Data Refinement in Object-Oriented Verification

by

Rosemary Monahan, BSc. MSc.

A dissertation presented to Dublin City University

in fulfilment of the requirements for the Degree of

Doctor of Philosophy

School of Computing

Dublin City University

Supervisor: Professor Joseph Morris

September 2010

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Degree of Doctor of Philosophy is entirely my own work, that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Rosemary Monahan ID No: 99145774

September 22, 2010

Permission to Lend and/or Copy

I, the undersigned, agree that Dublin City University Library may lend or copy this thesis upon request.

Rosemary Monahan ID No: 99145774

September 22, 2010

Acknowledgments

I would like to express my gratitude to my supervisor Professor Joseph Morris for his guidance and encouragement over the duration of this research. Thanks also to my examiners, Dr Joseph Kiniry and Dr David Sinclair, whose constructive feedback shaped the final version of this dissertation.

I would also like to express my gratitude to my colleagues at NUIM for their continued encouragement and support. A particular mention must go to the members of the Principles of Programming research group for keeping me enthusiastic and motivated. James and Paul - you made it sound so easy!

Special thanks are due to Dr. Rustan Leino, Professor Dominique Mery and Dr. Greg Nelson for hosting internships at Microsoft Research, LORIA and Hewlett Packard respectively. You introduced me to research away from the solitude of my desk and encouraged me to interact and collaborate where possible. Extended gratitude to Rustan for helping me dive into the world of Spec# with the best assistance I could have imagined.

Thanks to my parents, Patsy and Marie, and my sisters Margaret, Patricia, Teresa, Suzann and Amanda for being there when I needed proof readers, reason and perspective. Thanks also to the Casey clan for distracting me with lots of family gatherings and great nights out when I needed them most. A big thank you to all my friends for knowing when to ask and when not to ask how work was going but, most of all, for always being there. Special mentions go to Sinead for all

the chats, Karen for the laughs and coordination on Tuesday nights, Johanne for understanding when I arrived in London with work tucked under my arm, Fergal for his entertaining stories, and the lunchtime runners for keeping my head clear and my legs incapable of moving me from my desk. I would also like to express my gratitude to both Patrick and Suarla for motivating and encouraging me each time we met.

Finally, my deepest thanks goes to Kevin for his patience, his support, and his ability to make me smile no matter how pressured I feel. Without his support, this work would never have reached completion. All of your support and encouragement is very much appreciated.

Rosemary Monahan

Dublin City University
September 2010

Contents

Data Refinement in Object-Oriented Verification

Rosemary Monahan, Ph.D.
Dublin City University, 2010

Supervisor: Professor Joseph Morris

Data refinement is a special instance of refinement where a specification is refined by replacing the data type used in the specification. The theory of data refinement guarantees that this replacement does not adversely affect the functional behaviour of the programs that use these specifications.

Object-oriented programming languages such as JML and Spec# support the specification and verification of object-oriented programs. We research their capabilities, identifying their strengths and weaknesses from both a specification and a tool-support point of view. This leads us to the conclusion that object-oriented specification languages should support a view of objects that abstracts away from the implementation details. We examine the specification and verification of programs that are written in this way, making use of existing language features, so that data refinements can be verified using existing verification tools.

We propose a framework for the specification and verification of modular data

refinement within an object-oriented environment. Objects are specified in terms of one data type and implemented in terms of another. Clients who interact with these objects are never concerned with the underlying implementation details as they interact directly with the abstract specification. A proof-of-concept tool is developed to demonstrate the viability and effectiveness of our proposed framework. This tool takes the form of an application that checks whether or not a program conforms to our framework for the modular data refinement of object-oriented programs.

Chapter 1

Introduction

Formal methods of software development provide maximal levels of software reliability. However, methods such as program derivation and program verification [39, 31, 32, 73], have received much criticism due to their increased emphasis on proof, either during program construction or program verification.

More recently, a more pragmatic approach seeks to integrate the basic mechanisms of program verification, such as pre-conditions, postconditions and loop invariants, with routine software testing and model checking. Characterised as *lightweight formal methods* [43], they are most commonly found in the *design-by-contract* approach of Eiffel [68], and more recently in extensions to programming languages such as Java and C#. In this approach, pre-conditions and postconditions clearly assert the obligations of the software client and the software supplier, ensuring that there is no doubt about the environment in which the software should be executed in order to guarantee the expected results. These assertions provide the software specification that the implementation must satisfy if it is to be verified as correct.

Formal reasoning about programs provides extensive and detailed tool support that can be used to validate specifications against system requirements, check the consistency of specifications, and assist in the refinement of specifications into executable programs. The main advantage of a tool that performs these tasks is that it leads to a well-structured and safe approach to program construction.

During the last decade, experience in the specification and the verification of moderately scaled software systems has been achieved: interactive program verifiers,

such as B [1] and Spark Ada[10], have been applied to control software and other critical applications, software model checking with tools such as the Static Driver Verifier [8], has had an impact on industrial applications, and a number of research tools that detect errors in software have been built using semi-automatic program-verification technology (e.g. Spec# [13, 12, 14], JML [50], Spark [10], ESC/Java2 [46], Perfect Developer [24] Why [33], Krakatoa [67], KeY [15]).

1.1 Verifying Compilers

The Grand Challenges Exercise [45], an enterprise of the UK Computing Research Committee, revived the challenge of the construction and application of a verifying compiler: "the time is ripe to embark on an international Grand Challenge project to construct a program verifier that would use logical proof to give an automatic check of the correctness of programs submitted to it" [42]. Such a compiler could come in many varieties, from systems that generate provably correct code from specifications, to systems that ask users to guide an interactive theorem prover to produce a replayable proof script.

Two successful verifying compilers for procedural languages are those for the SPARK programming language and for the B specification language. SPARK is a high-level programming language based on a subset of the Ada language, and is designed in such a way that all SPARK programs are legal Ada programs. Although SPARK does not support many dynamic language features like references, memory allocation, and subclassing, it is useful for many embedded industrial applications as demonstrated by Praxis Critical Systems [3]. The SPARK tool-set offers a selection of static tools, from lightweight checking to full verification with an interactive theorem prover.

The B-Tool supports writing specifications in B and provides a machine-aided process for step-wise refining [78] these specifications into programs that can be compiled and executed. The resulting programs are similar in expressiveness to SPARK programs. This methodology, which has been used with success in constructing the braking system software for the Paris Metro, produces only correct programs. However, the interactive proofs and the skills needed during the refinement process make for a steep learning curve, which is a barrier for many programmers.

1.2 Contributions of this Research

This thesis contributes to work on the automatic verification of object-oriented programs. Two of the most successful projects in this area are the Extended Static Checker for Java (ESC/Java2) [46], and the Spec# programming system[13, 12, 14]. Both systems use the design-by-contract approach to annotate programs with assertions so that tools, such as compilers and theorem provers, can use these assertions to generate the proof obligations required to verify that a program satisfies its specification. Verification tools are then used to automatically discharge the proof obligations that have been generated. ESC/Java2 supports Java programs with assertions written in the Java Modeling Language (JML) [50] while the Spec# programming system supports C# programs with assertions written in the Spec# specification language.

The central emphasis in these projects is *Extended Static Checking*, i.e., support for traditional static verification techniques extended with support for dynamic analysis of programs. By static verification we mean checking the consistency of a program with its specification without executing the program. A typical example is type checking. Extended static checkers allow the compiler to emit run-time checks at compile time, recording assertions in the specification as meta-data for consumption by downstream tools such as SMT solvers like Simplify [28] or Z3 [27]. The result is the static detection of errors (null dereferences, casting errors, array bound errors, etc) that would traditionally be detected at run-time.

In this research we contribute to the verification of data refinement in existing verification tools for object-oriented programs. By verification, we mean proving that an implementation satisfies its specification in every possible execution. We are primarily concerned with accomplishing this proof via static reasoning, although the tools that we use also support dynamic checking. Our work provides an in-depth analysis of the current support for data refinement in the specification languages, Spec# and JML. We propose a framework for expressing and verifying modular data refinement, which could be used to extend existing languages such as these, so that their support for data refinement is improved. The feasibility of these proposed extensions is shown through the development of a prototypical tool. This tool accepts JML programs as input and checks whether they conform to the rules

governing a data refinement, as defined by our framework.

Some publications from our related research are listed below:

- K. Rustan M. Leino and Rosemary Monahan, "Using Boogie 2 in the Verification of Spec# Programs". In 13th Brazilian Symposium on Formal Methods (SBMF 2010)
- Rosemary Monahan, "Verification of C# programs using Spec# and Boogie 2". In 8th International Conference on integrated Formal Methods Workshops (iFM Workshops 2010)
- K. Rustan M. Leino and Rosemary Monahan, " Dafny meets the Verification Benchmarks Challenge". In 3rd International Conference on Verified Software: Theories, Tools, Experiments (VSTTE 2010)
- Rosenary Monahan and Yan Xu, "Implementing the Verified Software Initiative Benchmarks using Perfect Developer". In China-Ireland International Conference on Information and Communications Technologies (CIICT 2010)
- K. Rustan M. Leino and Rosemary Monahan, "Reasoning about Comprehensions with First-Order SMT Solvers". In 24th Annual ACM Symposium on Applied Computing (SAC 2009)
- K. Rustan M. Leino and Rosemary Monahan, " Program Verification Using the Spec# programming system". In European Conference on Object-Oriented Programming Summerschool Series (ECOOP Summer-school 2009)
- K. Rustan M. Leino and Rosemary Monahan, " Program Verification Using the Spec# programming system". In European Joint Conferences on Theory and Practice of Software, Summer-school Series (ETAPS-Summerschool 2008)
- K. Rustan M. Leino and Rosemary Monahan, " Automatic verification of textbook programs that use comprehensions". In Formal Techniques for Java-like Programs (FTfJP 2007)
- Gareth Carter, Rosemary Monahan and Joseph Morris, "Software Refinement with Perfect Developer". In Software Engineering and Formal Methods (SEFM 2005)

1.3 The Problem Description

A software client should only be concerned with the software specification. They do not need to know details about the implementation and the verification process. This separation of concerns is achieved via data abstraction. In all approaches to data abstraction the key idea is to provide an abstract view of a program which we can provide to the client without exposing implementation details. The specification of the program is typically written in terms of some data type whose relationship with the data type actually used in the implementation is only known to the programmer. We refer to the data type used in the specification as the abstract data type and to the data type used in the implementation as the concrete data type.

By data refinement we mean the process of transforming a specification written in terms of the abstract data type into a specification that is written in terms of the concrete data type. We refer to this latter specification as the implementation. The theory of data refinement guarantees that this replacement of types does not adversely affect the behaviour of the programs that use these specifications. We say that the implementation *refines* the specification. The key idea is to use an abstraction relation to describe the connection between the abstract and the concrete data types.

A standard example of data refinement is to replace a mathematical set with a sequence representation such as an array or a singly linked list. A suitable abstraction relation would map the contents of the sequence to the contents of the set. A consequence is that all operations defined on the abstract data type must be redefined in terms of the concrete data type. For example, if we specify a library as a set of books, the operation that adds a book might use the union operator, whereas if we specify a library as a sequence of books this would require use of the sequence concatenation operators.

The refinement calculus provides a set of laws that, using the abstraction relation, transforms a specification written in terms of the abstract data type into a specification that is written in terms of the concrete data type. The abstraction relation must take into account how the abstract variables have been used in the specification. The major advantage of data refinement is that specifications can be written and reasoned about in a way that is independent of their implementations.

In this way we can prove properties of the specification and then, through the laws of data refinement [72], we can prove that the implementation is consistent with the specification. Proving the correctness of a data refinement relies on a mapping that relates the abstract data and the operations defined on it, to the concrete data and its operations. In this work, we determine how existing specification languages and their corresponding verification tools should support data refinement in object-oriented programs.

1.3.0 The Significance of this Problem

A specification language typically contains sophisticated data types that are expensive or even impossible to implement. Their replacement with simpler or more efficiently implementable types during the programming process is necessary in order to achieve an implementation. This replacement and its verification is not fully supported in object-oriented programming languages at present. Supporting data refinement in object-oriented programming languages and their verification tools will allow greater flexibility when writing specifications. It will also support reasoning about specifications in a way that is independent of their implementations. Reasoning that applies to the specification is also applicable to the implementation, as a correct implementation and its specification will satisfy the same properties even if these properties are expressed in terms of different data types.

1.3.1 Current Approaches to this Problem

Current approaches to support for data abstraction in specification languages include ghost variables, model fields, logic functions and pure methods. In all of these approaches the idea is to present a higher-level, more abstract view of what a program implements. While these approaches support writing the specification in terms of an abstract view that is somehow mapped to its implementation, we believe that these approaches have shortcomings in supporting full data refinement verification in an object-oriented environment. In particular, the abstract specification and its concrete implementations are seldom modular. Hence the client's view of the specification is not guaranteed to be free of implementation details. Even when the client's view of the specification is free from these details, the substitution of one

implementation for a specification, and the substitution of one implementation for another, is not an easy task. In an object-oriented programming language, topics such as object mutability, aliasing, subtyping and modularity all play a considerable role in this substitutability.

1.4 The Goals of this Research

The first goal of this work is to provide and evaluate, a framework for modular data refinement in behavioural interface specification languages for object-oriented programs. The second goal is to provide improved support for data refinement in existing verification tools for object-oriented programs, making use of existing language structure and verification techniques where possible. To achieve these goals we first analyse existing specification languages and their support for data abstraction.

1.5 Thesis Overview

In this chapter we have introduced the motivation and the context of our work. We have described the problem that we wish to solve and the goals of our research. The remainder of this dissertation is laid out as follows.

In chapter 2, we present an overview of object-oriented programs and their specifications. We introduce the idea of software contracts, which provide the obligations for both the software and the software supplier. A notation in which to express these contracts is introduced. This notation is similar to that of popular object-oriented specification languages, JML and Spec#. Verification tools for object-oriented programs and the key factors that influence the modular verification of these programs are also presented.

We review the main components of the refinement calculus in chapter 3. Refinement laws for procedural refinement and data refinement are presented as well as strategies by which data refinements can be verified. In chapter 4, we discuss the verification of data abstractions in an object-oriented environment, focusing on support in JML and Spec#. As a result, we provide a suite of proposals to assist

the verification of modular data refinement in an object-oriented programming environment. These proposals are summarised in section 4.3. We provide an overview of Spec# notation in chapter 5 and proposes a strategy for the modular verification of data refinements in it's programming system, using existing language features to achieve a two-class approach to data refinement.

In chapter 6 we present an object-oriented programming framework to support a modular approach to data refinement. This framework completely decouples an object's specification from its implementation maintaining a client and a supplier view of the software. The language is designed to meet the proposals discussed in chapter 4. A proof-of-concept tool was developed to demonstrate the viability and effectiveness of our framework. This tool, described in chapter 7, takes the form of an application that checks whether or not a program conforms to our framework for the modular data refinement of object-oriented programs. Finally, we summarise our research in chapter 8 and suggest interesting topics for future work in this area.

Chapter 2

Object-Oriented Verification

In this chapter, we present an overview of object-oriented programs and their specifications. We introduce the idea of software contracts, which provide the obligations for both the software and the software supplier. A notation in which to express these contracts is introduced. This notation is similar to that of popular object-oriented specification languages such as the Java Modeling Language (JML) and Spec#. Verification tools for object-oriented programs and the key factors that influence the modular verification of these programs are also presented.

2.1 Object-Oriented Programs

An object-oriented program is typically defined as a set of classes where each class provides the structure and implementation for a family of objects. We work with typed object-oriented programming languages which support the definition of classes, objects, object references, interfaces and abstract classes with single (rather than multiple) inheritance, and a type system that supports subtyping and polymorphic types. The notation that we use throughout our work is similar to Java and C#.

Classes are similar to modules in Modula-3, packages in Ada and aspects in AspectJ, in that they encapsulate variables that define state and functions that define the allowed behaviour on that state. As is normal in object-oriented programming, we refer to these variables and functions respectively as fields and methods and

collectively as features.

Classes support information hiding through providing a private component of the class which keeps the fields and the method implementations hidden from the client (the programmer who uses the class) yet available for the supplier (the programmer who writes the class). A public interface for the class, in the form of method signatures, is visible to all clients. Clients may use the class by creating instances of the class, called *objects*. An object's state is given by the values of its fields and its behaviour is given by the methods that can be called (or executed) on the object. When a method m is called on an object *obj* (written as *obj.m()*) we refer to *obj* as the receiver object.

2.2 Behavioural Interface Specifications

The challenge in writing a specification for a class is to provide information about its allowed behaviour, without exposing implementation details to the client. The design-by-contract approach[68], pioneered by Bertrand Meyer in the Eiffel Programming language, embraces this challenge. A class contract outlining the client and supplier obligations provides the class specification without exposing implementation details. This class specification is provided in the form of assertions. These are boolean expressions that are written in predicate logic and side-effect free. Typically the language of assertions is a super-set of the programming language, supporting expressions that include quantifiers and data types that are not available for the implementation. As the implementation details are not exposed by the class contract, the programmer has the flexibility to change the implementation as long as it satisfies the specification. Wing [86, 87, 37] and Lamport [47] called these contracts *interface specifications*. They are also referred to as *behavioural interface specifications* [21, 38] to reflect the recording of the behavioural aspect of these specifications.

In their most basic form, behavioural interface specifications describe the functional behaviour of a class by describing the relationship between the inputs and outputs of its methods. For example a class containing a method that calculates the minimum of two positive integers might specify that the method has two positive values as input and that the smaller of those two inputs is returned as the result.

A class contract typically consists of a object invariant and its method contracts. A method contract consists of its signature, pre-conditions, postconditions and frame conditions. It does not include the implementation. Pre-conditions, postconditions and object invariants are expressed as assertions whereas frame conditions simply list the parts of the object's state that a method is allowed modify. The obligation of the client is that when they execute a method, they do so in a state that satisfies the object invariant and the pre-conditions of that method. In return, the supplier guarantees that the method's postconditions and the object invariant will be satisfied after the method has been executed.

In the sections that follow we discuss these components of class contracts in detail. We support our discussion with examples and in this way we introduce the notation that we use throughout this document.

2.2.0 Behavioural Interface Specification Languages

Specification languages which are used to express specifications can be general (as in the Z language) but are often specialised to precise methodologies and verification tools. We are primarily interested in specification languages that are specialised for object-oriented languages. Examples include Eiffel, JML and Spec#. These languages provide extensions to their underlying programming languages which allow assertions, such as method pre-conditions and postconditions, to be expressed in a syntax that is a super-set of the programming language.

Assertions may be processed by verification tools that perform static and dynamic analysis of programs. We will discuss existing tool support for these in section 2.2.7. First, we discuss the components of the class contracts and provide notation in which to express them. This notation is similar to that used in JML and Spec#.

