Copy in lib.copies()) {
     env.println(" "+c.copyNo()+": copy of item "
                 +c.itemNo());
    }
    env.println("Loans:");
```

```
      foreach (l:Loan in lib.loans()) {
       env.println(" loan of "+l.copyNo()+" to "
                        +l.memberNo());
      }
    }
}
```

This application can be executed to perform limited testing of the application. The check viewer can be used to see exactly which assertions have been checked:



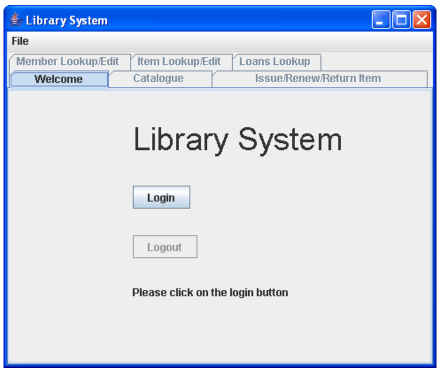
```
Run-time Checking - 0 tests, 1036 assertions (148 passed, 888 unchecked) via 295 checks
  Test Cases - 0 tests
  Run-time Checks - 1036 assertions (148 passed, 888 unchecked)
    Class Book - 11 assertions (1 passed, 10 unchecked)
    Class Copy - 1 assertion (1 passed)
    Class Date - 21 assertions (10 passed, 11 unchecked)
      Method newDate - 3 assertions (3 passed)
      Method nDaysLater - 3 assertions (3 passed)
      Method isLeapYear - 2 assertions (1 passed, 1 unchecked)
      Method isNewYear - 2 assertions (1 passed, 1 unchecked)
      Method isNewMonthAfterThirty - 2 assertions (1 passed, 1 unchecked)
      Method isNewMonthAfterThirtyOne - 2 assertions (1 passed, 1 unchecked)
      Method calcDaysLate - 3 assertions (3 unchecked)
      Method laterThan - 4 assertions (4 unchecked)
    Class DVD - 9 assertions (9 unchecked)
    Class ISBN - 15 assertions (4 passed, 11 unchecked)
    Class Member - 7 assertions (3 passed, 4 unchecked)
    Class ItemStatus - 10 assertions (2 passed, 8 unchecked)
    Class MemberStatus - 15 assertions (2 passed, 13 unchecked)
    Class Lib - 947 assertions (125 passed, 822 unchecked)
```

This tells us that there were 1036 run-time assertions generated. 148 of them were checked at least once. The remaining 888 were not checked by this particular test.

A separate GUI

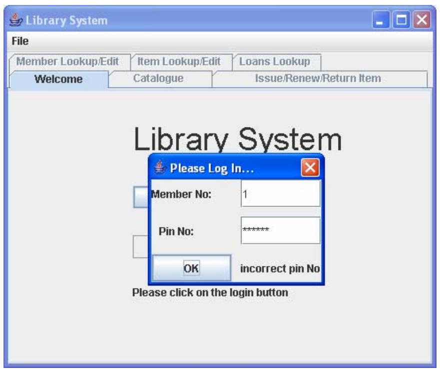
Another way to test the application is through the use of a GUI written in another language. The student implemented a GUI for the Library Omnibus application in Java. This can be used to interact with the application and perform tests.

The following screenshots show the use of the GUI to login, loan out a book, lookup the details of the loan, renew it on the same day without a fine and then return it late with a fine.



Library System

Please click on the login button



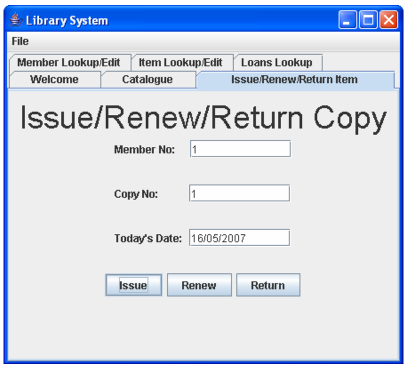
Library System

Member No: 1

Pin No: ******

incorrect pin No

Please click on the login button

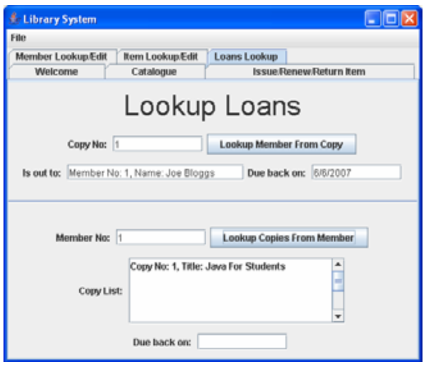


Issue/Renew/Return Copy

Member No: 1

Copy No: 1

Today's Date: 16/05/2007



Lookup Loans

Copy No: 1

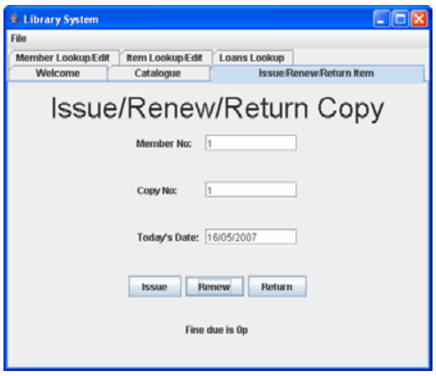
Is out to: Member No: 1, Name: Joe Bloggs

Due back on: 6/6/2007

Member No: 1

Copy List: Copy No: 1, Title: Java For Students

Due back on:



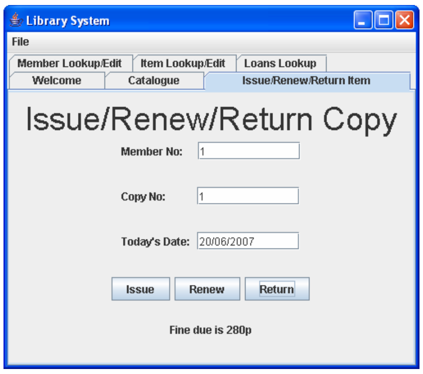
Issue/Renew/Return Copy

Member No: 1

Copy No: 1

Today's Date: 16/05/2007

Fine due is 0p



Issue/Renew/Return Copy

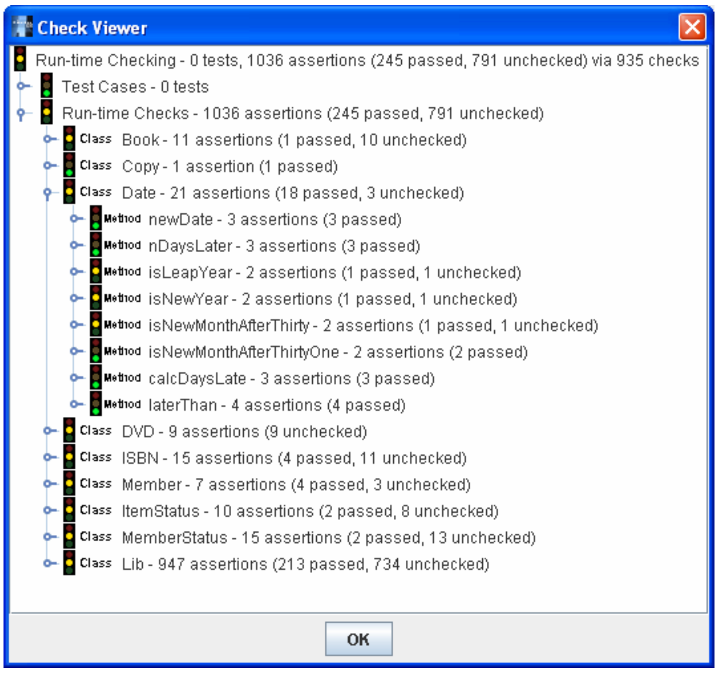
Member No: 1

Copy No: 1

Today's Date: 20/06/2007

Fine due is 280p

The check viewer (shown below) can be used in the same way to check the details of the run-time assertion checks that are made during this manual test. This test covered 245 of the 1036 assertions in the file. However, to repeat the test the user would have to repeat their actions with the GUI.



```
Run-time Checking - 0 tests, 1036 assertions (245 passed, 791 unchecked) via 935 checks
  Test Cases - 0 tests
  Run-time Checks - 1036 assertions (245 passed, 791 unchecked)
    Class Book - 11 assertions (1 passed, 10 unchecked)
    Class Copy - 1 assertion (1 passed)
    Class Date - 21 assertions (18 passed, 3 unchecked)
      Method newDate - 3 assertions (3 passed)
      Method nDaysLater - 3 assertions (3 passed)
      Method isLeapYear - 2 assertions (1 passed, 1 unchecked)
      Method isNewYear - 2 assertions (1 passed, 1 unchecked)
      Method isNewMonthAfterThirty - 2 assertions (1 passed, 1 unchecked)
      Method isNewMonthAfterThirtyOne - 2 assertions (2 passed)
      Method calcDaysLate - 3 assertions (3 passed)
      Method laterThan - 4 assertions (4 passed)
    Class DVD - 9 assertions (9 unchecked)
    Class ISBN - 15 assertions (4 passed, 11 unchecked)
    Class Member - 7 assertions (4 passed, 3 unchecked)
    Class ItemStatus - 10 assertions (2 passed, 8 unchecked)
    Class MemberStatus - 15 assertions (2 passed, 13 unchecked)
    Class Lib - 947 assertions (213 passed, 734 unchecked)
```

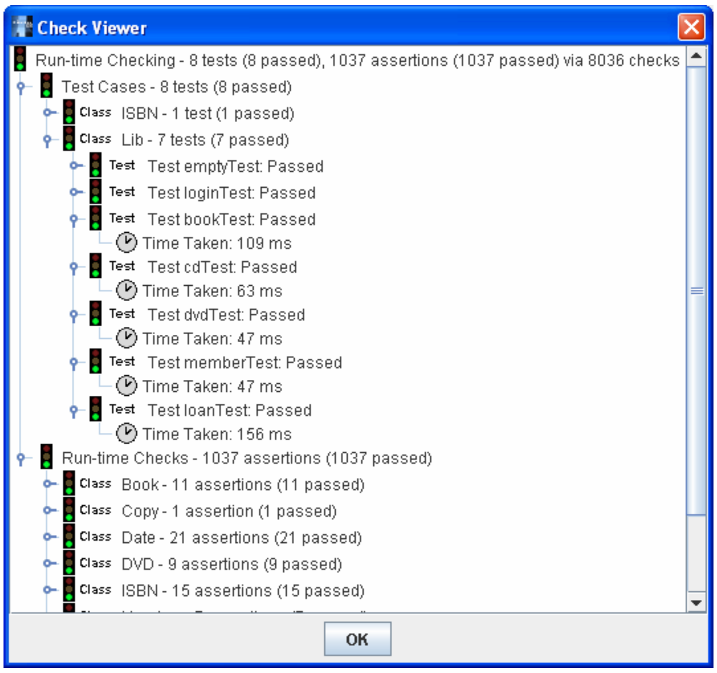
Test cases

Omnibus provides the facility to define test cases within each class. These can then be automatically re-executed by the user at any time.

To cover all the assertion checks we can use 7 tests: `emptyTest`, `loginTest`, `bookTest`, `cdTest`, `dvdTest`, `memberTest` and `loanTest`. Each of these tests a particular aspect of the example. As an example, the `bookTest` is displayed below. This tests the addition, retrieval, updating and changing of the loan status of books.

```
  test bookTest {
    var lib:Lib := Lib.testDataLibrary();

    // check addition of book
    // declare new book details
    var cplTitle:String
              := "Comparative Programming Languages";
    var cplISBN:ISBN := ISBN.fromString("0-201-71012-0");
    var rgcName:Name := Name.newName("Robert G.", "Clark");
    var lbwName:Name := Name.newName("Leslie B.",
                                     "Wilson");
    var cplAuths:List[Name] :=
              List[Name].empty().add(rgcName).add(lbwName);
    var cplPub:String := "Addison Wesley";
    var cplPubDate:Date := Date.newDate(6,11,2000);
    var cplItemNo:integer := lib.getNewItemNo();
    var cplCopyNo:integer := lib.getNewCopyNo();
    // add the new book
    lib.acquireBook(cplTitle, cplISBN, cplAuths, cplPub,
                     cplPubDate);

    // check details of added book
    var cplItem:Item := lib.lookupItem(cplItemNo);
    assert cplItem is Book;
    var cplBk:Book := lib.lookupBook(cplItemNo);
    assert cplItem = cplBk;
    // check search functions
    assert lib.lookupItemFromTitle(cplTitle) = cplBk;
    assert lib.lookupBookFromISBN(cplISBN) = cplBk;
    var cplOnlyList:Collection[Book]
                   := Collection[Book].empty().add(cplBk);
    assert lib.lookupBooksFromAuthor(rgcName).size() = 1;
    assert lib.lookupBooksFromAuthor(rgcName)
                                    .contains(cplBk);
    var rgcBooks:Collection[Book]
                   := lib.lookupBooksFromAuthor(rgcName);
    assert lib.lookupBooksFromAuthor(rgcName)
                                      = cplOnlyList;
    assert lib.lookupBooksFromPublisher(cplPub)
                                      = cplOnlyList;
    assert lib.lookupBooksFromPubYear(cplPubDate.year())
                                      = cplOnlyList;
    // check details of copy
    assert lib.lookupCopy(cplCopyNo).copyNo() = cplCopyNo;
    assert lib.lookupCopy(cplCopyNo).itemNo() = cplItemNo;
    assert lib.lookupItemFromCopy(cplCopyNo) = cplBk;

    // change book details
    var cplTitle2:String :=
      "Comparative Programming Languages (Third Edition)";
    lib.changeItemTitle(cplItemNo, cplTitle2);
    assert lib.lookupBook(cplItemNo).title() = cplTitle2;
    var cplISBN2:ISBN := ISBN.fromString("9780201710120");
    lib.changeBookISBN(cplItemNo, cplISBN2);
    assert lib.lookupBook(cplItemNo).isbn() = cplISBN2;
    var cplAuths2:List[Name] :=
                         List[Name].empty().add(rgcName);
    lib.changeBookAuthors(cplItemNo, cplAuths2);
    assert lib.lookupBook(cplItemNo).authors() = cplAuths2;
    var cplPub2:String := "Addison-Wesley";
    lib.changeBookPub(cplItemNo, cplPub2);
    assert lib.lookupBook(cplItemNo).publisher() = cplPub2;
    var cplPubDate2:Date := Date.newDate(1,1,2001);
    lib.changeBookPubDate(cplItemNo, cplPubDate2);
    assert lib.lookupBook(cplItemNo).pubDate()
                                        = cplPubDate2;

    // change book copy status
    assert lib.lookupCopy(cplCopyNo).status()
                                      .isAvailable();
    assert !lib.lookupCopy(cplCopyNo).status().isLost();
    lib.changeCopyStatus(cplCopyNo, ItemStatus.lost());
    assert lib.lookupCopy(cplCopyNo).status().isLost();
    assert !lib.lookupCopy(cplCopyNo).status()
                                  .isAvailable();
    lib.changeCopyStatus(cplCopyNo,
                          ItemStatus.available());
    assert lib.lookupCopy(cplCopyNo).status()
                                  .isAvailable();
    assert !lib.lookupCopy(cplCopyNo).status().isLost();

    // delete book
    var oldCopiesCount:integer := lib.copies().size();
    lib.disposeOfCopy(cplCopyNo);
    assert !(exists (c:Copy in lib.copies()):
                   c.copyNo() = cplCopyNo);
    assert lib.copies().size() = oldCopiesCount - 1;
  }
```

The check viewer (shown below) shows us that all the assertions have been checked this time. To achieve this, we had to produce tests totalling 240 lines excluding comments. These cover every generated run-time assertion check at least once.



Check Viewer

```
Run-time Checking - 8 tests (8 passed), 1037 assertions (1037 passed) via 8036 checks
  Test Cases - 8 tests (8 passed)
    Class ISBN - 1 test (1 passed)
    Class Lib - 7 tests (7 passed)
      Test Test emptyTest: Passed
      Test Test loginTest: Passed
      Test Test bookTest: Passed
        Time Taken: 109 ms
      Test Test cdTest: Passed
        Time Taken: 63 ms
      Test Test dvdTest: Passed
        Time Taken: 47 ms
      Test Test memberTest: Passed
        Time Taken: 47 ms
      Test Test loanTest: Passed
        Time Taken: 156 ms
  Run-time Checks - 1037 assertions (1037 passed)
    Class Book - 11 assertions (11 passed)
    Class Copy - 1 assertion (1 passed)
    Class Date - 21 assertions (21 passed)
    Class DVD - 9 assertions (9 passed)
    Class ISBN - 15 assertions (15 passed)
```

8.3.4 Errors detected

As you would expect, some errors were made in the specification and implementation of the library system by the student. In this section we will discuss some of the errors that the run-time assertion checking process enabled us to uncover.

Missing olds

The most common error that was detected by the run-time assertion checking process was missing `old` operators in `ensures` clauses of operations. For example, consider the specification of the `disposeOfCopy` operation as originally produced by the student.

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

This operation should remove the `Copy` with the specified number and, if it is the last `Copy`, delete the associated `Item`. On run-time assertion checking this operation, we get the following error reported where line 248 is the location of the closing curly bracket at the end of the `disposeOfCopy` operation.

```
   ** Executing bookTest test...
C:\Program
     Files\eclipse\workspace\OmnibusIDE\myprojects\Impl4-
     rac\Lib.obs:248: Failure of public requires clause of the
     lookupCopy function declared in Lib at line 780
Source:
     exists ((c:Copy) in copies()): c.copyNo() = copyNo
Attributes:
     members: {omni.lang.Collection: [Member: memberNo: 2,
     name: [Name: fName: Kate, sName: Brown], address1: 1A
     London Road, address2: Stirling, postcode: FK9 1BB,
     telNo: 1786477557, isAdult: true, isStaff: true, status:
     [MemberStatus: value: 0], fineDue: 0, pinNo: 5555],
     [Member: memberNo: 1, name: [Name: fName: Joe, sName:
     Bloggs], address1: 17 Queen Street, address2: Stirling,
     postcode: FK9 1AA, telNo: 1786466666, isAdult: true,
     isStaff: false, status: [MemberStatus: value: 0],
     fineDue: 0, pinNo: 5555]}
     loans: {omni.lang.Collection: }
     copies: {omni.lang.Collection: [Copy: copyNo: 1, itemNo:
     1, status: [ItemStatus: value: 1]]}
     items: {omni.lang.Collection: [Book: isbn: [ISBN: value:
     01-303-23-772], authors: {omni.lang.List: [Name: fName:
     Doug, sName: Bell]}, publisher: Prentice Hall, pubDate:
     [Date: day: 1, month: 7, year: 2001], itemNo: 1, title:
     Java For Students]}
     cards: {omni.lang.Collection: [Card: cardNo: 1, memberNo:
      1]}
Parameters:
     copyNo: 2
Call stack:
  at Lib.disposeOfCopy(Lib:248)
  at Lib.test_bookTest(Lib:1307)
       -- Test aborted after 93 ms
```

The functions `isOnlyOneCopy` and `lookupCopy` are used in the condition of the `if` expression in the `ensures` clause. The intention was to assert that if there was only one copy of the associated `Item` before the operation then the size of the `items` collection is reduced by one; otherwise it stays the same. However, the calls of `isOnlyOneCopy` and `lookupCopy`, without an `old` operator prefix, refer to the library after the operation, not before it. Thus, the evaluation of `lookupCopy` with the specified `copyNo` after the operation will result in a violation of its `requires` clause since the `Copy` with that `copyNo` will have been removed by the end of the method. Instead, the user should have used the `old` operator before both of these function calls to indicate that these functions should have been evaluated at the start of the implementation. The `ensures` should have been written as follows:

```
     ensures copies().size() = old copies().size() - 1,
      if (old isOnlyOneCopy(
                    old lookupCopy(copyNo).itemNo())) then
       items().size() = old items().size() - 1
      else
       items() = old items()
```

This alteration allows the operation to be successfully run-time assertion checked.

Genuine mistakes

There were a number of conventional mistakes that were made in the specification of the library system and picked up by the run-time assertion checker. A good example is from the `acquireBook` operation.

```
  public operation acquireBook(title:String, isbn:ISBN,
      authors:List[Name], pub:String, pubDate:Date)
     changes items, copies
     ensures
```

```
    if (forall (i:Item in old items() where i is Book):
               ((i as Book).title() != title
                 && (i as Book).authors() != authors)) then
     items().size() = old items().size() + 1
     && copies().size() = old copies().size() + 1
    else
     copies().size() = old copies().size() + 1
     && items() = old items()
  {
    var countOfCopies:integer := 0;
    foreach (i:Item in items){
     if (i is Book){
      var b:Book := i as Book;
      if (b.title() = title && b.authors() = authors){
        countOfCopies := countOfCopies + 1;
      }
     }
    }
    if (countOfCopies = 0){
     items.add(Book.newBook(getNewItemNo(), title, isbn,
                              authors, pub, pubDate));
    }
    copies.add(Copy.newCopy(getNewCopyNo(),
        lookupItemFromTitle(title).itemNo()));

  }
```

Again, `items()` was originally used instead of **`old`** `items()` but we have made that correction first. Even after that correction, the run-time assertion checker reports a failure of the `ensures` clause at the end of the operation.

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

```
   ** Executing bookTest test...
C:\Program
     Files\eclipse\workspace\OmnibusIDE\myprojects\Impl4-
     rac\Lib.obs:172: Failure of public ensures clause of the
     acquireBook operation declared in Lib at line 145
Source:
     if (forall ((i:Item) in old items() where i is Book): ((i
     as Book).title() != title && (i as Book).authors() !=
     authors)) then (items().size() = old items().size() + 1
     && copies().size() = old copies().size() + 1) else
```

```
      (copies().size() = old copies().size() + 1 && items() =
      old items())
Attributes:
     members: {omni.lang.Collection: [Member: memberNo: 2,
      name: [Name: fName: Kate, sName: Brown], ...], [Member:
      memberNo: 1, name: [Name: fName: Joe, sName: Bloggs],
      ...]}
     loans: {omni.lang.Collection: }
     copies: {omni.lang.Collection: [Copy: copyNo: 3, itemNo:
      3, ...], [Copy: copyNo: 2, itemNo: 2, ...], [Copy:
      copyNo: 1, itemNo: 1, ...]}
     items: {omni.lang.Collection: [Book: authors:
      {omni.lang.List: [Name: fName: Robert G., sName: Clark]},
      title: Programming in Ada: A First Course, ...], [Book:
      authors: {omni.lang.List: [Name: fName: Robert G., sName:
      Clark]}, title: Comparative Programming Languages (Third
      Edition), ...], [Book: authors: {omni.lang.List: [Name:
      fName: Doug, sName: Bell]}, title: Java For Students,
      ...]}
     cards: {omni.lang.Collection: [Card: cardNo: 1, memberNo:
      1]}
Old Attributes:
     old members: {omni.lang.Collection: [Member: memberNo: 2,
      name: [Name: fName: Kate, sName: Brown], ...], [Member:
      memberNo: 1, name: [Name: fName: Joe, sName: Bloggs],
      ...]}
     old loans: {omni.lang.Collection: }
     old copies: {omni.lang.Collection: [Copy: copyNo: 2,
      itemNo: 2, ...], [Copy: copyNo: 1, itemNo: 1, ...]}
     old items: {omni.lang.Collection: [Book: authors:
      {omni.lang.List: [Name: fName: Robert G., sName: Clark]},
      title: Comparative Programming Languages (Third Edition),
      ...], [Book: authors: {omni.lang.List: [Name: fName:
      Doug, sName: Bell]}, title: Java For Students, ...]}
     old cards: {omni.lang.Collection: [Card: cardNo: 1,
      memberNo: 1]}
Parameters:
     title: Programming in Ada: A First Course
     pubDate: [Date: day: 31, month: 5, year: 1985]
     isbn: [ISBN: value: 9780521257282]
     authors: {omni.lang.List: [Name: fName: Robert G., sName:
      Clark]}
     pub: Cambridge University Press
Call stack:
  at Lib.acquireBook(Lib:172)
  at Lib.test_bookTest(Lib:1312)
       -- Test aborted after 109 ms
```

The implementation iterates through the `items` collection and counts the number of `Book`s that have the specified title and authors. If this is zero, it adds a new `Item` and then in all cases it adds a new `Copy`.

The ensures clause consists of an if expression. The condition checks whether all the Books have a different title and list of authors from those passed to acquireBook. This is inconsistent with the implementation which counts the number of Books whose title and list of authors are the same as those passed. Instead, the condition of the if expression in the ensures clause should check that either the title <u>or</u> the authors are different from those passed. The corrected ensures clause is shown below.

```
  ensures
    if (forall (i:Item in old items() where i is Book):
              ((i as Book).title() != title
                || (i as Book).authors() != authors)) then
     items().size() = old items().size() + 1
     && copies().size() = old copies().size() + 1
    else
     copies().size() = old copies().size() + 1
     && items() = old items()
```

After this correction, the operation can be run-time assertion checked without errors.

This error was only picked up because of the comprehensiveness of the bookTest test case. The inconsistency between the quantification in the condition of the if expression and the implementation is only exposed when a new Book is added with the same title but different authors or same authors but different title. It would have been easy to have missed this scenario in our test case, in which circumstance the error would have gone undetected by the run-time assertion checking process.

8.3.5 Conclusions

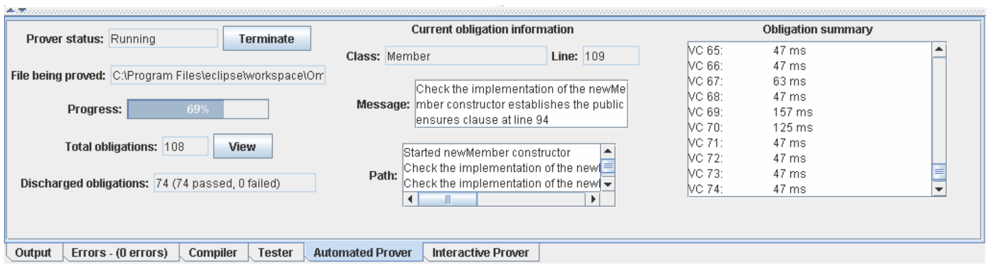
The student was able to produce an implementation of the described library system in Omnibus and we were able to use run-time assertion checking to verify it. This allowed us to identify a range of errors in the original application. However, as we will see in later sections, there are other errors which the run-time assertion checking approach was not able to detect.

8.4 Extended static checking

Next we will consider the use of the extended static checking approach to check the library project. We will take as our starting point the Omnibus implementation with lightweight specifications produced in the previous section. We will see that ESC allows us to uncover errors that were not detected by RAC.

8.4.1 Using the ESC tool

ESC tools are straightforward to use: the programmer simply selects the ESC policy in the Policy Selector and then clicks ‘Verify Project’. The tool generates appropriate theories in the generic logic and then invokes the automated Simplify prover. While the prover is running, its progress can be tracked via the Prover pane.



Prover status: Running
File being proved: C:\Program Files\eclipse\workspace\Om
Progress: 69%
Total obligations: 108
Discharged obligations: 74 (74 passed, 0 failed)

Current obligation information
Class: Member
Line: 109
Message: Check the implementation of the newMember constructor establishes the public ensures clause at line 94
Path:
```
Started newMember constructor
Check the implementation of the newI
Check the implementation of the newI
```

Obligation summary
```
VC 65:     47 ms
VC 66:     47 ms
VC 67:     63 ms
VC 68:     47 ms
VC 69:     157 ms
VC 70:     125 ms
VC 71:     47 ms
VC 72:     47 ms
VC 73:     47 ms
VC 74:     47 ms
```

8.4.2 New errors detected

The ESC tool uncovers a range of errors and omissions which were not detected by the run-time assertion checking process.

Missing private specifications for public model functions

In the specifications produced for the RAC approach, we did not give private specifications for the public model functions. These were not needed to run-time check the classes and so in accordance with the freedoms of lightweight specifications we were allowed to omit them. For example, the public model functions for the `Item` class were declared as follows:

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

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

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

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

On attempting to ESC-check this file we get the following error:

```
C:\Program
      Files\eclipse\workspace\OmnibusIDE\myprojects\Impl4-
      esc\Item.obs:23: Unable to verify the implementation of
      the setTitle operation respects the public changes clause
      at line 20
Attributes:
     title: t_0
     itemNo: itemNo_0
Old Attributes:
     title: title_0
     itemNo: itemNo_0
Parameters:
     t: t_0
Knowledge:
     {itemNo_0:integer},
     {title_0:String},
     {t_0:String},
     true
Assertion to check (unevaluated):
     old itemNo() = itemNo()
Assertion to check (evaluated):
     this{itemNo := itemNo_0, title := title_0}.itemNo() =
      this{itemNo := itemNo_0, title := t_0}.itemNo()
Path:
     Started setTitle operation
     Check the implementation of the setTitle operation
      respects the public changes clause at line 20
1 error
```

The problem is that the ESC tool is unable to deduce the relationship between the `itemNo()` function and the `itemNo` attribute because there is no specification of this relationship. Remember that our ESC tool uses modular checking and can only reason about other methods using their specifications. The relationship between the model function and the attribute is clear from the trivial implementation of the function, but the tool cannot refer to this from within the `setTitle` operation.

To address this problem, we must add a private specification to the public model functions expressing the relationship between the public model functions and the private attributes. We could adjust the definitions of the public model functions in `Item` to be the following:

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

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

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

Missing requires clauses

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

Consider the `Book` class as implemented by the student and RAC-checked. It is declared with an invariant that there are no duplicates in the authors.

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

Two methods have the ability to change the authors: the `newBook` constructor and the `setAuthors` operation.

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

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

If we try to ESC-check this file we get the following errors:

```
C:\Program
     Files\eclipse\workspace\OmnibusIDE\myprojects\Impl4-
     esc\Book.obs:51: Unable to verify the implementation of
     the newBook constructor establishes the public invariant
     at line 35
...
C:\Program
     Files\eclipse\workspace\OmnibusIDE\myprojects\Impl4-
     esc\Book.obs:68: Unable to verify the implementation of
     the setAuthors operation maintains the public invariant
     at line 35
...
2 errors
```

The problem is that the `author` lists that are passed into the `newBook` and `setAuthors` methods are not constrained to not contain duplicates. The ESC tool correctly picks up this omission.

We can address the problem by adding the following `requires` clause to both the `newBook` constructor and `setAuthors` operation.

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

The RAC approach was not able to detect this missing `requires` clause. The first reason for this was that the associated test cases were written to test correct execution for expected values, not graceful failure for invalid inputs. We did not define a test case to check the rejection of the addition of a Book with duplicate authors. Let us consider what would have happened if we had defined such a test. We could allow for the addition of a second book by RGC into `bookTest` just before the deletion of the first book.