2.2.1 Pre-conditions and Postconditions

Hoare logic [4, 39] and Dijkstra's weakest pre-conditions calculus [76, 31] lay the foundation for the design-by-contract approach. Hoare logic provides a set of logical rules in order to reason about the correctness of software using the rigour of mathematical logic. Proof that an implementation satisfies its contract is presented in the form of a triple $\{Q\}P\{R\}$. The meaning of this triple is: if program P is

executed in a state satisfying the pre-condition Q, and if P terminates, then the state achieved will satisfy the postcondition R. For example, the triple

$$\{x > 4\}\ x := x + 1\ \{x > 5\}$$

where $:=$ represents assignment, means that the program $x := x + 1$ executed in a state where x has a value that is greater than 4 will achieve a state where x has a value that is greater than 5.

Hoare's programming rules prove properties of programming constructs such as skip (do nothing), assignment $(:=)$, sequential composition $(;)$, if statements and loops. His logical rules allow the manipulation of pre-conditions and postconditions through strengthening or weakening them and through distributing logical operators over them.

An alternative formulation of these rules, that is used in many program verification tools, is Dijkstra's weakest pre-conditions calculus. In this calculus the operator $wp(P, R)$ is used to construct the weakest pre-condition that can establish the postconditions R for the program P. An assertion Q is weaker that assertion `S` if $S \Rightarrow Q$. Intuitively this means that Q is less restrictive than `S` and hence if $\{S\}P\{R\}$ is a valid triple in Hoare logic then $\{Q\}P\{R\}$ is also valid.

The notation that we use to represent pre-conditions and postconditions is close to that of Spec# and JML. To see how this compares to the traditional Hoare style format, we write a class in Fig. 2.1 that encompasses our previous example as a method, i.e., $\{x > 4\}x := x + 1\{x > 5\}$. Note that the method pre-conditions is annotated with the keyword **requires** and its postconditions are annotated with the keyword **ensures**. We refer to these as **requires** and **ensures** clauses respectively. A method contract may contain any number of **requires** clauses, which provide the method's pre-conditions when they are conjoined. Likewise a method's postconditions is the conjunction of its **ensures** clauses. Note also, the symbol that we use for assignment is $=$ and the symbol that we use for equality is $==$. This is in keeping with the notation for assignment and equality in popular specification languages such as Spec# and JML.

Pre-conditions express the constraints under which the method will execute cor-

```
public class Inc{
        private int x;

        public void Increment()
        requires x > 4;
        ensures x > 5;
        {
                x = x+1;
        }
}
```

Figure 2.1: The specification and implementation of a method that increments an integer variable.

rectly and postconditions express what will happen as a result of the methods proper execution. Since the client must establish the pre-conditions before calling a method, it is reasonable to demand that every feature appearing in the pre-conditions of a method must be available to every client to which the method is available. Meyer[68] refers to this as the *Pre-condition Availability Rule*. So, for example, a method's pre-conditions should not refer to the private fields of the class. Postconditions may refer to features that are not directly available to the client, as they should record all effects of the method. These effects could impact on the client even if they have no access to the features that are modified. If a pre-conditions is violated the error is known to be in the client's code whereas if a postconditions is violated the error is known to be in the supplier's code.

2.2.2 Quantification

Specifications may contain quantifiers such as the universal and existential quantifiers. These quantifiers are specification constructs that are not directly implementable in our language. In our specification notation, we represent universal quantification and existential quantification using the keywords **forall** and **exists** respectively. For example an assertion that expresses that all values in an array called A are greater than zero could be written as

$$\textbf{forall}\{\textbf{int } k \ \textbf{ in } (0 : A.Length); A[k] > 0\};$$

whereas the assertion that there is at least one positive value in the array A could be written as

$$\textbf{exists}\{\textbf{int } k \textbf{ in } (0 : A.Length); A[k] > 0\};$$

An example of a specification containing universal quantification is presented in Fig. 2.2.

```
public class Sq{
        private int []! a;

        public int  Square()
        modifies  a[*];
        ensures forall {int i in (0:a.Length); a[i]==i*i};
        {
                int  x = 0;
                int y = 1;
                for (int n = 0; n <a.Length; n++){
                        a[n] = x; x +=y; y + =2;
                }
        }
}
```

Figure 2.2: An example of a postcondition that contains the universal quantifier.

2.2.3 Relating Values Before and After Method Executions

Postconditions may relate the value of a variable after a method's execution with its value prior to that method's execution, i.e., in its pre-state. We use the notation *old* x to denote the value of the variable x in its pre-state. The method's post-condition, **ensures** $x >$ *old* x, expresses that the value of the variable x has been incremented. This notation is used in the same way in Eiffel, JML, and Spec#. Some other popular notations, with which the reader might be familiar, are x' to denote the value of x in the post-state and x_0 to denote the value of x in the pre-state. Logical variables have also been used in Hoare logic and specification languages like Z to relate the values of variables in the pre-state and post-state. These variables can be used in specifications but cannot be assigned values in the implementation.

2.2.4 Non-Null Types

Errors in modern programs often manifest themselves as null-dereference errors. Many specification languages try to eradicate null dereferences through supporting non-null types. For example, in C#, each reference type T includes references to objects of type T and references to the null value. In Spec#, type T! contains only references to objects of type T. We adopt this notation in our specification language. The supplier delegates to the client, the responsibility of ensuring that non-null arguments are used in a method call. This decision is recorded using an exclamation point. For example, in Spec# the declaration **int** []! xs declares an integer array called xs which cannot be initialized to null. Verification tools enforce this decision at call sites returning a message such as " Error: null is not a valid argument" if a null value is passed to a method that requires a non-null argument.

2.2.5 Frame Conditions

Frame conditions specify the parts of the object's state that a method is allowed to modify. In Fig. 2.3 the annotation **modifies** x specifies the frame condition and hence limits the parts of the program state that the method is allowed to modify to the variable x. In our notation, the **modifies** keyword is followed by a list of variables that the method is permitted to change. We refer to this list as the method $frame$. In this case the example in Fig. 2.2 illustrates how a **modifies** clause can refer to an array of values. The notation $a[*]$ refers to all values in the array a. Similarly, the notation $b.*$, used in a **modifies** clause refers to all fields of the object b while **modifies** $b.x$, refers to the allowed modification of field x of the object b.

2.2.6 Return Values

The postconditions of the $Increment()$ method of Fig. 2.3 specifies that the method returns the value x. This is specified by writing **result** $== x$, where **result** denotes the value returned by the method. Recall that the symbol that we use for equality is == (as used in Java and C#).

```
public class Inc{
        private int x;

        public int Increment()
        requires x > 4;
        modifies x;
        ensures x > old x && result == x;
        {
                x = x+1;
                return x;
        }
}
```

Figure 2.3: A method that increments an integer variable and returns its value.

2.2.7 Object Invariants

An object invariant is an assertion that specifies properties of the object's state. An object is valid if it satisfies the object invariant at all *stable* times. By this we mean that the object invariant must be established when the object is constructed and must be re-established before a method execution terminates. These are the responsibilities of the supplier. In return the client can assume that the object invariant holds when a method is executed. A method implementation may violate an object invariant but must re-establish it before its execution terminates, ensuring that the object is in a state where other methods may be called without error.

Assertions within a class definition, labelled with the keyword **invariant** are called object invariants. The conjunction of these assertions provides the overall object invariant. See Fig. 2.4 for an example, where the constructor establishes the object invariant by assigning the values 0 and *true* to the variables c and *even* respectively. The method $Inc()$ temporarily violates the object invariant within the method body, but re-establishes it before the method execution terminates. Another example is a class that describes a linked list, where a suitable object invariant is that each node in the list must be linked to another. A method that inserts a node into the centre of the list must break the invariant so that the node is inserted. When the insert method has completed, the object invariant must be re-established.

Object invariants strengthen method contracts by adding properties that the object should satisfy and that the client should maintain. All object invariants are public in our notation.

```
public class Counter{
        private int c;
        private bool even;

        invariant 0 ≤ c;
        invariant even ⟺ c mod 2 == 0;

        public Counter()
        ensures c == 0 && even == true;
        {
                c = 0; even = true;
        }
        public void Inc()
        modifies c, even;
        ensures c == old(c)+1 && even ≠ old(even);
        {
                c++; even = !even;
        }
}
```

Figure 2.4: The method *Inc()* violates the object invariant after it increments c and before it changes the value of *even* as $even \iff c \bmod 2 == 0$ is false.

2.3 Object-Oriented Program Verification

Program verification means proving that a program implementation satisfies its specification for every possible execution path. This is usually achieved through static reasoning which establishes whether an implementation satisfies its specification through examining all possible execution paths before the implementation is executed (i.e., at compile time). Static reasoning can also be combined with dynamic analysis techniques such as run-time assertion checking. Run-time assertion checking discovers any inconsistencies that exist between the specification and the implementation by executing assertions to determine their validity at run-time. This execution of assertions does not change the behaviour of the implementation, as specifications are not permitted to have any side-effects [16].

Verification tools for languages like JML and Spec# provide both static and run-time assertion checking of specifications and their implementations. If the proof obligations can be discharged, the program is proved to be a correct implementation of the specification. Failed proof attempts are reported as error messages, to which a user responds by fixing errors or omissions in the program and its specifications.

There are several tools for statically checking JML assertions, providing different levels of automation and supporting different levels of expressiveness in specifica-

tions. One example is ESC/Java2 which checks assertions and automatically detects errors such as dereferencing null, indexing an array outside its bounds, or casting a reference to an incorrect type. JML specifications can be dynamically checked by the JML run-time assertion checker (JML RAC). JML RAC tries to find inconsistencies between the specification and the implementation by executing JML assertions and notifying the user of any assertion violations.

The Spec# static program verifier, Boogie [55], generates logical verification conditions from a Spec# program. It achieves this by first translating compiled Spec# programs into the intermediate verification language BoogiePL. BoogiePL is a simple first-order language that includes mathematical functions, arithmetic, and logical quantifiers. The Spec# compiler performs run-time checks for method contracts and invariants. Internally, the Spec# programming system uses an automatic theorem prover that analyses the verification conditions to prove the correctness of the program or find errors if they exist.

The main advantage of static and dynamic analysis of programs is that potential errors can be identified even before the program is complete or executable. A further advantage is that the program is better designed and documented due to the assertions that describe the properties that are statically checked.

There are two fundamental properties of any proof method: soundness and completeness. A proof system is sound if what it proves is also true. Therefore, if no error is reported we are sure that no error exists. A program verification system that is not sound may fail to produce errors about incorrect programs . Such a system is still useful if the user knows the type errors that the system fails to report. For example, ESC/Java fails to report errors arising from modular arithmetic and/or multi-threading.

A proof system is complete if everything that is true can be proven. Therefore, a verification system that is complete will report a positive result if a program verification is true for all possible inputs. If the verification system is not complete it may produce errors about correct programs, hence complaining that an implementation does not satisfy its specification when it does.

2.3.0 Modular Verification and Behavioural Subtyping

In any object-oriented program we expect class invariants, method pre-conditions, frame conditions and method postconditions to be specified and verified. The aim is to achieve modular verification of classes and methods so that they may be reused in software development. Specification techniques must be capable of handling language features for modular object-oriented programming such as subtyping, dynamic dispatch and inheritance.

Subtype Polymorphism

A *type* defines a set of values and the operations that are allowed on those values. We write $x : T$ to express that a variable x has type T. `S` is a *subtype* of T if a value of type `S` can be substituted anywhere that a value of type `T` is expected without causing type errors. Conversely we say that `T` is a *supertype* of `S`. We write `S<:T` to denote that the type `S` is a *subtype* of T and understand this as "every value described by `S` is also described by `T`" or "the set of values of `S` is a subset of the set of values of `T`". One of the consequences of subtyping is that if `S<:T`, then every element t of type `S` also has type T. We refer to this property as *subsumption* and express it by the following rule:

$$\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T} \quad (\textsc{Subsumption})$$

where $\Gamma \vdash t : S$ expresses the type judgement $t : S$ (the variable t is of type S) in the typing context Γ. A typing context is just a set of type judgements like $\{x_0 : T_0,\ x_1 : T_1, ...\}$ where x_i refer to variables and T_i refer to types and $0 <= i$.

Due to subsumption, `S<:T` then a variable of type `T` may be assigned a value of type `S` without causing any type errors. For example a language might allow floating point values to be used wherever integer values are expected ($Float <: Integer$) so given the function $f : Integer \rightarrow Integer$, the function call $f(v)$ with $v : float$ does not result in any type errors.

In a programming language that supports subtyping, variables and expressions can therefore denote values of several different but related types at run-time. We

refer to this as *subtype polymorphism*. Note that if `S<:T` then a variable of type T can be assigned the value of a variable of type `S`. However a variable of type `S` cannot be assigned a value of type `T` as it is possible that some operations are allowed on `S` which are not permitted on `T`. The subtyping relation is reflexive and transitive as expressed by the rules below:

$$S <: S \qquad \text{(Subtyping Rule 1:Reflexivity)}$$

$$\frac{S <: U \qquad U <: T}{S <: T} \quad \text{(Subtyping Rule 2: Transitivity)}$$

Inheritance and Subtyping

Clients of a class must satisfy the class contract in order to obtain guarantees about an object's behaviour. The client is typically a class which uses another class in its definition (as in Fig. 2.6). Another popular way to reuse a class definition is via extension where a class inherits features from an existing class. An example can be seen in Fig. 2.7, where a `ColouredPoint` inherits from the class `Point` (which is defined in Fig. 2.5).

As is standard in object-oriented programming, when a class inherits from another we refer to the inheriting class as the *subclass* and the inherited class as the *superclass*. The advantages of subclassing are mostly due to reuse. The subclass can reuse features from the superclass, may add new features to extend the class, or may change the behaviour of the inherited methods. When a class inherits from another it inherits the class contract as well as the implementation. We refer to this as *specification inheritance*. The resulting design is more modular and hence easier to reason about. A key mechanism that enables these benefits is method overriding, which specialises the behaviour of inherited methods.

Dynamic Dispatch and Behavioural Subtyping

In object-oriented programming, the *Liskov Substitution Principle* is a definition of subtyping that was introduced by Barbara Liskov in a 1987 conference keynote address [63]. In a follow-up paper [66] Liskov stated the principle as: "Let `q(x)`

```
public class Point{
        private int X;
        private int Y;

        invariant 0 ≤ X && 0 ≤ Y;

        public Point()
        ensures X == 0 && Y == 0;
        {
                X = 0; Y = 0;
        }

        public void SetPoint(int x1, int y1)
        requires 0 ≤ x1 && 0 ≤ y1;
        modifies X, Y;
        ensures X == x1 && Y == y1;
        {
                X = x1;  Y = y1;
        }

        public int getX()
        ensures result == X;
        {
                return X;
        }

        public int getY()
        ensures result == Y;
        {
                return Y;
        }
}
```

Figure 2.5: The specification and implementation of a class that defines a two-dimensional point.

be a property provable about objects x of type `T`. Then `q(y)` should be true for objects y of type `S` where `S` is a subtype of `T`."

The Liskov Substitution Principle means that if `S<:T` then objects of type `T` in a program may be replaced with objects of type `S` without changing any desirable properties of that program.

In most class-based object-oriented languages, subclasses give rise to subtypes, i.e., if `A` is a subclass of `B`, then an instance of `A` may be used in any context where an instance of B is expected. Therefore, we say `A` is a subtype of `B`. The consequence is that any variable declared as having type `B` might, at run-time, hold a value of type `A`. We say that the variable's static type is `B` and its dynamic type is `A`. Examples of where subclasses do not give rise to subtypes include private inheritance in C++ and operations on derived types in Eiffel where features inherited from a subclass

```
public class Line{
        private Point a;
        private Point b;

        public Line()
        ensures a.getX == 0 && a.getY == 0;
        ensures b.getX == 0 && b.getY == 0;
        {
                a.SetPoint(0,0);  b.SetPoint(0,0);
        }

        public void  SetLine(int x1, int y1, int x2, int y2)
        requires 0 ≤ x1 && 0 ≤ x2;
        requires 0 ≤ y1 && 0 ≤ y2;
        modifies a.*, b.*;
        {
                a.SetPoint(x1, y1); b.SetPoint(x2, y2);
        }
}
```

Figure 2.6: The specification and implementation of a line.

```
public class ColouredPoint:Point{
        private int colour;
        public ColouredPoint(){
                colour = 0;
        }

        public void SetColour(int c1)
        ensures colour == c1;
        {
                colour == c1;
        }
}
```

Figure 2.7: The specification and implementation of a class which inherits from the class defined in Fig. 2.5.

can be modified.

The Liskov substitution principle is closely related to the design-by-contract methodology, leading to some restrictions on how contracts can interact with inheritance. For example, method pre-conditions cannot be strengthened in a subclass and method postconditions cannot be weakened in a subclass.

Dynamic dispatch, a characteristic feature of object-oriented languages, allows a method call to have different effects depending on the dynamic type of the receiver. A receiver's dynamic and static types can be identical. However, due to subsumption, the receivers dynamic type can also be any subtype of its static type. Dynamic dispatch allows the class of the receiver to be determined at run-time. The version

of the method corresponding to that class definition is then executed.

Dynamic dispatch generates a problem for static verification as the method to be executed, and hence its specification, is unknown at compile time. One approach to solving this problem is to verify a method call for each possible subtytpe. This approach has efficiency problems as every time a new subtype is added to the language a new suite of verification tasks need to be performed. A more popular approach is to place restrictions on the behaviour of subtypes so that subclass objects are capable of behaving according to the specifications of superclass objects. This methodology is known as behavioural subtyping [66] and allows reasoning about the behaviour of all possible subclasses in terms of the superclass specification. Most object-oriented specification languages enforce behavioural subtyping via their rules for specification inheritance. The necessary restrictions on the contracts of inherited specifications will be discussed below in the context of modular verification.

As explained by the Liskov Substitution Principle, a subtype object may be substituted where a superclass object is expected. Therefore, restrictions are placed on inheriting specifications as follows:

0. The object invariant for objects of the subclass must imply the object invariant for objects of the supertype (inheriting classes may strengthen object invariants).

1. The postcondition of any overriding method must imply the postconditions of the method that it overrides in the superclass (overriding methods may strengthen postconditions).

2. The pre-condition of any method in the superclass must imply the pre-condition of the corresponding overriding method (overriding methods may weaken pre-conditions).

3. For every method that is overridden, its frame must be a subset of the frame of the method that it overrides from the superclass.