```
test bookTest {
    ...

    // add second book by RGC
    var paTitle:String
                  := "Programming in Ada: A First Course";
    var paISBN:ISBN := ISBN.fromString("9780521257282");
    var paAuths:List[Name] :=
                      List[Name].empty().add(rgcName);
    var paPub:String := "Cambridge University Press";
    var paPubDate:Date := Date.newDate(31,5,1985);
    var paItemNo:integer := lib.getNewItemNo();
    var paCopyNo:integer := lib.getNewCopyNo();
    // add the new book
    lib.acquireBook(paTitle, paISBN, paAuths, paPub,
                      paPubDate);

    // delete book
    ...
}
```

Without the `requires` clause in the newBook constructor we would get the following error when executing the test case:

```
   ** Executing bookTest test...
C:\Program
     Files\eclipse\workspace\OmnibusIDE\myprojects\Impl4-
     rac\Book.obs:36: Failure of public invariant declared in
     Book at line 24 at end of newBook constructor
Source:
     !authors().containsDuplicates()
Parameters:
     auths: {omni.lang.List: [Name: fName: Robert G., sName:
     Clark], [Name: fName: Robert G., sName: Clark]}
     iNo: 3
     isbnNo: [ISBN: value: 9780521257282]
     t: Programming in Ada: A First Course
     pub: Cambridge University Press
     pDate: [Date: day: 31, month: 5, year: 1985]
Call stack:
  at Book.newBook(Book:36)
  at Lib.acquireBook(Lib:165)
  at Lib.test_bookTest(Lib:1312)
      -- Test aborted after 78 ms.
```

So, we get a failure report for the invariant at the end of the `newBook` constructor. Since this is not a `requires` clause failure, this should indicate an error by the implementer of the `newBook` constructor. However, it is not an error in the implementation, it is the omission of a suitable `requires` clause. This is a much more awkward way to detect missing `requires` clauses than the ESC tool.

Truisms in ensures clauses

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

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

This was obviously an entry error and what they intended was:

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

The later specification asserts that the `fineDue` function is equal to the passed `fine` value after the operation whereas the former specification simply asserts that the `fineDue` function is equal to itself, which tells us nothing useful about its value.

However, the error was not picked up by the run-time assertion checker since `fineDue() = fineDue()` always evaluates to true and so no assertion failure is ever triggered in the execution of the test cases. Error detection within run-time assertion checking is built upon the failure of assertion checks.

The same problem occurs in the ESC-checking of the `Member` class. The `ensures` clause is checked at the end of the implementation of the `changeFineDue` operation and it evaluates to true, as it would regardless of what the implementation of the method did.

Unlike in the RAC-checking, the error is picked up though when we check the `Lib` class, specifically the `changeMemberFine` operation. The lightweight specification of this operation asserts that the `fineDue` for the specified `Member` should be set to the passed value. This is implemented via a call of the `changeFineDue` operation in the body of the method.

```
public operation changeMemberFine(memberNo:integer,
   fineDue:integer)
  requires exists (m:Member in members()):
              m.memberNo() = memberNo
  changes members
  ensures members().size() = old members().size(),
   lookupMember(memberNo).fineDue() = fineDue
{
  var memberObj:Member := lookupMember(memberNo);
  members.remove(memberObj);
  memberObj.changeFineDue(fineDue);
  members.add(memberObj);
}
```

As always, when reasoning about another method in ESC, the specification of that method is used. Here the specification of the `changeFineDue` operation is used to reason about how `memberObj` is changed. But that specification does not give us the information we require to prove that the `fineDue` function of `memberObj` is changed to `fineDue`. Thus, an error is reported, picking up the mistake in the `changeFineDue` operation.

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

8.4.3 Limitations of the ESC tool

The Omnibus ESC tool is not able to verify the `Lib` and `Date` classes. If the programmer attempts to verify these files then the prover hangs. The specification of the `Lib` class is too sophisticated to be ESC-checked. The problem with the `Date` class is that it was not completely specified.

8.4.4 Conclusions

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

8.5 Full formal verification

Finally, we consider the full formal verification of the library project. We will see that FFV discovers further issues that even ESC did not detect.

Instead of considering an implementation with lightweight specifications, in this section we consider a heavyweight specification. Using RAC and ESC, the behaviour of methods can be described using very incomplete `ensures` clauses. Requirements such as `invariant`s are verified separately at the end of each constructor/operation. In FFV, the behaviour of methods should be described more completely in the `ensures` clauses. Crucially, in our FFV approach the requirements must follow from the behaviour specifications. This helps ensure a greater level of completeness in the specifications.

8.5.1 Automatic and interactive FFV

Full formal verification can be performed using either an automated or interactive theorem prover.

FFV with an automated prover

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

The verification of the central `Lib` class with the automated prover is less successful. The first 7 VCs corresponding to the proofs of each of the invariants for the `empty` constructor are proved in a total of just over 4 seconds, but then the prover hangs whilst trying to prove the main operations. This is not an unexpected result and is consistent with previous attempts to use the Simplify prover for full formal verification projects.

FFV with an interactive prover

To fully verify the main `Lib` class we must use an interactive prover. The Omnibus IDE uses PVS as its interactive prover.

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

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

The `memberNo` for the new `Member` is calculated using the `getNewMemberNo` function. The specification of this function simply states that it returns an integer which is not being used as the `memberNo` of any `Member` in the `members` collection.

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

Now let us consider the verification of the invariants. We can assume that the invariants hold before the operation is called and we must demonstrate that they hold at the end of the operation. The `changes` clause specified that only the public `members` model function is changed, so any invariant which does not refer to `members` will be trivially maintained because its truth will not be altered by the method. This covers the 5 of the 7 invariants from the `Lib` class, which are repeated below.

```
  public invariant itemNosInItemsUnique:
    forall (i:Item in items()):
     forall (i2:Item in items()):
      i.itemNo() != i2.itemNo() || i = i2
  public invariant copyNosInCopiesUnique:
    forall (c:Copy in copies()):
     forall (c2:Copy in copies()):
      c.copyNo() != c2.copyNo() || c = c2
  public invariant cardNosInCardsUnique:
    forall (c:Card in cards()):
     forall (c2:Card in cards()):
      c.cardNo() != c2.cardNo() || c = c2
  public invariant allCopiesOnLoanInCopies:
     forall (l:Loan in loans()):
      exists (c:Copy in copies()):
        l.copyNo() = c.copyNo()
  public invariant noCopiesOnLoanMoreThanOnce:
    forall (l:Loan in loans()):
     forall (l2:Loan in loans()):
      l.copyNo() != l2.copyNo() || l = l2
```

Next, we have the invariant that for all `memberNo`s referred to in an entry in the `loans` collection there is a `Member` with that `memberNo` in the `members` collection. This is also relatively trivial to demonstrate since we have made an addition to the `members` collection, and hence all elements that were previously in the `members` collection will still be in it.

```
  public invariant allMembersWithLoansInMembers:
    forall (l:Loan in loans()):
      exists (m:Member in members()):
        l.memberNo() = m.memberNo()
```

Finally, consider the invariant that there are no two `Member`s in the `members` collection with the same `memberNo`. This is slightly more complicated and relies on the specification of the `getNewMemberNo` function. The `getNewMemberNo` function returns an integer value which is not used as the `memberNo` of any of the `Member`s in the `members` collection before the operation is called, so it should be allowable to add a new `Member` with that `memberNo`.

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

As an illustration of the PVS proof process, the verification of this invariant is worked through in detail in appendix E.

8.5.2 New errors detected

The full formal verification process requires more sophisticated specifications, leading to detection of further errors and omissions.

Missing ensures clauses

As part of the ESC process, we had to add `requires` clauses to the `newBook` and `setAuthors` methods in order to ensure the invariant. However, we did not have to provide `ensures` clauses since the invariant was checked at the end of the implementation of each method. The specifications sufficient to ESC-check the `Book` class were:

```
  public constructor newBook(iNo:integer, t:String,
     isbnNo:ISBN,  auths:List[Name], pub:String, pDate:Date)
    requires !auths.containsDuplicates()

  public operation setAuthors(auths:List[Name])
    requires !auths.containsDuplicates()
    changes authors
```

When using FFV, we must verify that the specification alone is sufficient to verify the invariant. If we attempt to verify this class using automated FFV, we get the following errors:

```
C:\Program
     Files\eclipse\workspace\OmnibusIDE\myprojects\Impl4-
     ffv\Book.obs:13: Unable to verify the public behaviour of
     the newBook constructor satisfies the public invariant at
     line 7
C:\Program
     Files\eclipse\workspace\OmnibusIDE\myprojects\Impl4-
     ffv\Book.obs:27: Unable to verify the public behaviour of
     the setAuthors operation satisfies the public invariant
     at line 7
2 errors
```

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

```
  public constructor newBook(iNo:integer, t:String,
     isbnNo:ISBN,  auths:List[Name], pub:String, pDate:Date)
    requires !auths.containsDuplicates()
    ensures itemNo() = iNo,
     title() = t,
     isbn() = isbnNo,
     authors() = auths,
     publisher() = pub,
     pubDate() = pDate

  operation setAuthors(auths:List[Name])
    requires !auths.containsDuplicates()
    changes authors
    ensures authors() = auths
```

On the addition of these assertions, the file can be automatically verified.

Ensures clauses not strong enough

We have already made a number of corrections to the `acquireBook` operation. Using RAC we detected missing `old` operators in the `ensures` clause and a mistake in the checking for an existing instance of the `Book`. Using ESC we detected a missing `requires` clause. However, there is still a remaining error in the heavyweight specification produced by the student. This specification with all the existing corrections is shown below.

```
public operation acquireBook(title:String, isbn:ISBN,
     authors:List[Name], pub:String, pubDate:Date)
  requires !authors.containsDuplicates()
  changes items, copies
  ensures if (forall (i:Item in old items()
                                            where i is Book):
              (i.title() != title
                || (i as Book).authors() != authors)) then
     items() = old items().add(
                   Book.newBook(old getNewItemNo(), title,
                           isbn, authors, pub, pubDate))
     && copies() = old copies().add(
              Copy.newCopy(old getNewCopyNo(),
                old lookupItemFromTitle(title).itemNo()))
   else
     copies() = old copies().add(
              Copy.newCopy(old getNewCopyNo(),
                old lookupItemFromTitle(title).itemNo()))
```

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

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

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

The problem can be addressed by the addition of an explicit assertion that `items` does not change in the else part of the existing assertion.

```
   else
     copies() = old copies().add(
                Copy.newCopy(old getNewCopyNo(),
                  old lookupItemFromTitle(title).itemNo()))
     && items() = old items()
```

This permits the invariant to be verified.

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

8.5.3 Conclusions

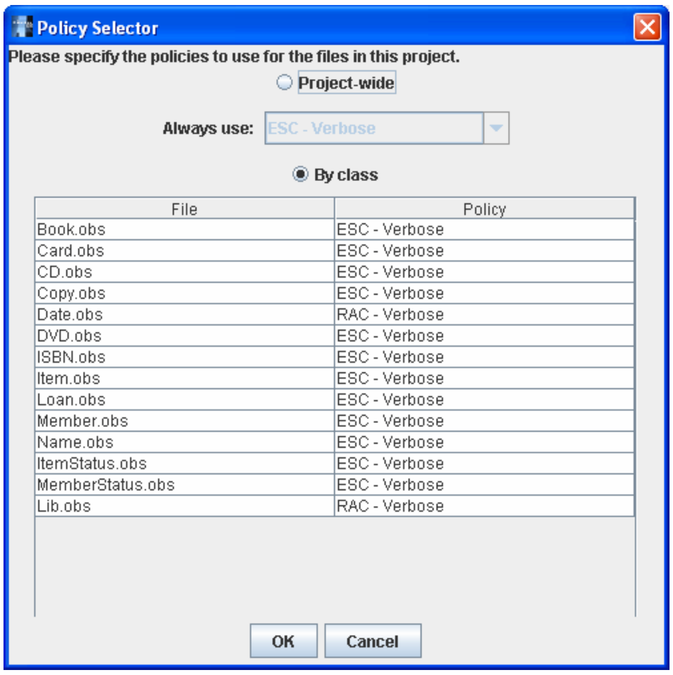
FFV allows the specifications of the project to be improved still further. Richer specifications can be used since the user can guide the PVS prover through proofs that are beyond the automated Simplify prover. However, the costs are much higher. Appendix E describes the proof of a single VC. This proof took considerable time and mathematical ingenuity and would be beyond many users.

8.6 Integrated verification

In the preceding sections we considered the use of the different approaches, in turn, to verify the entirety of the library project. In this section we consider how we can use the different verification approaches offered by Omnibus in a more integrated fashion. We use the example to demonstrate how the different features of the integrated support can be used to fully exploit the integration.

8.6.1 Using different policies for different files

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



Please specify the policies to use for the files in this project.

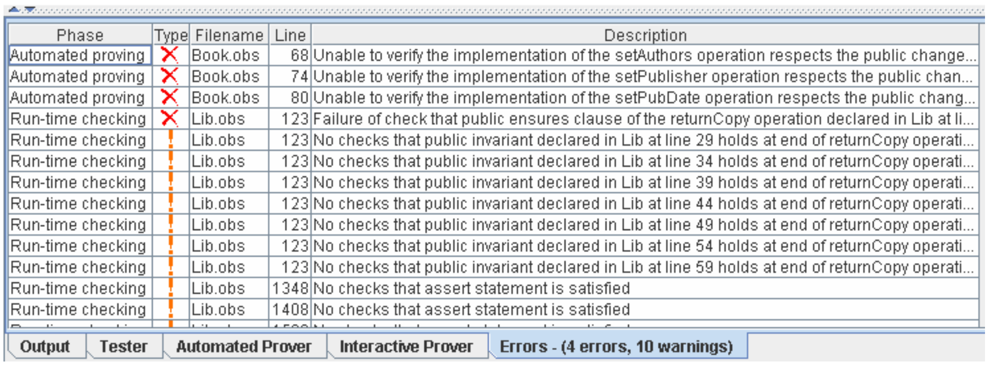
○ Project-wide

Always use: ESC - Verbose

● By class

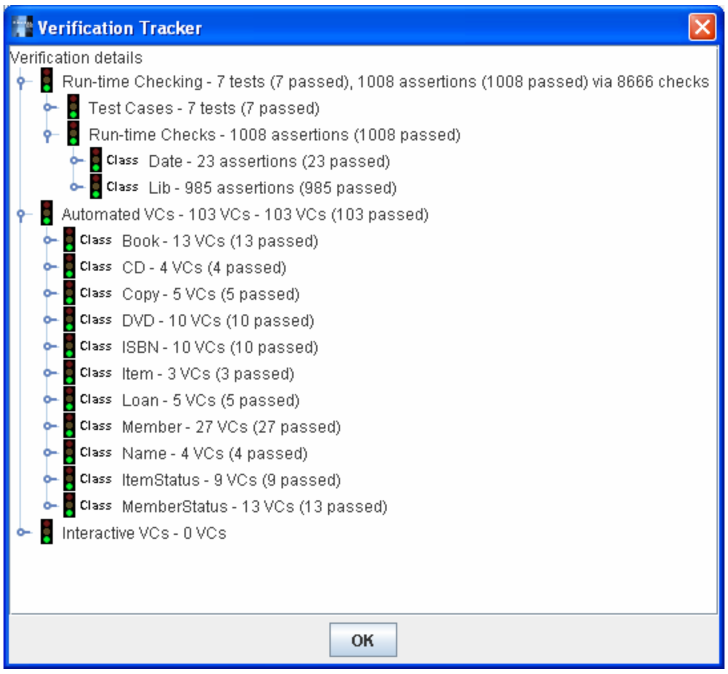
File	Policy
Book.obs	ESC - Verbose
Card.obs	ESC - Verbose
CD.obs	ESC - Verbose
Copy.obs	ESC - Verbose
Date.obs	RAC - Verbose
DVD.obs	ESC - Verbose
ISBN.obs	ESC - Verbose
Item.obs	ESC - Verbose
Loan.obs	ESC - Verbose
Member.obs	ESC - Verbose
Name.obs	ESC - Verbose
ItemStatus.obs	ESC - Verbose
MemberStatus.obs	ESC - Verbose
Lib.obs	RAC - Verbose

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



Phase	Type	Filename	Line	Description
Automated proving	✗	Book.obs	68	Unable to verify the implementation of the setAuthors operation respects the public change...
Automated proving	✗	Book.obs	74	Unable to verify the implementation of the setPublisher operation respects the public chan...
Automated proving	✗	Book.obs	80	Unable to verify the implementation of the setPubDate operation respects the public chang...
Run-time checking	✗	Lib.obs	123	Failure of check that public ensures clause of the returnCopy operation declared in Lib at li...
Run-time checking	!	Lib.obs	123	No checks that public invariant declared in Lib at line 29 holds at end of returnCopy operati...
Run-time checking	!	Lib.obs	123	No checks that public invariant declared in Lib at line 34 holds at end of returnCopy operati...
Run-time checking	!	Lib.obs	123	No checks that public invariant declared in Lib at line 39 holds at end of returnCopy operati...
Run-time checking	!	Lib.obs	123	No checks that public invariant declared in Lib at line 44 holds at end of returnCopy operati...
Run-time checking	!	Lib.obs	123	No checks that public invariant declared in Lib at line 49 holds at end of returnCopy operati...
Run-time checking	!	Lib.obs	123	No checks that public invariant declared in Lib at line 54 holds at end of returnCopy operati...
Run-time checking	!	Lib.obs	123	No checks that public invariant declared in Lib at line 59 holds at end of returnCopy operati...
Run-time checking	!	Lib.obs	1348	No checks that assert statement is satisfied
Run-time checking	!	Lib.obs	1408	No checks that assert statement is satisfied

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

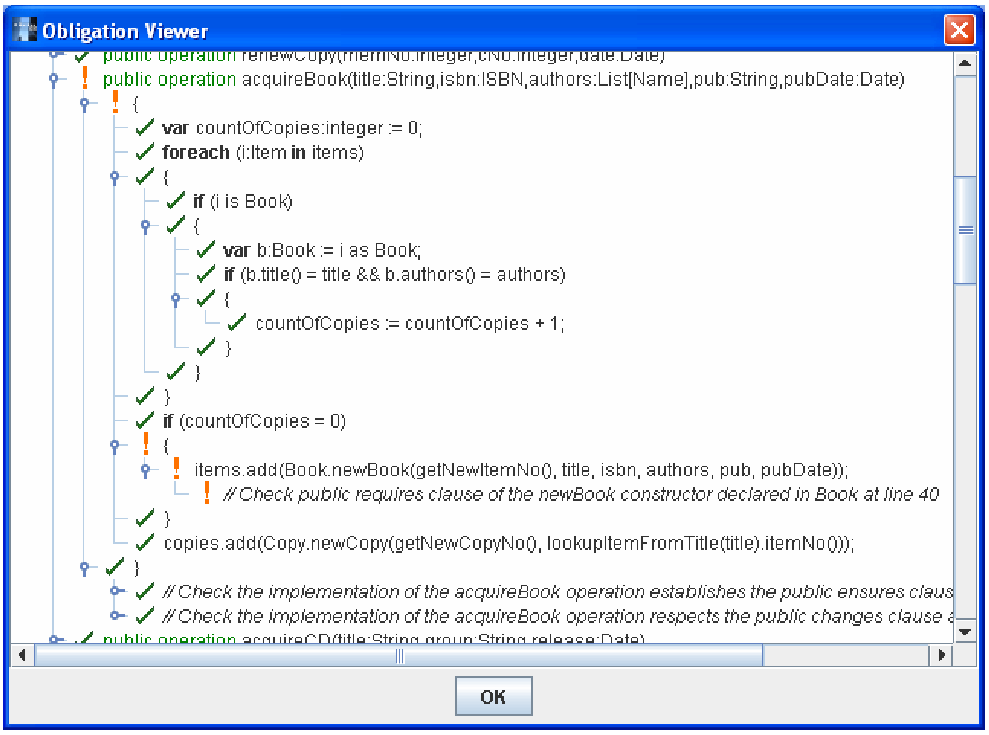


Verification details
- Run-time Checking - 7 tests (7 passed), 1008 assertions (1008 passed) via 8666 checks
  - Test Cases - 7 tests (7 passed)
  - Run-time Checks - 1008 assertions (1008 passed)
    - Class Date - 23 assertions (23 passed)
    - Class Lib - 985 assertions (985 passed)
- Automated VCs - 103 VCs - 103 VCs (103 passed)
  - Class Book - 13 VCs (13 passed)
  - Class CD - 4 VCs (4 passed)
  - Class Copy - 5 VCs (5 passed)
  - Class DVD - 10 VCs (10 passed)
  - Class ISBN - 10 VCs (10 passed)
  - Class Item - 3 VCs (3 passed)
  - Class Loan - 5 VCs (5 passed)
  - Class Member - 27 VCs (27 passed)
  - Class Name - 4 VCs (4 passed)
  - Class ItemStatus - 9 VCs (9 passed)
  - Class MemberStatus - 13 VCs (13 passed)
- Interactive VCs - 0 VCs

All the classes were translated into Java and compiled to bytecode, but only `Lib` and `Date` had their assertions translated into run-time checks because only those classes used a policy with run-time checks enabled. The ESC policy used by the other classes incorporates no run-time checks.

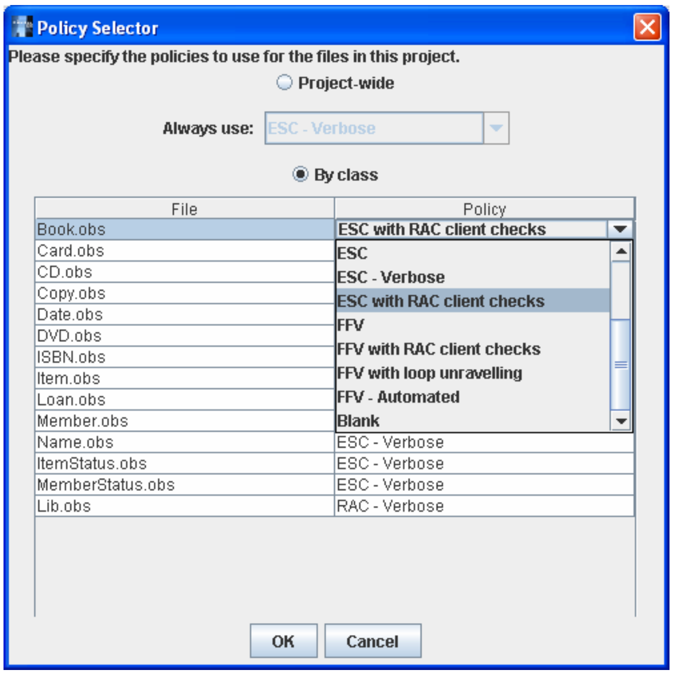
This absence of run-time pre-condition checks violates guideline 3 from section 5.2.2 and causes a clash between the approaches. The tests used to run-time check the assertions in the `Lib` class use instances of the other classes which were statically verified and have no run-time pre-condition checks. This means the tests may be silently violating the pre-conditions of the other classes, invalidating their verification which was based on the assumption that the pre-conditions of their methods are respected by calls.

Consider the `acquireBook` operation in the `Lib` class as an example. There are correctness obligations for the `changes` and `ensures` clauses at the end of the method body and these have been checked via the generated run-time checks for the `Lib` class. However, there is also a correctness obligation that the pre-condition of the `newBook` constructor is respected when the new `Book` object is created. This has not been checked because the ESC policy was used for the `Book` class which has no run-time assertion checks at all. So, the tool is right to present a warning as can be seen by the exclamation mark next to the relevant statement in the screenshot below.



```
✓ public operation renewCopy(memNo:integer,cNo:integer,date:Date)
! public operation acquireBook(title:String,isbn:ISBN,authors:List[Name],pub:String,pubDate:Date)
  ! {
    ✓ var countOfCopies:integer := 0;
    ✓ foreach (i:Item in items)
    ✓ {
      ✓ if (i is Book)
      ✓ {
        ✓ var b:Book := i as Book;
        ✓ if (b.title() = title && b.authors() = authors)
        ✓ {
          ✓ countOfCopies := countOfCopies + 1;
        ✓ }
      ✓ }
    ✓ }
    ✓ if (countOfCopies = 0)
    ! {
      ! items.add(Book.newBook(getNewItemNo(), title, isbn, authors, pub, pubDate));
        ! // Check public requires clause of the newBook constructor declared in Book at line 40
    ✓ }
    ✓ copies.add(Copy.newCopy(getNewCopyNo(), lookupItemFromTitle(title).itemNo()));
  ✓ }
  ✓ // Check the implementation of the acquireBook operation establishes the public ensures claus
  ✓ // Check the implementation of the acquireBook operation respects the public changes clause a
✓ public operation acquireCD(title:String,group:String,release:Date)
```

To address this problem we should follow guideline 2 and generate run-time checks of the pre-conditions of the `Book` class since we use it in an RAC-verified class. We can do this by changing the verification policy for the `Book` class to be ‘ESC with RAC client checks’. This policy includes run-time checks of the pre-conditions as well as statically verifying the implementation. The change is made via the Policy Selector, as shown below.



Please specify the policies to use for the files in this project.

Project-wide

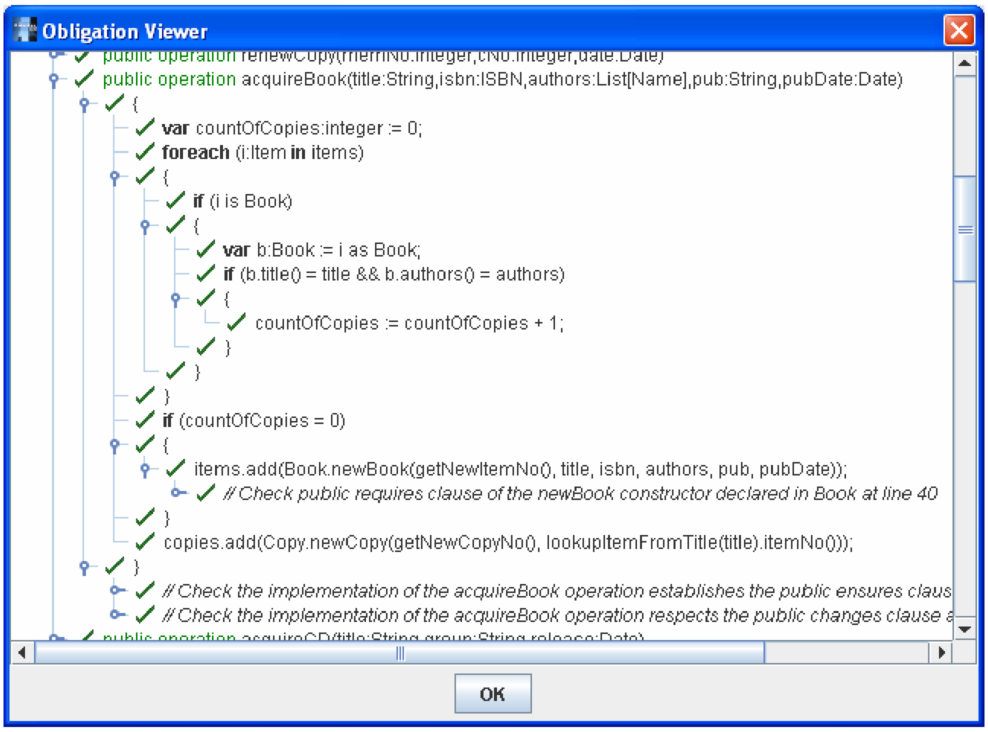
Always use: ESC - Verbose

[✓ By class]

File	Policy
Book.obs	ESC with RAC client checks
Card.obs	
CD.obs	
Copy.obs	
Date.obs	
DVD.obs	
ISBN.obs	
Item.obs	
Loan.obs	
Member.obs	
Name.obs	ESC - Verbose
ItemStatus.obs	ESC - Verbose
MemberStatus.obs	ESC - Verbose
Lib.obs	RAC - Verbose

ESC
ESC - Verbose
ESC with RAC client checks
FFV
FFV with RAC client checks
FFV with loop unravelling
FFV - Automated
Blank

The pre-condition of the `newBook` constructor will now be checked when the call is made. This provides justification for the correctness obligation when the project is re-verified. Note how in the screenshot below, the exclamation mark has been replaced by a tick.