Restriction 0 is necessary to support inheritance. All inherited methods assume that the object invariant for the superclass holds in the subclass. Therefore, object

invariants cannot be weakened by the subclass. For example, the object invariant in the class `Point` in Fig. 2.5 is

$$0 <= X \;\&\&\; 0 <= Y$$

The class `ColouredPoint` in Fig. 2.7 inherits this invariant from the class `Point`. A method call such as $Cp.SetPoint(2, 4)$, where Cp is a `ColouredPoint` object, expects the object invariant to hold. In method calls like this weakening the object invariant by replacing it with

$$0 <= X$$

where there is no constraint on Y could cause an error (as a negative value for Y would be acceptable).

The object invariant in the subclass will normally be strengthened as constraints are added on new or inherited fields. For example the `ColouredPoint` object invariant could be strengthened to become

$$0 <= X \;\&\&\; 0 <= Y \;\&\&\; 0 <= c$$

If an inherited method is called on an object of the subclass, errors can occur as the inherited method does not know to establish the potentially stronger invariant of the subclass. A number of solutions have been suggested: object invariants that mention variables of the superclass could be prevented [75], overriding methods, that have the potential to modify the object invariant of the subclass, could be required [82], all inherited methods that have not been overridden could be re-verified [77] or restrictions on when a field can be modified and when an object invariant must hold can be put in place [13, 11, 60].

Restrictions 1, 2 and 3 ensure that all overriding subtype methods satisfy the contract of the method that they override. This means that method contracts can be verified at compile time. As the pre-conditions cannot be strengthened, we are guaranteed that the pre-conditions of an overriding subclass method, which could be called at run-time, is also satisfied. The postconditions of overriding methods establish at least what the overridden methods establish. Hence, we are guaranteed that a method call whose receiver class is a subclass object at run-time, will establish

the postconditions expected from a similar method call when the receiver class is a superclass object. As the number of locations that an overriding method can modify cannot increase from those that the overridden method can modify, we are guaranteed that locations that are left unchanged by a superclass method call are also left unchanged by a subclass method call. Hence, once again, a run-time substitution of a subclass object will satisfy the compile time contract.

In summary, all overriding subtype methods must satisfy the contract of the method that they override. This is necessary for sound modular reasoning using a supertype's method contract [29, 51]. Most existing specification languages enforce behavioural subtyping through specification inheritance. By this we mean that subtypes inherit their specification from their supertypes but may add their own specifications to an overridden method.

The rules for behavioural subtyping [65] can be enforced by generating the overridden methods *effective* contract as follows:

- The effective pre-conditions becomes the disjunction of its pre-conditions and the pre-conditions of the methods which is overrides. This guarantees that the overriding method's pre-conditions is weaker that the overridden methods pre-condition.

- The effective frame condition is the intersection of its frame condition and the frame conditions of the methods which it overrides. This guarantees that the overriding methods frame condition is a subset of the overridden method's frame condition.

- The effective postconditions becomes the conjunction of its postconditions and the postconditions of the methods which it overrides. This guarantees that the overriding methods postconditions is stronger than the overridden method's postcondition.

However, these rules can hide errors [35]. Given a method with the pre-conditions $x < 0$, overridden by a method with the pre-conditions $x >= 0$, a the effective pre-conditions of the method is $x < 0 \mid\mid x >= 0$

2.3.1 Verification Issues

The most prominent issues that arise in program verifications are modular correctness, frame conditions, modular verification of class invariants and the extended state problem [74].

A method invocation can lead to the execution of code that is declared outside the module containing the invocation. This can occur due to dynamic method binding. Therefore, the correctness of a module relies on properties of the context in which it can be reused.

Frame properties list fields that can be modified by a method execution. Proving that fields these are the only fields that are modified by the method is difficult, as assignments in class extensions often have the knock-on effect of modifying a field that is not present in the frame conditions.

Class invariants lead to proof obligations for all public methods of a program as each method must reestablish the class invariant. An inherited method must establish the class invariant of both the superclass and subclass. Since re-verification of inherited methods is prevented by information hiding, invariants have to be restricted such that they can be proved based on the specifications of the inherited methods.

Subclasses inherit from their superclasses and can introduce additional fields, which are referred to as extended state. To be able to refine the behaviour of inherited methods, their specifications of frame properties must be loose enough to allow subclasses to modify the extended state.

Other verification problems that arise from inheritance include method callbacks when invariants are not satisfied, superclasses breaking subclass invariants, overriding methods can break contracts, and overriding methods can violate the frame conditions. These will be discussed in detail in chapter 4.

2.3.2 Assisting the Verification Process

In addition to the assertions that define class contracts (as discussed in section 2.3) program verification tools use other assertions to check properties of the implementation other than those visible to the client. These assertions are not concerned with specifying the external client contract. Instead, they that assist the verifica-

tion tools in proving that a specification satisfies its implementation. We discuss these assertions here.

Loop Invariants

```
public class Sq{
        private int n;

        public int Square()
        requires 0 ≤ n;
        ensures result == n*n;
        {
                int r = 0;
                int x = 1;
                for(int  i = 0; i < n; i++)
                invariant i ≤ n && r == i*i;
                invariant x == 2*i + 1;
                {
                        r = r+x; x = x+2;
                }
                return r;
        }
}
```

Figure 2.8: Example of a method specification containing a loop invariant.

Loop invariants specify the conditions that hold on each loop iteration. The Hoare logic rule for loops state that the loop invariant holds before the loop guard is evaluated, is maintained by the loop body and holds when the loop terminates. Loop invariants are not part of a method's external contract with the client. However, they are used to verify that the containing method establishes the method's postcondition. Examples are presented in Fig. 2.8 and Fig. 2.9.

Assertions and Assumptions

Assertions such as method pre-conditions and postconditions are used to describe a method contract. However, assertions may be added to a class for reasons other than providing a contract for the client. For example, static analysis tools can be used to verify assertions that are embedded into an implementation. This can be seen in the Java programming language and in specification languages like Spec# where assertions are written as executable code that may be executed during testing and removed afterwards. Likewise, SPARK supports the verification of assertions,

```
public class Summation{
        private int []! a;

        public int Sum()
        requires 0 ≤ n;
        ensures result == sum{int i in (0: a.Length); a[i]};
        {
                int  s = 0;
                int  n = 0;
                while (n < a.Length)
                invariant n ≤ a.Length;
                invariant s == sum {int i in (0: n); a[i]};
                {
                        s += a[n]; n++;
                }
                return s;
        }
}
```

Figure 2.9: Example of loop invariants with quantifiers.

although these are added to program comments rather than to the implementation. In these systems, the execution of an assertion has no effect if the assertion holds in the state in which it is evaluated. If the assertion does not hold then the program execution is stopped and an error is reported.

Like JML and Spec#, we support the addition of an assertion to a method implementation as illustrated in Fig. 2.10. The notation **assert** E requires that the pure boolean expression E is verified via static reasoning. The assertion **assert** $x < y$ in Fig. 2.10 evaluates to *true*. If it is replaced with **assert** $x > y$ an error should be reported at compile time.

We also support the addition of assertions that verification tools can assume to hold. These are typically assertions that cannot be verified in the given context but are needed for the verification. The statement **assume** E is like **assert** E at runtime, but the static program verifier checks the assert whereas it blindly assumes the assume statement.

Issues Associated with Behavioural Interface Specification Languages

The behavioural interface specification style used in JML and Spec# is in contrast with specification languages, such as those from the Larch family [37, 86], which provide a special mathematical syntax that is ideally suited for input into automatic verification tools. These mathematical languages have the advantage that specifica-

```
public class Minimum{
        private int x;
        private int y;
        public int Min()
        requires true;
        ensures (x<y) ⟹ result == x;
        ensures (x>=y) ⟹ result == y;
        {
                int m;
                if (x<y){
                        assert x<y; m = x;
                } else {
                        assume x>=y; m = y;
                }
                return m;
        }
}
```

Figure 2.10: Example of assertions added using **assert** and **assume**.

tions can be written in terms of abstract values [40]. This allows the specifications to be an abstraction of the concrete state of the program. Their disadvantage is the overhead created for programmers, who need to learn, understand and express both the specification language and the implementation language.

The approach taken in JML and Spec# eliminates this overhead but there is a trade-off. Specifications which are written in an implementation-like syntax are not suited for input to automatic verification tools. Hence, the overhead becomes an issue for the tool developer, as specifications must be encoded in a form that automated tools can understand and verify.

A further issue arises due to the lack of distinction between the specification language and the implementation language. The issue is that the specification often gets written in terms of the concrete data types. This has a number of drawbacks. First, it exposes implementation details in the specification, letting clients know how the specification is implemented. Second, this coupling between the specification and the implementation creates a dependency which requires that the specification is rewritten every time that the implementation details are changed. Third, verification tools may need to execute some of the code in the implementation to determine the state of variables used in the specification. This has the negative effect of slowing down the tool's performance.

We address these issues by providing support for data refinement so that implementations can be written in terms of a concrete data type and can be proved

to satisfy a specification that is written in terms of another data type. This work is relevant to languages where the syntax used in specifications and implementations is shared. The verification that the implementation satisfies the specification is supported through a framework for data refinement.

2.4 Data Abstraction Challenges for Object-Oriented Programming Languages

Leavens, Leino and Müller describe several important specification and verification challenges for sequential object-oriented programming languages [49]. They draw on their experience of specifying and verifying code using JML and Spec#, and static checking and verification tools for programs written in these languages.

The challenges of most interest to our work concern data abstraction in specifications. By data abstraction we mean that specifications are written in an implementation-independent way. Two main challenges in this area are identified. The first challenge,is the development of a specification technique for modeling types. Modeling types that describe mathematical types such as bags, sets, sequences, relations and maps were added to JML so that specifications could be written in terms of abstract mathematical types [48]. While these built-in types work well for run-time assertion checking, they are difficult to specify in a way that is useful for static verification.

Rather than using built-in modeling types for specification, we use the data types that are already supported in our programming language. Our focus is on the data abstraction techniques that are used to relate modeling types to their implementations. We provide a stand-alone specification that is expressed in terms of one data type, and which may be implemented in terms of another. In this way the client can view and use the specification without knowing any implementation details. The data type used in the implementation is referred to as the concrete data type and the data type used in the specification is referred to as the abstract data type.

Through the laws of data refinement [72] we can prove that the implementation is a correct implementation of the specification. This proof relies on a mapping that

relates the abstract and the concrete data types. We say that the implementation *refines* the specification. The major advantage of providing data abstraction in specifications is that specifications can be written and reasoned about in a way that is independent of their implementations. However, the verification of a data refinement must also consider the specification's implementation. In an object-oriented programming language these verifications will be influenced by issues such as mutable objects, aliasing, subtyping and modularity.

The second challenge identified in the area of data abstraction is the development of a verification technique for general quantifiers and comprehensions that is suitable for automatic verification tools. These tools encode proof obligations as first-order formulae that are passed to automatic theorem provers like Simplify [28] or Z3 [27]. Our work in this area is documented in [57, 59] where a technique for translating common comprehension expressions (**sum**, **count**, **product**, **min**, and **max**) into verification conditions that can be tackled by two off-the-shelf first-order SMT solvers is presented. The technique has been implemented in the Spec# program verifier.

2.5 Conclusion

In this chapter, we presented an overview of object-oriented programs and their specifications. We introduced behavioural interface specifications using a notation that is similar to that of popular object-oriented specification languages JML and Spec#. The factors that influence the modular verification of object-oriented programs were identified and some specification and verification challenges for data abstraction in sequential object-oriented programming languages were presented.

We aim to address these challenges through the provision of a framework which supports the modular data refinement of object-oriented programs. This framework will provide a stand-alone specification that is expressed in terms of one data type, and which may be implemented in terms of another. In this way the client can view and use the specification without requiring any implementation details. Related work, where we added support for comprehensions to the Spec# programming system, provides for the improved specification of abstract data. We discuss the specification and verification of data refinements in Spec# in chapter 4. First, we provide

an overview of the refinement calculus, which is concerned with the transformation of specifications into executable programs through a series of correctness-preserving steps.

Chapter 3

The Refinement Calculus

Refinement is a style of development that leads from a formal specification to a working implementation in a sequence of correctness-preserving steps. The refinement calculus [5, 6], based on the weakest pre-condition approach to program correctness, provides a set of laws that supports the derivation of a program that satisfies a given specification. Data refinement arises when we decide on a change of data representation and want to derive the new version of the specification from the old one. The change of data representation can arise for a variety of reasons. The classical case is employing one type of data to facilitate the specification process and replacing it with a more concrete type to allow implementation. We refer to this as type transformation in our work in [17]. In this chapter, we review the main components of the refinement calculus. Refinement laws for procedural refinement and data refinement are presented as well as strategies by which a data refinement can be verified.

3.1 Refinement

The language in which we write specifications and their implementations is a language of typed terms. We view the specification language as a programming language with extra terms that improve the expressiveness of our language. These more expressive terms are used in the early stages of software development to describe the program that we wish to implement. The notation used in the specification is richer

in two ways. Firstly, it has more expressive terms, such as those that involve quantified expressions. Secondly, the notation has richer data types than those available to the executable program. Such data types allow specifications to be written in a way that is independent of implementation details.

Refinement is a multi-step process which transforms a specification into a more algorithmic one through a series of correctness-preserving transformations. These transformations rewrite the non-executable statements in terms of executable constructs while maintaining the original meaning of the specification. The two specifications are not necessarily functionally equivalent as the refined specification might eliminate some of that non-determinism offered by the original specification. The main types of refinement that we consider here are procedural refinement and data refinement. We discuss other refinement techniques in [17]. *Procedural refinement* [6, 69, 71] focuses on transforming the more expressive statements of specifications into executable statements while *data refinement* focuses on a replacing the data types used in a specification with more concrete representations. Together they achieve executable specifications that may be expressed in terms of data types other than those used in the original specification.

Refinement offers a method of programming where the implementation is *correct-by-construction*. By this we mean that the implementation is constructed from the specification through a series of steps that are known to be correctness preserving. An alternative method is to generate an implementation and to prove that it satisfies its specification using a technique such as the weakest pre-condition calculus.

3.2 Notation

We take Dijkstra's language of guarded commands as our programming language. Our specification language is a super-set of this language allowing more expressive statements (which take the form of predicate calculus formulae), as well as a richer set of data types. We adopt the notation of Morgan [70] where specifications have the form

$$\vec{w} : [pre, post]$$

Like contracts in the design-by-contract methodology, these specifications consist of three parts. First, the pre-condition, denoted by the keyword *pre*, provides the initial conditions that must hold for the implementation to function correctly. Next, the postcondition, denoted by the keyword *post*, specifies the state that is achieved if the implementation is executed in a state where its pre-condition is true. Third, the frame condition, denoted by $\vec{w}$, lists the variables that the implementation may modify in order to achieve the postcondition. The specification

$$y : [x >= 0 \wedge y > 0, y >= x]$$

provides the pre-condition $x >= 0 \wedge y > 0$, the postcondition $y >= x$ and the frame variable y. When this specification is refined by an executable program, its execution from a state that satisfies the pre-conditions will achieve the corresponding postconditions through modification of variable y.

Terms

The language in which we write specifications is a language of typed terms. The symbols T, $T1$, $T2$ etc represent general types in the language. We write $E : T$ when the term E has the type T. Terms, also called expressions, are either

- variables which are denoted by a small letter a - z
- constants such as 0 or
- functions that are applied to other terms e.g., x+1

We associate types with a variable through variable declarations using the keyword *var*; the declaration $var\ x : T$ declares variable x as having type T. The symbols $\|$ and $\|$ are delimiters for the local *scope* of a declaration. Any declarations made within $\|$ and $\|$ exist only within that scope. The formulae to which they apply are separated from the declarations by the symbol $\bullet$. We write

$$\|var\ x : T \bullet\ w, x : [pre,\ post]\|$$

to declare variable x with type T in the scope of the specification $w, x : [pre, post]$.

Constants are declared using the keyword *con* and used to refer to values that do not change within the scope of the declaration.

The sequential composition operator may be used to associate assumptions with variable declarations: we write $var\ x : T;\ and\ inv$ to declare x with type T and that the assumption inv is true. For example, $var\ x : N;\ and\ x> = 1$ declares x a natural number, which has a value $> = 1$. While the variable declaration is part of both the specification and the programming language, the *and inv* declaration is only part of the specification language and hence must be removed during refinement.

Terms can be combined with predicate symbols (such as the relational operations) to form boolean terms. These terms are combined, using the standard boolean operators $\wedge, \vee, \neg, \Rightarrow$ and $\Leftrightarrow$, to generate propositions which are used to express assertions such as pre-conditions and postconditions.

The value of a term may be determined through its **state,** the mapping of variables to their values. E.g., The term

$$x = 5 \wedge y = 10 \wedge x + y = z$$

describes a state in which x has the value 5, y has the value 10 and the value of z is 15.

The quantifiers $\forall$ and $\exists$ have their usual meaning, allowing for the consideration of a collection of values. Given a formula A,

- $(\forall x \bullet A)$ is true when A is true for all values of the bound variable x, and
- $(\exists x \bullet A)$ is true when A is true for at least one value of the bound variable x.

All free occurrences of x in A become bound occurrences in the overall quantified formulae. Typed quantifications are written $(\forall x : T \bullet A)$ and $(\exists x : T \bullet A)$, where T denotes a set of values which variable x ranges over in the term A. Types in our language include the Real numbers (R), the Naturals (N), the Integers (Z) and the Boolean type (B) with the usual operators. We also include Sets, Bags and Sequences which use the following traditional notation:

- Set elements are written between { and }. Basic set operations include set union ($\cup$), set intersection ($\cap$), set difference (-), cartesian product ($\times$), membership (ϵ), inclusion ($\subseteq$), strict inclusion ($\subset$) and cardinality (#), while the empty set is represented by $\emptyset$ or { }. To describe sets of terms we use set comprehension of the form {x:T | P • t } where x is a variable of type T and which may occur in predicate P. The elements of the set are the terms t which satisfy the predicate P. For example, the set $\{n : N \mid \exists m : N \bullet n = 2m \bullet n\}$ contains the natural numbers that are even.

- Bag elements are written between $\lfloor$ and $\rfloor$ and the basic operations are bag union ($\sqcup$), bag intersection ($\sqcap$), bag difference (-), bag addition (+), bag cardinality (#) and membership (ϵ), while the empty bag is represented by $\lfloor\rfloor$.

- Sequence elements are written between $<$ and $>$ with operators including: concatenation of a single element to a list (:), concatenation of two lists (##), head (hd), tail (tl), front (ft), last (lt) and length (#). The empty sequence is represented by $<>$.

Functions and Relations

Mathematically, functions and relations may both be regarded as sets of pairs. All the pairs are elements of the Cartesian product of its source and its target type. We assume the usual definitions of functions and relations with the notation $var\ f : T1 \nrightarrow T2$, $var\ g : T1 \rightarrow T2$ and $var\ r : T1 \longleftrightarrow T2$ denoting the declaration of partial functions, total functions and relations respectively. Relations map an element of the domain type to many elements in the range whereas functions map an element of the domain type to at most one element in the range.