```
✓ public operation renewCopy(memNo:integer,cNo:integer,date:Date)
✓ public operation acquireBook(title:String,isbn:ISBN,authors:List[Name],pub:String,pubDate:Date)
    ✓ {
        ✓ var countOfCopies:integer := 0;
        ✓ foreach (i:Item in items)
        ✓ {
            ✓ if (i is Book)
            ✓ {
                ✓ var b:Book := i as Book;
                ✓ if (b.title() = title && b.authors() = authors)
                ✓ {
                    ✓ countOfCopies := countOfCopies + 1;
                ✓ }
            ✓ }
        ✓ }
        ✓ if (countOfCopies = 0)
        ✓ {
            ✓ items.add(Book.newBook(getNewItemNo(), title, isbn, authors, pub, pubDate));
                ✓ // Check public requires clause of the newBook constructor declared in Book at line 40
        ✓ }
        ✓ copies.add(Copy.newCopy(getNewCopyNo(), lookupItemFromTitle(title).itemNo()));
    ✓ }
        ✓ // Check the implementation of the acquireBook operation establishes the public ensures claus
        ✓ // Check the implementation of the acquireBook operation respects the public changes clause a
✓ public operation acquireCD(title:String,group:String,release:Date)
```

8.6.2 Fully integrating the verification approaches

Special cases and targeted tests can be used to support higher levels of integration between the different approaches. This section describes how these facilities can be used to fulfil a number of purposes. A summary of the uses we consider is given in the following table. The contents of the table are described in the rest of this section.

Policy for class	**Feature**	**Purpose**
RAC	ESC/FFV special case	Statically verify to omit non RAC-compatible obligations
"	"	Statically verify pre-condition of method without any run-time checks
"	ESC targeted test	Check ESC-compatibility
ESC	RAC special case and RAC targeted test	Run-time checking non ESC-compatible obligations
"	FFV special case	Interactive proof of non ESC-compatible obligation
Automated FFV	FFV special case	Interactive proof of non ESC-compatible obligation (not discussed)
ESC/FFV	RAC targeted test	Check presence and RAC-compatibility of run-time pre-condition checks
FFV	ESC targeted test	Check ESC-compatibility

Using static verification special cases to omit non RAC-compatible obligations

Policy for class: RAC

Feature used: ESC/FFV special case

Certain obligations in an RAC-checked project may not be RAC-compatible. They may use constructs that cannot be translated into run-time checks or may be too costly to evaluate. Assertions in methods that are called frequently can often be too costly to run-time check. So it may be worthwhile to statically verify particular obligations in an RAC-checked project.

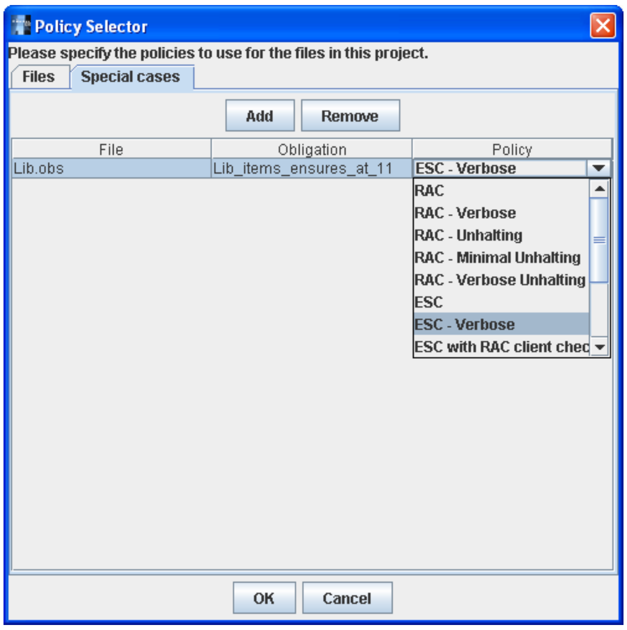
As an example, consider `items` model function of the `Lib` class. We did not produce any specification for this method because none was required to RAC-verify the class. However, we may wish to add one in order to make the specification of the class more amenable to ESC verification of client code. To use ESC, private `ensures` clauses are essential for the public model functions. So let us add a private `ensures` clause like the following.

```
public model function items():Collection[Item]
  private ensures result = items
{
  return items;
}
```

If we now re-verify the application the Omnibus IDE runs out of memory and crashes whilst reading the run-time check report file. The problem is that these functions are called so often in the application that their check reports flood the check report file. Using the standard test cases which cover all the assertions in the project, the private `ensures` clause of the `items` function alone is executed 1,531 times.

One option would be to remove the assertion. We may deem that it is not RAC-compatible because it cannot be efficiently run-time checked. When using RAC, we must be conscious of the limitations of the verification approach and adjust our specifications to what can be efficiently run-time checked. Of course, then there would be no specification of how the public `items` function relates to the private `items` attribute. This could prevent us using ESC within any part of the file because we would need to know that to use ESC.

A better option would be to define the verification of the private `ensures` clause of the `items` function as a special case. We could then use ESC to verify it instead of RAC. The method is easy to statically verify, and this approach enables us to retain the specification and ensure that it is respected. This can be done using the Policy Selector as shown in the screenshot below.

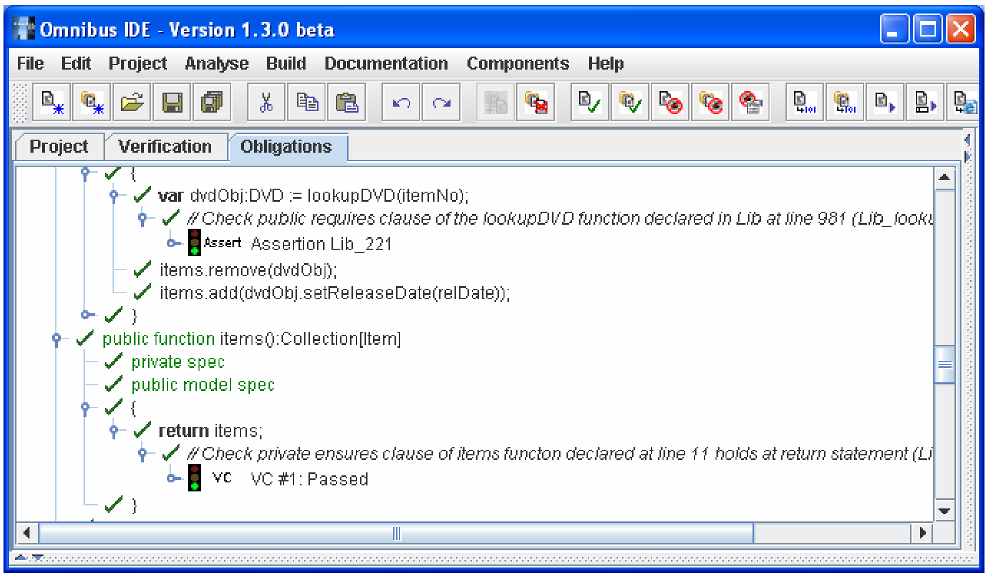


Please specify the policies to use for the files in this project.

File	Obligation	Policy
Lib.obs	Lib_items_ensures_at_11	ESC - Verbose

RAC
RAC - Verbose
RAC - Unhalting
RAC - Minimal Unhalting
RAC - Verbose Unhalting
ESC
ESC - Verbose
ESC with RAC client chec

If we open the obligations, we can see that it is justified by a VC, unlike the other obligations which are all run-time checked:



```
{
  var dvdObj:DVD := lookupDVD(itemNo);
    // Check public requires clause of the lookupDVD function declared in Lib at line 981 (Lib_looku
      Assert  Assertion Lib_221
  items.remove(dvdObj);
  items.add(dvdObj.setReleaseDate(relDate));
}
public function items():Collection[Item]
  private spec
  public model spec
  {
    return items;
      // Check private ensures clause of items functon declared at line 11 holds at return statement (Li
        VC    VC #1: Passed
  }
```

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

Statically checking missing run-time pre-condition checks

Policy for class: RAC

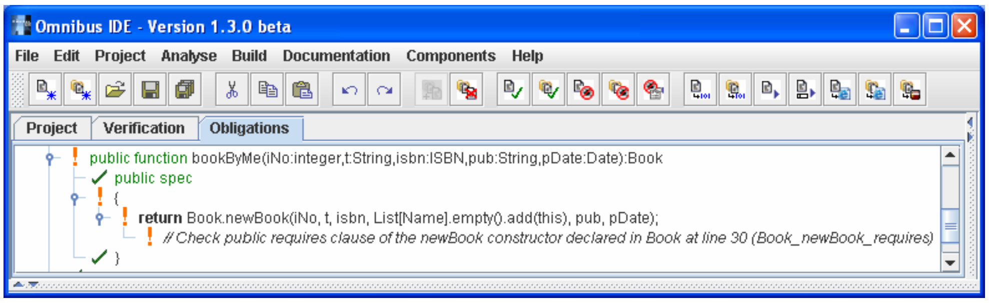
Feature used: ESC/FFV special case

We have already seen that statically verified classes must contain run-time checks of their pre-conditions if they are to be called from RAC-checked code or else unjustified correctness obligations will be reported. If the class is to be reusable by classes that are RAC-checked then the distributor of the component should generate their code with run-time pre-condition checks, but what if they do not? One option open to users of the component is to define the correctness obligation to check the pre-condition of the method as an ESC or FFV special case.

Consider the `Name` class in the Library project. We have seen that a `Book` is constructed with, among other things, a list of author names. Suppose we want to define a new function in `Name` which constructs a `Book` with the specified parameters and `this Name` as the only author. We could implement this as follows.

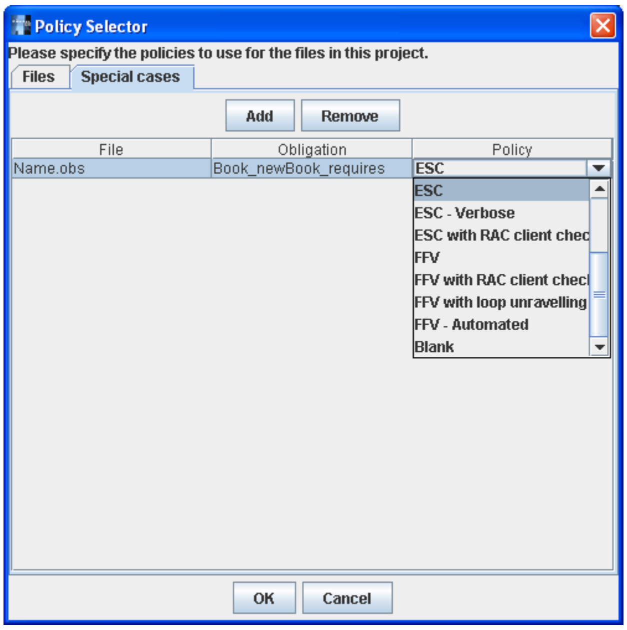
```
public class Name {
  ...
  function bookByMe(iNo:integer, t:String, isbn:ISBN,
          pub:String, pDate:Date):Book {
    return Book.newBook(iNo, t, isbn,
                        List[Name].empty().add(this),
                        pub,pDate);
  }
  ...
}
```

Now suppose that the `Book` class uses the ESC policy which has no run-time pre-condition checks. We would have an outstanding proof obligation that the `requires` clause of the `Book` constructor is respected. This can be seen in the screenshot below as an exclamation mark next to the return statement and in the comment following it.


Omnibus IDE - Version 1.3.0 beta
File Edit Project Analyse Build Documentation Components Help
Project Verification Obligations

```
public function bookByMe(iNo:integer,t:String,isbn:ISBN,pub:String,pDate:Date):Book
  public spec
  {
    return Book.newBook(iNo, t, isbn, List[Name].empty().add(this), pub, pDate);
      // Check public requires clause of the newBook constructor declared in Book at line 30 (Book_newBook_requires)
  }
```

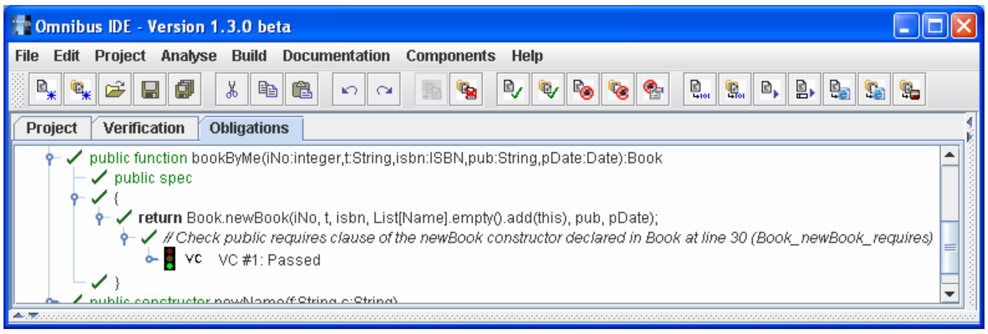
We could define this obligation as a special case and use ESC for it (as shown below). While RAC requires that pre-condition checks are generated within the called method, ESC checks the pre-condition independently, using the specification of the method. So the fact that the `newBook` constructor does not have run-time pre-condition checks does not cause it problems.



Please specify the policies to use for the files in this project.

File	Obligation	Policy
Name.obs	Book_newBook_requires	ESC

We can see below that this technique enables us to justify the correctness obligation that the `requires` clause of the `newBook` constructor is respected. The justification is provided by ESC even though the rest of the file is RAC-checked.


Omnibus IDE - Version 1.3.0 beta
File Edit Project Analyse Build Documentation Components Help
Project Verification Obligations

```
✓ public function bookByMe(iNo:integer,t:String,isbn:ISBN,pub:String,pDate:Date):Book
  ✓ public spec
  ✓ {
    ✓ return Book.newBook(iNo, t, isbn, List[Name].empty().add(this), pub, pDate);
      ✓ // Check public requires clause of the newBook constructor declared in Book at line 30 (Book_newBook_requires)
        VC   VC #1: Passed
  ✓ }
```

Check ESC-compatibility of RAC-verified class

Policy for class: RAC

Feature used: ESC targeted test

RAC specifications are often not ESC-compatible. The problem is generally that they do not have expressive enough `ensures` clauses. We can use ESC targeted tests to detect these insufficiencies.

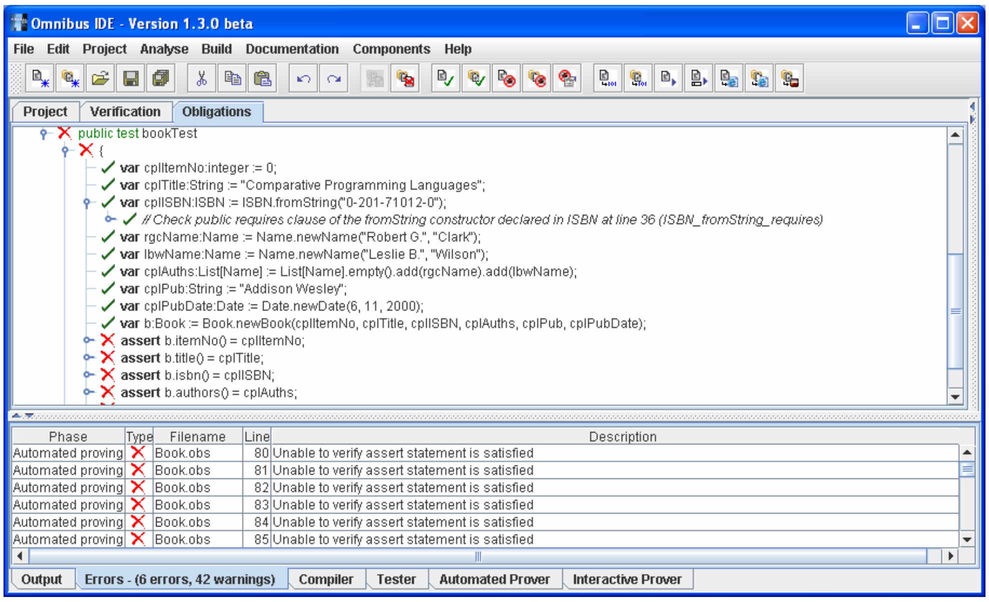
Consider the lightweight specification of the RAC-checked `Book` class. It has extremely minimal specifications. That is acceptable to RAC-check the file, and RAC does not report errors for omissions in specifications.

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

This specification is acceptable for RAC, but is it acceptable for ESC? It would be desirable to discover the answer to this question before distributing the class. We may do this using an ESC targeted test.

```
test bookTest
  policy "ESC"
{
  var cplItemNo:integer := 0;
  var cplTitle:String := "Comparative Programming Languages";
  var cplISBN:ISBN := ISBN.fromString("0-201-71012-0");
  var rgcName:Name := Name.newName("Robert G.", "Clark");
  var lbwName:Name := Name.newName("Leslie B.", "Wilson");
  var cplAuths:List[Name] :=
     List[Name].empty().add(rgcName).add(lbwName);
  var cplPub:String := "Addison Wesley";
  var cplPubDate:Date := Date.newDate(6,11,2000);
  // add the new book
  var b:Book := Book.newBook(cplItemNo, cplTitle, cplISBN,
     cplAuths, cplPub, cplPubDate);
  // check details of book
  assert b.itemNo() = cplItemNo;
  assert b.title() = cplTitle;
  assert b.isbn() = cplISBN;
  assert b.authors() = cplAuths;
  assert b.publisher() = cplPub;
  assert b.pubDate() = cplPubDate;
}
```

When we now re-verify the file, the tool will check the main body of the class with RAC but verify the ESC targeted `bookTest` test case using the specification of the `Book` class. This process checks that the specification is expressive enough for use with ESC. In this case it is not. The `newBook` constructor has no `ensures` clause, so we cannot deduce that the model functions of `Book` have the respective values passed into the constructor. As a result we get errors shown below in the ESC-compatibility test.


Omnibus IDE - Version 1.3.0 beta
File Edit Project Analyse Build Documentation Components Help
Project Verification Obligations
Output Errors - (6 errors, 42 warnings) Compiler Tester Automated Prover Interactive Prover

```
public test bookTest
  {
    var cplItemNo:integer := 0;
    var cplTitle:String := "Comparative Programming Languages";
    var cplISBN:ISBN := ISBN.fromString("0-201-71012-0");
      // Check public requires clause of the fromString constructor declared in ISBN at line 36 (ISBN_fromString_requires)
    var rgcName:Name := Name.newName("Robert G.", "Clark");
    var lbwName:Name := Name.newName("Leslie B.", "Wilson");
    var cplAuths:List[Name] := List[Name].empty().add(rgcName).add(lbwName);
    var cplPub:String := "Addison Wesley";
    var cplPubDate:Date := Date.newDate(6, 11, 2000);
    var b:Book := Book.newBook(cplItemNo, cplTitle, cplISBN, cplAuths, cplPub, cplPubDate);
    assert b.itemNo() = cplItemNo;
    assert b.title() = cplTitle;
    assert b.isbn() = cplISBN;
    assert b.authors() = cplAuths;
```

Phase	Type	Filename	Line	Description
Automated proving	✗	Book.obs	80	Unable to verify assert statement is satisfied
Automated proving	✗	Book.obs	81	Unable to verify assert statement is satisfied
Automated proving	✗	Book.obs	82	Unable to verify assert statement is satisfied
Automated proving	✗	Book.obs	83	Unable to verify assert statement is satisfied
Automated proving	✗	Book.obs	84	Unable to verify assert statement is satisfied
Automated proving	✗	Book.obs	85	Unable to verify assert statement is satisfied

To be ESC-compatible, a specification should be sufficient to ESC-check simple properties such as these.

Run-time checking non ESC-compatible obligations

Policy for class: ESC

Feature used: RAC special case and RAC targeted test

There are limits to what can be proved with automated provers. For example, the automated Simplify prover used by Omnibus is poor at handling recursion, arithmetic and complex quantified expressions. This can lead to VCs which should be provable being reported as failures by the ESC tool. Within ESC we could combat this by adjusting the specifications to remain within the bounds of what the tool can do or using assumption constructs. Alternatively, we could define the troublesome VCs as special cases and use RAC with associated tests to justify them.

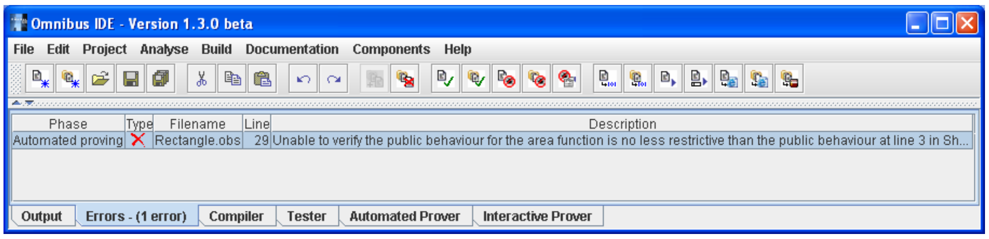
To illustrate one of the limitations of ESC, consider the simple `Shape` and `Rectangle` classes shown below. This example is not, in fact, part of the case study described in the rest of this chapter but we include it here because it illustrates the point we are discussing nicely.

```
public abstract class Shape {
  public abstract function area():integer
```

```
    ensures result >= 0
}

public class Rectangle isa Shape {
  ...
  public invariant width() >= 0 && height() >= 0
  ...
  public function area():integer
    ensures result = width() * height()
  {
    return width * height;
  }
}
```

The abstract `area` function in the `Shape` class has an `ensures` clause asserting that the result of the function should be non-negative. This method is overridden in the `Rectangle` class where its value is defined as being equal to the product of the `width` and `height`. Both the `width` and `height` are constrained to be non-negative by an invariant and so the behaviour of the `area` method in `Rectangle` should be consistent with the definition in `Shape`.

However, the Simplify prover is not very good at arithmetic and cannot, by default, deduce that the product of two non-negative numbers is non negative. This shows up via the error shown below.


Omnibus IDE - Version 1.3.0 beta
File Edit Project Analyse Build Documentation Components Help
Output
Errors - (1 error)
Compiler
Tester
Automated Prover
Interactive Prover

Phase	Type	Filename	Line	Description
Automated proving	✗	Rectangle.obs	29	Unable to verify the public behaviour for the area function is no less restrictive than the public behaviour at line 3 in Sh...

To address this we can define the correctness obligation as a special case and use RAC. However, without the definition of any test cases, the obligation will still be unjustified. We could add a normal test case to the file but then while we could use the test to execute the assertion check, we would also have to ESC-check it because all normal tests in an ESC-checked class must be ESC-checked. What we need is an RAC targeted test which we can use to define a test to cover the run-time assertion check, without the burden of also having to be ESC-checked.

This technique can also be used to define assumptions within the ESC process which are then justified by RAC checks. A similar facility is available directly through the verification policy system. A policy can define separate handling of assumption constructs for static verification and run-time checking. Specifying that the static verifier should treat them as assumptions and the run-time checker should check them gives the same thing.

Interactive proof of non ESC-compatible obligation

Policy for class: ESC

Feature used: FFV special case

When faced with the limitations of the ESC tool, another alternative is to use the interactive prover to verify the obligation.

This works similarly to the use of RAC in the previous section except that it does not require the use of a targeted test since FFV does not require the production of a test suite. This technique can be used to give a form of lemma reuse across approaches. An `assume` statement can be used to introduce an assumption in ESC with the corresponding VC being proved as an FFV special case.

Check presence and RAC-compatibility of run-time pre-condition checks

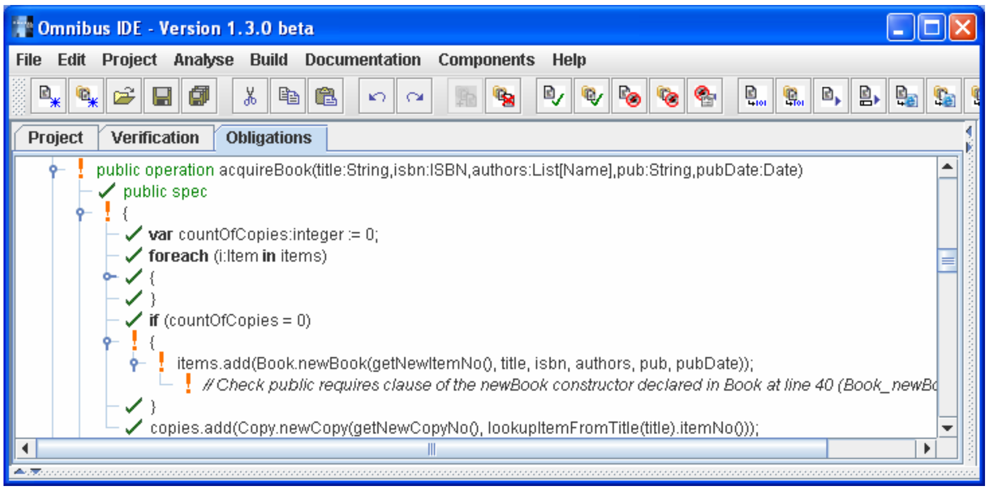
Policy for class: ESC/FFV

Feature used: RAC targeted test

We must ensure that any statically verified classes have RAC-compatible pre-conditions that are run-time checked if they are to be called from RAC-verified classes. We can use RAC targeted tests to check for this.

Consider the `Book` class in the Library project. The abstract state of this class consists of an `itemNo`, `title`, ISBN, list of authors, `publisher` and publish date. There is an invariant that the list of authors should not contain duplicates and corresponding pre-condition checks to guard the constructor and `setAuthors` operation. Suppose that we are ESC-verifying this class. The ESC tool reports no errors as long as we include the pre-conditions of the constructor and the `setAuthors` operation, so we may conclude that the class is correct and distribute it to clients.

However, clients that use the class from RAC-verified code will encounter unjustified correctness obligations corresponding (indicated in the screenshot below via exclamation marks) for the pre-conditions of the constructor and `setAuthors` operation of `Book`. This is because, while we statically verified the `Book` class, we used the ESC policy which does not include run-time pre-condition checks. The users of the class may send us feedback of this problem and we could then re-build the project with these checks. But it would be better for the developers of the component to detect these problems before distribution of the component.



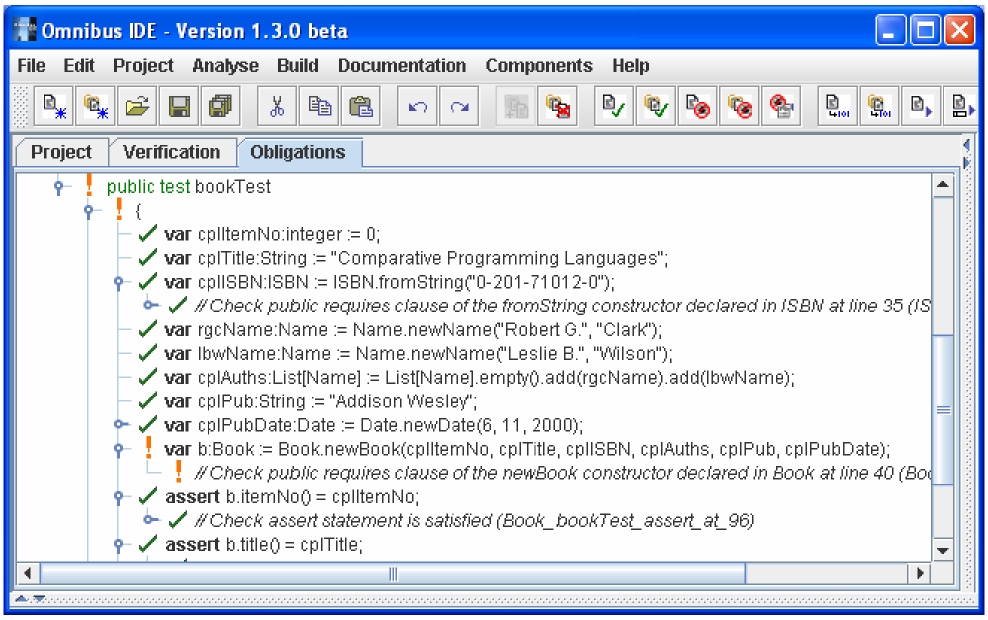
```
! public operation acquireBook(title:String,isbn:ISBN,authors:List[Name],pub:String,pubDate:Date)
  ✓ public spec
  ! {
    ✓ var countOfCopies:integer := 0;
    ✓ foreach (i:Item in items)
    ✓ {
    ✓ }
    ✓ if (countOfCopies = 0)
    ! {
      ! items.add(Book.newBook(getNewItemNo(), title, isbn, authors, pub, pubDate));
        ! // Check public requires clause of the newBook constructor declared in Book at line 40 (Book_newBo
    ✓ }
    ✓ copies.add(Copy.newCopy(getNewCopyNo(), lookupItemFromTitle(title).itemNo()));
```

RAC targeted test cases give us a way of simulating the conditions of RAC-verified client code. For example, we could define a test in our `Book` class and specify the use of the RAC policy to verify it. This test carries out a simple construction of a new `Book`.