Given a function $f : T1 \nrightarrow T2$ and some $x : T1$ we apply a function f to its argument x by writing $f(x)$. A function application is a term that evaluates to an element in the range of the applied function. We can override a function f by another function by writing $f \oplus g$ which behaves as follows:

$$(f \oplus g)(x) = g(x) \quad \text{if } x \in dom\ g$$
$$(f \oplus g)(x) = f(x) \quad otherwise$$

3.2.0 Reasoning about Specifications

The weakest pre-condition that must be satisfied for a specification to achieve a postcondition R, is defined using the wp function

$$wp((\vec{w} : [pre, post]), R) \triangleq pre \wedge (\forall \vec{w} \bullet post \rightarrow R)[\vec{v_0} \backslash \vec{v}]$$

where $\vec{v}$ is a vector containing all the specifications variables, the vector $\vec{v_0}$ represents those variables values in the initial state and $[\vec{v_0} \backslash \vec{v}]$ represents the substitution of all values in $\vec{v_0}$ with the corresponding values in $\vec{v}$.

When reasoning about specification we relate different predicates so that properties may be proved about the specification. Two predicates, A and B, are equivalent if, in every state, A is true if and only if B is true; we write $A \equiv B$. We write $A \Rightarrow B$ if in every state, if A is true then B is true; and A $\Leftarrow$ B if, in every state, A is true if B is true.

Predicates that evaluate to $true$ are satisfied by all states while predicates that evaluate to $false$ are not satisfied by some states. These relations, together with the laws of predicate calculus (e.g., associativity and commutativity of $\wedge$ and $\vee$) allow us to reason about specifications.

3.2.1 Programming Constructs

The programming language is an implementable subset of the specification language. Specifications are refined by executable specifications which are written in terms of assignments ($:=$), sequential composition ($;$), skip statements, if statements and loops. We also provide for the declaration of procedures and modules which allow us to group language constructs into one component that can be reused within a program. The conventional meaning applies for all of this programming constructs. We outline our notation below.

Assignment: The notation $w := E$ represents an *assignment* where the variable w is mapped to the value of expression E. All other variables are unchanged. A multiple assignment changes many variables at the same time:

$w_0, w_1, ...w_n := E_0, E_1, ...E_n$ assigns E_0 to w_0, E_1 to $w_1, ..., E_n$ to w_n simultaneously.

Skip: The *skip* command specifies when no change in behaviour is required. An equivalent program is one that assigns variables to themselves achieving no change in state.

Sequential composition: We write $S_0; S_1$ to mean term S_0 followed by term S_1. Refinement laws allow the use of ; to combine terms into a refined term.

Conditional Statements: The programming language with which we work is the language of guarded commands. Guarded commands have the form $G \rightarrow P$ where G is a boolean valued term, referred to as the guard. If G is true then $G \rightarrow P$ behaves in the same way as the term P. We use the traditional guarded command notation for these statements: $if\ G_0 \rightarrow P_0\ []\ G_1 \rightarrow P_1\ []\ \ldots\ []\ G_n \rightarrow P_n\ fi$ where $G_0 \ldots G_n$ are guards. We write $if([]i.G_i \rightarrow P_i)fi$ as shorthand. The meaning of this statement is that if exactly one guard G_i, is true its corresponding command P_i, will be executed. If several guards are true, then one of them is selected and its corresponding command executed. If the alternation has been designed properly, it will not matter which of the guards is executed and no assumption about which one will be chosen will be possible. If no guard is true, the full statement aborts.

Iterations: Iterations are built from a single guarded command: $do\ G \rightarrow P\ od$ which repeatedly executes P when G evaluates to *true*.

Procedures: Procedures are similar to functions and methods, in that they provide a way of naming a group of language constructs that may be reused many times. They differ from functions in that they do not return a result when they are called. While procedures help to control the size and structure of a specification, their main advantage is due to the reuse of their refinements: whenever a procedure definition is reused in a specification, it may be replaced by that procedure's refinement.

Procedures declared using the syntax $procedure\ N(value\ f : T) \triangleq P$ have N

```
procedure Swap(value x:N, value y:N) ≙ x,y:[x=X ∧ y=Y,x=Y ∧ y=X]
procedure Square_root(value x:R, result r:R) ≙ r := √(x)
```

Figure 3.1: Procedure declarations.

as the name of the procedure, f, which has type T, as the name of the procedure's formal parameter and P as an executable term or a specification that forms the procedure body. In the context of refinement we use the symbol $\triangleq$ for "is defined as" and the symbol $=$ for equality. As shown in Fig. 3.1 procedure bodies may have the form of a non-executable specification or the form of an executable program. Procedures that contain specifications may have the form

$$procedure\ N(value\ f : T) \triangleq w, f : [pre, post]$$

whereas procedures that contain code may have the form

$$procedure\ N(value\ f : T) \triangleq f := E.$$

Calling a procedure requires that the actual parameter is substituted for the formal parameter; this substitution can be achieved in a number of alternative ways. These are *substitution-by-value*, *substitution-by-result* or *substitution-by-value-result*. Parameters are labelled with the substitution type expected Fig. 3.1.

Substitution-by-value allows the direct substitution of the actual parameter for the formal parameter. Given the definition of the procedure $Swap$ as in Fig. 3.1, the procedure call $Swap(a, b)$ results in the values of the actual parameters a and b being substituted for the formal parameters x and y. As before, we use the notation $a\backslash x$ to represent the substitution of a for x. The actual parameter must be a term that evaluates to a legal value of the appropriate type. Variable capture is avoided in the case of variable name clashes.

Substitution-by-result is complementary to substitution-by-value, It takes a value out of a procedure, rather than passing a value in. As procedure calls are not terms, they do not return a result. Instead the result of executing the procedure call can be stored in a procedure parameter using substitution-by-result (Fig. 3.1). Substitution-by-value-result is used when we want to combine the functionality of

substitution-by-value and substitution-by-result in the parameter.

Modules: Modules allow the programmer to bring together related parts of a program, in a similar way to procedures, so that the details that do not concern the client can be hidden. In this way, modules may be used for data abstraction. Modules typically contain local variable declarations, procedure declarations and variable initialisations as in 3.2. Refinement of a modules individual procedures, results in the overall refinement of the module.

Procedures are local to the module but may be exported for use in other modules and programs. Procedures that are not exported are only available for use within the module. We refer to these as *local procedures*. Variables can be exported for reading but they cannot be changed outside a module. They can only be changed by local procedures of a module. Both variables and procedures can be imported for reuse and re-declared within that module (see Figure 3.2). This re-declaration allows reasoning about the module to be independent of the overall program.

```
module Date{
        export AddDay;
        import GetDay;

        var day:N;
        var month:N;
        var year:N;

        init() ≙ day, month, year: [True, day = month = year = 0;]
        procedure AddDay() ≙ day:[True, day = old(day) +1;]
        procedure GetDay(result d:N) ≙ :[True, d = day;]
}
```

Figure 3.2: An example of a module that specified a Date.

3.3 The Refinement Calculus

Specifications are refined through applying refinement laws from the refinement calculus [5, 69, 71]. The refinement calculus originates with the stepwise refinement method of program construction [30, 41]. In this section, we recall the refinement calculus of Morgan [70], where a specification is refined through the application

of refinement laws. Through the refinement calculus, the refined specification is guaranteed to be a correct refinement of the original specification.

3.3.0 The Refinement Relation

As is normal in refinement, we define a relation $\sqsubseteq$ on terms, such that $s_0 \sqsubseteq s_1$ formalises the statement that "term s_0 is refined by term s_1" or that "term s_1 refines term s_0". A refinement law is an inference rule that allows us to deduce that a refinement $s_0 \sqsubseteq s_1$ is valid. As equality is a stronger property than refinement, we can also treat any rule which concludes that two terms are equal as a refinement law. We use refinements laws to establish that the individual refinement steps are correct in a derivation $s_0 \sqsubseteq s_1 \sqsubseteq s_2 \sqsubseteq ... \sqsubseteq s_n$.

The refinement process is possible because of two properties of the $\sqsubseteq$ relation. First, it is transitive. By this we mean that $s_0 \sqsubseteq s_1 \wedge s_1 \sqsubseteq s_2 \Rightarrow s_0 \sqsubseteq s_2$. Another important property of refinement is *monotonic replacement.* This permits specifications to be refined through refining their components in isolation. The refined components may then be substituted back into the specification to achieve an overall refinement. We express this property as follows: for specification S and terms s_0, s_1 and s_2 if $s_1 \sqsubseteq s_2$ then $S(s_0 \backslash s_1) \sqsubseteq S(s_0 \backslash s_2)$. Note that where $S(s_0 \backslash s_2)$ denotes S with each occurrence of s_0 replaced by s_2.

The result of the refinement process is the generation of a sequence of specifications where any of the specifications may contain a mixture of executable and non-executable constructs. The initial specification is typically written in terms of the richer non-executable terms, while the refined specification contains only executable terms. If the refined specification, s_n, is fully executable, we call it a program and say that the specification s_0 *is refined by* s_n.

Through studying the refinement relationship between specifications, laws which transform a specification into its refinement has been documented. We examine these in the section that follows.

3.3.1 Refinement Laws for Reasoning about Specifications

The laws presented here, and others, are available in [5, 69, 71]. These laws provide for the refinement of initial specifications into executable specifications. Equivalent

rules are applied in program verification tools to reason about the implementations when verifying that they satisfy their specification.

Strengthening postcondition: A specification of the form $w : [pre,\ post]$ is refined by strengthening the postcondition as follows:

$$\text{if post' } \Rightarrow \text{ post then } w : [pre,\ post] \sqsubseteq w : [pre,\ post']$$

Strengthening the postcondition improves the specification from the client's point of view, as it achieves what the original specification achieved and more. For example:

$$\begin{array}{rl} & y : [x >= 0 \wedge y > 0, y >= x] \\ \sqsubseteq & y : [x >= 0 \wedge y > 0, y >= x \wedge x > 10] \end{array}$$

Weaken Pre-condition: A specification may be refined by weakening its pre-condition as follows:

$$\text{if pre' } \Rightarrow \text{ pre then } w : [pre',\ post] \sqsubseteq w : [pre,\ post]$$

The resulting specification will achieve the original postcondition but under a broader range of conditions than the original specification. For example:

$$\begin{array}{rl} & y : [x >= 0 \wedge y > 0, y >= x] \\ \sqsubseteq & y : [x >= 0, y >= x] \end{array}$$

Assumption: When the frame is empty and the postcondition is true we call the pre-condition an assumption. We write the assumption pre as $\{pre\}$ and define it as $[pre, true]$ i.e. if an assumption is true, the postcondition is achieved. Otherwise the program fails to terminate. Assumptions before a specification can be absorbed into its pre-condition as follows:

$$\{pre'\}\ w : [pre, post] \triangleq w : [pre' \wedge pre, post]$$

Sequential Composition: A specification is refined by the sequential compo-

sition of two other specifications if the postcondition of one specification implies the pre-condition of the other. Formally, we write:

$$w : [pre, post] \sqsubseteq w : [pre, inter]; w : [inter', post]$$

where $inter \Rightarrow inter'$

Introduce Local Variable: Local variables, invariants and constants may be introduced to specifications. For example, a local variable x may be introduced into a specification using the following law:

If x does not occur in w, pre or $post$ then

$$w : [pre, post] \sqsubseteq [|\ var\ x : T \bullet w, x : [pre, post]\ |]$$

Remove Constant: Constants may be removed from a specification during refinement as follows:

If c occurs nowhere in the program P then $[|\ con\ c : T \bullet P\ |] \sqsubseteq P$

Expand Frame: Recall from section 2.2.3 that we use the convention that terms with a 0 subscript in a postcondition refer to the initial value of these terms e.g., x_0 refers to the initial value of x. The following law expands the frame to include extra variables that can be modified:

$$w : [pre; post] = w, x : [pre, post \wedge x = x_0]$$

3.3.2 Some Interesting Specifications

The following specifications also describe interesting situations which we need to be aware of in verifications.

Abort $w : [false, true]$.
From the clients point of view, *Abort* describes the worst specification of all, as it

is never guaranteed to terminate (pre-condition false) and allows any refinement to have complete freedom in setting any variables.

Choose w $w : [true, true]$.
Choose w always terminates, but guarantees no particular result. It can be refined by any terminating program that changes only w.

Skip $: [true, true]$.
The specification *Skip* always terminates, changing nothing.

Magic $w : [true, false]$.
The infeasible specification $Magic$ always terminates and establishes the impossible postcondition $false$. This specification cannot be executed on a computer and no contract based on it could ever be met.

Note that if the pre-condition of a specification is true, we may omit the pre-condition from the specification. We write $w : [post]$ instead of $w : [true, post]$.

3.3.3 Laws for Refining Specifications to Executable Programs

If the *feasibility* of a specification can be established, it can be refined by an executable program. The specification $w : [pre;\ post]$ is feasible if and only if the pre-condition implies that there exists a w such that the postcondition is established. It is important to establish the feasibility of a specification, as infeasible specifications cannot be refined to code. This is achieved by introducing programming language components such as assignment statements, if statements, loops and procedure calls. We now describe some of the laws that permit these transformations. Examples of their use can be seen in Appendix A.

Assignment:

Code that refines a specification may be produced by applying the following assignment law:

$$\text{If } pre \rightarrow post[w\backslash E] \text{ then } w : [pre, post] \ \sqsubseteq \ w := E$$

The assertion $post[w\backslash E]$ stands for the assertion $post$ with each free occurrence of variable w replaced by the term E. If E and w are lists, the rule applies to corresponding elements of the lists. Provided E contains no w, the assignment $w := E$ is equivalent to $w : [true, \ w = E]$. From the client's point of view, the statement $w := E$ is better than $w : [pre, post]$ because $post$ may have allowed several final values for w, whereas now only one is allowed and the client knows exactly which one it will be. An example that shows a specification that is refined by an assignment is $x : [x >= 0, x > 0] \sqsubseteq x := x + 1$ as

$$\begin{aligned} & x >= 0 \rightarrow (x >= 0)[x\backslash x + 1] \\ \equiv\ & x >= 0 \rightarrow x + 1 >= 0 \\ \equiv\ & true \end{aligned}$$

Skip:

We specify that no changes occur in a specification by using the $skip$ command:

$$\text{if } pre \rightarrow post \text{ then } w : [pre, \ post] \sqsubseteq skip$$

An equivalent program is one that assigns variables to themselves:

$$w : [pre, post] \sqsubseteq w : [pre, post]; w := w.$$

Alternation:

An alternation statement may be introduced into a specification by applying the following refinement law:

$$\begin{aligned} & \text{if } pre \rightarrow G_0 \vee G_1 \vee ... \vee G_n \text{ then} \\ & w : [pre, post] \quad \sqsubseteq \quad \text{if } (\ [\ ]i.G_i \rightarrow w : [G_i \wedge pre, \ post]) \text{ fi} \end{aligned}$$

The meaning of $([\ ]\ i.G_i \rightarrow w : [G_i \wedge pre,\ post])$ is that if exactly one guard G_i, is true the specification $w : [G_i \wedge pre,\ post]$ is the specification of the code that will be executed. This specification can be refined by one that contains an alternation statement as described in section 3.2.1.

Iteration:

Iteration is used to implement the repeated execution of a command (the loop body), while the loop guard evaluates to true. A *loop invariant* is a formula which, if true initially, is true after every iteration of the loop.

We write $w : [G \wedge inv, inv] \sqsubseteq P$ where G represents the guard, inv represents the loop invariant and P represents the loop body. To guarantee termination, the loop body must also establish the negation of the guard. Therefore, if the invariant conjoined to the negation of the guard implies the loop postcondition, then the specification may be refined as follows:

$$w : [inv,\ inv \wedge \neg G] \sqsubseteq \text{do } G \rightarrow\ P \text{ od}$$

We include a *variant* function, which determines when the loop iterations will terminate. In general, some integer valued term, bound by a lower bound and that is strictly decreasing on each iteration of the loop is chosen. The refinement law has the following format:

$$w : [inv, inv \wedge \neg G] \sqsubseteq do\ G \rightarrow w : [inv \wedge G, inv \wedge 0 <= V < V_0]\ od.$$

where neither G or inv contain initial variables, V is the variant function and V_0 is the initial value of the variant function ($V[w \backslash w_0]$).

Procedures:

Specifications may be refined by procedure calls, given appropriate procedure definitions. At the specification level, a procedure declaration has the form

$$procedure\ P(f : T) \triangleq w, f : [pre, post]$$

with local variable w and parameter f, while procedures defined at the code level have the form

$$procedure\ P(f : T) \triangleq w, f := E, F$$

The assignment $w, f := E, F$ assigns the local variable w the term E, and the parameter variable f the term F, simultaneously. We often use the symbol ? to denote a yet undetermined term for assignment.

The refinement laws used when refining specifications to procedure calls depend on the parameter substitution method used and whether the procedure is specified at the specification level or at the code level. The following rules are used when refining procedure calls that have their parameters passed by value.

Value Assignment: Given the procedure $procedure\ P(value\ f : T) \triangleq w, f := E, ?$, then $w := E[f\backslash A] \sqsubseteq P(A)$ where the actual parameter A is a term of type T and variables w and f are disjoint.

Value Specification: Given the procedure $procedure\ P(value\ f : T) \triangleq w, f : [pre, post]$ then $w : [pre[f\backslash A], post[f_0\backslash A_0]] \sqsubseteq P(A)$ where A_0 is $A[w\backslash w_0]$ and $post$ contains no f.

Similar refinement laws for procedure calls that use substitution-by-result may be found in [70].

Modules:

If $M \sqsubseteq N$ where M and N are modules then we call M a specification module and N an implementation module. The specification module is made public to the client and the implementation module remains private to the supplier. As a module can be refined by many different implementations, a specification module may have many implementation modules associated with it. Morgan [70]defines refinement of modules by the following law.

Let E be the list of exported procedures from module M, I its imported procedures, and *initially* its variables initialisation. A module M' refines M if the following three conditions are satisfied:

- Its exported variables are unchanged

- Its exported procedures E' refine E
- Its initialisation *initially'* refines *initially*

The following changes can also be made as long as the conditions above are not invalidated. Conditions specified on imported variables' declarations can be weakened and a modules imported procedures I' can be refined by I. In addition, an imported procedure I can be replaced by a local (neither imported or exported) procedure I' that refines I. Refinement of procedures is described above.