```
test bookTest
  policy "RAC"
{
  var cplItemNo:integer := 0;
  var cplTitle:String := "Comparative Programming Languages";
  var cplISBN:ISBN := ISBN.fromString("0-201-71012-0");
  var rgcName:Name := Name.newName("Robert G.", "Clark");
  var lbwName:Name := Name.newName("Leslie B.", "Wilson");
  var cplAuths:List[Name] :=
     List[Name].empty().add(rgcName).add(lbwName);
  var cplPub:String := "Addison Wesley";
```

```
   var cplPubDate:Date := Date.newDate(6,11,2000);
   // add the new book
   var b:Book := Book.newBook(cplItemNo, cplTitle, cplISBN,
      cplAuths, cplPub, cplPubDate);
   // check details of book
   assert b.itemNo() = cplItemNo;
   assert b.title() = cplTitle;
   assert b.isbn() = cplISBN;
   assert b.authors() = cplAuths;
   assert b.publisher() = cplPub;
   assert b.pubDate() = cplPubDate;
}
```

If we now re-verify the `Book` class we get an unjustified obligation (indicated in the screenshot below via an exclamation mark) that the `requires` clause of the constructor is respected. This is the problem of the missing run-time pre-condition checks, but the test has enabled us to detect it ourselves before distribution.



```
! public test bookTest
  ! {
    ✓ var cplItemNo:integer := 0;
    ✓ var cplTitle:String := "Comparative Programming Languages";
    ✓ var cplISBN:ISBN := ISBN.fromString("0-201-71012-0");
      ✓ // Check public requires clause of the fromString constructor declared in ISBN at line 35 (IS
    ✓ var rgcName:Name := Name.newName("Robert G.", "Clark");
    ✓ var lbwName:Name := Name.newName("Leslie B.", "Wilson");
    ✓ var cplAuths:List[Name] := List[Name].empty().add(rgcName).add(lbwName);
    ✓ var cplPub:String := "Addison Wesley";
    ✓ var cplPubDate:Date := Date.newDate(6, 11, 2000);
    ! var b:Book := Book.newBook(cplItemNo, cplTitle, cplISBN, cplAuths, cplPub, cplPubDate);
      ! // Check public requires clause of the newBook constructor declared in Book at line 40 (Boo
    ✓ assert b.itemNo() = cplItemNo;
      ✓ // Check assert statement is satisfied (Book_bookTest_assert_at_96)
    ✓ assert b.title() = cplTitle;
```

The RAC tests also allow us to check other aspects of the RAC-compatibility of the class such as the efficiency of the checks and the use of assertion constructs that cannot be converted into run-time checks.

The RAC tests also permit conventional test cases to be defined which can be used to check the performance of the class for specific concrete values. Even when a class has been statically verified, it is still a good idea to define conventional tests for concrete values. RAC targeted tests can be used to do this.

Check ESC-compatibility of FFV-verified class

Policy for class: FFV

Feature used: ESC targeted test

Like RAC specifications, FFV specifications may not be ESC-compatible. With FFV specifications the problem is generally that the `ensures` clauses are too expressive and specify properties beyond what can be checked by ESC's automated prover. Again, ESC targeted tests can be used to detect ESC-compatibility.

Consider the following extract of the specification of an FFV-verified `List` class. It has a relatively rich specification. While it can be interactively verified using FFV, it is not clear if it is ESC-compatible. Again, this example is not taken directly from the library case study but is used here for convenience.

```
class List[Element] {
  model function size():integer
  model function elementAt(i:integer):Element
    requires i >= 0 && i < size()

  invariant size() >= 0

  function contains(e:Element):boolean
    ensures result = (exists (i:integer := 0 to size()-1):
     elementAt(i) = e )
  constructor empty()
    ensures size() = 0
  operation add(e:Element)
    changes size,elementAt
    ensures size() = old size() + 1,
      elementAt(size()-1) = e,
      forall (j:integer := 0 to old size()-1):
        elementAt(j) = old elementAt(j)
  function indexOf(e:Element):integer
    ensures if !contains(e) then
        result = -1
      else
        result >= 0 && result < size()
        && elementAt(result) = e
        && !(exists (i:integer where i >= 0 && i < result):
          elementAt(i) = e )
  operation removeFirstOf(e:Element)
    requires contains(e)
    changes size, elementAt
    ensures size() = old size() - 1,
     forall (j:integer := 0 to old indexOf(e) - 1):
      elementAt(j) = old elementAt(j),
     forall (k:integer := old indexOf(e) to size()-1):
      elementAt(k) = old elementAt(k+1)
}
```

To check the ESC-compatibility of this specification, we can write an ESC targeted test. We can manipulate an instance of the class in this test and the system will attempt to ESC-check it.

```
test lstTest
  policy "ESC"
{
  var int0:IntegerObject := IntegerObject.withValue(0);
  var int1:IntegerObject := IntegerObject.withValue(1);
  var int2:IntegerObject := IntegerObject.withValue(2);

  var lst:List[IntegerObject] := List[IntegerObject].empty();
  assert lst.size() = 0;
  assert !lst.contains(int0);
  lst.add(int1);
  assert lst.size() = 1;
  assert lst.elementAt(0) = int1;
  deny lst.elementAt(0) = int0;
  assert lst.contains(int1);
  deny lst.contains(int0);
  lst.add(int2);
  assert lst.size() = 2;
  assert lst.elementAt(1) = int2;
  assert lst.elementAt(0) = int1;
  assert lst.contains(int2);
  assert lst.contains(int1);
  deny lst.contains(int0);
  assert lst.indexOf(int1) = 0;
  deny lst.indexOf(int2) = 2;
  assert lst.indexOf(int0) = -1;
  lst.removeFirstOf(int2);
  assert lst.size() = 1;
  assert lst.elementAt(0) = int1;
}
```

The ESC tool is able to prove all but the final assertion. After some further experimentation we can deduce that the specification of the `removeFirstOf` operation is not ESC-compatible. The quantifications are too complicated for the automated prover to handle. If this class is to be used in ESC-checked classes, it should provide a lightweight substitute via a `which ensures` clause.

Such a substitute could be the following which simply describes whether an element is contained in the `List` after the removal but says nothing about the indices.

```
  which ensures size() = old size() - 1,
    forall (e2:Element):
     if e = e2 then
      ! contains(e2)
     else
      contains(e2) = old contains(e2)
```

8.7 Summary

The Library system described in this chapter was fully run-time assertion checked relative to the lightweight specification described in section 8.2. A set of test cases was developed which were sufficient to exercise every assertion check generated by the tool from the lightweight assertion annotations. However, a range of errors still remained after this process and they were picked up by the use of the other approaches.

The ESC approach was used to check all of the classes in the Library system apart from the `Lib` and `Date` classes. This checking was relative to the lightweight specification based on the one produced for the RAC approach with enhancements described in section 8.3. This offered increased error coverage which allowed further errors to be detected.

The FFV approach was used to fully verify a heavyweight specification of the Library system. The classes other than `Lib` and `Date` were fully verified. This involved interactively proving that the behaviour specifications satisfied the requirements specifications. Selected parts of the `Lib` class were also fully verified. The FFV approach offered the greatest error coverage and required the production of more complete behaviour specifications.

Finally, a number of ways of using verification policies were explored which enable greater levels of integration between the different verification approaches.

Chapter 9

Conclusions and future work

This section starts with a presentation of the conclusions of this thesis. These are split in to two sections: one on the language and one on the integrated tool support. Some areas of possible future work are then discussed.

9.1 Conclusions on the Omnibus language

This thesis presented the details of the new Omnibus language. In this section we assess the contribution of the language and discuss the conclusions we have drawn from its development.

9.1.1 Suitability of the language for different purposes

Good for modelling classes

The Omnibus language is well suited to the specification of modelling classes. The model function approach provides adequate facilities to describe the base levels of the specifications of a project. The contribution by Omnibus is the `changes` clause semantics which extend the classical base function approach with a useful frame-condition facility. This makes it suitable for static verification approaches without the need for the developer to manually write frame-conditions. Appendix B describes how the language can be used to model core set, sequence and map classes.

Usable for applications

While being well suited for modelling classes, the Omnibus language is also usable as an implementation language for applications. This was illustrated by the library case study. The example was a reasonable size with the implementation consisting of 2278 lines of Omnibus code. The code generation seems to be reasonably efficient, with the full tests involving the execution of over 8000 assertion checks and taking only 734 milliseconds. The code generation also seemed to fare relatively favourably to the similar PerfectDeveloper tool when compared using benchmarks [14]. Of course, there would be the potential for efficiency problems in larger applications but we are encouraged by these results.

Problems with use for applications

Our library case study showed that it was possible to use Omnibus for the implementation of applications but Omnibus was not ideal for this task. The main problems stemmed from the use of value semantics. The library application needed to manage a series of mutable data structures which cross-referenced each other. Such a scenario would be naturally representable using reference semantics but does not naturally fit the value semantics model. Using reference semantics we could simply store references to objects in fields of other objects. The built-in aliasing would allow all the objects accessed via these references to be automatically updated. We could not use the same approach in our value semantics language. The problem is that an update to one of the references to an aliased object would create a new object and leave the value of the object accessed via the other references unchanged.

To get around this, the student had to essentially manually implement reference semantics. Special identification numbers were introduced for each of the referenceable objects (the `Member`s, `Item`s, `Copy`s, etc.). Collections of all the objects were held by the central `Lib` class. If a reference to another object was required (e.g. a reference to an `Item` within a

`Copy`), the identification number of the associated object was stored and functions were provided in Lib to lookup the values of objects from their identification number. The lookup functions provide a manual implementation of dereferencing.

Operations had to be added to the `Lib` class to update the details of any of the state elements. The referenceable objects could not be updated via objects retrieved from the lookup functions for the same reason. Attempts to update them would create new objects, leaving their values held by the `Lib` class unchanged. The requirement to provide update operations for all the state elements contributed to a large `Lib` class of nearly 1500 lines. Much of the code for the update operations was very similar, consisting of the same get-update-set pattern for the different elements of the referenceable objects. This is clearly undesirable. It directly inhibits the modularity of the system, which is a bad thing for a language to do.

Native reference semantics

This is further evidence that native support for references is certainly needed. This support must be managed in some kind of structured way, such as the Universe type system discussed in chapter 4, in order to make it amenable to static modular verification. However, some of the problems we encountered in our case study are likely to be shared by these approaches since they use a similar kind of hierarchical structuring mechanism.

9.1.2 Problems with the use of the language

There were some other problems with the use of the language in our case study which were not specific to our language itself and may apply equally to other languages.

Specifications more prone to errors than implementations

A disappointing aspect of the case study was the lack of implementation errors that were detected. The vast majority of the errors that were detected using the different approaches were mistakes or omissions in the specifications. Part of the reason for this was that the implementation was tested conventionally without assertion annotations first, before the full verification support was available. Thus, the implementations had been checked in some way by the time we received them while the specifications had not.

Despite this bias, we can still speculate that the specifications produced by most programmers are more prone to errors than their implementations. This should not be a particularly surprising result since these programmers have been taught to think in the imperative style of implementations. By far the most common error that was made in the case study was missing instances of the `old` keyword. This seemed to result from the developer thinking in an imperative style about the declarative specifications.

We believe this problem will always be present in some form until assertions are incorporated more prominently into the software development world and featured more prominently in software engineering courses.

Repetition between heavyweight specifications and implementations

When using the Omnibus language to perform full formal verification, the developer can end up writing the same description of the method twice in two slightly different ways: once in the heavyweight specification and once in the implementation. All assertion-based verification approaches require some level of redundancy in order to perform their verification, but in our full formal verification approach, this seems to have gone too far. It is not so much of a problem in our RAC and ESC approaches since the developer only writes lightweight specifications that express some particular properties of the implementation. In FFV, for both the specification and implementation, the developer attempts to completely describe the desired behaviour of the method.

There seem to be two alternative approaches to this problem: automatically generating specifications from implementations or automatically generating implementations from specifications. A number of tools have aimed to automatically generate specifications for implementations. Daikon [39] and Houdini [40] seem to be the most prominent of these, both having achieved some success at assisting a developer in adding annotations. However, a crucial flaw is that these processes generate specifications over the private attributes of the class rather than the public methods. As a result, they are better suited to the detection of invariants than pre- or post-conditions. There are different ways that an implementation could be automatically generated from a specification. PerfectDeveloper [33] uses a fairly standard assertion language and automatically generates implementations for some specifications. If an implementation cannot be automatically generated, then the user must give one.

Assertion expressiveness

The Omnibus assertion language is built upon the expression language with a few additions which include an `old` operator, quantifiers (universal and existential) and additional logical operators like implication. Most assertion languages are based upon the same principles. Unfortunately, an assertion language containing only these facilities is not expressive enough to fully describe the behaviour of all the methods programmers write. In our work with Omnibus we sometimes encountered methods that we could implement but were unable to specify using the assertion language without the use of recursion. We tried to avoid the use of recursion in specifications wherever possible because we found static proofs involving recursion to be extremely difficult and time-consuming.

JML incorporates a number of additions that help address this issue. For example, generalised quantifiers allow you to express that a value is equal to the sum, product, maximum, minimum or count of the values satisfying a given predicate and allows specifications to be given in terms of “model programs”, simple imperative descriptions given using the implementation language. Many JML tools, such as LOOP [49], also allow unspecified Java methods to be reasoned about using their implementation. OCL [98] provides a range of special Collection operations which help express many properties which cannot otherwise be specified. The `iterate` operation is the most general of these, representing a generic loop operation. These offer the greatest expressiveness but are difficult to perform proofs with for the same reasons that it is hard to prove properties of recursive specifications.

We believe it would be useful to have a specification construct for iteratively describing properties and verifying them inductively. The challenge is to combine expressive power with ease of verification.

9.2 Conclusions on our integrated verification approach

The thesis also described an integrated verification approach which allows RAC, ESC and FFV to be used together in different parts of a single application. In this section we assess the contributions of this work.

9.2.1 Combining approaches to address limitations

Different error coverage and ease of use of approaches

For the library case study, we started by using RAC, ESC and FFV, in turn, for the project in its entirety. We found that the approaches offered increasing error coverage in return for increasing user burdens such as the need for more comprehensive specifications and, for FFV, user assistance in the proof process. This provides justification for our claim that the approaches offer different balances between error coverage and ease of use, making them suitable for different parts of a project with different reliability requirements.

Main problems with the different approaches

All of the approaches we support have key problems that prevent them being an ideal solution on their own.

The main problem with RAC is the need for users to produce test harnesses to sufficiently exercise the project to check all the assertion annotations. Part of the problem is that it may not be clear whether all the assertion annotations are being checked. The Omnibus IDE’s check viewer helps somewhat by allowing the user to easily identify which assertion checks have been covered by a specific test harness. The problem of producing the test cases still remains, but users should produce test cases anyway and the assertion check coverage provides a useful measure for the sufficiency of these tests. As such, we see RAC as the most practically accessible of the verification approaches we support. There is work on the automated generation of test cases which could help address this problem [17, 100].

ESC is a very powerful approach allowing projects to be statically verified automatically. It does not require test cases or user-guided proofs. However, it has limits. For example, we were unable to verify the central `Lib` class of the library project using ESC. Many of the VCs were beyond the limits of the associated automated prover. This can lead to the situation where the user is not sure whether an ESC error report corresponds to a real error or results from a limitation in the prover. To determine which of these is the case requires intimate knowledge of the underlying prover.

Our FFV approach requires the use of the interactive PVS prover to discharge the generated VCs. Default proof attempts are generated by the system and these were sufficient to prove a handful of the VCs in the auxiliary classes of the library project. We were able to verify the rest of the auxiliary classes relatively easily. However, many of the VCs from the central `Lib` class were very challenging to prove and we were not able to complete proofs of them all. Appendix E includes the details of the proof of just one of these VCs. It is clear from this that the burden of our FFV approach is very high. This has prompted us to focus most of our efforts on the RAC and ESC approaches.

Using different approaches for different files

Our verification policy system enables us to use different verification approaches for different files within a single project. For example, we were unable to use ESC for the entire library case study, but we were able to use ESC successfully for the auxiliary files and RAC for the remaining classes. When doing this, it is possible for there to be conflicts between the different verification approaches. We defined a set of guidelines for avoiding these conflicts and our tool can be used to ensure that they are obeyed.

Greater levels of integration with special cases

Our tool offers the possibility of even greater levels of integration between the different approaches through special cases. The special cases facility hinges on the correctness obligations concept. The correctness obligations of a file are the abstract properties which must all be verified in some way in order to complete the verification process. A verification policy may define that some types of correctness obligations be run-time checked and others statically verified. For example, an ESC policy may be adjusted to include run-time checks of `assume` statements so that all correctness obligations will be covered. Specific correctness obligations may also be defined as special cases and have a different policy. This gives great flexibility and means other approaches can be used to directly address the key flaws in the other approaches.

9.2.2 Supporting the safe reuse of software components

The assertion-based verification approaches we considered provide for two of the basic requirements of safe software component reuse: clear descriptions of what the components should do, and some assurance that the implementations do this. The assertion annotations in the interface specifications provide the clear descriptions and the verification process provides the basis for trusting the implementations follow these descriptions. However, without specialised support:

1. users would have to either browse the source or re-generate the documentation to get clear descriptions,
2. users would have to manually re-verify the component to ensure that it was correct, and
3. component producers would have to develop their own means of packaging and distributing their components.

The Omnibus IDE provides dedicated tool support for addressing these three problems. The first two problems relate to the identified theoretical requirements for safe reuse, while the final one addresses a practical need for managing the distribution of components.

We follow the lead of JavaDoc and JMLdoc in providing facilities to automatically generate interface documentation from the input files. The documentation of each file contains the type signature details, any textual documentation provided in the files, and details of the interface specifications.

The assertion-based verification approaches provide the basis for trusting the hidden implementations. Verification certificates are used to present a summary of the verification of the component in an accessible form. A certification level system is proposed to distil the verification information into a single reliability value based on how the different verification approaches were used together to verify the component.

Finally, the tool allows project jar files to be built which contain the original Omnibus source, executable bytecode implementations, HTML interface documentation and the verification certificate of the component, among other things. These jar files are a convenient way of distributing components. These jar files can be used directly in Java applications. The IDE supports the automatic extraction and opening of the interface documentation of a project jar file and the checking of verification certificates. The checking of bytecode implementations and HTML interface documentation was also discussed.

9.3 Future work

There are a number of ways in which the work presented in this thesis could be extended. An overview of these is presented in this section.

9.3.1 Formal definition for the Omnibus language

It would be desirable to have a formal definition of the Omnibus language. This thesis described the semantics of the language in informal terms and indirectly through its translation to logic via the static verification process. It would be better to also have an independent formal definition for the language. This should include a formalisation of the type system as well as full details of the semantics of the language. One of the motivating goals of the Omnibus language was to produce a language which was well-suited to formal verification. If we have achieved this goal then the formal definition of the language should be comparatively straightforward. The presentation of such a relatively simple formal definition would help justify the language.

9.3.2 Combining value and reference semantics

The language presented in this thesis is primarily built upon value semantics. This enables us to reduce annotation burdens and make formal verification more amenable to automation. However, support for reference semantics is required to increase the expressiveness and efficiency of the language. It would be desirable for the language presented in this thesis to be extended to include facilities for modelling reference semantics. However, this would be challenging.

The first problem with combining support for value and reference semantics is a superficial one. In Java, all objects are implicitly accessed via reference. In Omnibus, all objects are implicitly treated as values. This makes things easy to parse and understand. Whenever the name of a class is used as a type then this is interpreted as a reference in Java and a value in Omnibus. In each language there is one concept which is the default. However, if we are to combine the value semantics and reference semantics then there will be two concepts. This immediately complicates the language and could lead to confusion and mistakes by the programmers. It may complicate the language sufficiently that a prospective user of the language finds it too alien and rejects it. On a syntactic level, how should the language differentiate between the two forms of objects? We could introduce additional type modifiers to indicate whether a type is to be treated as a value or reference. However, this increases the verbosity of the language, something which we should look to limit as much as possible. One measure we could take to combat this would be to adopt one of the approaches as the default and not require type modifiers in that case. But which form should be the default? Since our work is primarily focused on value semantics, it would be natural for us to adopt that as the default. However, if we adopted reference semantics as the default and required type modifiers where value semantics was intended, then this could be viewed by programmers as an extension of Java. We note that C++ [94] has the same problem. In JML, value semantics classes are manually implemented and tagged with the `pure` modifier.

The second, and more technical problem with extending our work to incorporate support for reference semantics is that we would need to extend our run-time checking and static verification techniques to handle the complexities of reference semantics. Currently we are able to make a range of simplifications through our use of value semantics exclusively. For example, we do not need any formal model of the heap for the static verification process and we can use simpler, more lightweight specification techniques. If we are to support reference semantics then we will have to develop a formal model of the heap and support the kinds of heavyweight specification techniques of languages like JML. However, there has been much work in this area and it has been well documented. In particular, the work on ownership type systems is of great relevance and could act as a foundation for a restricted form of reference semantics. We hope we can use the existing state-of-the-art techniques for reference semantics to provides us with a good starting point for our new support.

While we wish to add support for reference semantics, we do not simply wish to produce a replica of the JML language with specialised value semantics support. JML is designed to be fully backward compatible with Java. We would like to explore the development of a Java-like language with reference semantics but will not take full backward compatibility with Java as a hard requirement. This will allow us to explore restrictions that can be made which ease the task of formal verification by compromising backward compatibility.

We also need to extend the language to handle a range of other facilities that are essential for commercial programming. These include static data and exceptions.

9.3.3 Integrating latest theorem provers and model checkers

Currently the Omnibus IDE supports extended static checking through the fully-automated Simplify theorem prover and full formal verification through the interactive PVS theorem prover. Both of these theorem provers are used by a number of leading verification projects. The most notable project to use PVS is LOOP, while both ESC/Java2 and Spec# both still currently use Simplify. There are problems with both of these provers. The interactive PVS prover is only available for the Linux and Solaris Operating Systems, and uses its own language which people would also have to become familiar with in order to perform interactive verification with it. The main problem with the fully-automated Simplify prover is that it is no longer being maintained. Work stopped on the project some time ago and, while the source and binary versions are available, it is unable to keep pace with the demands of the verification tool developers. As a consequence of this, both the ESC/Java2 and Spec# projects have said they are looking for replacements. The fact that these two clearly imperfect theorem provers are being used by a number of current projects suggests that there are currently no ideal theorem provers for verification. This is a problem and we are dependent on people in the theorem proving field to develop better tools. The SMT-LIB project seems to be one of the most promising projects. We also note that the presentation of proofs in terms of the input language appears to make the proof process far more accessible to everyday programmers. However, to support this we would need to develop our own theorem prover which we are reluctant to do.

We currently only support run-time checking, extended static checking and full formal verification. Crucially, we provide no support for model checking. Traditionally, model checking has been used for verifying properties of concurrent or communicating systems, and has relied on the use of special temporal and modal logics for specification, e.g. the JavaPathFinder project. However, the SpEx project [88] has developed a way of applying model checking to JML. Their tool is built on top of the extensible Bogor model checker in a similar way that our ESC is built upon the Simplify prover and our FFV is built upon the PVS prover. The tool takes JML-annotated Java programs and translates them into the input languages of the Bogor model checker. It would be possible to use this approach to add model checking to our integrated verification suite. However, we note that the SpEx project had to implement a range of custom model checking optimisations in order to make the process suitably efficient.

9.3.4 Distribution of reusable software components

We have discussed our support for the description and certification of reusable components. However, component description and certification are just two of the challenges inhibiting the full adoption of software component reuse. Other questions involve the deployment and retrieval of reusable components. How should third parties make their components available? How can software developers find the components they need? There may be many versions of each component; how is this managed?

Specialised tool support could be provided to support the distribution and retrieval of components. This could focus on *repository manager* and *component finder* tools. The repository manager would support the automatic construction and maintenance of online component repositories. These could be built into web servers and provide access to the components through a web interface. The sites could be managed like e-commerce sites. When components are uploaded to a repository, their verification certificates could be checked so that users could be assured that all of the components in a repository can be trusted. We imagine that there should be a central repository for the standard library components, but support could be provided to allow anyone to set up their own. The repositories would provide dedicated support for the management of different versions of each component. The component finder would allow software developers to search component repositories for the components they needed. This tool could be integrated into the Omnibus IDE and allow a number of repositories to be searched at once.

A large amount of research has been carried out into the management and retrieval of software components. Early systems used primitive techniques such as keyword searches

[42], enumerated classification [1], and faceted classification [86]. The following projects enhance these basic approaches. CRECOR [61] provides a window-based drag-and-drop GUI that supports the location, adaptation, deployment and testing of components. ComponentRank [48] uses techniques from Internet search engines to rank components by significance when performing searches. Agora [91] adds introspection techniques to Internet search engine techniques which allow the type signature interfaces of components to be viewed within the search engine. The Semantic-Based Method Retrieval approach [95] uses domain knowledge to assist searches. These projects mentioned so far all depend upon the developer instigating a search request when they think a useful software component may exist. However, a developer may not be aware of the existence of all the different types of component and so may not know to instigate a search. To address this, tools like RASCAL [67] and CodeBroker [104] monitor the developer's actions, infer the need for reusable components, perform relevant searches in the background, and actively recommend the results.

9.3.5 Other future work

There are a few remaining areas that we would like to explore further.

Firstly, we would like to develop a range of specialised specification and verification support for a number of specific programming domains. Suitable domains are GUIs, databases and parsers. GUIs are essential parts of modern software applications yet have received relatively little attention from the formal verification community. We would like to develop an assertion-based specification and verification approach that is applicable to GUIs. We will embrace the point-and-click approach to GUI development present in IDEs like Visual Studio, jBuilder and Eclipse, augmenting it with the specification facilities of an assertion-based language, and allowing invariants to be defined over the display widgets. Databases could also be formally modelled with the intention of eliminating errors resulting in the evaluation of database queries. Parsers are also very applicable to formal verification, so we would like to provide specific support for them.

We have carried out a number of case studies, the largest one of which is reported in chapter 8. We would like to carry out further, commercial-scale case studies to further assess our work.

Finally, our Omnibus IDE is implemented as a stand-alone tool. We would like to investigate whether we could integrate the tool into an existing IDE such as Eclipse.

9.4 Concluding remarks

I decided to study for a PhD in order to pursue ideas that I had about assertion-based verification. The Omnibus language and its supporting tools are a realisation of these ideas. I hope that others in the field can find material of worth in my work which can be incorporated into their own languages and tools. I believe that we can make an important contribution together.

References

[1] ACM, "The Full Computing Reviews Classification System," *ACM*, 1992.

[2] J. R. Abrial, *Assigning Programs to Meaning*, Prentice Hall, 1993.

[3] W. Ahrendt, T. Baar, B. Beckert, et al, "The KeY tool," *Software and Systems Modeling*, vol. 4, no. 1, 2005.

[4] P. S. Almeida, "Balloon types: Controlling sharing of state in data types", *In Proceedings of European Conference on Object-Oriented Programming (ECOOP 97)*, 1997.

[5] NASA Earth Science Data Systems Reuse Working Group, NASA Earth Science Data Systems Reuse Working Group Survey 2005 preliminary results, Available at: Available from http://www.esdswg.org/softwarereuse, 2005.

[6] Computer Business Review Staff, NHS IT systems crisis: the story so far, Available at: http://www.computer-business-review.com/article_cbr.asp?guid=35AC0F09-6C33-4D0E-AC2C-D912E2AA6042, 2006.

[7] Forum on Risks to the Public in Computers and Related Systems, Available at: http://catless.ncl.ac.uk/risks, 2007.

[8] B. Auernheimer and R. A. Kemmerer, *ASLAN User's Manual*, Technical report TRCS84-10, Department of Computer Science, University of California, Santa Barbara, 1985.

[9] J. Barnes and J. G. Barnes, *High Integrity Software: The SPARK Approach to Safety and Security*, Addison-Wesley, 2003.

[10] M. Barnett, K. R. M. Leino and W. Schulte, "The Spec# programming system: An overview", *In Proceedings of International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS 2004)*, 2004.

[11] D. Bartetzko, C. Fischer, M. Moller, et al, "Jass - Java with Assertions," *Electronic Notes in Theoretical Computer Science*, vol. 55, no. 2, 2001.

[12] G. Barthe, L. Beringer, P. Crégut, et al, "Mobius: Mobility, ubiquity, security. objectives and progress report", *In Proceedings of Second symposium on Trustworthy Global Computing (TGC 2006)*, 2006.

[13] P. Behm, P. Benoit, A. Faivre, et al, "Météor: A Successful Application of B in a Large Project", *In Proceedings of Formal Methods in the Development of Computing Systems (FM 99)*, 1999.

[14] K. Bell, "Developing Reliable Software Using Omnibus," Honours Dissertation, Department of Computing Science and Mathematics, University of Stirling, 2005.

[15] J. P. Bowen and M. G. Hinchey, "Ten commandments of formal methods," *Computer*, vol. 28, no. 4, 1995.

[16] J. Bowen, Formal Methods Virtual Library, Available at: http://vl.fmnet.info/, 2005.

[17] C. Boyapati, S. Khurshid and D. Marinov, "Korat: automated testing based on Java predicates", *In Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2002)*, 2002.

[18] F. P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, vol. 20, no. 4, 1987.

[19] L. Burdy, A. Requet and J. L. Lanet, "Java Applet Correctness: a Developer-Oriented Approach", *In Proceedings of Formal Methods Europe (FME 2003)*, 2003.

[20] L. Burdy, Y. Cheon, D. R. Cok, et al, "An overview of JML tools and applications," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 7, no. 3, 2005.

[21] L. Burdy, M. Huisman and M. Pavlova, "Preliminary Design of BML: A Behavioral Interface Specification Language for Java bytecode*", In Proceedings of Fundamental Approaches to Software Engineering (FASE 2007)*, 2007.

[22] P. Chalin, "Logical foundations of program assertions: what do practitioners want?*", In Proceedings of Software Engineering and Formal Methods (SEFM 2005)*, 2005.

[23] P. Chalin, P. R. James and G. Karabotsos, *The Architecture of JML4, a Proposed Integrated Verification Environment for JML*, Technical report ENCS-CSE-TR 2007-006, Concordia University, 2007.

[24] R. Chapman, "Industrial Experience with SPARK," Ada Letters, 2000.

[25] R. N. Charette, "Why Software Fails," *IEEE Spectrum*, vol. 42, no. 9, 2005.

[26] Z. Chen, *JavaCard Technology for Smart Cards: architecture and programmer's guide*, Addison-Wesley, 2000.

[27] Y. Cheon, *A Runtime Assertion Checker for the Java Modeling Language*, Technical report TR #03-09, Iowa State University, 2003.

[28] E. M. Clarke and J. M. Wing, "Formal methods: state of the art and future directions," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4, 1996.

[29] D. G. Clarke, J. M. Potter and J. Noble, "Ownership Types for Flexible Alias Protection*", In Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 98)*, 1998.

[30] D. R. Cok and J. R. Kiniry, "ESC/Java2: Uniting ESC/Java and JML*", In Proceedings of CASSIS*, 2004.

[31] D. R. Cok and J. R. Kiniry, Esc/Java2 implementation notes, 2004.

[32] D. R. Cok, "Reasoning with specifications containing method calls and model fields," *Journal of Object Technology*, vol. 4, no. 8, 2005.

[33] D. Crocker, "Development of Formally Verified Object-Oriented Systems with Perfect Developer*", In Proceedings of Precise Modelling and Deduction for Object-oriented Software Development (PMD 01)*, 2001.

[34] D. Crocker, "Safe Object-Oriented Software: The Verified Design-By-Contract Paradigm", *In Proceedings of Twelfth Safety-Critical Systems Symposium*, 2004.

[35] D. L. Detlefs, *Extended Static Checking*, Technical report 159, Compaq Systems Research Center, 1998.

[36] D. Detlefs, G. Nelson and J. B. Saxe, "Simplify: a theorem prover for program checking," *Journal of the ACM (JACM)*, vol. 52, no. 3, 2005.

[37] W. Dietl and P. Muller, "Universes: Lightweight ownership for JML," *Journal of Object Technology (JOT)*, vol. 4, no. 8, 2005.

[38] E. W. Dijkstra, "The end of computing science?," *Communications of the ACM*, vol. 44, no. 3, 2001.

[39] M. D. Ernst, J. Cockrell, W. G. Griswold, et al, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, 2001.

[40] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for ESC/Java", *In Proceedings of Formal Methods Europe (FME 2001)*, 2001.

[41] C. Flanagan, K. R. M. Leino, M. Lillibridge, et al, "Extended static checking for Java," *ACM SIGPLAN Notices*, vol. 37, no. 5, 2002.

[42] W. B. Frakes and B. A. Nejmeh, "Software Reuse Through Information Retrieval", *In Proceedings of 20th Hawaii International Conference on System Sciences*, 1987.

[43] C. George, A. E. Haxthausen, S. Hughes, et al, *The RAISE Development Method*, Prentice Hall, 1995.

[44] W. W. Gibbs, "Software's chronic crisis," *Scientific American*, vol. 271, no. 3, 1994.

[45] M. G. Tassy, *The Economic Impacts of Inadequate Infrastructure for Software Testing*, Technical report , National Institute of Standards and Technology (NIST), 2002.

[46] T. Hachler, "Applying the Universe type system to an industrial application: case study," MSc Dissertation, Department of Computer Science, Swiss Federal Institute of Technology, 2005.

[47] J. Hogg, "Islands: aliasing protection in object-oriented languages*", In Proceedings of Object Oriented Programming Systems Languages and Applications (OOPSLA 91)*, 1991.

[48] K. Inoue, "Component Rank: Relative Significance Rank for Software Component Search*", In Proceedings of 25th International Conference on Software Engineering*, 2003.

[49] B. P. F. Jacobs and E. Poll, "A logic for the Java modeling language JML*", In Proceedings of Fundamental Approaches to Software Engineering (FASE 2000)*, 2000.

[50] B. Jacobs, H. Meijer and E. Poll, "VerifiCard: A european project for smart card verification," Newsletter 5 of the Dutch Association for Theoretical Computer Science (NVTI), 2001.

[51] B. Jacobs, C. Marche and N. Rauch, "Formal verification of a commercial smart card applet with multiple tools*", In Proceedings of Algebraic Methodology and Software Technology (AMAST 2004)*, 2004.

[52] J. M. Jézéquel and B. Meyer, "Put it in the Contract: The Lessons of Ariane," *IEEE Computer*, vol. 30, no. 7, 1997.

[53] M. Karaorman, U. Holzle and J. Bruno, "jContractor: A Reflective Java Library to Support Design by Contract*", In Proceedings of Meta-Level Architectures and Reflection (Reflection 99)*, 1999.

[54] M. Kaufmann and J. Moore, "Maintaining the ACL2 Theorem Proving System*", In Proceedings of FLoC'06 Workshop on Empirically Successful Computerized Reasoning*, 2006.

[55] J. C. King, "A Program Verifier," PhD Thesis, Carnegie-Mellon University, 1969.

[56] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, 1976.

[57] J. R. Kiniry, P. Chalin and C. Hurlin, "Integrating Static Checking and Interactive Verification: Supporting Multiple Theories and Provers in Verification*", In Proceedings of International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, 2005.

[58] R. Kramer and C. T. Partners, "iContract - the JavaTM Design by ContractTM tool*", In Proceedings of Technology of Object-Oriented Languages (TOOLS 26)*, 1998.

[59] G. T. Leavens, A. L. Baker and C. Ruby, "Preliminary design of JML: a behavioral interface specification language for Java," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 3, 2006.

[60] G. T. Leavens, K. R. M. Leino and P. Müller, "Specification and verification challenges for sequential object-oriented programs," *Formal Aspects of Computing*, vol. 19, no. 2, 2007.

[61] J. Lee, J. Kim and G. S. Shin, "Facilitating Reuse of Software Components using Repository Technology*", In Proceedings of Tenth Asia-Pacific SE Conference (APS03EC)*, 2003.

[62] K. R. M. Leino, "Extended static checking: A ten-year perspective," Informatics 10 Years Back, vol. 10, 2001.

[63] B. H. Liskov and J. M. Wing, "A behavioral notion of subtyping," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 6, 1994.

[64] B. H. Liskov and J. M. Wing, "Behavioural subtyping using invariants and constraints," Formal methods for distributed processing: a survey of object-oriented approaches, 2001.

[65] C. C. Mann, "Why software is so bad," *Technology Review*, vol. 105, no. 6, 2002.

[66] C. Marché, C. Paulin-Mohring and X. Urbain, "The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML," *Journal of Logic and Algebraic Programming*, vol. 58, no. 1-2, 2004.

[67] F. McCarey, M. O. Cinneide and N. Kushmerick, "RASCAL: A Recommender Agent for Software Components in an Agile Environment," *Artificial Intelligence Review*, vol. 24, no. 3, 2005.

[68] B. Meyer, *Eiffel: the language,* Prentice Hall, 1992.

[69] B. Meyer, "The Next Software Breakthrough," *Computer*, vol. 30, no. 7, 1997.

[70] B. Meyer, "On to Components," *Computer*, vol. 32, no. 1, 1999.

[71] B. Meyer, "Contracts for Components," *Software Development*, vol. 8, no. 7, 2000.

[72] J. Meyer and A. Poetzsch-Heffter, "An architecture for interactive program provers*", In Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000),* 2000.

[73] P. Müller, J. Meyer and A. Poetzsch-Heffter, "Making executable interface specifications more expressive," *Java-Informations-Tage (JIT)*, vol. 99, 1999.

[74] P. Müller and A. Poetzsch-Heffter, "Universes: A type system for controlling representation exposure," *Programming Languages and Fundamentals of Programming. Fernuniversitat Hagen*, vol. 176, 1999.

[75] P. Müller and A. Poetzsch-Heffter, *Universes: A Type System for Alias and Dependency Control*, Fernuniversitat Hagen, Fachbereich Informatik, 2001.

[76] P. Müller, "Modular specification and verification of object-oriented programs," PhD Thesis, Fernuniversitat Hagen, Fachbereich Informatik, 2002.

[77] P. Müller, A. Poetzsch-Heffter and G. T. Leavens, "Modular invariants for layered object structures," *Science of Computer Programming*, vol. 62, no. 3, 2006.

[78] S. Nageli, "Ownership in design patterns," MSc Dissertation, Software Component Technology Group, Department of Computer Science, ETH Zurich, 2006.

[79] G. C. Necula and P. Lee, "Safe Kernel Extensions Without Run-Time Checking", *In Proceedings of Second Symposium on Operating System Design and Implementation (OSDI)*, 1996.

[80] G. C. Necula, "Proof Carrying Code", *In Proceedings of Principles of Programming Languages (POPL 97)*, 1997.

[81] S. Owre, N. Shankar, J. M. Rushby, et al, *PVS Language Reference*, Technical report , Computer Science Laboratory, SRI International, 1999.

[82] T. Parr, "ANTLR Reference Manual," *MageLang Institute, document version*, vol. 2.

[83] P. B. Larkin, The Ariane 5 Failure - Computer-Related Incidents with Commercial Aircraft, Available at: http://www.rvs.uni-bielefeld.de/publications/compendium/incidents_and_accidents/ariane5.html, 1996.

[84] R. Plasmeijer and M. van Eekelen, *Functional programming and parallel graph rewriting*, 1993.

[85] J. S. Poulin, *Measuring software reuse: principles, practices, and economic models*, Addison-Wesley, 1996.

[86] R. Prieto-Diaz and P. Freeman, "Classifying Software for Reusability," *IEEE Software*, vol. 4, no. 1, 1987.

[87] A. D. Raghavan, *Design of a JML documentation generator*, Technical report TR #00-12, Iowa State University, 2000.

[88] E. Rodriguez, M. B. Dwyer and J. Hatcliff, "Checking JML specifications using an extensible software model checking framework," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 8, no. 3, 2006.

[89] D. Sannella, "Formal program development in Extended ML for the working programmer", *In Proceedings of 3rd BCS/FACS Workshop on Refinement*, 1990.

[90] B. Schoeller, T. Widmer and B. Meyer, "Making specifications complete through models," *Architecting Systems with Trustworthy Components*, vol. 3938, 2006.

[91] R. C. Seacord, S. A. Hissam and K. C. Wallnau, *Agora: A Search Engine for Software Components*, Technical report CMU/SEI-98-TR-011, Carnegie Mellon University, 1998.

[92] T. Standish, "Essay on software reuse," *IEEE Transactions on Software Engineering*, vol. 10, no. 5, 1984.

[93] S. Stepney, D. Cooper and J. Woodcock, *An Electronic Purse: Specification, Refinement, and Proof*, Oxford University Computing Laboratory, Programming Research Group, 2000.

[94] B. Stroustrup, *The C++ Programming Language: Special Edition*, Addison-Wesley, 2000.

[95] V. Sugurmaran and V. Storey, "A semantic-based approach to component retrieval," *ACM SIGMIS Database*, vol. 34, 2003.

[96] Sun Microsystems Inc, Javadoc tool home page, Available at: http://java.sun.com/j2se/javadoc/, 2002.

[97] T. Wilson, S. Maharaj and R. G. Clark, *Push-button tools for software developers, full formal verification for component vendors*, Technical report CSM-167, Department of Computing Science and Mathematics, University of Stirling, 2006.

[98] J. Warmer and A. Kleppe, *The object constraint language: precise modeling with UML*, Addison-Wesley Object Technology Series, 1998.

[99] M. A. Weiss, *Data structures and algorithm analysis in C++*, Benjamin-Cummings Publishing, 1994.

[100] E. Weyuker, T. Goradia and A. Singh, "Automatically generating test data from a Boolean specification," *IEEE Transactions on Software Engineering*, vol. 20, no. 5, 1994.

[101] T. Wilson, S. Maharaj and R. G. Clark, "Omnibus: A clean language and supporting tool for integrating different assertion-based verification techniques*", In Proceedings of Rigorous Engineering of Fault-Tolerant Systems Workshop (REFT 05), Technical report, University of Newcastle upon Tyne*, 2005.

[102] T. Wilson, S. Maharaj and R. G. Clark, "Omnibus Verification Policies: A flexible, configurable approach to assertion-based software verification*", In Proceedings of Third IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, 2005.

[103] T. Wilson, S. Maharaj and R. G. Clark, "Flexible and configurable verification policies with Omnibus," *Software and Systems Modeling*, 2007.

[104] Y. Yunwen and G. Fischer, "Information delivery in support of learning reusable software components on demand*", In Proceedings of 7th international conference on Intelligent user interfaces*, 2002.

[105] J. Zukowski, *Java collections*, Apress Berkely, 2001.

[106] M. de Mol, M. van Eekelen and R. Plasmeijer, "Theorem Proving for Functional Programmers - SPARKLE: A Functional Theorem Prover*", In Proceedings of 13th International Workshop on the Implementation of Functional Languages (IFL 2001)*, 2001.

Appendix A

Language syntax

This appendix gives a description of the syntax and informal discussion of the semantics of the Omnibus language.

A.1 Syntax notation

We use a variation of EBNF (Extended Backus-Naur Form) adapted from ANTLR syntax. This section on the syntax notation is adapted from the “ANTLR Reference Manual” [82].

Character literals are specified just like in Java. They may contain the usual special character escapes recognized by Java ('`\b`', '`\r`', '`\t`', '`\n`', '`\f`', '`\'`', '`\\`').

String literals are sequences of characters enclosed in double quotes. The characters in the string may be represented using the same range of characters that are valid in character literals.

The following table summarizes the punctuation used in our syntax descriptions.

Symbol	Description
(...)	subrule
(...)*	closure subrule, i.e. zero or more
(...)+	positive closure, i.e. subrule one or more

(...)?	Optional, i.e. zero or one
|	alternative operator
..	range operator
~	not operator
:	label operator, rule start
;	rule end

For convenience, in this document we will write keywords as unquoted bold tokens since we feel it makes the rules more readable.

A.2 Lexical form

A.2.1 Ignored elements

Whitespace

Whitespace characters such as spaces, tabs, carriage returns, formfeeds and newlines are ignored by the compiler except to separate tokens.