3.4 Data Refinement

Data refinement [7, 70, 71, 89, 88, 90] originates in Hoare's work on program correctness [39]. Specifications are typically written using a rich set of abstract data types such as sets, bags and mappings. We refer to these specifications as abstract specifications. Data refinement is a special instance of refinement where an abstract, possibly non-executable, specification is refined by modifying the data type used in the specification. This change is data representation can arise for a variety of reasons. Examples include the introduction of extra variables to improve efficiency and changing the data type, to a more concrete one, the implementation stage [17].

The theory of data refinement [72] provides for these systematic changes while maintaining consistency with the original specification. The overall effect of the data type substitution is the refinement of the specification in which the abstract data type is used. We refer to the refined data type as the *concrete* data type. A standard example of data refinement is to replace a mathematical set with a sequence representation such as an array or a singly linked list.

We aim for semantic implementation correctness, as defined in [36], where "Given two programs, one called concrete and the other called abstract, the concrete program implements the abstract program correctly whenever the use of the concrete program does not lead to an observation that is not also an observation of the abstract program."

Changing the data type used in a module definition results in the refinement of the module's external behaviour. Using the laws for data refinement we refine a module through a three step process:

0. The concrete variables are added to the module.

1. The module operations are transformed so that they refer to the concrete variables rather than the abstract variables. This transformation ensures that the abstraction invariant is satisfied throughout the specification.

2. The abstract variables are removed.

In order to transform the operations, a number of rules must be followed. These are outlined below and described fully in [70]. For simplicity, we assume that there is only one abstract variable a and one concrete variable c. There may, of course, be many. We use the shorthand AI, to refer to the abstraction invariant, which relates the abstract variable a to the concrete variable c. The notation $w : [pre,\ post]$ describes a procedures specification and *initially* refers to the procedure which initialises the module's data(as in section 3.3.3).

3.4.0 Augmentation Laws

These laws introduce concrete variables into abstract specifications, ensuring that the specifications do not change their meaning. The main laws are summarised as follows.

Augment Initialisation: The postcondition of the procedure that initialises module variables, is strengthened to include the abstraction invariant. Hence, a specification of the form $w : [true,\ w = 0]$ becomes $w : [true,\ w = 1 \wedge w = x + y]$ where w is the abstract variable and the concrete variables are x and y.

Augment Specification: The abstraction invariant is conjoined to the pre-conditions and postconditions of procedure specifications and the frame is extended to allow the new concrete variables to change. Hence, each procedure specification $w : [pre, post]$ becomes $w, c : [pre \wedge AI,\ post \wedge AI]$

Augment Assignment: Assignments in procedure bodies must preserve the abstraction invariant and hence, their specifications must be extended so that they can change the concrete variables. Therefore, an assignment such as $w := E$ is

replaced by $\{AI\}\ w, c := E, ?\ \{AI\}$ where $\{AI\}$ is the assumption that the abstraction invariant holds and ? represents a value assigned to the concrete variable c so that the abstraction invariant is preserved.

Augment Guard: Guards in specification statements can be rewritten in terms of the concrete variables. Hence, a guard G, that is written in terms of the abstract variables, may be replaced by another guard G', that is written in terms of the concrete variables, provided that $AI \rightarrow (G \equiv G')$.

3.4.1 Diminution Laws

When the concrete variables have been added to the specification and the specification has been rewritten in terms of the concrete variables, the abstract variables may be removed from the specification. The abstract variables declaration is simply deleted and the following rules are applied to remove all occurrences of the abstract variables in the refined specification. We refer to the abstract variable by the letter a and assume that it has type A. We write $a : A$ to express that variable a has type A.

Diminish Initialisation: The specification of the procedure that initialised variables is modified so that the postcondition no longer refers to the abstract variables. Instead it specifies the existence of an abstract variable which allows the specification to be satisfied. Therefore, the specification $w : [pre,\ post]$ becomes $: [pre,\ \exists a : A \bullet post]$. For example, the specification $w : [true,\ w = 1 \wedge w = x + y]$ becomes $: [true,\ 1 = x + y]$ when the concrete variables x and y are added and the abstract variable w is removed.

Diminish Specification: The specification $w, a : [pre, post]$ becomes $w : [\exists a : A \bullet pre, \forall a_0 : A \bullet pre_0 \Rightarrow (\exists a : A \bullet post)]$ where pre_0 is $pre[w, a \backslash w_0, a_0]$. The frame beforehand must include the variable a. The pre-condition becomes $\exists a : A \bullet pre$, as we know that there exists a value of a although we cannot say what that value is. The postcondition becomes $\forall a_0 : A \bullet pre_0 \Rightarrow (\exists a : A \bullet post)$, as we specify the existence of an abstract variable a which allows the specification to be

satisfied. The quantification involving a_0 indicates that the postcondition depends on all values of a, not just its initial value a_0.

Diminish Assignment: Assignments to abstract variables can be removed as follows. The assignment $w, a := E, F$ can be replaced with $w := E$ if E contains no variable a.

Note that all references to abstract variables in guards must also be removed. This is achieved by the refinement law for alternation discussed earlier.

3.4.2 A Data Refinement Example

An example of data refinement in [70] specifies a calculator in terms of a bag of real numbers and implements it in terms of a pair of numbers. We now present this example to illustrate the data refinement process. Recall that bag elements are written between $\lfloor$ and $\rfloor$, the operator for bag addition is $+$ and the operator for bag enumerations is $\#$, while the empty bag is represented by $\bigsqcup$. The calculator operates by clearing any input, entering values one at a time and then taking the mean of those values. The abstract specification is shown in Fig. 3.3:

```
module Calculator{
        var b:Bag R;

        initially() ≙ b:[true, b := ⨆];
        procedure Clear() ≙ b := ⨆;
        procedure Enter(value r:R) ≙ b := b + ⌊r⌋;
        procedure Mean(result m:R) ≙ m:[b≠⨆, m = (Σb)/#b]
}
```

Figure 3.3: Specification of the calculator module.

To implement the bag in terms of its sum, s, and its size, n, we use the laws of data refinement from section 3.4.0 and section 3.4.1 with

- abstract variable: $b : bag\ R$,
- concrete variables: $s : R$ and $n : N$, and

- abstraction invariant: $s = \Sigma b \wedge n = \#b$.

The steps involved in the data refinement are:

0. Add declarations of the concrete variables $s : R,\ n : N$,

1. Use **Augment Initialisation** so that the initialisation $b : [true, b := \bigsqcup]$ becomes
   $b : [true,\ b := \bigsqcup \ \wedge\ s = \Sigma b\ \wedge\ n = \#b]$.

2. Use **Augment Assignment** to add assignments to the concrete variables while maintaining the abstraction invariant.

   - In the procedure *Clear* replace $b := \bigsqcup;$ with $b, s, n := \bigsqcup, 0, 0;$
   - In the procedure *Enter* replace $b := b + \lfloor r \rfloor;$ with $b, s, n := b + \lfloor r \rfloor, s + r, n + 1;$

3. Use **Augment Specification** to include the abstraction invariant in procedure specifications.

```
module Calculator {
        var b:Bag R;
        var s:R;
        var n:N;

        initially() ≙ b:[true, b := ⨆ ∧ s = Σ b ∧ n = #b];
        procedure Clear () ≙ b,s,n:= ⨆,0,0;
        procedure Enter(value r:R) ≙ b,s,n := b+⌊r⌋, s+r, n+1;
        procedure Mean(result m:R) ≙
                        m,s,n:[b≠⨆, s = Σ b ∧ n = #b, m = (Σ b)/#b]
}
```

Figure 3.4: Specification of the calculator module contained both abstract and concrete data.

The resulting module, shown in Fig. 3.4 contains a mixture of abstract and concrete data. The *Mean* procedure is refined as follows.

$$
\begin{array}{ll}
 & m, s, n : [b \neq \bigsqcup,\ s = \Sigma b \wedge n = \#b,\ m = \Sigma b / \#b] \\
\sqsubseteq & m, s, n : [n \neq 0,\ s = \Sigma b \wedge n = \#b,\ m = s/n] \\
\sqsubseteq & m : [n \neq 0,\ s = \Sigma b \wedge n = \#b,\ m = s/n] \\
\sqsubseteq & m : [n \neq 0,\ s = \Sigma b \wedge n = \#b,\ m = s/n] \\
\sqsubseteq & m : [n \neq 0,\ m = s/n]
\end{array}
$$

Removing the abstract variables (using the laws from section 3.4.1) produces the implementation in Fig. 3.5:

```
module Calculator{
        var s:R;
        var n:N;

        initially() ≙ :[true, n = 0 ∧ s = 0];
        procedure Clear() ≙ s,n:= 0,0;
        procedure Enter(value r:R) ≙ s,n := s+r, n+1;
        procedure Mean(result m:R) ≙ m:[n ≠ 0, m = s/n]
}
```

Figure 3.5: Implementation of the calculator module.

3.4.3 Abstraction Functions and Data Type Invariants

In a data refinement, the abstraction invariant is defined as a function from the concrete data type to the abstract data type. The classic example is representing a set as a sequence where a given set can be presented by many different sequences but every sequence is associated with just one set. However, we may want to specify further properties of the concrete data type. Perhaps we want to specify that the sequence is not permitted to contain duplicates. In this way we can make the abstraction functional so that for every sequence there corresponds at most one set. The general form of functional abstraction invariants such as these is:

$$a = f(c) \wedge dti\ c$$

where the *abstraction function* $f(c)$ maps the concrete type c to the abstract type a and the *data type invariant, dti* c, specifies properties of the concrete data.

The abstraction invariant described above is both total and functional. The abstraction function is often partial, meaning that it is not defined for all input values. Abstraction invariants need not be functional either. This means that it can map its input to more than one representation. For example, a *bag* may be the abstract representation of a real number that represents the mean of a list of numbers. In this case the number 5 could have many abstract representations as bags have no ordering on their elements and they allow multiplicity of elements.

If the abstraction invariant is functional then data refinement can achieved by completing the augmentation and the diminutation sets together. Therefore, the data refinement steps become:

0. Replace all declarations of abstract variables by their concrete counterparts.

1. Refine the initialisation procedure so that the concrete variables are initialised and the abstraction invariant is maintained.
   $b : [true, post]$ becomes
   $b : [true, post[a \backslash f(c)] \ \wedge dti(c)]$.

2. Refine assignments so that $w, a := E, F$ is replaced by $w, c := E[a \backslash f(c)], G$ provided the expression G contains no abstract variable a and $dti\ c \Rightarrow F[a \backslash f(c)] = f(G)$ and $dti\ c \Rightarrow dti\ G$

3. Refine procedure specifications so that the abstraction invariant is maintained.
   $w, a : [pre,\ post]$ becomes
   $w, a, c : [pre, a = f(c) \wedge dti\ c,\ post]$ which becomes
   $w, c : [pre[a \backslash f(c)] \wedge dti\ c,\ post[a_o \backslash f(c_o),\ a \backslash f(c)] \wedge dti\ c]$

4. Refine guards so that all abstract variables are replaced by $f(c)$ and the data type invariant is maintained.

3.4.4 Data Types

A data type provides a collection of related operations for manipulating data while isolating users of the data type from the details of how the data is represented. Each data type has three parts: a *signature,* which gives the types of its operations, a *representation type,* which is the type used to represent the data and give the data

type state, and a *body,* which gives details about the operation's implementation. The key idea here is data abstraction. The software client is only concerned with the signature of the data type. They are not concerned with the representation type and the implementation of the operations. These are provided by the supplier.

Generally, we write a signature Sig and a data type D as follows:

$$Sig \triangleq \{type\ L_0; ...; type\ L_m; val\ Op_0 \in T_0; ...val\ Op_n \in T_n;\}$$
$$D \triangleq \{Sig\}\{type\ L_0 \triangleq U_0; ...type\ L_m \triangleq U_m; let\ Op_0 \triangleq u_0; ...; let\ Op_n \triangleq u_n;\}$$

```
Signature ≙ {
        type Num;
        val init ∈ Set N;
        val add ∈ Set N times N  →  Set N;
        val remove ∈ Set N times N  →  Set N;
}

Sales ≙ {Sig} {
        type Num ≙ Seq N;
        let init ≙ Num :[True, Num = {}];
        let add ≙ Num:[a ∉ Num, Num = Num ∪ {a}];
        let remove ≙ Num:[True, Num = Num /{a}];
}
```

Figure 3.6: An example of a signature used to specified a Sales type.

A specification S that uses a data type D, of signature Sig_D is said to be a Sig_D *client* (or a client, when the signature is understood). The example in Fig. 3.6 shows Sales which is a $Sig_{Signature}$ *client*. We define the term $S[Sig_D]$ to represent a client S who uses the a data type of signature Sig_D and the term $S[Sig_D.Op_i]$ to represent a client S who uses the operation Op_i, as defined in the signature Sig_D.

An abstract specification $S[Sig_A]$ can be refined by replacing data type A with a more concrete data type C. Note that the signatures (the operations available) in the specification remain. It is the underlying data representation and the implementation of the operations that changes. This requires that the operations A_j from A are replaced by corresponding operations C_j which belong to a more concrete data type C. We concentrate on the two data types, rather than the programs, so that the focus is on determining when the concrete operations correctly refine the abstract operations.

3.4.5 Simulation

Let A and C be data types which both have the signature Sig. Let $S[Sig_A]$ represent a client specification S, which uses the the signature Sig, and data type A in its specification. We say that A is data refined by C, written $A \sqsubseteq C$, if and only if $S[Sig_A] \sqsubseteq S[Sig_C]$ for all specifications S. This definition of data refinement involves a quantification over all client specifications, making proofs expensive. We use simulation as a client-independent way of establishing a data refinement. Two simulation techniques enable us to verify data refinement [41]. These are forward (downward) and backward (upward) simulations. When the data types have different representation types we must find a relation that relates the representation of one of the data types with the representation of the other.

A simulation is established between data types A and C, which have the same signature Sig, through two steps. First, for each representation type L in Sig, we define a correspondence between the two implementations of L. Secondly, for each operation Op_i in Sig, we prove that the correspondences are preserved. Therefore, there is a proof obligation corresponding to each operation of Sig.

Let signature Sig and data types A and C be defined as in Fig. 3.7.

```
Sig ≙ {
        type L_{0}; ...;
        type L_{m};
        val Op_{0} ∈ T_{0}; ...;
        val Op_{n} ∈ T_{n};
}

A ≙ {Sig} {
        type L_{0} ≙ U_{0}; ...;
        type L_{m} ≙ U_{m};
        let Op_{0} ≙ u_{0}; ...;
        let Op_{n} ≙ u_{n};
}

C ≙ {Sig} {
        type L_{0} ≙ V_{0}; ...;
        type L_{m} ≙ V_{m};
        let Op_{0} ≙ v_{0}; ...;
        let Op_{n} ≙ v_{n};
}
```

Figure 3.7: Data-types `A` and `C` with signature `Sig`.

We assume that there are no dependencies between the components i.e., no

U_i or V_i involves L_j, and no u_i or v_i involves either L_j or l_k. We establish a correspondence between A and C by a set of abstraction functions from the representation types of C to the representation types of A. We call this set of functions the *abstraction invariant*. This invariant is typically written as a backward simulation (also called L-simulation or upward simulation [44, 41]) that maps the concrete data type to the abstract data type. The simulation is written in this direction as the concrete type may offer many representations of the abstract data. If the invariant is written in the opposite direction, mapping the abstract data type to the concrete data type, then we refer to this as forward simulation.

In implementing the specification, all references (assignments, declarations, expressions) to the abstract data type are replaced with references to the concrete data type, so that the abstraction invariant is maintained throughout. If the specifications A and C, have the same signature, and AI is the abstraction invariant, then we write the simulation relation $A \sqsubseteq_{AI} C$ to mean that A is data refined by C via the abstraction invariant AI.

For example, if a specification $S[Sig_A]$, where the abstract data type A is a set, is refined by replacing the set by a list C, we write the simulation relation as $S[Sig_A] \sqsubseteq_{AI} S[Sig_C]$. We define the abstraction invariant AI so that A is defined by function f as the range of C, e.g.,

- $f(<>) \triangleq \{\}$
- $f(append(x, C)) \triangleq \{x\} \cup f\{C\}$

The symbols $<>$ and $\{\}$ represent the empty list and the empty set respectively, *append* is an operation that adds an element (in this case x) to a list and $\cup$ is an operation on sets, which returns the union of its parameters. An example is the following lists $< 1, 2, 3 >, < 1, 3, 2 >, < 2, 1, 3 >, ...$ which all correspond to the set $\{1, 2, 3\}$.

The data type C *simulates* the data type A if and only if:

- for each data type in the signature of A and C there is a correspondence defined via the abstraction invariant and
- the abstraction invariant is preserved for each operation in A and C.

This *simulation condition* must be proved for each operation in the specification's signature. We prove that the initial states of each operation correspond, and that each operation in the concrete representation behaves in the same way as the abstract representation. Hence the concrete operation succeeds whenever the abstract one does and if the abstract and concrete operations start in states equivalent states they produce the same results. By equivalent states we mean states initial states that are related through data refinement. A technique for simulation is called *sound* if the existence of a simulation between data types A and C guarantees that A is data refined by C. This is, if $A \sqsubseteq_{AI} C$ then $A \sqsubseteq C$.

3.5 Data Refinement in Object-Orientation

In an object-oriented programming language a user can define their own data types by providing class definitions. A class consists of three parts: a representation type which represents the data, method signatures which describe the allowed operations on the data, and method bodies which provide the operations implementations. The software client is only concerned with method signatures as they provide the behaviour associated with objects of the class. A class can be refined by replacing its representation type with a more concrete representation and its method bodies with implementations that are written in terms of the concrete representation. The signature of both the abstract class and the concrete class remains the same. Therefore, as far as the client is concerned, no change in behaviour will occur due to a data refinement. To verify the data refinement, we establish a simulation between the abstract and the concrete representations, establishing a correspondence between the two representation types (the abstraction invariant) and prove that this correspondence is preserved for each operations on the data.

Support for data abstraction in object-oriented programs is offered through their specification languages and verification tools. These will be discussed in detail in chapter 4. Verification of simulations typically involve transforming a program into a logical expression, corresponding to the proof obligations that must be discharged for a program to be proved correct, and passing these expressions to an underlying theorem prover or SMT solver to be discharged. The logical expression represents the weakest pre-condition of the program relative to its specification (section 2.2.1).

3.6 Conclusion

In this chapter we provided an overview of the laws for refining a specification into an executable program. While both procedural refinement and data refinement were discussed, we focus on data refinement as we are concerned with providing a framework for modular data refinement in object-oriented programming languages.

Data refinement proofs are achieved via simulation. These proofs require the definition of a mapping between the data types used in the refinement, and verification that this mapping is preserved for each operation in the specifications signature. In the chapter that follows we investigate how specification languages assist in this verification in an object-oriented environment. In particular we examine how these languages support data abstraction so that a clear separation between the abstract properties of a program's specification and the concrete properties of its implementation is achieved.

Chapter 4

Data Abstraction

Support for data abstraction in object-oriented programs is offered through specification languages and their verification tools. We introduce the main approaches and analyse how JML and Spec#, two of the more popular specification languages, support data abstraction. Our aim is to support the verification of data refinement in a behavioural interface specification language, where the client view is written completely in terms of the abstract data and the implementation is provided in terms of the concrete data. We discuss the issues that impact upon the verification of data abstractions in an object-oriented environment. As a result, we provide a suite of proposals to assist the verification of modular data refinement in an object-oriented programming environment. These proposals are listed throughout the chapter and are summarised in section 4.3.

In this chapter we assume some familiarity with the syntax of JML and Spec#. The notation used in these languages is close to the notation introduced in chapter 2. We will assist the reader with comments about the syntax where necessary but refer them to [52] for JML syntax and [14] for Spec# syntax should more information be required.

4.1 Data Abstraction

Through data abstraction we enforce a clear separation between the abstract properties of a program's specification and the concrete properties of its implementation.

The abstract properties are those that are visible to the client while the implementation is kept entirely private and can change for example, to make the implementation more efficient. These changes should not impact the client since they involve no difference in the abstract behaviour. We focus on changes from the specification to the implementation, where the core data types used in the specification are replaced in the implementation.

There are four popular approaches to data abstraction in specification languages. These are pure methods, ghost fields, model fields and logic functions. We discuss each of these with examples, focusing on pure methods and model fields as they are most relevant to our work.

4.1.0 Pure Methods

Pure methods [23, 48] are methods that have no observable side effects on a program's state. These methods are typically annotated with the keyword **pure**. An example of a pure method definition in Spec# follows:

```
[Pure] public boolean Even(int x)
requires true;
ensures result == (x % 2 == 0);
{
        return x % 2 == 0;
}
```

These methods may be used in specifications to return a value without revealing any implementation details. Note that the **modifies** clause is empty as no variables are changed by the method. For example, the *Even* method could be used in the postcondition of a method that returns the sum of all even values in the array a as follows:

```
ensures result == sum{int k in(0:a.Length); (Even(a[k])};
```

The pure methods pre-conditions and postconditions are used when reasoning with assertions that involve them. Hence implementation details are concealed from the client. However pure methods do create difficulties for verification. One issue is checking that they really are free from observable side effects. This is difficult to assess as pure methods may have side effects that are not observable to the client, e.g., creating local objects and modifying their state. A second issue is that they

are not necessarily deterministic as a pure method may return different results if called twice in succession. This is because unobserved side effects of the first call could have caused a different start state for the method execution. Solutions to this problem have been suggested in [61] where different return values are allowed but callers are prevented from assuming anything other than an equivalence between the results. A third complication is ensuring they are well-defined and do not lead to an unsound axiomatisation [25, 2, 83, 56]. Despite these complications, pure methods are a useful technique for hiding implementation details from the client and are often used in combination with ghost and model fields as we will see in further examples.

We note also that pure methods have been used to build up theories of data types in the form of model classes [20] in JML. While this expands the scope for data abstraction by allowing the use of model classes in assertions, it appears to generate difficulties for static verification as these classes must be mapped to the verifiers underlying logic [49, 26].

4.1.1 Ghost Fields

Ghost fields are specification-only variables which cannot be directly referred to in the implementation. These variables are primarily used to represent the abstract view of the specification and are sometimes used in method specifications to link a variable's pre-state to its post-state. The values which ghost fields are assigned must be pure, i.e., the evaluation of the expression which they are assigned must have no side effects. Although ghost variables cannot be referred to in the implementation, some specification languages provide special assignment statements which can be used to update their values. These assist the verification of properties which ghost variables specify. As ghost variables are not used in implementations, they are not included in the executable code that is generated for compilation.

In Fig. 4.1 the class *Swap* declares two variables and a method that swaps the values stored in those variables. The property that the sum of these two variables should remain the same throughout the class is specified by declaring a ghost variable and setting it equal to their sum. In JML, ghost fields are declared using the keyword *ghost*. The relationship between ghost fields and concrete fields is speci-

```
public class Swap{
        private /*@ spec_public @*/ int x;
        private /*@ spec_public @*/ int y;
        //@  ghost public int Sum;
        //@  public invariant Sum == x+y;

        /*@ assignable x,y,Sum;
          @ ensures Sum == x+y;
          @*/
          Swap(int a, int b){
                x = a; y = b;
                //@ set Sum = x+y;
        }

        /*@ public normal_behavior
          @ assignable x,y, Sum;
          @ ensures x== \old(y) && y == \old(x) && Sum == \old(x+y);
          @*/
          public void swapxy(){
                int t;
                t = x; x = y; y = t;
                //@ assert Sum == x+y;
        }
}
```

Figure 4.1: Using *ghost* fields in JML. JML assertions start with the characters `//@` or may be written between assertion delimiters `/*@` and `@*/`. The syntax of JML follows Java syntax closely but excludes any operators that have side effects, e.g., `++`. Other operators have the same syntax and semantics as those in Java. The keyword **assignable** proceeds a list of variables that may be modified in the frame of a method.

fied using an object invariant and must be maintained by updating the ghost fields (using *set* in JML) within the implementation where necessary.

Linking ghost fields and concrete fields in this manner originates from the theory of abstraction functions which describes the relationship between abstract and concrete fields in data refinement. In the data refinement calculus (as discussed in chapter 3), ghost fields are used to introduce concrete fields into a specification. Through the refinement process these fields become part of the implementation, replacing the abstract fields entirely. The abstract fields are then removed from the specification to generate the final implementation [7]. Rather than supporting the transformation of ghost variables into variables that are used in the implementation, specification languages support their use for the documentation of specification details for the client. Verification tools ensure that the abstraction specification is

satisfied by a given implementation. One disadvantage of using ghost variables in this way is that verification relies on the supplier to change the value of ghost variables. Their advantages include abstraction and their simple semantics.

4.1.2 Model Fields

Model fields are similar to ghost fields in that they are specification-only variables that are used to provide an abstract view of the specification. They differ in that they cannot be directly assigned values. Instead, the value of a model field is defined as a function of its corresponding concrete fields.

In program verification, this relationship between model fields and concrete fields is often referred to as its *representation*. As we have already used this term with respect to data structures in chapter 3 we will use the term "abstraction" (as used in data refinement) instead. This relationship can be in the form of an abstraction function or an abstraction relation.

The value of a model field changes as soon as the concrete fields in its abstraction are modified. Therefore, as far as the client is concerned, there is no difference between a ghost field and a model variable. As far as the supplier is concerned, the implementation is easier when a model field is used, as model fields are automatically updated when their representation changes.

As model fields provide an abstract view of the specification they, and the specifications in which they are used, can be made visible to the software client. The concrete fields and the abstraction function remain private as they reveal information about the implementation. This private information is used when verifying that the implementation satisfies its abstract specification.

Both JML and in Spec# support the use of model fields. In both languages the keyword **model** is used to identify these fields. JML provides a **represents** clause to define the abstraction function that maps the model field values to the concrete field value's while Spec# declares constraints on model field declaration that relate them to concrete values. Fig. 4.2 provides a simple example in JML, that shows how the model field *Name* is simply renamed from its public representation to the non-null string *Code* which is its implementation. This is a simple example of how model fields could be used to hide implementation details.

```
//@ public model string Name;
private /*@ non_Null @*/ string Code;
//@ private represents Name  →  Code;
```

Figure 4.2: Using **model** fields in JML.

The mapping from abstract to concrete data is not always a one-to-one mapping, as illustrated in this example. It often requires the definition of an abstraction relation which translates the concrete data representation into its abstract counterpart. An example of this is the classic data refinement example where an abstract set data type is represented as a sequence data type. As the set data type does not distinguish between sets that have their elements in a different order, there are many sequence representations of the same set. JML distinguishes between abstraction functions and abstraction relations by the notation used in the **represents** clause. It uses a $\leftarrow$ for abstraction functions and the **such_that** keyword to identify abstraction relations. See section 4.2.5 for more details. Note also that JML includes a library of modelling types that describe primitive types such as integers, characters and doubles, as well as mathematical types such as bags, sets, sequences, relations and maps. These types allow the specifier to describe abstract specifications using standard mathematical notation. Objects created from this library are immutable and pure, as they are designed for use in specifications.

A number of complications arise when verifying specifications that contain model fields. One arises from the use of partial abstraction functions, as these can lead to undefined variables. Another complication is the effect that changing a variable may have. When a variable is modified all model fields which depend on that variable are also modified, i.e., it has a potential impact on the entire upward closure [62] of this variable. A solution to this problem is proposed in [61] where Leino and Muller present an encoding in which model fields are not updated immediately when concrete fields are modified. Instead model fields are only updated when the object invariant is guaranteed to be true. The methodology guarantees that the abstraction function holds for objects when they are in a valid state. This methodology is currently being implemented in the Spec# programming system.

4.1.3 Logic Functions

Logic functions are mathematical functions and predicates, which are defined using the expression language of the specification language. These functions are non-executable and used directly by the specification logic during a verification. Languages that use logic functions include Larch [37], C programs that have been annotated for verification by the Caducceus tool [34], Java programs that have been annotated for verification using the Krakatoa tool [67], VeriCool [85] and Dafny [54]. Logic functions must be checked to ensure that their definitions are well-defined and logically consistent.

KML (Krakatoa Modelling Language) is a specification language for Java programs that is similar to JML. It is used for writing specifications for Java programs that can be verified using the Krakatoa tool suite. KML does not allow pure method definitions but it does allow the definition of logic functions, predicates and lemmas as follows:

```
//@  logic T id(T1 x1; ... ; Tn xn) = e ;
//@  predicate id(T1 x1; ... ; Tn xn) = p ;
//@  lemma id: p;
```

An example is presented in Fig. 4.3 when the logic function

```
//@  logic integer sq(integer x) = x * x;
```

is used in the specification of a method that calculates the square root of an integer.

4.2 Data Abstraction in JML and in Spec#

We present an example of data abstraction in both JML and Spec# to analyse how these languages, and the verification of programs written in these languages, support data abstraction and its verification. Both languages support the use of model fields in specifications. An example from [18] is presented in both JML and Spec# notation in Fig. 4.4 and Fig. 4.5. Using this example we explore the use of model fields in both JML and Spec#.

This example uses the model field *time* to provide an abstract view of the specification while its concrete representation is implemented in terms of three integers:

```
//@ + CheckArithOverflow = no

/*@ lemma distribute_right:
  @   \forall integer x y z; x*(y+z) == (x*y)+(x*z);@*/

/*@ lemma distribute_left:
  @   \forall integer x y z; (x+y)*z == (x*z)+(y*z);
  @*/

//@ logic integer sqrt(integer x) = x * x;

public class Isqrt {
        /*@ requires x >= 0;
          @ ensures \result >=0 && sqrt(\result) ≤ x && x < sqrt(\result + 1);
          @*/

        public static int isqrt(int x) {
                int count = 0, sum = 1;
                /*@ loop_invariant count >= 0;
                  @ loop_variant x >= sqrt(count) && sum == sqrt(count+1);
                  @ loop_variant x - sum;
                  @*/
                while (sum ≤ x) {
                        count++; sum = sum + 2*count+1;
                }
                 return count;
        }
}
```

Figure 4.3: Using Logic Functions in the Krakatoa Modelling Language (KML).

hour, *minutes* and *seconds*. We examine the notation used and discuss its implication for verification in both specification languages.

4.2.0 Using Model Fields

Both JML and Spec# annotate model fields with the keyword **model**. In JML the **represents** keyword is used to identify the abstraction function that maps the model field *time* to the concrete value given in terms of the variables *hour*, *minute* and *second*.

The model field is immediately updated when a variable in this abstraction changes. However this immediate update can cause modularity problems in specifications. For example, how can a method specification name all variables that a method will modify without revealing implementation details? It can also lead to unsoundness if an abstraction function is applied to an object whose object invariant is not satisfied. These situations must be checked during verification.

Recent advances in the treatment of model fields within the Spec# programming system have simplified the verification of programs that use model fields by restricting their updates to special statements within the language [61]. The value of each model variable is constrained using a **satisfies** clause and a boolean expression of the programming language. Rather than applying immediate updates Spec# uses a special **pack/unpack** statement that updates values of model fields whenever an object's invariant holds. This method guarantees that the abstraction function is true whenever the object invariant holds making reasoning about model fields easier to manage.

One of the main ways that JML and Spec# differ in the expression of data abstraction is in the details that are available in the client view of the specification. JML conceals the implementation details by keeping abstraction functions, concrete fields, and their invariants private as well as concealing concrete fields in data-groups in method frames. In Spec#, model fields and their satisfies clauses are publicly available (access modifiers such as private, public, and protected are not supported on model fields) so implementation details can be viewed by the client through consulting the satisfies clause.

4.2.1 Invariants

In JML, object invariants expressing properties of the model fields can be expressed publicly, e.g., $0 <= time \;\&\&\; time < 24*60*60$ while invariants which express properties of the concrete fields can be kept private, e.g., $0 <= hour \;\&\&\; hour <= 23$. In Spec# there is no distinction between private and public invariants. All invariants are available to clients of the class. However Spec# does not allow invariants to refer to model fields. Instead such constraints are added to the model field's **satisfies** clause.

4.2.2 Method Contracts

If implementation details are to be keep separate from the specification no concrete fields should appear in the method contract. JML supports this through using data groups (see section 4.2.8) which hide concrete fields from the specification. Spec#

```
public class Clock {
     //@ public model long time;
     //@ private represents time == second + (minute * 60) + (hour * 60 * 60);
     //@ public invariant time ==
                         getSecond() + getMinute() * 60 + getHour() * 60 * 60;
     //@ public invariant 0 ≤ time && time < 24 * 60 * 60;
     //@ private invariant 0 ≤ hour && hour ≤ 23;
         private int hour; //@ in time;

     //@ private invariant 0 ≤ minute && minute ≤ 59;
         private int minute; //@ in time;

     //@ private invariant 0 ≤ second && second ≤ 59;
         private int second; //@ in time;

     //@ ensures time == 12 * 60 * 60;
         public /*@ pure @*/ Clock(){hour = 12; minute = 0; second = 0; }

     //@ ensures 0 ≤ result && result ≤ 23;
         public /*@ pure @*/ int getHour(){return hour; }

     //@ ensures 0 ≤ result && result ≤ 59;
         public /*@ pure @*/ int getMinute(){return minute; }

     //@ ensures 0 ≤ result && result ≤ 59;
         public /*@ pure @*/ int getSecond(){return second; }

     /*@ requires 0 ≤ hr && hr ≤ 23 && 0 ≤ min && min ≤ 59;
       @ assignable time;
       @ ensures time == hr * 60 * 60 + min * 60;
       @*/
         public void setTime(int hr, int min){
                this.hour = hr; this.minute = min; this.second = 0;
         }

     //@ assignable time;
     //@ ensures time == \old(time + 1) % 24 * 60 * 60;
        public void tick(){
                second = second + 1;;
                if (second == 60){second = 0; minute = minute + 1;}
                if (minute == 60){minute = 0; hour = hour + 1;}
                if (hour == 24){hour = 0;}
         }
}
```

Figure 4.4: JML specification and implementation for a clock illustrating model fields, represents clauses and assignable clauses.

```
public class Clock{
        model long time{
        satisfies time ==
                second + (minute * 60) + (hour * 60 * 60) &&
                        time == getSecond() + getMinute() * 60 + getHour() * 60 * 60;
        }

        invariant 0 ≤ hour && hour ≤ 23;
        protected int hour;
        invariant 0 ≤ minute && minute ≤ 59;
        protected int minute;
        invariant 0 ≤ second && second ≤ 59;
        protected int second;

        public Clock()
        ensures time == 12 * 60 * 60;
        {
                hour = 12; minute = 0; second = 0;
        }

        [Pure] public int getHour()
        ensures 0 ≤ result && result ≤ 23;
        {
                return hour;
        }

        [Pure] public int getMinute()
        ensures 0 ≤ result && result ≤ 59;
        {
                return minute;
        }

        [Pure] public int getSecond()
        ensures 0 ≤ result && result ≤ 59;
        {
                return second;
        }

        public void setTime(int hr, int min)
        requires 0 ≤ hr &&hr ≤ 23 && 0 ≤ min && min ≤ 59;
        modifies time, hour, minute, second;
        ensures time == hr * 60 * 60 + min * 60;
        {
                this.hour = hr; this.minute = min; this.second = 0;
        }

        public void tick()
        modifies time, hour, minute, second;
        ensures time == old(time + 1) % 24 * 60 * 60;
        {
                second = second + 1;
                if (second == 60){second = 0; minute = minute + 1;}
                if (minute == 60){minute = 0; hour = hour + 1;}
                if (hour == 24){hour = 0;}
        }
}
```

Figure 4.5: Spec# specification and implementation for a clock illustrating model fields and satisfies clauses.

allows the use of both model and concrete fields in the frame conditions of a method contract.

In the following we refer to both the Spec# and the JML approaches as we discuss the issues that impact the verification of data abstractions in an object-oriented environment. In particular we provide a suite of recommendations for how best to support the verification of data refinement.

4.2.3 Providing Client and Supplier Views

While the above approaches to data abstraction provide some distinction between abstract specifications and concrete implementations, both the client and the supplier view are usually presented together in a class.

An alternative is to use an interface to provide the client view of a class, i.e., the public methods, their signatures and their specifications. The model fields used in the specification and properties of those variables, expressed as an invariant, must also be made available to the client. Traditionally no data fields or invariants would appear in an interface as these reveal implementation details of which the client should not be aware. However model fields and their invariants form part of the client contract without revealing implementation details, so they should be made available in the interface in this case. Verification tools require that the interface is implemented in order to verify that the implementation satisfies the specification. This implementation, together with the interface specification, provides the supplier's view of the software. An example showing an interface and its implementation written in JML are presented in Fig. 4.6 and Fig. 4.7. Note that the abstraction function and the concrete fields remain private in the class definition so that they are not available to the client. The semantics of implementing an interface ensures that the signature of the abstract specification and the concrete implementation stay the same.

Such a separation of client and supplier views is not as easy to achieve in Spec# as the abstraction that relates model fields to the concrete fields appears in the **satisfies** clause of the model fields. One option is to write the invariant on the model field in the specification and to strengthen the satisfies clause by adding the abstraction in the implementation, keeping this strengthening hidden from the client

(we achieve this by overriding the model variable).