```
whitespace : ' ' | '\t' | '\f' | newline;

newline : ('\r' ('\n')? | '\n');
```

Comments

Omnibus supports Java-style comments, either from “//” until the end of the line or between “/*” and “*/”.

```
singlelinecomment : "//" (~('\r' | '\n'))* newline;

multilinecomment:
  “/*”
    ( '*' ~’/’
    | newline
    | ('*'|'\n'|'\r')
    )*
  “*/” ;
```

A.2.2 Special tokens

Operators

&&	:	:=	,	.	=	!	>	>=
(	{	<==>	==>	[	<	<=	-	!=
%	+	?	)	}	]	;	/	*
||								

Keywords

abstract	also	alters	as
assert	assume	attribute	behaviour
boolean	changes	character	class
constraint	constructor	deny	description
else	elseif	ensures	exists
false	for	forall	foreach
function	if	in	initially
integer	invariant	is	isa
library	maintains	model	native
nothing	null	old	operation
out	package	prev	private
protected	public	requires	return
returns	spec	static	test
then	to	true	unreachable
uses	var	where	which
while			

A.2.3 Identifiers

Identifiers consist of a letter followed by zero or more letters, digits and underscores. Note that identifiers starting with underscores are not permitted; these are reserved for internal use by the compiler.

```
ident : letter (letter | digit | '_')*;

digit : '0'..'9' ;

letter : 'a'..'z' | 'A'..'Z' ;
```

A.2.4 Literals

Character literals

Similar to Java, character literals are written between single quote characters and can be an ordinary single character or an escaped character. Currently only newline, tab, single quote and backslash characters are supported.

```
characterliteral : '\'' singlecharacter '\'' ;

singlecharacter :
    '\\' ('n' | 't' | '\'' | '\\' )
    | ~'\\' ;
```

String literals

String literals are written as a sequence of characters written between double quote characters.

```
stringliteral : '\"' (~'\"')* '\"'
```

Integer literals

Integer literals simply consist of one or more digits.

```
integerliteral : (digit)+;
```

A.2.5 Primitive types and expressions

The Omnibus type system is partitioned into primitive types and objects like Java. Here we discuss the primitive types and expressions.

Primitive types and literals

Omnibus supports three primitive types: **`integer`**, **`boolean`** and **`character`**. These correspond to the `int`, `boolean` and `char` types in Java. Omnibus does not support the shortened `byte` or `short` types, the widened `long` type or the floating point `float` or `double` types from Java. Numeric overflow of integers is not handled within the static verification modules.

Standard operators

Five logical operators are provided within the expression language: negation, disjunction, conjunction, implication and bi-implication. As in Java, the disjunction, conjunction and implication operators are short-circuited, i.e. their second operands are only evaluated if, given the value of the first operand, they can affect the value of the expression. The symbols '`==>`' and '`<==>`' are used for implication and bi-implication, respectively.

Six standard integer operators are provided: addition, subtraction, multiplication, division, modulus and minus. Division by zero is handled and an assertion failure results if a divisor evaluates to zero. This is checked as part of the static verification process.

Four relational (or inequality) operators are provided: less than, less than or equal to, greater than, and greater than or equal to.

Equality operators

In Java, the '`==`' and '`!=`' operators are used for equality of primitive values and its negation. In Omnibus we use '`=`' and '`!=`'. The equality operators are overloaded and also used for object equality. We will consider the use of the equality operators in this way in section 3.3.

If .. then .. else ..

An '**`if`** `..` **`then`** `..` **`else`** `..`' operator is provided within the Omnibus expression language. This is a textual form of the ternary '`?:`' operator from Java. The first operand should provide a boolean condition whose truth determines whether the value taken in the `then` or `else` clause is used as the value of the expression. The types of the expressions in the `then` and `else` parts of the expression must be the same.

Variables

Variables can be referred to in expressions via their corresponding identifier in the standard way. There are different kinds of variables. For example, variables are used for parameters, fields, quantifier variables and special variables such as `this` and `result`. We will discuss each of these as appropriate in the rest of this section.

Quantifiers

The Omnibus language supports universal and existential quantifiers within its expression language. Quantification expressions consist of a series of variable declarations, restrictions over those variables, and an assertion over the variables. For objects, the declared variables are interpreted to range over all possible valid values of the specified type, not all allocated values of the specified type as they are in JML. Each variable declaration can be directly restricted to range over a delimited integer range or the contents of a specified collection. A further assertion can be used to restrict the values of the variables that are quantified over through a `where` clause. The assertion provided in a `where` clause can refer to any of the quantified variables. A single assertion is calculated from the restrictions (if present) from the variable declarations and `where` clause. This is then combined with the assertion over the variables. In the case of existential quantifiers, the conjunction operator is used. In the case of universal quantifiers, the implication operator is used.

The syntax for quantifier expressions is given below:

```
quantifierexpression :
  (forall | exists)
      “(“ quantvardeclarations (whererestriction)? “)” “:”
    expression;

quantvardeclarations :
  (quantvardeclaration (“,” quantvardeclaration)*)?;

quantvardeclaration :
  vardeclaration (quantvarrestriction)?;

quantvarrestriction :
  rangerestriction | inrestriction;

rangerestriction :
  “:=” expression to expression;

inrestriction :
  in expression;

vardeclarations : (vardeclaration (“,” vardeclaration)*)?;

vardeclaration : ident “:” expression;

whererestriction :
  where expression;
```

Evaluation order

The evaluation order of the previously discussed operators is as follows (from lowest to highest precedence):

Quantifiers

If expression

Logical Implication, Logical OR, Logical AND

Equality operators

Relational operators

Addition, Subtraction

Multiplication, Division, Modulus

Logical NOT

Arithmetic negation

As in Java, the evaluation order can be overridden by parenthesising sub-expressions. The type of a parenthesised expression is the type of the expression contained within the parentheses.

Full expression syntax

The full syntax of the expression language is given below.

```
expression : quantifierexpression;

quantifierexpression :
  (forall | exists) “(“ quantvardeclarations
     (whererestriction)? “)” “:” expression
  | iffexpression;

iffexpression : impliesexpression ( “<==>”
     impliesexpression)?;

impliesexpression : orexpression ( “==>” orexpression)?;

orexpression : andexpression ( “||” orexpression)?;

andexpression : eqcompareexpression ( “&&” andexpression)?;

eqcompareexpression : compareexpression ( (“=” | “!=”)
     compareexpression)?;

compareexpression : addsubexpression ( (“<” | “>” | “<=” |
     “>=”) addsubexpression)?;

addsubexpression : multdivexpression ( (“+” | “-“)
     addsubexpression)?;

multdivexpression : isasexpression ( (“*” | “/” | “%”)
     isasexpression)?;

isasexpression : notexpression ( (is | as) notexpression)?;

notexpression : (“!”)? negexpression;

negexpression : (“-“)? suffixexpression;

suffixexpression : atomicexpression (“.”
     variableorfunctioncall)*;

atomicexpression :
  “(“ expression “)”
  | true | false
  | integer | boolean | character
  | ident (actualparameters | actualsquareparameters)?;
  | old ident (actualparameters)?
```

```
   | stringliteral | characterliteral | integerliteral
   | if expression then expression else expression
   ;
```

A.3 Source files

As in Java, Omnibus applications are made up of a collection of class definitions. Each Omnibus class must be defined in a separate source file. A package definition and a number of uses clauses may be provided. The syntax of an Omnibus source file is given below.

```
file : (packagedef)? (usesclause)* classdecl;
```

Package definitions

A package definition gives the package that this class should be part of. Like C# and unlike Java, the package path does not have to match the directory structuring of the source files. No terminating semi-colon is needed.

```
packagedef : package path;

path : ident (“.” ident)*;
```

Uses clauses

A `uses` clause is equivalent to a Java import statement. To import all the files in a package, the name of the package should simply be specified, without the “.*” suffix used in Java. The type checker will report an error if there is not a class or directory with the specified path.

```
usesclause : uses path;
```

Class definitions

For presentation purposes, we consider the class definitions in two parts, the class header and the class body.

```
classdecl : classhead classbody;
```

Class headers

The headers of Omnibus classes are similar to the headers of Java classes. As with Java classes, accessibility and `abstract` modifiers are provided. If no accessibility is specified, it defaults to public. The `abstract` modifier behaves as in Java, permitting the class to define abstract methods and preventing the class from being directly instantiated.

An Omnibus class can also be declared with either the `spec` or `native` modifiers, but not both. We refer to a class declared with the spec modifier as a *spec class* and a class declared with the native modifier as a *native class*. A spec class is a class without an implementation. Spec classes can be used to declare the specifications and reason about them before proceeding to produce an implementation. Without the `spec` modifier, the presence of non-abstract methods without implementations would result in a type checking error. A spec class defines no implementation and so cannot be compiled to bytecode. A type checking error will result if an attempt is made to compile a spec class to bytecode. A native class, like a spec class, does not have implementations for its methods. However, these classes can be used by other non-specification classes. The system assumes that a corresponding native bytecode implementation will be provided in the run-time system. Of course, such an implementation must be compatible with the principles of the bytecode generated from Omnibus classes and should be independently verified to be consistent with the specification given in the native Omnibus class. This is similar to the use of non-bytecode implementations in Java through Java's `native` modifier.

Like Java 1.5, Omnibus supports generics. In Omnibus, template parameters are surrounded by square brackets. The template parameter list, if present, specifies a sequence of identifiers to hold generic type parameters. Within the body of the class, variables with these names will be introduced holding class values. This allows these identifiers to be used wherever a type is expected.

Inheritance is supported via the `isa` clause. The keyword `isa` is used in place of `extends` because the underlying concept of inheritance is slightly different in Omnibus and

Java. In Java, inheritance is often used in ways that do not obey the principle of substitutability relative to the natural full specifications of correctness. This can lead to problems in modular verification. To avoid this problem, Omnibus enforces behavioural inheritance, sometimes referred to as the "is a" relationship. A class inherits non-static functions and operations from the superclass if they are not redefined in the current class. A class inherits all the requirements of its superclass. Where methods are overridden, the specifications of the two methods must be consistent as defined later. As of yet, only single behavioural inheritance is supported. There is no support for multiple superclasses (like those in C++ or Eiffel) or Java-style interfaces.

Classes may include a description clause which can be used to provide a short description of the class. This is similar to a JavaDoc comment and is treated in an analogous manner by the Omnibus documentation generator. The description must be given as a String literal.

```
classhead : (accessibility)? (abstract)? (spec | native)?
                 class ident (templateparameters)?
                               (isa ident)?
                 (description)? ;

accessibility : (public | protected | private);

templateparameters : [ identifierlist ];

identifierlist : ident (, ident)*;

description : description text;

text : stringliteral;
```

Class bodies

The body of a class definition consists of a sequence of zero or more member declarations which declare attributes, methods or requirements. There three forms of method: functions, operations and constructors. Requirements are discussed in section 2.3.

```
classbodyelement :
  attributedecl
  | functiondecl
  | constructordecl
  | operationdecl
  | requirementdecl
```

```
   | testdecl;
```

Attributes

Attribute declarations are used to declare the concrete fields used to implement an object. The value of the attribute can be accessed within the body of the class via a variable of the same name.

```
attributedecl : (accessibility)?
                       attribute ident “:” expression
                             (description)?;
```

Constructors

Constructors are used to create instances of classes. An implementation for the constructor should be given unless the containing class is declared with the `spec` or `native` modifier. Omnibus constructors are implemented as class methods and create a new object instance of the containing class. They are equivalent in function to Java constructors but are called like Java static methods. Unlike Java, there are no default constructors because of the possibility of invariants. The specification of constructors is discussed in section 2.2.

```
constructordecl : (accessibility)?
                       constructor ident parameters
                             (description)?
                             behaviour
                       (code)?;
```

Functions

Functions accept a number of parameters and calculate a single value to return to the caller without any side-effects. Functions can be declared with the static modifier. Static functions, like in Java, are used to define class methods. An implementation should be given for each static function unless the class is declared with the `spec` or `native` modifier. Static functions cannot also be declared as `abstract`. Functions declared without the `static` modifier are standard object methods. Implementations for non-static functions should be provided unless the `abstract` modifier is used or they appear in a spec or native class. The specification and use of the `model` modifier in function declarations is discussed in section 2.2.

```
functiondecl : (accessibility)? (static | abstract)?
                                (model)?
                  function ident parameters “:” expression
                      (description)?
                      behaviour
                  (code)?;
```

Operations

Operations are used to manipulate the values of objects. There are two different types of operations: simple operations and complex operations. Simple operations are operations without any var/out parameters. These are equivalent to functions which return a new object that is calculated from the initial object by applying the corresponding manipulation. These methods are semantically indistinguishable from functions returning values of the same type as the class in which the declaration occurs. This allows them to be used in expressions and assertions as if they were functions as well as being used in operation call statements. Complex operations are operations which contain at least one var or out parameter. These cannot be converted to functions which, by our definition, should calculate a single value without side-effects. As such, complex operations cannot be used in expressions.

```
operationdecl : (accessibility)? (abstract)?
                  operation ident parameters
                      (description)?
                      behaviour
                  (code)?;
```

Parameters

Parameters consist of a parenthesised set of declarations, optionally preceded by `var` or `out`. Parameters without a `var` or `out` prefix correspond to call-by-constant-value input parameters. Omnibus does not permit the values of these parameters to be changed within the body of a method. Parameters with a `var` prefix correspond to call-by-value-result input-output parameters. The values of these parameters can be changed within the body of a method, and their resulting values will be copied to the input variables after the call completes. Parameters with an `out` prefix are like var parameters except that they are initially undefined and have no value passed in. Omnibus only allows `var` and `out` parameters to be used in conjunction with operations.

```
parameters : “(” (declarations)? “)”;

declarations : (declaration (“,” declaration)*)?;

declaration : (var | out)? ident “:” expression;
```

Tests

Omnibus allows unit tests to be defined within a class. These can be used to express the intended behaviour of object instances of the current class for specific concrete values. The tests are treated like static methods with no parameters and no return value. Test data must be constructed within the body of the test, and can then be used to manipulate instances of the class and check properties of its behaviour. By default, a test case is verified using the same verification policy as the class which contains it. However, a `policy` clause may be provided, specifying an alternative policy to use.

```
testdecl : (accessibility)? test ident (policy)? code;

policy : policy text;
```

A.4 Classes as types and objects

In Omnibus, programmers can define their own types through the class definitions discussed in the previous section. The methods defined in the class definitions can be used to create, query and manipulate instances of the classes which are referred to as objects.

Classes as types

Wherever a type is required, a class expression can be provided. Like Java, all the classes that are used by a class (i.e. which are imported by uses clauses either directly or are in a package that is imported) are made available through variables with the same name as the class. The top-level packages are also made available as variables and can be dereferenced to yield sub-packages and contained classes. Template parameters can be specified in square brackets. Class expressions should be provided as the template parameters.

```
classexpression :
  (ident “.”)* ident
                 (“[“ (expression (“,” expression)*)? “]”)?
```

Some examples of class expressions are given below:

```
Environment
omni.lang.String
Collection[String]
omni.lang.Collection[omni.lang.String]
Map[String, Collection[String] ]
```

Using Objects

Objects are created using constructors. Object instances of a class are initially created by calling one of the constructor methods of the class. In Omnibus, constructors are named class methods and, unlike Java, there is no `new` operator. When creating an instance of a class with some generic parameters, concrete type values should be passed for those parameters in square bracketed parameters following the name of the class. An object can be queried by calling one of its functions and updated by calling one of its operations. Functions and operations are object methods, called using standard dot notation. Method calls can be made in expressions using the following method syntax. Constructor calls are made by providing a class expression as the first expression.