Proposal 1:

Separate the client and the supplier view to ensure a modular approach to data refinement. From the client perspective, the specification is completely independent of implementation details. From the supplier perspective, the specification can be implemented by many different concrete classes, thus ensuring specification reuse.

```
public interface Coin{
        /*@ public instance model int i;
          @ public invariant 0 ≤ i && i < 2;
          @*/

        /*@ public normal_behavior
          @ requires true;
          @ assignable i;
          @ ensures i == (old(i) + 1) % 2;
          @*/

        public void flip();
}
```

Figure 4.6: An abstract specification for a coin.

```
public class Flip_Coin implements Coin{
        private boolean b;
        //@ in i;

        /*@ private represents i ← (b ? 1:0); @*/

        public void flip()
        {
                if (b) b = 0; else b = 1;
        }
}
```

Figure 4.7: A concrete implementation for coin.

4.2.4 Client View of the Concrete Specification

Even when the client and the supplier views are separated, the client's view is written in terms of the abstract data and the supplier's view is a mixture of abstract (in the form of the specification) and concrete data (in the form of the implementation).

Indeed, the abstraction function is written in terms of both as it provides the mapping from the concrete data to the abstract data. The lack of a specification, that is written entirely in terms of the concrete fields, makes the work of the supplier difficult as they are implementing the class in terms of one data type while ensuring that the implementation satisfies a specification that is written in terms of another. It also makes verification difficult as the implementation and the specification are written in terms of two different types of data. An example follows:

```
/*@ requires true;
  @ assignable i;
  @ ensures i == (old(i)+1) % 2;
  @*/
public void flip(){
        if (b) b = 0; else b = 1;
}
```

Proposal 2:

Provide a version of the specification in terms of the concrete data to assist the supplier of the implementation and verification tools. This specification should not be written as a mixture of abstract and concrete fields but entirely in terms of the concrete data. Its generation could be formalised through the application of data refinement laws as presented in section 3.4.0. Having such a concrete specification would make both the implementation and its verification easier as the specification and the implementation is written in terms of the same data .

4.2.5 Abstraction Invariants

The mapping between abstract and concrete fields can be a function where each concrete state corresponds to exactly one abstract state, or it can be a relation where each concrete state can map to any number of abstract states.

JML support both forms using the **represents** clause. The first form uses the $\leftarrow$ notation to give an explicit definition of the model fields in terms of the concrete fields through an abstraction function. The type of right-hand side of the represents clause must be compatible to the type of left-hand side by assignment. For example, the interface in Figure 4.6 could be implemented by mapping the model integer variable i to the concrete boolean variable b as follows:

```
private boolean b;
//@ in i;
//@ private represents i ← (b ? 1:0);
```

The second form uses the **such_that** notation to provide an abstraction relation between the model field and the concrete field. For example:

```
private int y;
public model int x;
//@ private represents x such_that 0 ≤ x && x ≤ y;
```

The mapping from abstract to concrete data can be supported by the definition of a method which translates the concrete data representation into its abstract counterpart. This method must be **pure** as it is used in the specification. We refer to these methods (that are used to define the abstraction function) as model methods.

The **satisfies** clause associated with a model field declaration in Spec# provides a boolean expression which constrains the values that a model variable may have. An example follows where the model field *Total* is constrained in two ways. The constraint that it must have a value that is greater than or equal to zero is equivalent to an object invariant on the abstract data in JML, while the constraint that its value is equal to the sum of the elements in the array *A*provides the abstraction function which would be expressed in a **represents** clause in JML.

```
public int[] A;
model int Total{
satisfies 0 ≤ Total && Total == sum{int i in (0: count); A[i]};
}
```

The **satisfies** clause can refer to the model field to which it belongs, other fields of the class in which the model field is declared, and fields of any sub-objects which are "owned" by the class which declares the model field. The concept of ownership is explained in section 4.2.7. A **satisfies** clause can refer to other model fields as long as dependencies are not cyclic.

Proposal 3:

*Provide separate declarations of properties of abstract fields (public invariants in JML) and their mapping to concrete fields (***represents** *clauses in JML).* This will

achieve separate client and supplier views of data refinement properties.The client should be able to view the properties of the abstract fields but not their relationship to the concrete fields.

Proposal 4:

Provide support for an abstraction invariant which maps the abstract data type that is used in the specification to the concrete data type that is used in the implementation.

4.2.6 Object Invariants

As explained in chapter 2, an object invariant is an assertion that specifies properties of the object's state. The object invariant must be established when an object is constructed and may be violated during a method execution as long as it is reestablished before that method execution terminates. Therefore, we are guaranteed that the object invariant holds when a method is executed. An example written in Spec# follows:

```
class Car{
        [SpecPublic] protected int speed;
        invariant 0 ≤ speed;
        public Car() { speed = 0; }

        public void set_Speed(int kmph)
        modifies Speed;
        requires 0 ≤ kmph;    //The invariant must hold in the pre
        ensures speed == kmph; //The invariant must hold in the post
        {
                speed = kmph;
        }
}
```

In data refinement, object invariants that are visible to the client will be defined on model data. Other invariants, visible to the supplier and for use in verification, will be specified on the concrete data. For verification purposes, the concrete invariant must hold as described above so that the implementation satisfies the concrete specification. The abstract invariant must also hold, so that the implementation satisfies the abstract specification as seen by the client. In the Spec# methodology for model fields, constraints on model fields are included in their satisfies clauses

and model fields are disallowed from object invariants (due to the need to maintain ownership invariants).

The verification of object invariants is further complicated by method re-entrancy. If we consider object invariants as assertions that must be true at a method's pre-conditions and postcondition, then it is possible that a method could call another method when the object invariant does not hold. Method re-entrancy occurs when a method call within the body of a method call backs into the object while the object is in an inconsistent state [13]. Adding the following method to the class *Car* provides an example of a method which violates the object invariant.

```
public void bad_Speed(int kmph)
modifies speed;
requires 0 ≤ kmph;
ensures speed == kmph;
{
        speed = speed-1;
        speed = kmph;
}
```

The object invariant requires that 0 <= *speed*, and the method *bad_Speed* can break the invariant when it decrements *speed*. The value of *speed* is reset to be equal to *kmph* before the method terminates. This re-establishes the object invariant. However if a method is called after decrementing *speed*, then an error could occur because the invariant 0 <= *speed* is not satisfied.

The Boogie methodology is used to verify Spec# programs. In order to control where an object invariant must hold, this methodology distinguishes between *valid* and *mutable* objects. Spec# provides **pack**/**unpack** statements in the form of **expose** blocks to identify if an object is valid or mutable. An object is in a *valid* state when it is guaranteed to satisfy its object invariant. Fields of a valid object may only be updated if the updates maintain the object invariant. An object is in a *mutable* state when it is allowed to undergo change and its invariant is allowed to be violated [38]. An object is mutable until the constructor has been fully executed. After its construction expose statements are used to temporarily change an object from valid to mutable.

In Spec# the expose statement, **expose** (**this**), puts the **this** object into a mutable state, allowing the invariant to be violated by the code enclosed by parenthesis { and }. The syntax can be seen in Fig. 4.8. The invariant may be violated within

the expose block as long as the object invariant is re-established by executing the code enclosed by the expose statement. This provides control of when an object may be modified.

```
public void bad_Speed(int kmph)
modifies speed;
requires 0 ≤ kmph;
ensures speed == kmph;
{
        expose(this){
             speed = speed-1;
               speed = kmph;
        }
}
```

Figure 4.8: Using **expose** Blocks in Spec#.

Although the use of expose blocks has no direct impact on the client of a class that is data refined, it does have an impact on the verification of a concrete specification.

Proposal 5:

Provide an object invariant which specifies properties of the abstract variables (the abstract invariant). This object invariant should be available to the client as it specifies details which are relevant to their view of the specification.

Proposal 6:

Provide an object invariant which specifies properties of the concrete fields (the concrete invariant). This object invariant should only be available to the supplier as it specifies details which are relevant to the specification's implementation. These details are of no concern to the client as the client only need be aware of the specification.

Proposal 7:

Provide verification support so that proving the concrete invariant holds implies that the abstract invariant also holds.

Proposal 8:

Distinguish between valid *and* mutable *objects, as in the Boogie methodology, and use this information to determine when it is safe to violate the object invariant.*

4.2.7 Ownership

The data fields of an object often reference other objects, which are often instances of different classes and those objects reference further objects. The combination of these sub-objects form an overall object, which we refer to as the aggregate object. The Boogie methodology handles aggregate objects by guaranteeing that if an object is *valid* (as discussed above) then that implies that its component objects are also valid. This requires some form of aliasing control which controls the use of object references. Ownership provides this control associating every object with a unique owner object. An aggregate owner is the owner of its component objects. Objects outside the aggregate are allowed to reference component objects, but these references are only of limited use as they cannot modify them with owning them.

When model fields are objects and objects are implemented by other objects in Spec#, the same discipline of ownership is needed to keep track of which objects may specify and modify properties of other objects [74]. The satisfies clauses for a model field may refer to fields of sub-objects as long as they are "owned" by the aggregate object. Note that methods that are private to the class definition may violate the class invariant at any time, as they do not have to honour the class contract.

Spec# uses the concept of ownership to keep track of objects that reference others. We say that an aggregate object owns its sub-objects. This is indicated in a Spec# specification by using the annotation [**Rep**]. The owner of this [**Rep**] field is granted permission to modify it. Fig. 4.9 shows an example where methods in the *Summation* class are granted permission to modify the array a.

We have just seen that in order to be clear about when class invariants hold, the Boogie methodology distinguishes between objects in a valid state where the object is guaranteed to satisfy its invariant and objects in a mutable state where its invariant is allowed to be violated. As the object invariant is allowed to specify properties of component objects, Spec# organises objects further into a dynami-

```
public class Summation{
        [Rep] [SpecPublic] private int []! a;
        invariant forall {int k in (0:a.Length); 0 ≤ a[k]}

        [Pure] public int SegSum(int i, int j)
        requires 0 ≤ i && i ≤ j && j ≤ a.Length;
        modifies a[*];
        ensures result == sum{int k in (i:j); a[k]};
        {
                int s = 0;
                for(int n = i; n < j; n++)
                        invariant i ≤ n && n ≤ j;
                        invariant s == sum{int k in (i: n); a[k]};
                        {
                                s += a[n];
                        }
                        return s;
        }
}
```

Figure 4.9: A Sample Spec# Class.

cally changeable ownership hierarchy. Each valid object is further categorised as *committed* (when the object's owner is in the valid phase) or *consistent* (when the object's owner is in the mutable phase). This is explained in [38] as

> "Typically, public methods are applied to objects that are consistent, and the transition to mutable takes place after entry to public methods. A committed object cannot not be mutated directly and, because it is not accessible except through its owner, it does not accept method calls; rather, it is in a phase where its state is constrained by the objects' owners' invariant."

If such constraints on object invariants are part of the specification, they must be translated into equivalent constraints at the implementation level. The categorisation of objects as committed or consistent is also taken advantage of in the modifies clause in Spec#.

Proposal 9:

In the specification of aggregate objects provide aliasing control so that every sub-object has a unique owning object, and
(a) that sub-object can only be modified through its owner

(b) the abstraction invariant and the invariant on the abstract data may refer to fields of sub-objects as long as they are "owned" by the aggregate object

4.2.8 Framing Conditions

The specification should name the abstract fields that are modified by a method. Traditionally these variables are modified when the concrete fields, on which their values depend, are modified. Rather than applying automatic updates Spec# uses a special **pack/unpack** statement that updates values of model fields whenever an object's invariant holds. As this methodology treats model fields in the same way as concrete fields that are updated by a **pack** statement (at the end of an **expose** block), the normal rule for modifies clauses in Boogie apply.

Frame conditions typically name the abstract and the concrete fields that a method may modify. However concrete fields should not be named in the client's view of the frame condition, as this will reveal details about the implementation. Furthermore, when a specification is being written it is most likely that the implementation will not be provided, hence the concrete fields are not available as they are unknown.

The three main approaches to specifying frame conditions in specification languages are:

0. Use a **modifies** clause [64] or a similar construct listing the variables that are modified by a method, i.e., a method where the modifies clause **modifies** x specifies that the method is permitted to change the variable x.

1. Use the postconditions to specify the variables that are not modified by a method, i.e., a method where the postcondition **ensures** $x == old(x)$ specifies that the method does not change the variable x. This approach is problematic as the method specifications need to know information about the environment in which the method will be called, and that information is not often available when the method is being specified.

2. Use the pre-conditions to restrict the variables a method has access to modify, e.g., in separation logic [80] if a pre-conditions specifies that the method is dependent on a memory location then the method can modify its contents.

The semantics of the language ensures that a method cannot modify a memory location that it does not have permission to modify.

Using the **modifies** clause to list the variables that are modified by a method is the approach that is used in specification languages like JML and Spec#. A number of approaches that have been suggested, allow the modifies clause represent all the variables that the procedure is permitted to modify without revealing implementation details (or having to predict concrete field names). These approaches include downward-closures, data groups, dynamic frames, regions and ownership.

Model Fields and their Downward-Closure

When a modifies clause lists a model variable it makes clients of the specification aware that the method may modify the model variable. As the model variable's value changes when the concrete fields on which it depends change, then the modifies clause must also grant permission to the method to modify those concrete fields. Leino and Nelson [62] refer to these variables as the *downward-closure* of the model variable. When the model fields are object fields and objects are implemented by other objects, the fields on which the model fields depend are influenced by the program state as well as the program text. Hence a mechanism like ownership may be useful to track object components [74].

Data-Groups

A *data-group* is used to name sets of locations in a way that does not expose representation details. When a model variable is declared in a specification a data-group of that name is automatically created. The concrete attributes, used in the implementation, are declared to be **in** that data-group so that, whenever modifications to the data-group are allowed by the specification the concrete attributes may be modified in the implementation. This technique is used in JML. It avoids exposing implementation details as the concrete fields do not need to be explicitly named in the method's frame condition. In the examples in Fig. 4.7, the concrete variable b is declared to be **in** i, the data group formed when the corresponding model variable i is declared.

An alternative to data-groups in JML is the **depends** clause which is used with the **represents** clause to indicate the variables that the represents is dependent on. In the above example we would have

```
private boolean b;
/*@ private represents i ← (b ? 1:0);
  @ depends b;
  @*/
```

Dynamic Frames

Dynamic frames allow the specification of variables that a method can modify through naming sets of locations in the modifies clause. For example VeriCool [84] specifies dynamic frames using pure methods. Dafny [54] is a language that explores the use of dynamic frames in specifications. It uses ghost variables, which must be explicitly updated to overcome issues regarding recursively defined pure methods or logic functions. Banerjee et al. [9] offers a variant on dynamic frames based on regions which are used for memory management.

Over-Approximation of a Method's Set of Modified Locations

In Spec# the modifies clause lists the fields that may be modified by a method. In addition to these variables, the Boogie methodology [13, 60] implicitly allows each method to have a net effect on the state of committed objects. To modify an object, the method body must first obtain a reference to the object and must then follow the rules to make that object mutable (the object must be a [**Rep**] object that is exposed). One advantage of this technique is that it requires only a modest set of abstraction features for use in modifies clauses. A disadvantage is that sometimes the over-approximation is too coarse.

Proposal 10:

In the client's view frame conditions should name the abstract fields that a method may modify.

Proposal 11:

Concrete fields should not be named in the client's view of the frame condition as this will reveal details about the implementation.

Proposal 12:

Naming the abstract fields in a frame condition should grant permission for the modification of concrete fields on which they depend (via the abstraction).

These three proposals are primarily concerned with the data representation that is available to the client and the supplier. The client should be aware of the abstract fields that methods can modify (Proposal 10). He should not be aware of the underlying concrete representation (Proposal 11). The supplier must be aware of the concrete data that she is allowed to modify. As the method's frame condition is written in terms of the abstract data, and the abstraction invariant relates the abstract and the concrete data, she should be allowed to modify the concrete data to which the abstract data relates (Proposal 12).

4.2.9 Inheritance

A specification written in terms of model fields can be inherited by another. To provide modular verification we must be able to guarantee that subtype methods behave according to the specifications of supertype methods i.e they are behavioural subtypes [65].

As a subtype object may be substituted where a superclass object is expected, the normal restrictions (as discussed in chapter 2) are placed on inheriting specifications. In summary:

0. Object invariants may be strengthened by the subclass as constraints are added on new attributes or on attributes inherited from its superclass.

1. Method pre-conditions may be weakened in the overriding methods.

2. Method postconditions may be strengthened in the overriding methods.

In the case of Spec# specifications **satisfies** clauses may be strengthened as these correspond to invariants that are defined on the model fields.

Behavioural Subtyping in Spec#

In Spec# the class specification and the run-time checks associated with it are inherited by the inheriting class. Spec# achieves behavioural subtyping by supporting the strengthening of object invariants and method postconditions through inheritance. However it does not allow any changes in the method's pre-conditions, as clients expect the method's specification at compile time (its static resolution) to agree with the specification at run-time (dynamic checking). Similarly, it does not allow any changes in the modifies clause, as extending the frame would be unsound and shrinking the frame can be achieved with an added postcondition that states that a variable has not changed.

An example of an inheritance relationship in Spec# is shown in Fig. 4.10. Inheritance is denoted by the symbol **:**.

```
class Car{
        [SpecPublic] protected int speed;
        invariant 0 ≤ speed;
        public Car(){
                speed = 0;
        }

        public virtual void SetSpeed(int kmph)
        requires 0 ≤ kmph;
        ensures speed == kmph ;
        {
                speed = kmph;
        }
}

public class LuxuryCar:Car{
        [SpecPublic] private int cruiseControlSettings;
        invariant cruiseControlSettings == -1 || speed == cruiseControlSettings;

        public LuxuryCar(){
                cruiseControlSettings = -1;
        }
}
```

Figure 4.10: Inheritance in Spec# where a the subclass *LuxuryCar* inherits from the superclass *Car*.

Overriding Abstractions

The JML compiler (release 5.0 rc1) permits the overriding of represents clauses in JML. JML semantics do not specify that there must be a relation between the represents clause of the overridden and overriding class. Without a behavioural subtyping constraint for represents relations, inconsistencies can be introduced easily.

In Spec# a model field's **satisfies** clause can be strengthened via a subclass. This is accomplished by repeating the declaration of the model field and providing a further satisfies clause. The **satisfies** clause then equates to the conjunction of all satisfies clauses in the inheritance hierarchy for that model variable. This permits strengthening the invariant that is specified on the model fields and strengthening abstractions as both are specified in the **satisfies** clause.

Proposal 13:

Support inheritance between abstract specifications, ensuring that the rules of behavioural subtyping are adhered to.

Proposal 14:

Support the overriding of abstractions to strengthen them in inheriting classes (this only applies to implementations as abstractions will not be present in specififcations).

4.3 Conclusion

Through our examination of how specification languages support data abstraction we have generated a suite of proposals for representing modular data refinement in object-oriented verification environments. These proposals are summarised below:

Proposal 1: (section 4.2.3)
Separate the client and the supplier view to ensure a modular approach to data refinement.