```
objectmemberexpression :
  expression “.” ident
        (“(“ (expression (“,” expression)*)? “)”)? ;
```

For example, we can calculate the balance of a savings `BankAccount` which was opened with a deposit of `300`, had `50` withdrawn and then a further `80` deposited using the following expression:

```
BankAccount.openSavings(300).withdraw(50).deposit(80)
```

```
           .balance()
```

As in Java, within the body of a class definition, the methods of the current class can also be referred to directly.

```
functionexpression :
  ident “(“ (expression (“,” expression)*)? “)”
| old ident “(“ (expression (“,” expression)*)? “)”
```

Special variables

There are a range of special variables that are available within the Omnibus language. For example, the **`this`** keyword is used to refer to the instance of the containing class that a method is being applied to. The **`result`** keyword is used to refer to the value to be returned from a function in an ensures clause. The local attributes of a class can also be referred to via variables.

Casting

As with any statically typed object-oriented language, at times it will be necessary to check the run-time class of an object against some other class and perform a cast to convert it to an instance of that class. We support both of these functions through `is` and `as` operators, respectively. These correspond to the `instanceof` operator and cast expression in Java.

The `is` expression returns whether the left operand is an instance of the right operand, i.e. is its dynamic class equal to the specified class or a subclass of it.

```
isexpression :
  expression is expression
```

The `as` expression casts the left operand to an object instance of the right operand.

```
asexpression :
  expression as expression
```

A.5 Specifications

Omnibus specifications are discussed in detail in chapter 2. Here we present the details of the syntax of Omnibus behaviour and requirement specifications

Behaviour specifications

```
behaviour:
  lightbehaviour | fullbehaviour;

lightbehaviour:
  (requiresclause)? (changesclause)?
  (ensuresclause)? (whichensuresclause)?;

fullbehaviour:
  (behaviourlevel)+;

behaviourlevel:
  (public | protected | private) (model)? (spec)?
      (requiresclause)? (changesclause)?
      (ensuresclause)? (whichensuresclause)?;

requiresclause : requires assertion (“,” assertion)*;

changesclause : changes expression (“,” expression)*;

ensuresclause : ensures assertion (“,” assertion)*;

whichensuresclause : which ensures assertion (“,” assertion)*;

assertion :
  text “:” expression
  | expression;
```

Requirements specifications

```
requirementdecl :
  invariantdecl
  | constraintdecl
  | initiallydecl;

invariantdecl : (accessibility)? invariant assertion;

constraintdecl : (accessibility)? constraint assertion;

initiallydecl : (accessibility)? initially assertion;
```

A.6 Implementation language

A.6.1 Code

Omnibus implementations consist of a number of statements enclosed by curly brackets.

```
code : “{” (statement)* “}”;
```

A.6.2 Statements

The Omnibus implementation language is fairly simple, consisting of 11 statements.

```
statement :
  assignmentstatement | callstatement | declarestatement
  | assertstatement  | assumestatement | ifstatement
  | forstatement | foreachstatement | whilestatement
  | returnstatement | unreachablestatement ;
```

Assignment statements

The assignment statement is used to assign the value of an expression to a variable.

```
assignmentstatement:
  ident “:=” expression ‘;’ ;
```

A variable with the specified name must exist, the variable must be assignable (i.e. be an attribute in a constructor or operation, local variable, var parameter, out parameter or one of the special variables `result` or `this`), the expression must be executable (e.g. no quantifiers without variable restrictions) and the type of the expression must be acceptable in place of the type of the identifier.

Operation call statements

There are two types of operation call statement: local operation call statements and object operation call statements. Local operation call statements update the current object (`this`) by applying one of the operations from the local class. Object operation call statements update the value of a specified object variable by applying an operation to that object.

```
callstatement:
  ident “(“ (callparameter (“,” callparameter)*)? “)”
  | ident “.” ident “(“ (callparameter (“,” callparameter)*)?
     “)”) ‘;’;
```

The parameters to an operation call statement are either expressions or variable names prefixed with `var` or `out`, depending on the declaration of the corresponding parameter.

```
callparameter :
  var ident | out ident | expression;
```

If no object is specified, there should be a local operation with the specified name and the parameters should be acceptable. This check includes checks of the types of the parameters and whether they are var/out parameters. For example, if a formal parameter is declared with the `var` modifier then the corresponding actual parameter should also start with the `var` modifier. If an object is specified, a variable with the specified name should exist and should be assignable, the type of the variable should be an object, and that object should contain an operation with the specified name. We enforce this in an attempt to make whether a parameter can be changed more explicit to readers of the calling code. The parameters should also match as was described before.

Declaration statements

A declaration statement is used to declare a new local variable. The new variable may also be given an initial value.

```
declarestatement :
  var ident “:” expression (“:=” expression)? “;”;
```

There should not be an existing local variable with the specified name, the expression after the colon should be a type. If an initial value is specified then it should be executable and should be acceptable as a value of this type.

Assert statements

Assert statements are used to specify assertions that should hold at the point when the statement is reached. These assertions will be checked either statically or at run-time to ensure they are valid. Like other assertions, the assertion specified can optionally be given a label.

```
assertstatement :
  assert (text “:”)? expression “;”;
```

The type of the expression representing the assertion to check should be boolean. If run-time checking of `assert` statements is turned on, the assertion should also be executable.

Assume statements

Assume statements are also used to specify assertions that should hold at the point when the statement is reached. However, unlike assert statements, their assertions are taken by the system to be true without further justification. Again, the assertion specified can optionally be given a label.

```
assumestatement :
  assume (text “:”)? expression “;”;
```

The type of the expression representing the assertion to check should be boolean. If run-time checking of `assume` statements is turned on, the assertion should also be executable.

If statements

If statements are used to execute one of a number of sections of code depending on the values of a number of conditions.

```
ifstatement :
  if “(” expression “)” code
  (elseif “(” expression “)” code)*
  (else code)?;
```

The type of the conditions should be boolean and they should be executable.

While statements

The while statement is used to execute a block of code repeatedly while a condition evaluates to true. Loop invariants can be provided using `alters` and `maintains` clauses.

```
whilestatement :
  while “(“ expression “)”
     ((altersclause)? maintainsclause)?
    code;
```

The loop condition should be a value of type boolean and should be executable.

Loop invariants

Loop invariants are specified using `alters` and `maintains` clauses. Alters clauses are similar to `changes` clauses, and `maintains` clauses are similar to `ensures` clauses. While `changes` clauses can refer to model functions and attributes, `alters` clauses may only refer to attributes. Alters clauses make the specification of invariants easier just as changes clauses make the specification of operations easier.

```
altersclause : alters ident (“,” ident)*;

maintainsclause : maintains assertion (“,” assertion)*;

assertion :
  text “:” expression
  | expression;
```

All of the identifiers specified in an `alters` list should be the names of existing variables that are assignable. The expressions specified in each of the assertions in a `maintains` clause should be of type boolean. There are no loop variants in Omnibus.

For statements

For statements can be used to iterate over a range of integer values. For statements implicitly declare a new local variable. An integer iterator variable is declared with the specified name and initialised to the low value. Execution of the loop proceeds while the value of the iterator variable is less than or equal to the high value. At the end of each iteration, the iterator variable is incremented. Loop invariants can be provided using `alters` and `maintains` clauses. If a loop invariant is present and run-time checking of loop invariants is enabled, then the invariant is checked before each iteration of the loop body.

```
forstatement :
  for “(“ ident “:=” expression to expression “)”
     ((altersclause)? maintainsclause)?
    code;
```

As was mentioned, for statements implicitly declare new local variables. As such, there should not be an existing local variable with the name specified before the `:=` operator. The expressions before and after the `to` keyword should be integer expressions.

Foreach statement

Foreach statements can be used for iterating over the values in a subclass of the `omni.lang.Iterable` class. Again, Foreach statements implicitly declare a new local variable. At the start of the loop, an iterator is retrieved from the `Iterable` object. While the iterator has a next element it retrieves the next value, places it in a local variable with the same name as the iterator variable, executes the body of the loop, and then removes the next element. If a loop invariant is present then the invariant is checked before each iteration of the loop body.

```
foreachstatement :
  foreach “(” ident “:” expression in expression “)”
     ((altersclause)? maintainsclause)?
    code;
```

Since foreach statements also implicitly declare a new local variable, there should not be an existing local variable with the name specified before the `:` operator. The expression before the `in` keyword should be an object type and the expression after the `in` keyword should be an `omni.lang.Collection` value. Additionally, the second expression should be a `Collection` of instances of the first expression. This is overly restrictive and should be replaced with a generic iteration concept.

Return statements

The return statement is used within functions to return a value from the method and halt its execution. Each path through the function should end with a return statement. The `ensures` clause of the function should hold at the return statement with the special `result` variable assigned the specified value.

```
returnstatement :
  return expression “;”;
```

This statement can only appear in the body of a function. The type of the expression should match the return type of the function.

Unreachable statements

The unreachable statement is used in place of a return statement when the path through the method ending at the statement should never be reached. It is needed to avoid warnings of missing return statements in the generated Java that are known to be spurious from the method pre-conditions.

```
unreachablestatement :
  unreachable “;”;
```

Run-time assertion checks are generated in place of unreachable statements, reporting a failure if execution reaches the statement.

Appendix B

Modelling types

The model function specification approach of the Omnibus language is well-suited to the specification of modelling types that can be used to describe the rest of a system. These provide core specification facilities such as sets, maps and sequences. This appendix describes basic versions of the specifications for some modelling types provided in the Omnibus standard libraries. We present the public specifications of the classes and describe unit tests to ensure the specifications are ESC-compatible. The implementations of the classes are not shown although implementations are provided in the standard libraries to allow run-time assertion checking of specifications described using modelling types.

B.1 Collection class

The `omni.lang.Collection` class is used to represent a logical set of values. It is a parameterised class which can be used to store values of a specific class, referred to internally as `Element`. In this section we present a simplified version of this class. The specification of the class is built around a `contains` function which yields the boolean value true if the passed element is in the collection and false otherwise. The class header and `contains` model function are shown below.

```
spec class Collection[Element] {
  model function contains(elem:Element):boolean
```

The `empty` constructor can be used to construct a collection containing no values. This is described by saying that for all elements, the `contains` function yields false. The specification of this operation is shown below.

```
  constructor empty()
    ensures forall (e:Element): !contains(e)
```

The `add` operation is used to add a value to the collection. Its specification is relatively straightforward using the Omnibus frame condition logic. The `contains` model function is adjusted so that it yields true for the passed element and whether all other elements are contained in the collection is unchanged. To express this we simply state that the model function `contains` is changed for the parameter value `e` and that the assertion `contains(e)` holds at the end of the operation, i.e. that `contains(e) = true`. This can be done using the following specification.

```
  operation add(e:Element)
    changes contains(e)
    ensures contains(e)
```

The operation could also be specified in Omnibus without the use of the parameterised changes clause entry. We could instead specify that the `contains` function is changed for all values (indicated by the lack of a parameter) and then add an additional assertion in the ensures clause to manually express the frame condition. This is the approach that would have to be used in JML. Such an alternative specification for the `add` operation is given below.

```
  operation add(e:Element)   // Alternative version
    changes contains
    ensures contains(e),
     forall (e2:Element where e != e2):
      contains(e2) = old contains(e2)
```

The `remove` operation is used to remove a value from the collection. It is similarly straightforward to specify in Omnibus. Again, we can specify that the `contains` model function is changed for the passed value, and this time assert that the `contains` function yields false for that value after the operation. The specification is given below.

```
  operation remove(e:Element)
    changes contains(e)
    ensures !contains(e)
```

Again we could provide an alternative specification with a manual frame condition.

We need to ensure that the specification we have produced expresses the class we intended and that the automated prover can effectively reason about it, i.e. there is ESC-compatibility. Both of these things can be checked using test cases. We can write a test case to create and manipulate object instances of the class and check that the functions can be proved to have the values we would expect. This checks the meaning and ESC-compatibility. The test starts with its header, including the policy clause indicating that “ESC” should be used, and the declaration of object which will be used in the following code. This code is shown below.

```
  test escCompatibilityTest
    policy “ESC”
  {
    var int0:IntegerObject := IntegerObject.withValue(0);
    var int1:IntegerObject := IntegerObject.withValue(1);
```

The `empty` constructor is then used to construct a new collection. This collection should not contain any objects and so it should be possible to prove that the objects `int0` and `int1` are not contained in the collection. The tool is able to do this. The code is shown below.

```
    var s:Collection[IntegerObject] :=
     Collection[IntegerObject].empty();
    assert !s.contains(int0);
    assert !s.contains(int1);
```

The `add` operation is then used to add the object `int0`. This should result in the `int0` object being contained in the collection after the operation but the `int1` object should still not be in the collection. The code to test this is shown below.

```
    s.add(int0);
    assert s.contains(int0);
    assert !s.contains(int1);
```

The `add` operation is then used again to add the `int1` object. This should result in both `int0` and `int1` being contained by the collection. The code to check this is shown below.

```
    s.add(int1);
    assert s.contains(int0);
    assert s.contains(int1);
```

The `remove` operation is then used to remove `int0`. The `int0` object should then no longer be contained in the collection but `int1` should still be.

```
    s.remove(int0);
    assert !s.contains(int0);
    assert s.contains(int1);
```

Finally, a `deny` statement is used to ensure that false cannot be proved. This allows contradictions to be detected.

```
    deny false;
  }
}
```

This completes the test and the class which the Omnibus verification tool is able to verify successfully. The full specification of the `omni.lang.Collection` class also contains a `size` model function and methods for comparing and combining collections in different ways.

B.2 Map class

The `omni.lang.Map` class is used to represent a mapping of keys to values. It has two type parameters, one for the key, `Key`, and one for the value, `Value`. In this section we present a simplified version of this class. The specification is built around a `contains` function yielding true iff there is a value for the passed key, and a `value` function which returns the value assigned to a specific key. Only values for keys that are contained by the map can be retrieved. The class header and model function declarations are shown below.

```
spec class Map[Key,Value] {
  model function contains(key1:Key):boolean
  model function value(key2:Key):Value
    requires contains(key2)
```

The `empty` constructor is similar to that of the Collection class. It asserts that for all keys, the `contains` function evaluates to false after the constructor. This also implies that there are no valid parameters for the `value` model function.

```
  constructor empty()
    ensures forall (k:Key): !contains(k)
```

The `put` operation is used to add a new key-value pair to the map. This can be specified by stating that the `contains` model function is changed for the parameter `k` and that after the operation, `contains(k)` is true and `value(k)` is equal to the specified value. The `value` model function does not need to be referred to in the `changes` clause since no values that are valid before and after the operation change (as was discussed in section 3.1.2).

```
  operation put(k:Key, v:Value)
    changes contains(k)
    ensures contains(k),
     value(k) = v
```

The `remove` operation is used to remove a key-value pair from the map. Again, the `contains` model function is changed for the specified key value, but this time `contains(k)` is false after the operation.

```
  operation remove(k:Key)
    changes contains(k)
    ensures !contains(k)
```

Again, an ESC-compatibility test can be defined. The preliminaries are the same:

```
  test escCompatibilityTest
    policy “ESC”
  {
    var int0:IntegerObject := IntegerObject.withValue(0);
    var int1:IntegerObject := IntegerObject.withValue(1);
```

The test starts by creating a new empty `Map` of `IntegerObject` to `IntegerObject`. Both `int0` and `int1` should not be contained by this map.

```
    var m:Map[IntegerObject,IntegerObject] :=
     Map[IntegerObject,IntegerObject].empty();
    assert !m.contains(int0);
    assert !m.contains(int1);
```

The put operation is then used to put the key int0 into the map with the value int0. From the frame condition, int1 is still not contained by the map.

```
    m.put(int0, int0);
    assert m.contains(int0);
    assert m.value(int0) = int0;
    assert !m.contains(int1);
```

The put operation can be used again to put the key int1 into the map with the value int1. The int0 and int1 objects should now both be contained in the map with the appropriate values.

```
    m.put(int1, int1);
    assert m.contains(int0);
    assert m.value(int0) = int0;
    assert m.contains(int1);
    assert m.value(int1) = int1;
```

The remove operation can then be used to remove the int0 key from the map. The int0 object is then no longer contained in the map but int1 is, with the same value.

```
    m.remove(int0);
    assert !m.contains(int0);
    assert m.contains(int1);
    assert m.value(int1) = int1;
```

Finally, a deny statement is used to check that there are no contradictions.

```
    deny false;
  }
}
```

This completes the test and the class which is successfully verified. The full specification of the omni.lang.Map class also contains a size model function and methods for combining maps in different ways.

B.3 List class

The `omni.lang.List` class is used to represent a sequence of values. It has a single type parameter referring to the types of the elements it can contain. In this section we present a simplified version of this class. The specification is built around a `size` function which returns the current size of the list and an `elementAt` function which returns the element currently at a specified index in the list. The valid indexes of the list run from `0` up to `size()-1`. An invariant is used to assert that the size of the list is always non-negative. The class header and model function and invariant declarations are shown below.

```
spec class List[Element] {
  model function size():integer
  model function elementAt(ind:integer):Element
    requires ind >= 0 && ind < size()

  invariant size() >= 0
```

This time derived functions are also defined in the specification. The first of these is the `contains` function which yields true iff the specified element is contained in the list. The specification asserts that the result of the function is true iff there exists a valid index whose element at that index is equal to the specified element. The specification is shown below.

```
  function contains(e:Element):boolean
    ensures result = (exists (i:integer := 0 to size()-1):
     elementAt(i) = e )
```

The `empty` constructor is used to create an empty list. It can be described by simply asserting that the size is equal to zero at the end of the constructor. This implies that elements are not accessible at any index.

```
  constructor empty()
    ensures size() = 0
```

The `add` operation is used to add an element to the end of the list. As was described in section 3.1.2, we can specify such an operation including the `size` model function in the changes clause, asserting that this is increased by one and asserting that the newly accessible index value is equal to the passed element.

```
   operation add(e:Element)
     changes size
     ensures size() = old size() + 1,
      elementAt(size()-1) = e
```

The swap operation is used to interchange the values at two indices of the list. This specification can exploit the ability to include a model function in the changes clause with different parameters. Here, the elementAt model function is asserted to be changed for indices j and k. The ensures clause describes the interchange. The specification is shown below.

```
   operation swap(j:integer, k:integer)
     requires j >= 0 && j < size(),
      k >= 0 && k < size()
     changes elementAt(j), elementAt(k)
     ensures elementAt(j) = old elementAt(k),
      elementAt(k) = old elementAt(j)
```

The indexOf function is used to return the first index at which a specified element occurs in the list or -1 if it is not in the list. This can be specified using an if expression asserting that if the specified element is not in the list then -1 is returned. Otherwise a valid index with the specified element at that position, and no occurrence of the element before it, is returned.

```
   function indexOf(e:Element):integer
     ensures if !contains(e) then
         result = -1
       else
         result >= 0 && result < size()
         && elementAt(result) = e
         && !(exists (i:integer where i >= 0 && i < result):
           elementAt(i) = e )
```

The removeFirstOf operation is used to remove the first occurrence of a specified element. This can be specified using quantifiers asserting that the elements previously before the first occurrence of the element are unchanged, and those after are shuffled forward one place. We will see during the testing of this operation that the full specification is not ESC-compatible and so a lightweight substitute is provided via a which ensures clause. The specification is shown below.

```
  operation removeFirstOf(e:Element)
    requires size() > 0, contains(e)
    changes size, elementAt
    ensures size() = old size() - 1,
     forall (j:integer := 0 to old indexOf(e) - 1):
      elementAt(j) = old elementAt(j),
     forall (k:integer := old indexOf(e) to size()-1):
      elementAt(k) = old elementAt(k+1)
    which ensures size() = old size() - 1,
     forall (e2:Element):
      if e = e2 then
        ! contains(e2)
      else
        contains(e2) = old contains(e2)
```

Again, we use an ESC-compatibility test with the same preliminaries:

```
  test escCompatibilityTest
    policy “ESC”
  {
    var int0:IntegerObject := IntegerObject.withValue(0);
    var int1:IntegerObject := IntegerObject.withValue(1);
    var int2:IntegerObject := IntegerObject.withValue(2);
```

The test starts by creating an empty list and checking that the size is zero and all the declared objects are not contained in it.

```
    var lst:List[IntegerObject] :=
     List[IntegerObject].empty();
    assert lst.size() = 0;
    assert !lst.contains(int0);
    assert !lst.contains(int1);
    assert !lst.contains(int2);
```

The `add` operation is then used to add the `int0` object to the end of the list. This makes the size equal to one and the element at index `0` equal to `int0`. The `int1` and `int2` objects are still not in the list.

```
    lst.add(int0);
    assert lst.size() = 1;
    assert lst.elementAt(0) = int0;
    assert !lst.contains(int1);
    assert !lst.contains(int2);
```

The `add` operation is then used again to add the `int1` object to the end of the list. The size of the list is now two with `int1` now at index `1` and `int0` unchanged at index `0`. The `int2` object is still not contained in the list.

```
    lst.add(int1);
    assert lst.size() = 2;
    assert lst.elementAt(1) = int1;
    assert lst.elementAt(0) = int0;
    assert !lst.contains(int2);
```

The `swap` operation is then used to swap the elements at indexes `0` and `1`, and back again.

```
    lst.swap(0, 1);
    assert lst.size() = 2;
    assert lst.elementAt(0) = int1;
    assert lst.elementAt(1) = int0;
    lst.swap(0, 1);
    assert lst.size() = 2;
    assert lst.elementAt(0) = int0;
    assert lst.elementAt(1) = int1;
```

Now the `indexOf` function is tested. The index of `int0` should be `0`, the index of `int1` should be `1` and the index of `int2` should be `-1` because it is not in the list.

```
    assert lst.indexOf(int0) = 0;
    assert lst.indexOf(int1) = 1;
    assert lst.indexOf(int2) = -1;
```

For convenience we now add `int2` to the end of the list.

```
    lst.add(int2);
    assert lst.size() = 3;
    assert lst.elementAt(0) = int0;
    assert lst.elementAt(1) = int1;
    assert lst.elementAt(2) = int2;
```

Everything to this point is verified successfully by the tool.

Finally, we attempt to remove the first occurrence of `int1`.

```
    lst.removeFirstOf(int1);
    assert lst.size() = 2;
    assert lst.elementAt(0) = int0; // fails
    assert lst.elementAt(1) = int2; // fails
```

The tool is not able to prove the final two assert statements because the specification of `removeFirstOf` is not ESC-compatible. Its complex use of quantifiers is beyond the automated prover we use. However, the lightweight substitute does at least allow the elements that are contained in a list to be reasoned about after the `removeFirstOf` operation is called although the ordering information is lost.

```
  }
}
```

This completes the test and the class which is successfully verified. The full specification of the `omni.lang.List` class also contains methods for comparing and combining lists in different ways.

Appendix C

Verification policy definitions

This appendix defines the configurable options for the verification policy system, and then gives the full details of the built-in verification policies.

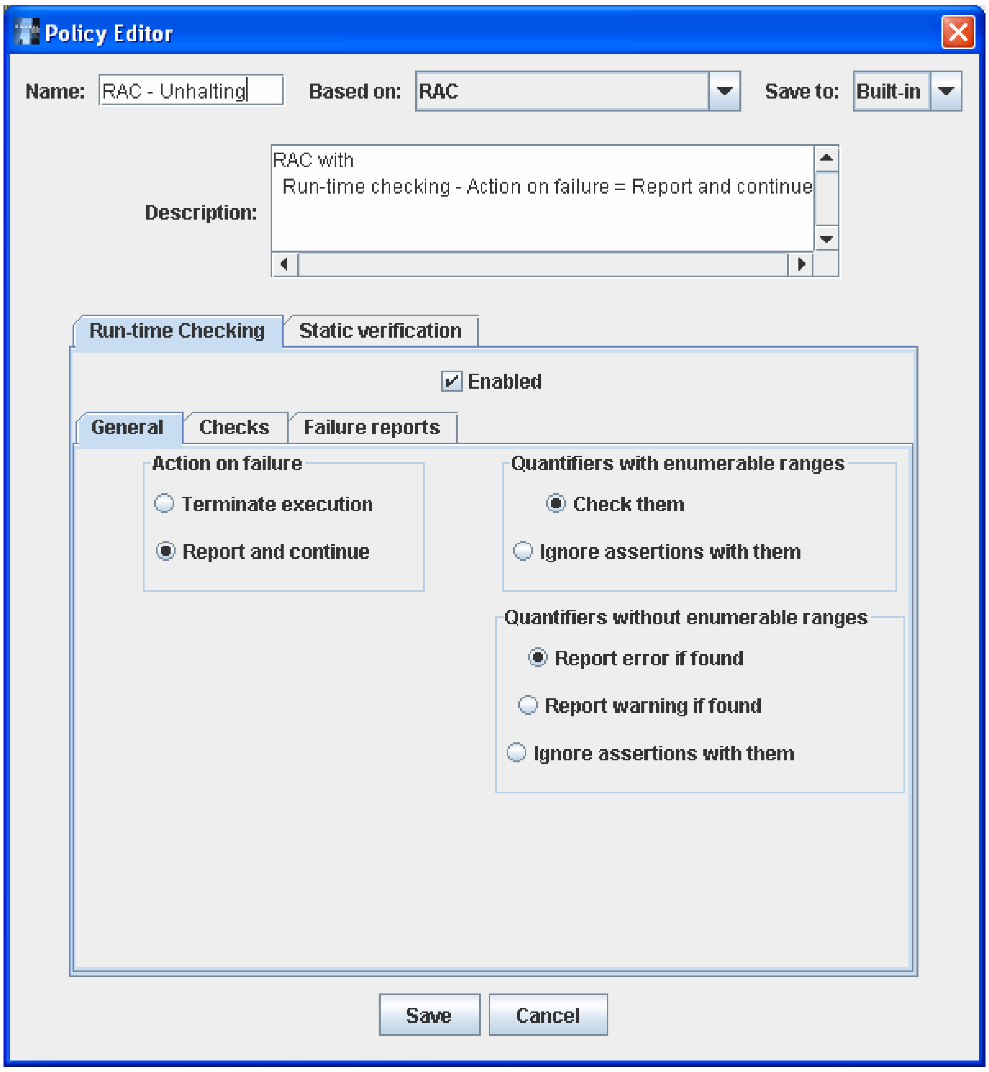
C.1 Configurable options for verification policies

Omnibus verification policies are defined in a hierarchical fashion, each policy being based on another policy with some changes. There is a root policy called ‘Blank’ which is at the root of all policy hierarchies and has no parent. For example, we define ‘RAC – Unhalting’, a kind of RAC that reports assertion failures but then continues, to be based upon the ‘RAC’ policy but with the ‘Action on failure’ property set to ‘Report and continue’. This makes it much easier to define related policies than if they all had to be configured separately.

Each policy has its own name and can be saved either within the settings of the IDE installation or in the currently open project. A description field keeps track of how the verification policy is different from the one it is based upon.

There is then a range of configurable options which are divided into two groups: run-time checking and static verification. Both these groups have a property for whether they are enabled and then three sub-groups of properties: general properties, checks and failure reports. Each property has either a boolean Y/N value specified via a check box or a value from a list selected via series of option buttons. If run-time checking or static verification is disabled then the settings used in that section are not used and that particular form of verification is not used for the class.

A screenshot of the Policy Editor is shown below.



Name: RAC - Unhalting
Based on: RAC
Save to: Built-in
Description: RAC with
Run-time checking - Action on failure = Report and continue

[✓] Enabled

Action on failure: ○ Terminate execution [● Report and continue]

Quantifiers with enumerable ranges: [● Check them] ○ Ignore assertions with them

Quantifiers without enumerable ranges: [● Report error if found] ○ Report warning if found ○ Ignore assertions with them

C.1.1 Run-time checking properties

The details of the run-time checking properties and their corresponding options are given in the table below.

Group	**Property**	**Options**
None	a. Enabled	Y/N
General	b. Action on Failure	1. Terminate execution 2. Report and continue
"	c. Quantifiers with enumerable ranges	1. Check them 2. Ignore assertions with them
"	d. Quantifiers without enumerable ranges	1. Report error if found 2. Report warning if found 3. Ignore assertions with them
Checks	e. Check requires clauses	Y/N
"	f. changes clauses	1. Check if possible, error if not 2. Check if possible, warn if not 3. Check if possible, ignore if not 4. Ignore them
"	g. Check ensures clauses	Y/N
"	h. Check requirements	Y/N
"	i. Check assert statements	Y/N
"	j. assume statements	1. Check them 2. Generate error if found 3. Ignore them
"	k. Check loop invariants	Y/N
Failure reports	l. Include description	Y/N
"	m. Include source	Y/N
"	n. Include stack trace	Y/N
"	o. Include parameter values	Y/N
"	p. Include attribute values	Y/N
"	q. Include local variable	Y/N

	values	
"	r. Include execution path info	Y/N

C.1.2 Static verification properties

The details of all the static verification properties and their corresponding options are given in the table below.

Group	**Property**	**Options**
None	a. Enabled	Y/N
General	b. Prover	1. Interactive PVS prover 2. Automated Simplify prover
"	c. Generate PVS lemma files	Y/N
"	d. Recursion	1. Check assertions with recursion 2. Generate error if found 3. Warn and check 4. Ignore assertions with recursion
"	e. which ensures clauses	1. Always use 2. Use when recursion in ensures 3. Never use
Checks	f. Check requires clauses	Y/N
"	g. Check changes clauses	Y/N
"	h. Check ensures clauses	Y/N
"	i. Requirements	1. Check from behaviour 2. Check from implementation
"	j. Check assert statements	Y/N
"	k. assume statements	1. Check them 2. Generate error if found 3. Ignore them

"	l. Loops without invariants	1. Unravel them 2. Warn and then unravel them 3. Generate error if found
Failure reports	m. Include description	Y/N
"	n. Include source	Y/N
"	o. Include stack trace	Y/N
"	p. Include parameter values	Y/N
"	q. Include attribute values	Y/N
"	r. Include local variable values	Y/N
"	s. Include execution path info	Y/N

C.2 Built-in verification policies

In this section we give the details of the built-in policies in terms of the properties defined in the previous section. First we give the full property settings for the 3 core built-in policies (RAC, ESC and FFV) and then we describe customisations of them.

C.2.1 Properties for RAC, ESC and FFV

The following tables present the policy settings for the RAC, ESC and FFV approaches. The column labels **a-r** and **a-s** correspond to the properties defined in C.1.1 and C.1.2, respectively. The values in the cells below these column labels correspond to the options from the properties which are used for that policy. The values in italics are not relevant for that policy because the appropriate form of verification is disabled but is relevant to some policies discussed in section C.2.2 which are based on these.

Run-time checking properties

Policy	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**	**n**	**o**	**p**	**q**	**r**
RAC	Y	1	1	1	Y	1	Y	Y	Y	1	Y	Y	Y	Y	N	N	N	N
ESC	N	*1*	*1*	*1*	*Y*	*1*	*Y*	*Y*	*Y*	*1*	*Y*	*Y*	*Y*	*Y*	*N*	*N*	*N*	*N*
FFV	N	*1*	*1*	*1*	*Y*	*1*	*Y*	*Y*	*Y*	*1*	*Y*	*Y*	*Y*	*Y*	*N*	*N*	*N*	*N*

Static verification properties

Policy	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**	**n**	**o**	**p**	**q**	**r**	**s**
RAC	N	*1*	*Y*	*1*	*1*	*Y*	*Y*	*Y*	*1*	*Y*	*1*	*1*	*Y*	*Y*	*Y*	*N*	*N*	*N*	*N*
ESC	Y	2	Y	2	1	Y	Y	Y	2	Y	3	1	Y	Y	Y	N	N	N	N
FFV	Y	1	Y	1	3	Y	Y	Y	1	Y	2	3	Y	Y	Y	N	N	N	N

C.2.2 Extensions of RAC, ESC and FFV

The full policy properties for the 3 core policies were defined in the previous section. In this section we describe some other policies that are based upon them. Instead of giving the full property details, we simply describe how they differ from the policy which they are based upon.

Policy	**Based on**	**Property changes**

RAC – Verbose	RAC	Run-time checking:
		o = Y
		p = Y
		q = Y
		r = Y

RAC – Unhalting	RAC	Run-time checking: b = 2

RAC – Minimal Unhalting	RAC – Unhalting	Run-time checking:
		l = N
		m = N
		n = N

ESC – Verbose	ESC	Static verification: p = Y

Formatted: Line spacing: 1.5 lines

		q = Y
		r = Y
		s = Y

ESC with RAC client checks	ESC	<u>Run-time checking:</u>
		a = Y
		f = 4
		g = N
		h = N
		i = N
		j = 3
		k = N

ESC with RAC assume checks	ESC	<u>Run-time checking:</u>
		a = Y
		e = N
		f = 4
		g = N
		h = N
		i = N
		k = N

FFV with RAC client checks	FFV	<u>Run-time checking:</u>
		a = Y
		f = 4
		g = N
		h = N
		i = N
		j = 3
		k = N

FFV with loop unravelling	FFV	<u>Static verification:</u> l = 2

FFV – Automated	FFV	<u>Static verification:</u>
		b = 2
		d = 3
		e = 2

Appendix D

Symbolic execution example

This appendix will work through the symbolic execution of a sorting implementation. The algorithm used is *insertion sort* [99]. The algorithm is implemented as a static function which accepts a `List` parameter and returns a sorted version of the `List`. Some simple correctness annotations are provided in an ensures clause. These express that the resulting `List` should be sorted and that the size of the input and output lists should be the same. The implementation involves the use of nested `while` loops. We consider the use of the ESC approach which involves the loop unravelling technique.

D.1 A sorting example

The source of the example is given below. This example has previously been presented [97] where the use of RAC, ESC and FFV to verify it was discussed.

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

```
 7:        result.size() = inList.size()
 8:  {
 9:   var a:List := inList;
10:   var m:integer := 1;
11:   while (m < a.size()) {
12:     var key:integer := a.elementAt(m);
13:     var l:integer := m-1;
14:     while (l >= 0 && l < a.size()
15:         && a.elementAt(l) >= key) {
16:       a.set(l+1,a.elementAt(l));
17:       l := l-1;
18:     }
19:     a.set(l+1,key);
20:     m := m+1;
21:   }
22:   return a;
23:  }
24: }
```

D.2 The symbolic execution process

The symbolic execution of this example begins with the processing of the `Sorter` class and the loading of the top-level packages, local and imported classes and local methods. The members, here only a single static function, are then processed.

The state of the symbolic executor consists of 3 components: the program counter, the path condition, and the current values for each variable. Initially, the program counter is the header of the function (line 2), the path condition is empty and there are no local variables.

```
Program Counter:
  2: public static function insertSort(inList:List):List
Path Condition:
  true
Variables:
  None
```

We then start processing the static function. New variables are declared for the parameters and reassigned arbitrary values. The typing information of the new symbol is recorded in the path condition. The program counter advances to the start of the code, ready for symbolic execution of the implementation.

```
Program Counter:
  8: {
Path Condition:
  {inList_0:List}
Variables:
  inList: inList_0
```

The first statements are two declarations, shown below:

```
 9:   var a:List := inList;
10:   var m:integer := 1;
```

These declare new local variables and assign them values. The `a` variable is assigned the current value of the `inList` parameter which is the symbol `inList_0`. The `m` variable is assigned the literal value `1`. The program counter then advances to line 11.

```
Program Counter:
  11:   while (m < a.size()) {
Path Condition:
  {inList_0:List}
Variables:
  inList: inList_0
  a: inList_0
  m: 1
```

The first `while` statement, shown below, is then reached.

```
11:   while (m < a.size()) {
```

This involves two branches. In the first, the condition is assumed to be false when the loop is first reached and the code following the loop is executed. In the second, the condition is assumed to be true when the loop is first reached, the body of the loop is executed and then the condition is assumed to be false after this iteration.

Branch 1 of while loop at line 11 (no iterations)

For branch 1, the condition is assumed to be false. This involves the evaluation of the negation of the loop condition being added to the path condition. Flow then moves to the statement immediately after the loop, the `return` statement.

```
Program Counter:
  22:   return a;
Path Condition:
  {inList_0:List},
  !(1 < inList_0.size())
Variables:
  inList: inList_0
  a: inList_0
  m: 1
```

The symbolic execution of the `return` statement first involves the declaration of a `result` variable which is assigned the evaluated of the expression to be returned. The post-conditions are then checked. This generates VCs that the path condition implies each ensures clause.

```
VC #1:
  {inList_0:List}
  && !(1 < inList_0.size())
==>
  forall (i:integer := 1 to inList_0.size()-1):
    inList_0.elementAt(i) >= inList_0.elementAt(i-1)
```

```
VC #2:
  {inList_0:List}
  && !(1 < inList_0.size())
==>
  inList_0.size() = inList_0.size()
```

This completes the symbolic execution of this branch as indicated by the **terminate** primitive in the symbolic execution rule of the `return` statement.

Branch 2 of while loop at line 11 (one iteration)

In branch 2, we consider the execution of one iteration of the body of the `while` loop. We start by assuming that the loop condition is true. So we know that `inList_0.size()` is greater than `1`.

```
Program Counter:
  12:      var key:integer := a.elementAt(m);
Path Condition:
  {inList_0:List},
  1 < inList_0.size()
Variables:
  inList: inList_0
  a: inList_0
  m: 1
```

We now reach the two declaration statements. These add new local variables and assign them values derived from the existing variables.

```
12:      var key:integer := a.elementAt(m);
13:      var l:integer := m-1;
```

The statement at line 12 involves a call of the `elementAt` function which contains a pre-condition. This generates a VC that this pre-condition is met.

```
VC #3:
  {inList_0:List}
  && 1 < inList_0.size()
==>
  inList_0.elementAt_pre(1)
```

After symbolic execution of these statements, the state of the symbolic executor is as follows:

```
Program Counter:
  14:      while (l >= 0 && l < a.size()
  15:           && a.elementAt(l) >= key) {
Path Condition:
  {inList_0:List},
  1 < inList_0.size()
Variables:
  inList: inList_0
  a: inList_0
  m: 1
  key: inList_0.elementAt(1)
  l: 1-1
```

We now reach the `while` loop. As described in the symbolic execution rule for `while` loops, the first stage is to check any pre-conditions in the loop condition. The `elementAt` function call requires such a check.

```
VC #4:
  {inList_0:List}
  && 1 < inList_0.size()
==>
  inList_0.elementAt_pre(1-1)
```

We can now consider the separate branches of the symbolic execution of the `while` loop: one for no iterations of the loop body and one for a single iteration of the loop body.

Branch 1 of while loop at line 14 (no iterations)

For branch 1, the condition is assumed to be false. This involves the evaluation of the negation of the loop condition being added to the path condition. Flow then moves to the statement immediately after the loop, the call of the `set` operation.

```
Program Counter:
  19:      a.set(l+1,key);
Path Condition:
  {inList_0:List},
  1 < inList_0.size(),
  !(0 >= 0 && 0 < inList_0.size()
     && inList_0.elementAt(0) >= inList_0.elementAt(1))
Variables:
  inList: inList_0
  a: inList_0
  m: 1
  key: inList_0.elementAt(1)
  l: 0
```

The loop is immediately followed by two statements: a call of the `set` operation and an assignment to increment the value of `m` by one.

```
19:      a.set(l+1,key);
20:      m := m+1;
```

The `set` operation has a pre-condition that the passed index is valid. A VC is generated to check this.

```
VC #5:
  {inList_0:List}
  && 1 < inList_0.size()
  && !(0 >= 0 && 0 < inList_0.size()
     && inList_0.elementAt(0) >= inList_0.elementAt(1))
==>
  inList_0.set_pre(0+1, inList_elementAt(1))
```

The statements on lines 19 and 20 can then be executed with the following consequences for the state of the symbolic executor. The program counter moves to the statement after the loop, the `return` statement.

```
Program Counter:
  22:    return a;
Path Condition:
  {inList_0:List},
  1 < inList_0.size(),
  !(0 >= 0 && 0 < inList_0.size()
      && inList_0.elementAt(0) >= inList_0.elementAt(1))
Variables:
  inList: inList_0
  a: inList_0.set(0+1, inList_0.elementAt(1))
  m: 1+1
  key: inList_0.elementAt(1)
  l: 0
```

VCs are then generated for the checks of the ensures clauses at the `return` statement.

```
VC #6:
  {inList_0:List}
  && 1 < inList_0.size()
  && !(0 >= 0 && 0 < inList_0.size()
      && inList_0.elementAt(0) >= inList_0.elementAt(1))
==>
  forall (i:integer := 1 to
         inList_0.set(1, inList_0.elementAt(1)).size()-1):
    inList_0.set(1,inList_0.elementAt(1)).elementAt(i)
    >= inList_0.set(1,inList_0.elementAt(1)).elementAt(i-1)
```

```
VC #7:
  {inList_0:List}
  && 1 < inList_0.size()
  && !(0 >= 0 && 0 < inList_0.size()
      && inList_0.elementAt(0) >= inList_0.elementAt(1))
==>
  inList_0.set(1,inList_0.elementAt(1)).size()
    = inList_0.size()
```

This completes this branch.

Branch 2 of while loop at line 14 (one iteration)

In branch 2, we consider the execution of one iteration of the body of the `while` loop. We start by assuming that the loop condition is true.

```
Program Counter:
  16:        a.set(l+1,a.elementAt(l));
Path Condition:
  {inList_0:List},
  1 < inList_0.size(),
  0 >= 0 && 0 < inList_0.size()
      && inList_0.elementAt(0) >= inList_0.elementAt(1)
Variables:
  inList: inList_0
  a: inList_0
  m: 1
  key: inList_0.elementAt(1)
  l: 0
```

Flow then moves to the statements in the body of the loop. These are another call of the `set` operation and an assignment to decrement `l` by one.

```
16:        a.set(l+1,a.elementAt(l));
17:        l := l-1;
```

The statement at line 16 involves calls of the `elementAt` function and `set` operation, both of which have pre-conditions. This generates two VCs:

```
VC #8:
  {inList_0:List}
  && 1 < inList_0.size()
  && 0 >= 0 && 0 < inList_0.size()
     && inList_0.elementAt(0) >= inList_0.elementAt(1)
==>
  inList_0.elementAt_pre(0)
```

```
VC #9:
  {inList_0:List}
  && 1 < inList_0.size()
  && 0 >= 0 && 0 < inList_0.size()
     && inList_0.elementAt(0) >= inList_0.elementAt(1)
==>
  inList_0.set_pre(0+1, inList_0.elementAt(0))
```

The statements at lines 16 and 17 can then be symbolically executed with the following consequences for the state of the symbolic executor. The program counter moves back to the header of the loop at line 14.

```
Program Counter:
  14:      while (l >= 0 && l < a.size()
  15:          && a.elementAt(l) >= key) {
Path Condition:
  {inList_0:List},
  1 < inList_0.size(),
  0 >= 0 && 0 < inList_0.size()
     && inList_0.elementAt(0) >= inList_0.elementAt(1)
Variables:
  inList: inList_0
  a: inList_0.set(1, inList_0.elementAt(0))
  m: 1
  key: inList_0.elementAt(1)
  l: 0-1
```

The pre-conditions of the method calls in the loop condition are then re-checked. The call of the `elementAt` function in line 15 generates another VC, shown below. The last two conjuncts of the path condition are derived from the evaluation order. The `elementAt` function will only be evaluated (and hence its pre-condition have to hold) if the first conjuncts of the loop condition are true. Since there is a contradiction (`-1 >= 0`) in the path condition for the VC, it is okay that the pre-condition of the `elementAt` function is not satisfied.

```
VC #10:
  {inList_0:List}
  && 1 < inList_0.size()
  && 0 >= 0 && 0 < inList_0.size()
     && inList_0.elementAt(0) >= inList_0.elementAt(1)
  && -1 >= 0
  && -1 < inList_0.set(1, inList_0.elementAt(0)).size()
==>
  inList_0.set(1, inList_0.elementAt(0)).elementAt_pre(-1)
```

The loop condition is then assumed to be false to complete the single iteration and the program counter advances to the statement immediately after the loop.

```
Program Counter:
  19:      a.set(l+1,key);
Path Condition:
  {inList_0:List},
  1 < inList_0.size(),
  0 >= 0 && 0 < inList_0.size()
     && inList_0.elementAt(0) >= inList_0.elementAt(1),
  !(-1 >= 0
     && -1 < inList_0.set(1, inList_0.elementAt(0)).size()
     && inList_0.set(1, inList_0.elementAt(0))
                                        .elementAt(-1)
          >= inList_0.elementAt(1))
Variables:
  inList: inList_0
  a: inList_0.set(1, inList_0.elementAt(0))
  m: 1
  key: inList_0.elementAt(1)
  l: -1
```

The next two statements are another call of the `set` operation and an assignment to increment `m` by one.

```
19:      a.set(l+1,key);
```

```
20:      m := m+1;
```

The `set` operation call at line 19 involves the generation of another VC:

```
VC #11:
  {inList_0:List}
  && 1 < inList_0.size()
  && 0 >= 0 && 0 < inList_0.size()
     && inList_0.elementAt(0) >= inList_0.elementAt(1)
  && !(-1 >= 0
     && -1 < inList_0.set(1, inList_0.elementAt(0)).size()
     && inList_0.set(1, inList_0.elementAt(0))
                                        .elementAt(-1)
          >= inList_0.elementAt(1))
==>
  inList_0.set(1, inList_0.elementAt(0)).set_pre(0,
                                      inList_0.elementAt(1)
```

The statements at lines 19 and 20 can then be symbolically executed with the following consequences for the state of the symbolic executor. The program counter then moves back to the header of the loop at line 11.

```
Program Counter:
  11:   while (m < a.size()) {
Path Condition:
  {inList_0:List},
  1 < inList_0.size(),
  0 >= 0 && 0 < inList_0.size()
     && inList_0.elementAt(0) >= inList_0.elementAt(1),
  !(-1 >= 0
     && -1 < inList_0.set(1, inList_0.elementAt(0)).size()
     && inList_0.set(1, inList_0.elementAt(0))
                                        .elementAt(-1)
          >= inList_0.elementAt(1))
Variables:
  inList: inList_0
  a: inList_0.set(1, inList_0.elementAt(0)).set(0,
                                    inList_0.elementAt(1)
  m: 1+1
  key: inList_0.elementAt(1)
  l: -1
```

The loop condition of this loop is then also assumed to be false to complete its single iteration and the program counter advances to the `return` statement following the loop.

```
Program Counter:
  22:   return a;
Path Condition:
  {inList_0:List},
  1 < inList_0.size(),
  0 >= 0 && 0 < inList_0.size()
     && inList_0.elementAt(0) >= inList_0.elementAt(1),
  !(-1 >= 0
     && -1 < inList_0.set(1, inList_0.elementAt(0)).size()
     && inList_0.set(1, inList_0.elementAt(0))
                                         .elementAt(-1)
          >= inList_0.elementAt(1))
Variables:
  inList: inList_0
  a: inList_0.set(1, inList_0.elementAt(0)).set(0,
                                     inList_0.elementAt(1)
  m: 1+1
  key: inList_0.elementAt(1)
  l: -1
```

VCs are then generated for the checks of the ensures clauses at the `return` statement.

```
VC #12:
  {inList_0:List}
  && 1 < inList_0.size()
  && !(0 >= 0 && 0 < inList_0.size()
     && inList_0.elementAt(0) >= inList_0.elementAt(1))
==>
  forall (i:integer := 1 to
        inList_0.set(1, inList_0.elementAt(0))
                   .set(0, inList_0.elementAt(1).size()-1):
    inList_0.set(1, inList_0.elementAt(0))
              .set(0, inList_0.elementAt(1).elementAt(i)
    >= inList_0.set(1, inList_0.elementAt(0))
              .set(0, inList_0.elementAt(1).elementAt(i-1)
```

```
VC #13:
  {inList_0:List}
  && 1 < inList_0.size()
  && !(0 >= 0 && 0 < inList_0.size()
     && inList_0.elementAt(0) >= inList_0.elementAt(1))
==>
  inList_0.set(1, inList_0.elementAt(0))
            .set(0, inList_0.elementAt(1).size()
    = inList_0.size()
```

This completes this branch and the symbolic execution of the example.

Appendix E

An interactive PVS proof

This appendix works through the details of the proof of a particular VC using PVS. The VC is from the full formal verification of the library project discussed in chapter 8.

E.1 Relevant Omnibus source code

We consider a proof for the following `registerMember` operation.

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

We will prove that it satisfies the `memberNosInMembersUnique` invariant.

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

These definitions make use of two local methods: the `members` model function and the `getNewMemberNo` function.

```
public model function members():Collection[Member]

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

The `newMember` constructor of the `Member` class is used to construct the new `Member` to add to the `members` collection.

```
public class Member {
  public model function memberNo():integer
  public model function name():Name
  ...

  public constructor newMember(memNo:integer, na:Name,
     ad1:String, ad2:String, pCode:String, tNo:integer,
     isAd:boolean, isSta:boolean)
    ensures memberNo() = memNo,
     name() = na,
     ...
  ...
}
```

The `members` model function of `Lib` holds a `Collection` object for which a simplified specification is given below:

```
public class Collection[Element] isa Iterable[Element] {
  public model function contains(e2:Element):boolean

  public operation add(e:Element)
    changes contains(e)
    ensures contains(e)
  ...
}
```

The post-condition of the `add` function includes the assertion that all elements other than `e` remain unchanged as implied by the `changes` clause.

E.2 Generated PVS

The verification process starts with the selection of ‘Interactive FFV’ as the verification process to use for the `Lib` class. On selecting ‘Verify Project’, the project is translated into the generic logic and then PVS. The axioms file containing the formalisation of the specifications of the project and used libraries is 6718 lines long. There is then a series of obligation files and default proof attempts for each file in the project which is to be verified using PVS. The obligations file for the `Lib` class is 2337 lines long and contains 231 VCs.

The VC we are concerned with is VC number 67 which is shown below.

```
  % VC #67:
  %     Source file: C:\Program
      Files\eclipse\workspace\OmnibusIDE\myprojects\Impl4-
      ffv\Lib.obs
  %     Line number: 160
  %     Method: registerMember
  %     Name:   Check the public behaviour of the
      registerMember operation satisfies the public invariant
      at line 23
  vc_67: CONJECTURE
    (FORALL (old_this:v_Lib), (name:v_Name),
           (ad1:v_omni_lang_String),
           (ad2:v_omni_lang_String),
           (pCode:v_omni_lang_String), (telNo:int),
           (isAdult:bool), (isStaff:bool), (this:Lib):
      (Lib_registerMember_pub_post(old_this, name, ad1,
                                 ad2, pCode, telNo, isAdult,
                                 isStaff, this)
      IMPLIES Lib_invariant_3(this))
    )
```

The VC is expressed using the function symbols defining the post-condition of the method and the invariant. `Lib_registerMember_pub_post` is declared and then specified as follows:

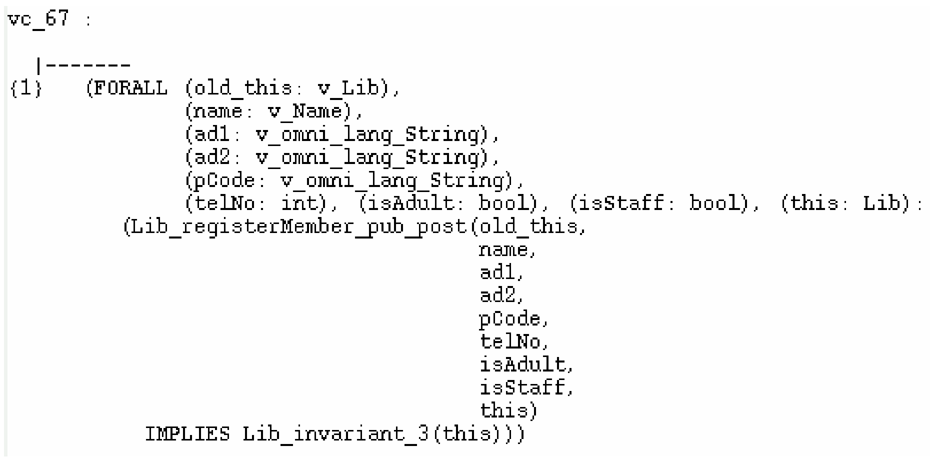
```
Lib_registerMember_pub_post(old_this:v_Lib, name:v_Name,
     ad1:v_omni_lang_String, ad2:v_omni_lang_String,
     pCode:v_omni_lang_String, telNo:int, isAdult:bool,
     isStaff:bool, this:Lib):bool

Lib_registerMember_pub_post_ax: AXIOM
  (FORALL (old_this:v_Lib), (name:v_Name),
              (ad1:v_omni_lang_String),
              (ad2:v_omni_lang_String),
              (pCode:v_omni_lang_String), (telNo:int),
              (isAdult:bool), (isStaff:bool), (this:Lib):
     (Lib_registerMember_pub_post(old_this, name, ad1, ad2,
                                  pCode, telNo, isAdult,
                                  isStaff, this)
    = ((((Lib_loans(old_this) = Lib_loans(this)) AND
        (Lib_copies(old_this) = Lib_copies(this))) AND
        (Lib_cards(old_this) = Lib_cards(this))) AND
        (Lib_members(this) =
          omni_lang_Collection_add(Lib_members(old_this),
            Member_newMember(Lib_getNewMemberNo(old_this),
                             name, ad1, ad2, pCode, telNo,
                                isAdult, isStaff)))))
  )
AUTO_REWRITE+ Lib_registerMember_pub_post_ax
```

`Lib_invariant_3` is declared and then specified as follows:

```
Lib_invariant_3(this:Lib):bool

Lib_invariant_3_ax: AXIOM
  (FORALL (this:Lib):
     (Lib_invariant_3(this) =
      (FORALL (m:v_Member):
        (omni_lang_Iterable_contains(Lib_members(this), m)
      IMPLIES (FORALL (m2:v_Member):
        (omni_lang_Iterable_contains(Lib_members(this), m2)
        IMPLIES
          ((Member_memberNo(m) /= Member_memberNo(m2))
          OR (m = m2)))
        ))
     ))
  )
AUTO_REWRITE+ Lib_invariant_3_ax
```

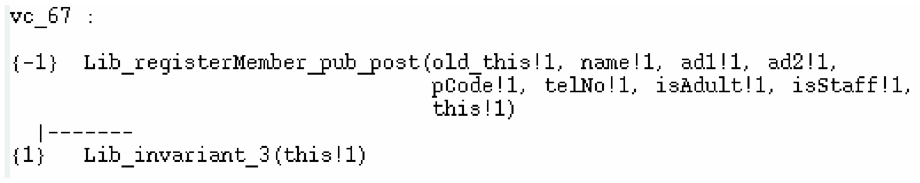
E.3 Details of the proof process

To start the proof process, we start PVS, open the `Lib` obligations file, move to VC 67 and select ‘prove’. The initial proof status is shown below:



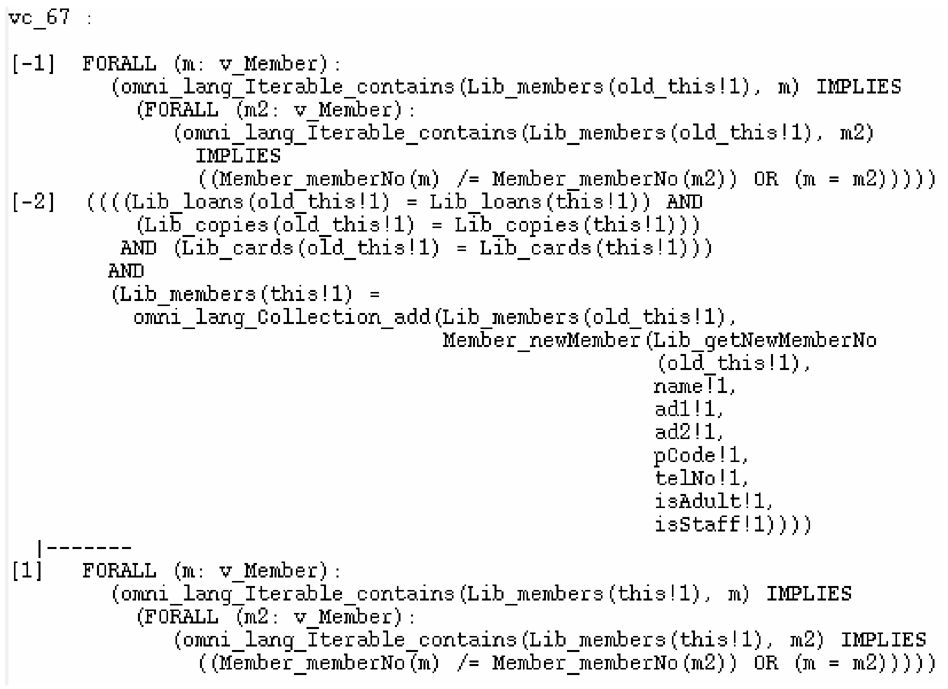
```
vc_67 :

  |-------
{1}   (FORALL (old_this: v_Lib),
              (name: v_Name),
              (ad1: v_omni_lang_String),
              (ad2: v_omni_lang_String),
              (pCode: v_omni_lang_String),
              (telNo: int), (isAdult: bool), (isStaff: bool), (this: Lib):
         (Lib_registerMember_pub_post(old_this,
                                      name,
                                      ad1,
                                      ad2,
                                      pCode,
                                      telNo,
                                      isAdult,
                                      isStaff,
                                      this)
           IMPLIES Lib_invariant_3(this)))
```

The proof then proceeds by skolemising and flattening to give us the following VC in terms of the introduced skolem constants.



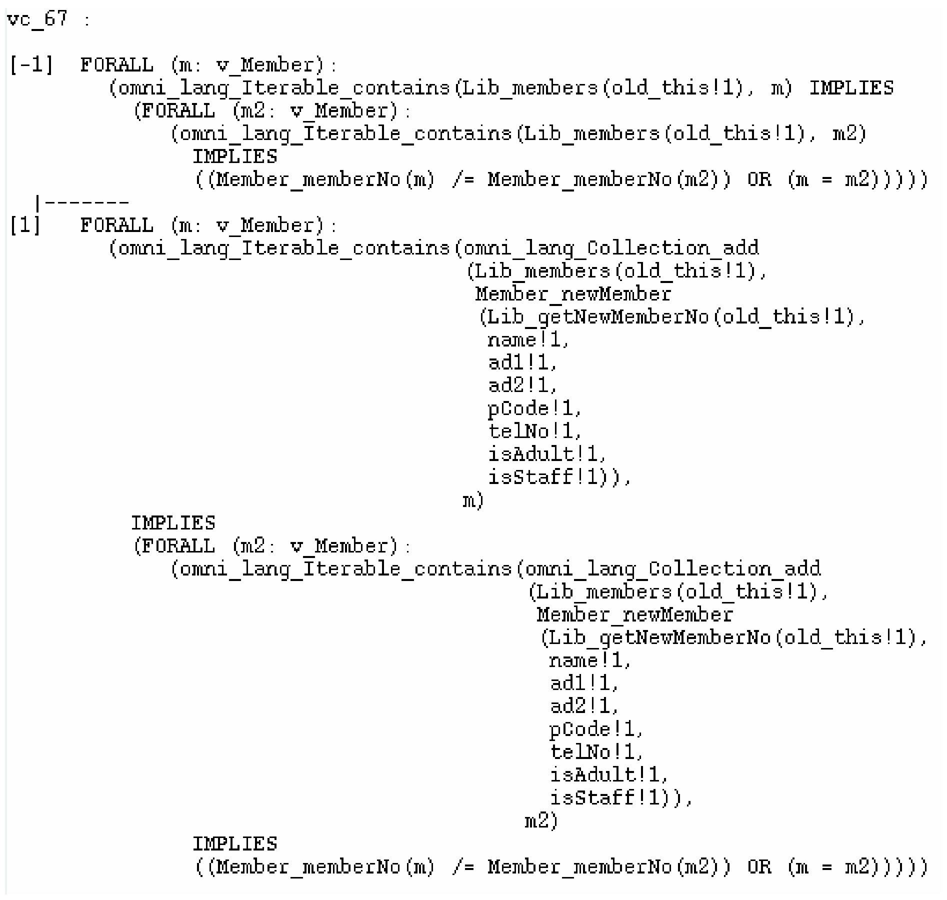
```
vc_67 :

{-1}  Lib_registerMember_pub_post(old_this!1, name!1, ad1!1, ad2!1,
                                  pCode!1, telNo!1, isAdult!1, isStaff!1,
                                  this!1)
  |-------
{1}   Lib_invariant_3(this!1)
```

Next we get the predicate for the validity of `old_this!1` using `typepred`. This gives us the knowledge that `Lib_invariant_3` holds over `old_this!1`. We then re-write the two instances of `Lib_invariant_3` (over `this!1` and `old_this!1`) using `Lib_invariant_3_ax` and do the same thing for `Lib_registerMember_pub_post` using `Lib_registerMember_pub_post_ax`. This gives us the following:



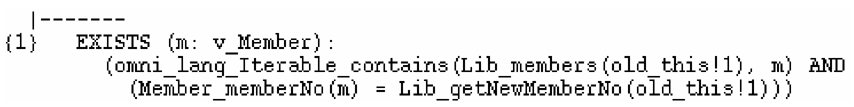
```
vc_67 :

[-1]  FORALL (m: v_Member):
        (omni_lang_Iterable_contains(Lib_members(old_this!1), m) IMPLIES
          (FORALL (m2: v_Member):
             (omni_lang_Iterable_contains(Lib_members(old_this!1), m2)
               IMPLIES
                ((Member_memberNo(m) /= Member_memberNo(m2)) OR (m = m2)))))
[-2]  ((((Lib_loans(old_this!1) = Lib_loans(this!1)) AND
          (Lib_copies(old_this!1) = Lib_copies(this!1)))
         AND (Lib_cards(old_this!1) = Lib_cards(this!1)))
        AND
        (Lib_members(this!1) =
          omni_lang_Collection_add(Lib_members(old_this!1),
                                   Member_newMember(Lib_getNewMemberNo
                                                     (old_this!1),
                                                    name!1,
                                                    ad1!1,
                                                    ad2!1,
                                                    pCode!1,
                                                    telNo!1,
                                                    isAdult!1,
                                                    isStaff!1))))
  |-------
[1]   FORALL (m: v_Member):
        (omni_lang_Iterable_contains(Lib_members(this!1), m) IMPLIES
          (FORALL (m2: v_Member):
             (omni_lang_Iterable_contains(Lib_members(this!1), m2) IMPLIES
               ((Member_memberNo(m) /= Member_memberNo(m2)) OR (m = m2)))))
```

We can then flatten `[-2]` and use the '`Lib_members(this!1) = ...`' part to rewrite `Lib_members(this!1)` in [1] in terms of `Lib_members(old_this!1)`. This gives us:



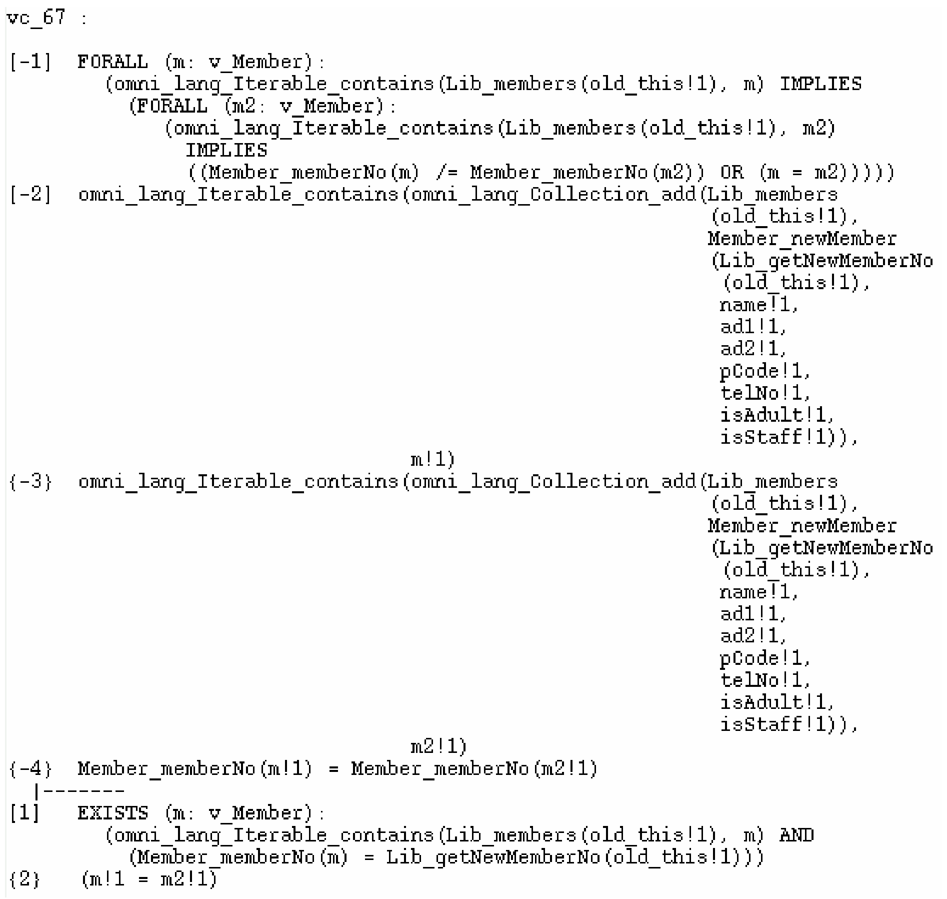
```
vc_67 :

[-1]  FORALL (m: v_Member):
        (omni_lang_Iterable_contains(Lib_members(old_this!1), m) IMPLIES
          (FORALL (m2: v_Member):
             (omni_lang_Iterable_contains(Lib_members(old_this!1), m2)
               IMPLIES
               ((Member_memberNo(m) /= Member_memberNo(m2)) OR (m = m2)))))
  |-------
[1]   FORALL (m: v_Member):
        (omni_lang_Iterable_contains(omni_lang_Collection_add
                                      (Lib_members(old_this!1),
                                       Member_newMember
                                        (Lib_getNewMemberNo(old_this!1),
                                         name!1,
                                         ad1!1,
                                         ad2!1,
                                         pCode!1,
                                         telNo!1,
                                         isAdult!1,
                                         isStaff!1)),
                                      m)
          IMPLIES
          (FORALL (m2: v_Member):
             (omni_lang_Iterable_contains(omni_lang_Collection_add
                                           (Lib_members(old_this!1),
                                            Member_newMember
                                             (Lib_getNewMemberNo(old_this!1),
                                              name!1,
                                              ad1!1,
                                              ad2!1,
                                              pCode!1,
                                              telNo!1,
                                              isAdult!1,
                                              isStaff!1)),
                                           m2)
               IMPLIES
               ((Member_memberNo(m) /= Member_memberNo(m2)) OR (m = m2)))))
```

We can then use `Lib_getNewMemberNo_pub_ax` to give us the knowledge that there is no existing `Member` in `members` with this `memberNo`. The negation is removed and the assertion is placed as the antecedent numbered `{1}` shown below.



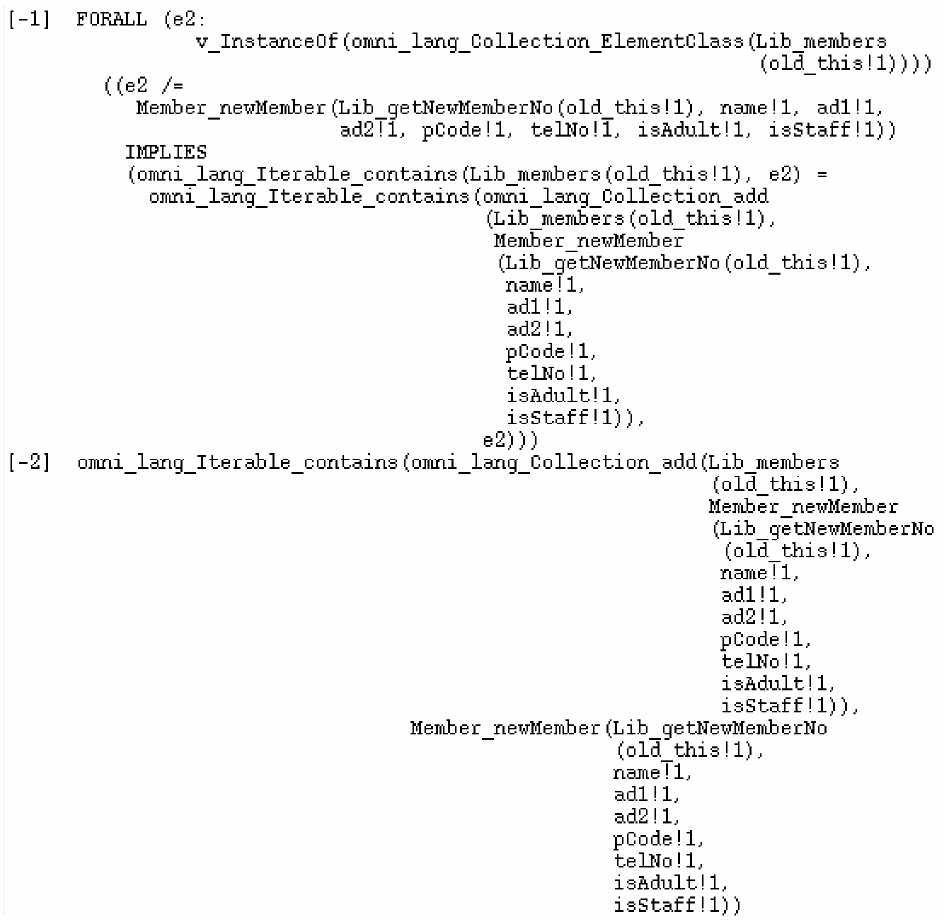
```
  |-------
{1}   EXISTS (m: v_Member):
        (omni_lang_Iterable_contains(Lib_members(old_this!1), m) AND
          (Member_memberNo(m) = Lib_getNewMemberNo(old_this!1)))
```

Next we skolemize and flatten the nested quantifiers in `[2]` which gives us arbitrary skolem symbols `m!1` and `m2!1` which we know are both contained in '`this.members()`' and we must prove are either equal or have different `memberNos`.



```
vc_67 :

[-1]  FORALL (m: v_Member):
        (omni_lang_Iterable_contains(Lib_members(old_this!1), m) IMPLIES
          (FORALL (m2: v_Member):
             (omni_lang_Iterable_contains(Lib_members(old_this!1), m2)
               IMPLIES
               ((Member_memberNo(m) /= Member_memberNo(m2)) OR (m = m2)))))
[-2]  omni_lang_Iterable_contains(omni_lang_Collection_add(Lib_members
                                                           (old_this!1),
                                                          Member_newMember
                                                          (Lib_getNewMemberNo
                                                            (old_this!1),
                                                           name!1,
                                                           ad1!1,
                                                           ad2!1,
                                                           pCode!1,
                                                           telNo!1,
                                                           isAdult!1,
                                                           isStaff!1)),
                                m!1)
{-3}  omni_lang_Iterable_contains(omni_lang_Collection_add(Lib_members
                                                           (old_this!1),
                                                          Member_newMember
                                                          (Lib_getNewMemberNo
                                                            (old_this!1),
                                                           name!1,
                                                           ad1!1,
                                                           ad2!1,
                                                           pCode!1,
                                                           telNo!1,
                                                           isAdult!1,
                                                           isStaff!1)),
                                m2!1)
{-4}  Member_memberNo(m!1) = Member_memberNo(m2!1)
  |-------
[1]   EXISTS (m: v_Member):
        (omni_lang_Iterable_contains(Lib_members(old_this!1), m) AND
          (Member_memberNo(m) = Lib_getNewMemberNo(old_this!1)))
{2}   (m!1 = m2!1)
```

Now we can use `omni_lang_Collection_add_pub_ax` to give us the specification for the `add` method which we can then use to determine whether the new `members` collection contains elements from whether the old `members` collection did. The two additional pieces of knowledge this gives us are as follows:



```
[-1]  FORALL (e2:
               v_InstanceOf(omni_lang_Collection_ElementClass(Lib_members
                                                              (old_this!1))))
        ((e2 /=
           Member_newMember(Lib_getNewMemberNo(old_this!1), name!1, ad1!1,
                            ad2!1, pCode!1, telNo!1, isAdult!1, isStaff!1))
          IMPLIES
          (omni_lang_Iterable_contains(Lib_members(old_this!1), e2) =
            omni_lang_Iterable_contains(omni_lang_Collection_add
                                        (Lib_members(old_this!1),
                                         Member_newMember
                                         (Lib_getNewMemberNo(old_this!1),
                                          name!1,
                                          ad1!1,
                                          ad2!1,
                                          pCode!1,
                                          telNo!1,
                                          isAdult!1,
                                          isStaff!1)),
                                        e2)))
[-2]  omni_lang_Iterable_contains(omni_lang_Collection_add(Lib_members
                                                          (old_this!1),
                                                          Member_newMember
                                                          (Lib_getNewMemberNo
                                                           (old_this!1),
                                                           name!1,
                                                           ad1!1,
                                                           ad2!1,
                                                           pCode!1,
                                                           telNo!1,
                                                           isAdult!1,
                                                           isStaff!1)),
                                Member_newMember(Lib_getNewMemberNo
                                                 (old_this!1),
                                                 name!1,
                                                 ad1!1,
                                                 ad2!1,
                                                 pCode!1,
                                                 telNo!1,
                                                 isAdult!1,
                                                 isStaff!1))
```

`[-2]` tells us that the new `members` collection contains the newly constructed `Member`. `[-1]` tells us that for all members other than the newly constructed `Member`, the new `members` collection contains the `Member` if the old collection contained the `Member`.

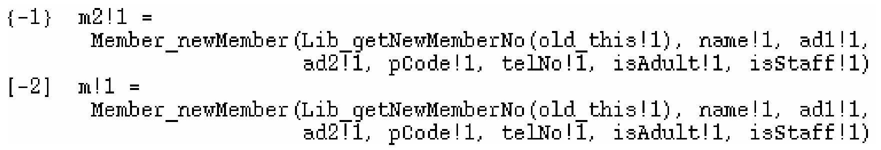
We have two `Member`s, `m!1` and `m2!1`, which we know are contained in the new `members` collection, and we have two rules, `[-1]` and `[-2]` for evaluating these. Each of `m!1` and `m2!1` can be rewritten using either of these rules. Thus we have three different cases:

1. Both are equal to the new Member

2. One of `m!1` and `m2!1` is equal to the new Member
3. Neither is equal to the new Member

Both are equal to the new Member

The easiest case is where both `m!1` and `m2!1` are equal to the new `Member`. In this case, we have the additional knowledge that:



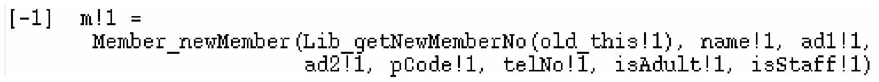
```
{-1}  m2!1 =
       Member_newMember(Lib_getNewMemberNo(old_this!1), name!1, ad1!1,
                        ad2!1, pCode!1, telNo!1, isAdult!1, isStaff!1)
[-2]  m!1 =
       Member_newMember(Lib_getNewMemberNo(old_this!1), name!1, ad1!1,
                        ad2!1, pCode!1, telNo!1, isAdult!1, isStaff!1)
```

Recall that we are trying to prove that either `m!1` and `m2!1` are equal or that they have different `memberNo`s. Well, this additional knowledge is sufficient to trivially prove that `m!1` and `m2!1` are equal.

One of `m!1` and `m2!1` is equal to the new Member

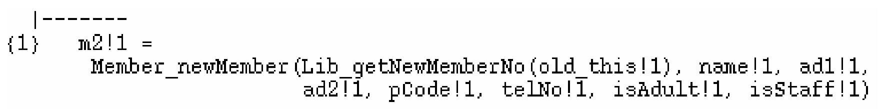
There are two symmetric subcases for this case. Either (a) `m!1` is equal to the new `Member` and `m2!1` is not or (b) `m2!1` is equal to the new `Member` and `m!1` is not. Here we will only discuss the proof of case (a) but the same general technique can be used to prove case (b).

The case analysis gives us two pieces of knowledge. The sequent that `m!1` is equal to the new `Member`:



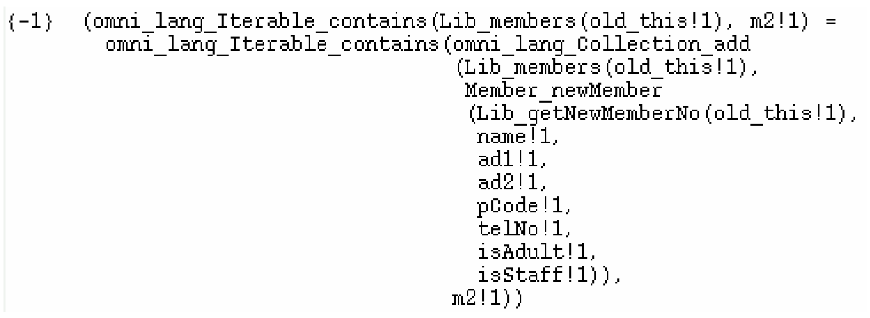
```
[-1]  m!1 =
       Member_newMember(Lib_getNewMemberNo(old_this!1), name!1, ad1!1,
                        ad2!1, pCode!1, telNo!1, isAdult!1, isStaff!1)
```

And the antecedent that `m2!1` is not.



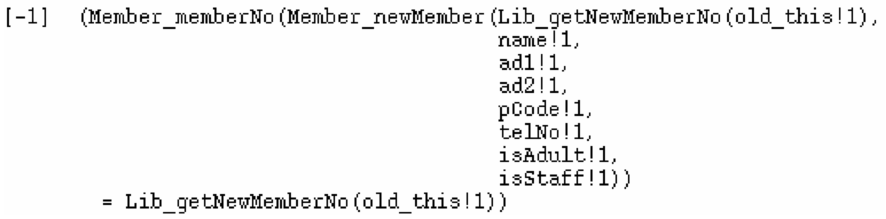
```
  |-------
{1}   m2!1 =
       Member_newMember(Lib_getNewMemberNo(old_this!1), name!1, ad1!1,
                        ad2!1, pCode!1, telNo!1, isAdult!1, isStaff!1)
```

We can start by instantiating the quantifier describing that `Member`s other than the new `Member` are contained in `members`, with `m2!1`. This gives us the knowledge that `m2!1` is in the new `members` if it was in the old `members`.



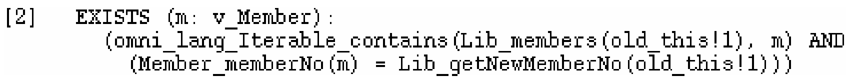
```
{-1}  (omni_lang_Iterable_contains(Lib_members(old_this!1), m2!1) =
        omni_lang_Iterable_contains(omni_lang_Collection_add
                                      (Lib_members(old_this!1),
                                       Member_newMember
                                        (Lib_getNewMemberNo(old_this!1),
                                         name!1,
                                         ad1!1,
                                         ad2!1,
                                         pCode!1,
                                         telNo!1,
                                         isAdult!1,
                                         isStaff!1)),
                                     m2!1))
```

Then we can use `Member_newMember_pub_ax` to deduce that the `memberNo` of the new `Member` is `getNewMemberNo(old_this!1)`. The sequent this gives is:



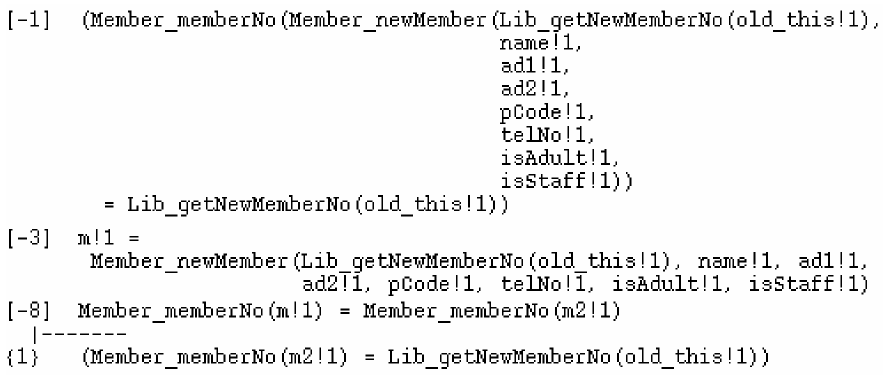
```
[-1]  (Member_memberNo(Member_newMember(Lib_getNewMemberNo(old_this!1),
                                         name!1,
                                         ad1!1,
                                         ad2!1,
                                         pCode!1,
                                         telNo!1,
                                         isAdult!1,
                                         isStaff!1))
        = Lib_getNewMemberNo(old_this!1))
```

The specification of the `getNewMemberNo` function earlier gave us the following proof goal:



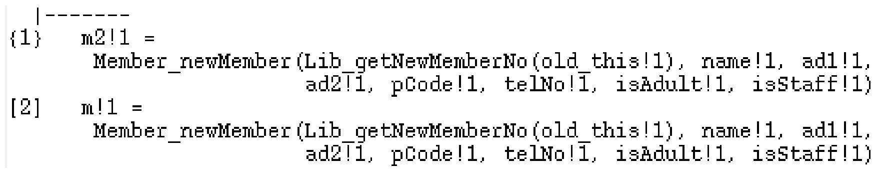
```
[2]   EXISTS (m: v_Member):
        (omni_lang_Iterable_contains(Lib_members(old_this!1), m) AND
          (Member_memberNo(m) = Lib_getNewMemberNo(old_this!1)))
```

We can now instantiate this with `m2!1`. To prove the resulting assertion we must prove both parts of the conjunction. We can prove that the old `members` collection contained `m2!1`, which we can do from the facts that we know the new collection contains `m2!1` and that we know the old collection contains `m2!1` iff the new one does. We can also prove that `MemberNo(m2!1)` is equal to `Lib_getNewMemberNo(old_this!1)` from the fact that we know `Member_memberNo(m!1) = Member_memberNo(m2!1)`, `m!1` equals the new `Member`, and the `memberNo` of the new `Member` is `Lib_getNewMemberNo(old_this!1)`. The relevant parts of the conjecture are shown below:

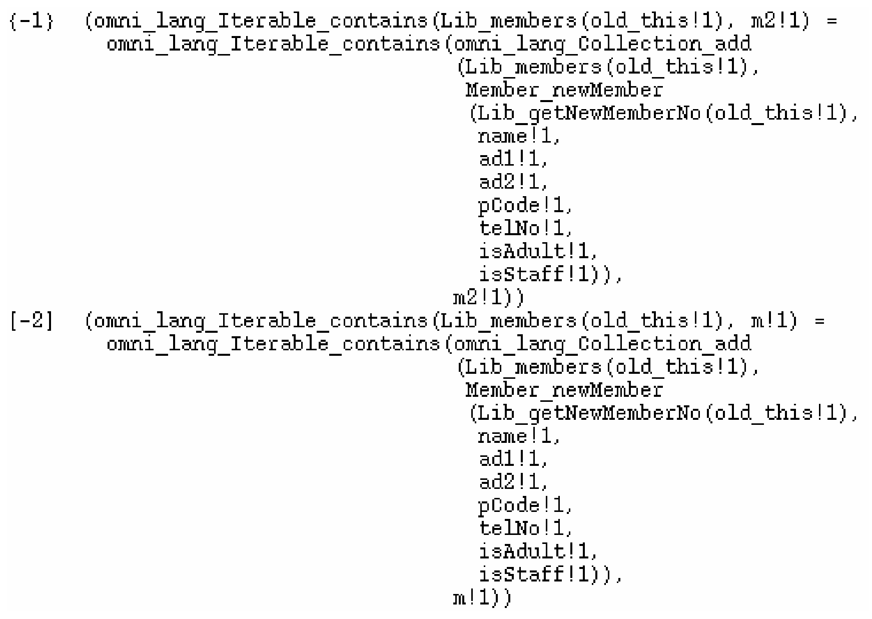

```
[-1]   (Member_memberNo(Member_newMember(Lib_getNewMemberNo(old_this!1),
                                        name!1,
                                        ad1!1,
                                        ad2!1,
                                        pCode!1,
                                        telNo!1,
                                        isAdult!1,
                                        isStaff!1))
         = Lib_getNewMemberNo(old_this!1))
[-3]   m!1 =
        Member_newMember(Lib_getNewMemberNo(old_this!1), name!1, ad1!1,
                         ad2!1, pCode!1, telNo!1, isAdult!1, isStaff!1)
[-8]   Member_memberNo(m!1) = Member_memberNo(m2!1)
  |-------
{1}    (Member_memberNo(m2!1) = Lib_getNewMemberNo(old_this!1))
```

Neither is equal to the new Member

The final case is that neither `m!1` or `m2!1` are equal to the new `Member`. This knowledge is represented by the following antecedents:

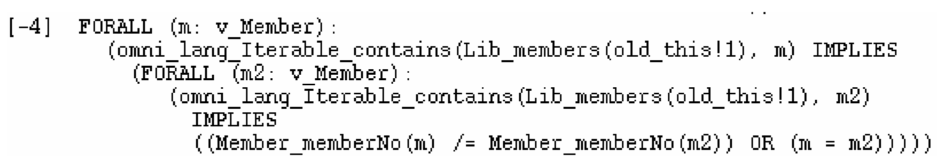

```
  |-------
{1}    m2!1 =
        Member_newMember(Lib_getNewMemberNo(old_this!1), name!1, ad1!1,
                         ad2!1, pCode!1, telNo!1, isAdult!1, isStaff!1)
[2]    m!1 =
        Member_newMember(Lib_getNewMemberNo(old_this!1), name!1, ad1!1,
                         ad2!1, pCode!1, telNo!1, isAdult!1, isStaff!1)
```

We can instantiate the quantifier which describes that `Member`s other than the new `Member` are contained in `members` with both `m!1` and `m2!1`, in turn. This gives us the knowledge that both `m!1` and `m2!1` are in the new `members` if they were in the old `members`. The corresponding sequents are:



```
{-1}  (omni_lang_Iterable_contains(Lib_members(old_this!1), m2!1) =
        omni_lang_Iterable_contains(omni_lang_Collection_add
                                     (Lib_members(old_this!1),
                                      Member_newMember
                                       (Lib_getNewMemberNo(old_this!1),
                                        name!1,
                                        ad1!1,
                                        ad2!1,
                                        pCode!1,
                                        telNo!1,
                                        isAdult!1,
                                        isStaff!1)),
                                     m2!1))
[-2]  (omni_lang_Iterable_contains(Lib_members(old_this!1), m!1) =
        omni_lang_Iterable_contains(omni_lang_Collection_add
                                     (Lib_members(old_this!1),
                                      Member_newMember
                                       (Lib_getNewMemberNo(old_this!1),
                                        name!1,
                                        ad1!1,
                                        ad2!1,
                                        pCode!1,
                                        telNo!1,
                                        isAdult!1,
                                        isStaff!1)),
                                     m!1))
```

We already have the knowledge that `m!1` and `m2!1` are in the new `members` collection, and from `[-1]` and `[-2]` we now know that they were in the old `members` collection. Thus, we can use the truth of the invariant over `old_this`, already loaded in `[-4]`, to deduce that `m!1` and `m2!1` are either equal or have different `memberNo`s.



```
[-4]  FORALL (m: v_Member):
        (omni_lang_Iterable_contains(Lib_members(old_this!1), m) IMPLIES
          (FORALL (m2: v_Member):
             (omni_lang_Iterable_contains(Lib_members(old_this!1), m2)
               IMPLIES
                ((Member_memberNo(m) /= Member_memberNo(m2)) OR (m = m2)))))
```

This completes the proof of the VC.

The entire proof consists of 104 steps and also involves a number of side-conditions.