Proposal 2: (section 4.2.4)
Provide a version of the specification in terms of the concrete data to assist the supplier of the implementation and verification tools.

Proposal 3: (section 4.2.5)
*Provide separate declarations of properties of abstract fields (public invariants in JML) and their mapping to concrete fields (***represents** *clauses in JML).*

Proposal 4: (section 4.2.5)
Provide support for an abstraction invariant which maps the abstract data type that is used in the specification to the concrete data type that is used in the implementation.

Proposal 5: (section 4.2.6)
Provide an object invariant which specifies properties of the abstract fields (the abstract invariant).

Proposal 6: (section 4.2.6)
Provide an object invariant which specifies properties of the concrete fields (the concrete invariant).

Proposal 7: (section 4.2.6)
Provide verification support so that proving the concrete invariant holds implies that the abstract invariant also holds.

Proposal 8: (section 4.2.6)
Distinguish between valid *and* mutable *objects, as in the Boogie methodology, and use this information to determine when it is safe to violate the object invariant.*

Proposal 9: (section 4.2.7)
In the specification of aggregate objects provide aliasing control so that every sub-object has a unique owning object, and
(a) that sub-object can only be modified through its owner

(b) the abstraction invariant and the invariant on the abstract data may refer to fields of sub-objects as long as they are "owned" by the aggregate object

Proposal 10: (section 4.2.8)
In the client's view frame conditions should name the abstract fields that a method may modify.

Proposal 11: (section 4.2.8)
Concrete fields should not be named in the client's view of the frame condition as this will reveal details about the implementation.

Proposal 12: (section 4.2.8)
Naming the abstract fields in a frame condition should grant permission for the modification of concrete fields on which they depend (via the abstraction).

Proposal 13: (section 4.2.9)
Support inheritance between abstract specifications, ensuring that the rules of behavioural subtyping are adhered to.

Proposal 14: (section 4.2.9)
Support the overriding of abstractions to strengthen them in inheriting classes (this only applies to implementations as abstractions will not be present in specifications).

Chapter 5

Representing Data Refinement in Spec#

We propose a framework for the modular verification of data refinements in the Spec# programming system [14]. This system consists of the Spec# programming language, the compiler, and the verifier. These tools support a sound programming methodology that permits specification and reasoning about object-oriented programs.

The Spec# language is a superset of C#, an object-oriented programming language similar to languages such as Java and C++. The C# features that are most relevant to this work are single inheritance, interfaces, object references and dynamically dispatched methods. Spec# builds on C#, adding support for program specification through a rich assertion language that includes quantifiers, thus allowing the specification of objects through object invariants, field annotations, and method specifications like pre- and postconditions. A class specification and its implementation are typically written together, within one Spec# class definition. The Spec# verifier translates the compiled programs into the intermediate verification language BoogiePL [11], from which it then generates verification conditions for various SMT (Satisfiability Modulo Theories) solvers.

In languages like Spec#, specifications are typically written in terms of fields, methods, and types that are intended to be private to the implementation, with the unfortunate consequence of exposing implementation details to the client. This

coupling between the specification and the implementation requires the specification to be rewritten every time the implementation details are changed. Furthermore, assertion checking tools may incur run-time performance penalties, as some code may require execution to determine the state of variables used in the specification.

To support data refinement object-oriented languages, we propose a framework for verification that completely decouples an object's specification from its implementation. This framework maintains a client and a supplier view of the software as discussed below. The remainder of this chapter is laid out as follows. section 5.1 provides an overview of our proposed framework, presenting a one-class approach, followed by a two-class approach to data refinement. section 5.2 provides an overview of Spec# notation, highlighting the main features that we will use in our presentation. section 5.3 discusses the representation of data refinement in Spec# in more detail, providing an example of the one class approach. section 5.4 provides an argument for a two class approach, where our improved solution links the specification and its implementation through inheritance. The proof obligations for the modular correctness of object-oriented programs, and their compatibility with the proof obligations generated by a data refinement, are discussed in section 5.5. In section 5.6, we use C# properties and Spec# model fields to explore alternative data refinement representations in Spec#. Finally, in section 5.7 we evaluate our representation, revisiting our suite of proposals for data refinement representations as listed in chapter 4.

5.1 Modular Data Refinement in an Object-Oriented Language

One possible approach to supporting data refinement in an object-oriented language combines the specification and the implementation in one class definition. However, rather than using the same data-type to specify and to implement the class, the specification is written in terms of one data type, while the implementation is written in terms of another. This is the approach that we have seen throughout many of the examples that illustrate data abstraction in specifications. While this approach achieves the separation of the client and supplier views, the resulting class

is complex. It contains both the abstract and the concrete data fields, invariants on both sets of data, method specifications and implementations, and the abstraction invariant which relates the two data types. A further disadvantage is that it is difficult to distinguish between the abstract and the concrete components without thoroughly examining the class. For example, the invariant on the abstract data the invariant on the concrete data and the abstraction invariant are all conjoined into one object invariant which contains a mixture of specification and implementation data types.

Specification languages typically provide specification-only variables that can be used to represent the abstract fields, and hence restrict them from been used in the implementation. However, no such mechanism exists to restrict concrete fields from use in the specification. This means that we cannot enforce the separation of abstract specifications and concrete implementations in the one-class approach described above.

An alternative approach puts the specification and the implementation into two classes that are related via data refinement. The specification class consists of three components: the declaration of the abstract data, the object invariant which specifies properties of the abstract data, and the constructor and the method specifications written in terms of the abstract data. The implementation class is similar, in that it provides the declaration of the concrete data, the object invariant specifying properties of the concrete data, and the constructor and the method implementations written in terms of the concrete data. This approach has the advantage of clearly distinguishing the view of the client from the view of the supplier. The client view is provided by the specification class while the supplier view is provided by a combination of the specification and the implementation classes.

However, the abstraction invariant, a vital component of the data refinement, has not yet been included in our representation. Including this invariant in the specification class would reveal implementation details to the client, as it refers to both the abstract and the concrete data. The supplier already has access to both the abstract and the concrete data, so including it in the specification class does not pose any problems as it does not reveal any implementation details to the client. Any methods that may be used in the definition of the abstraction invariant may also be included, because they will not affect the client's view of the specification.

There is currently no direct support in Spec# for linking and verifying a specification and implementation written in this way. In our proposal, we go out of our way to make use of existing Spec# features as much as possible. We represent the specification using an abstract class, and the implementation using a subclass that inherits this specification. Our aim is to achieve a modular representation of data refinement, where the specification is available to the client who should not be concerned about the implementation details of a class. Any implementation module should be substitutable as long as it satisfies the abstract specification.

Abstract classes provide the declaration of data fields and the signatures of the abstract methods which sub-classes must implement. In languages like Spec#, the specification of these methods may also be added to the abstract class. Abstract classes support single inheritance. By this, we mean that a subclass will inherit from only one abstract class. Through this inheritance relationship, we provide the implementation class. This subclass provides the concrete data and the implementation of all abstract methods that are presented in the abstract class. Method overriding is used to add implementations to methods that are specified in the specification class. This has the added advantage of guaranteeing that method specifications and method implementations have the same signature.

In data refinement, the specification is provided in terms of the abstract data and the implementation is provided in terms of the concrete data. This separation of specification and implementation is not directly supported through inheritance, although we could restrict the inheritance relationship to support this (section 7). Concrete data declarations may be added to the subclass and this concrete data may be used in method implementations. Verifying that each method implementation satisfies its specification is difficult, as the specifications are written in terms of the abstract data, and the implementations are written in terms of the concrete data. The abstraction invariant which links the two data types is required, and is added as an object invariant of the implementation class.

An abstract class can be instantiated via its subclass. This provides another similarity to specification classes, as the client chooses a specification which can only be instantiated through the implementation class. We choose abstract classes to represent our specifications rather than interfaces, since abstract classes may declare fields that correspond to abstract types that are used in specifications. The other key

difference between abstract classes and interfaces is that a class may implement any number of interfaces, but may inherit from only one abstract class. This provides a 1:1 correspondence between our specifications and our implementations.

There are some drawbacks to using inheritance to represent data refinement. First, the abstract class does not require the specification of a constructor. We require the specification of a constructor to establish the properties of the abstract data (the abstract class object invariant). These properties are established through the constructor of the implementation class. A second drawback resurrects one of the flaws of the one-class approach. As both the abstract and the concrete data are available in the implementation class, the implementation can be written in terms of either, or indeed both, sets of data.

Thirdly, inheritance does not require that all methods in the abstract class are implemented by its subclass. In data refinement, we insist that all specifications are implemented. Otherwise the data refinement is not complete. On a positive note, inheritance does allow the addition of extra methods to the subclass. This coincides with our data refinement representation, as suppliers may want to add methods to assist in the implementation. Since these methods are not visible to clients, they do not influence the data refinement.

A fourth drawback is that inheritance does not support the definition of an abstraction invariant. It is expected that the subclass methods will be implemented in terms of the same data type as that used in its specification. Since this is not the case in data refinement, an abstraction invariant is required to provide the link between the abstract and the concrete data. We include it as an invariant of the implementation class so that the supplier knows how the abstract and the concrete data are related. However, this does not distinguish the abstraction invariant from other object invariants, and assertion checkers have difficulty proving that it is a valid invariant in the inheritance hierarchy.

In summary, verifying that two classes are related via inheritance does not establish that they represent a data refinement. However, investigating the similarities and inadequacies leads us to suggest enhancements to Spec# to better support the verification of data refinements. First, we present a summary of the Spec# programming language.

5.2 The Spec# Language

The C# programming language, is an object-oriented language like C++ and Java. It includes features such as single inheritance, interfaces, object references, dynamically dispatched methods, and exceptions. Spec# extends C# with features that include non-null types, method contracts and object invariants.

5.2.0 Types

The primitive types in C# include the boolean and integer types, while the composite types include arrays and object types. A class describes an object type. Its components are fields and methods, standard in object-oriented languages. Additionally, object invariants provide an assertion that must be true for all objects of the class.

In C# a field can be declared as a value type in the case of the primitive types, or as a reference type, in the case of composite types. Like Java, each reference type `T` includes the value *null*. Spec# aims to eradicate all null dereference errors by distinguishing non-null object references from possibly-null object references. Hence in Spec#, type `T!` contains references to objects of type `T` that are not null. The Spec# system will enforce this typing by returning an error if a null value is used where a non-null value is required. Like other object-oriented languages, object types and array types in Spec# respect polymorphic subtyping. By this, we mean that if type `T` is a subtype of type `S`, then an expression of type `T` can be substituted for an expression of type `S`.

5.2.1 Attributes

Similar to other object-oriented languages, C# provides attributes that attach information to classes, fields, and methods. Spec# offers additional attributes, usually enclosed in square brackets, which provide additional specification details. The Spec# attributes that are of most interest to us are listed below:

- [**SpecPublic**]: A field that is not public may be annotated with [**SpecPublic**], so that the fields may be used in the specification. This allows the client

to see the field but prevents them from modifying it unless a public method is provided for that purpose. An example is provided in Fig. 4.9 where the private array a is used in a specification.

- [**Rep**]: In Spec#, a class has permission to modify its fields if they are declared as a value type, e.g., integer fields. It does not have permission to modify fields that are declared as reference types. This is because the object or array to which this field refers to could be referred to by other (reference type) fields. Declaring a field to be [**Rep**] ensures that the object which that field references cannot be referenced by any other field. As this [**Rep**] field is the only *owner* of the referenced object it is granted permission to modify it. Figure 4.9 shows an example where methods in the *Summation* class are granted permission to modify the array *a*.

- [**Pure**]: This annotation is used to identify methods which have no observable side effects. These methods may be used in specifications, as they do not change the state of any object fields. The method *SegSum*, Fig. 4.9, provides an example of a [**Pure**] method declaration.

5.2.2 Assertions

Spec# provides an assertion language with which we can write object invariants and method specifications. This assertion language includes quantified expressions which have the form

$$Q\{\textbf{int } k \ \textbf{ in } \ E;\ F\}$$

where Q is one of the quantifiers **forall**, **exists**, **sum**, **product**, **min**, **max**, **count**, k is a bound variable of type `int` in the range of integer values provided by the expression E, and the expression F is the term to which the quantifier is applied. The expression F has the type boolean for the quantifiers **forall**, **exists** and **count**, and the type *int* for quantifiers **sum**, **product**, **min** and **max**.

The bound variable k can occur free in F, but not in E. For example, the quantified expression

$$\textbf{forall}\{\textbf{int } k \textbf{ in } (0:10); 0 <= a[k]\}$$

evaluates to **true** if the values in the array *a*, in the range *a[0], ..., a[9]* are greater than or equal to *0*.

Every method and constructor can specify a contract between its clients and its suppliers. In Spec#, this contract has three components.

- The method pre-condition, which is introduced with the **requires** keyword. It provides the initial conditions that must hold for the method to function correctly.

- The method postcondition, which is introduced with the **ensures** keyword. It specifies the state that is achieved if the method is executed in a state where its pre-condition is true.

- The frame condition, which is introduced with the **modifies** keyword. It specifies the variables that the method may modify in its implementation.

An empty pre-condition is equivalent to **requires true** as there are no initial constraints on the method. Similarly, omitting the postcondition is equivalent to **ensures true**. Omitting the frame conditions is equivalent to stating that no attributes can be modified, while **modifies this**.$*$ permits all attributes of a class to be modified by the associated method.

As in Java, in C#, the keyword **this** provides a reference to the current object. It is an implicit method parameter, instantiated to the object which calls that method. Method parameters, including **this**, are in the scope of the specification and therefore they may be used in all method assertions. Variables in postconditions can be prefixed with the keyword **old**, e.g.,$x = \mathbf{old}(x) + 1$ indicates that the new value of x is the old value incremented by 1. The postcondition of methods may refer to the return value of a method using the keyword **result**. The type of the value stored in **result** must be a subtype of the method's return type. Method bodies may have additional assertions which assist in a verification. Loop invariants are introduced with the keyword **invariant**. They help the verifier to verify method postconditions. Assertions like **assert** E state a condition E that is expected to hold whenever execution reaches that assertion. Assertions like **assume** E are similar to assert statements, but the Spec# verifier checks the condition in the assert statement, whereas it blindly assumes the condition in the **assume** statement.

Object invariants are identified by the keyword **invariant**. They define properties that apply to the class as a whole, imposing proof obligations on the constructor, and on each public method of the class. When the invariant holds we say that the object is *valid*. First, the invariants must be established by the constructor, so that the object is valid when it is created. Second, the object invariant must hold prior to the execution of every public method of the class. Otherwise methods could be called on objects that are not valid. Finally, the object invariant must be re-established after the execution of every public method of the class. This ensures that the object is in a state where other methods may be called without error.

Spec# has a construct called **expose** blocks. These explicitly identify when an object is in a state where its object invariant must be true. We discuss this in more detail in section 4.2.6 where Spec#'s methodology for reasoning with object invariants is presented.

5.3 Data Refinement: A One-Class Approach

As outlined in section 5.1, one approach to representing data refinement in Spec# is to combine the specification and the implementation in one class definition. The resulting class contains:

- the abstract data, the methods, the constructors, and the object invariant specified in terms of the abstract data;
- the concrete data, the method's implementations, the constructor's implementation and the object invariant written in terms of the concrete data; and
- an abstraction invariant relating the abstract and the concrete data.

We present an example of the one-class approach to data refinement in Fig. 5.1. A *Bag* is specified in terms of a pair of integers: *bagsum* which records the sum of the elements in the bag, and *bagcount* which records the number of elements in the bag. The bag is implemented in terms of an integer array *elems*, which stores the values in the bag, and integer *count*, which records the number of elements that it contains.

```
public class Bag {
        [SpecPublic] private int bagsum;
        [SpecPublic] private int bagcount;
        [Rep] private int[] elems;
        private int count;

        invariant 0 ≤ bagcount;
        invariant 0 ≤ count && count ≤ elems.Length;
        invariant bagcount == elems.Length && bagsum == absSum();

        Bag()
        ensures bagcount == 0 && bagsum == 0;
        {
                this.count = 0;
                elems = new int [100];
        }

        [Pure] public bool isEmpty() //unsatisfied  postcondition - needs AI
        ensures result == (bagcount == 0);
        {
                if (count == 0) return true;
                else return false;
        }

        void add(int x) //unsatisfied  postcondition - needs AI
        requires 0 ≤ x;
        modifies this.*;
        ensures bagSum == old(bagsum) + x && bagcount == old(bagcount) + 1;
        {
                expose(this){
                        if (count == elems.Length){
                                int[]! b = new int[2*elems.Length+1];
                                Array.Copy(elems, 0, b, 0, elems.Length);
                                elems = b;
                        }
                        elems[this.count] = x;
                        count++;
                }
        }

        [Pure] public int absSum()
        ensures result == sum{int i  in (0:elems.Length); elems[i]};
        {
                int s = 0;
                for(int j = 0; j < elems.Length; j ++)
                        invariant j ≤ elems.Length;
                        invariant s == sum{int i in (0:j); elems[i]};
                {
                        s = s+elems[j];
                }
                return s;
        }
}
```

Figure 5.1: The One-Class approach to representing data refinement in Spec#. Both abstract and concrete components are present in the class with the bag represented as a pair of integers in the specification and as an array in the implementation.

The abstract data fields can be used in the specification, as indicated by the [**SpecPublic**] annotation, while the concrete data fields remain private to the implementation. The methods defined for use with objects of the class are:

- the constructor *Bag()* which initialises the class attributes
- *[Pure] public bool isEmpty()* which determines if the bag is empty
- *void add(int x)* which adds x to the bag

The invariants that are defined in the class are:

```
invariant 0 ≤ bagcount
invariant 0 ≤ count && count ≤ elems.Length
invariant bagcount == elems.Length && bagsum == absSum()
```

which describes the properties of the abstract data, the properties of the concrete data, and the abstraction invariant respectively. In keeping with the Boogie methodology (as discussed in section 4.2.6), all three invariants will be checked whenever Bag objects are in a valid state. Without reference to the abstract data in the implementation, the verification of the invariant on the abstract data, and the invariant which provides the abstraction invariant, is not possible as there is no code to establish these invariants.

Note that the method *[Pure] public int absSum()* is an abstraction function which maps the concrete data to its abstract counterpart. This method should be kept private from the client as it exposes details about the *Bag* implementation. However, in this one-class approach, we are forced to make the method public, as Spec# does not allow private methods to be used in invariants and does not allow methods to be annotated with [**SpecPublic**].

Using Spec# model fields to identify the abstract data does not significantly improve the one class representation of data refinement. Slight improvements include identification of the abstract data fields using the *model* keyword and using the **satisfies** clauses to provide the invariants for the abstract data. However, both the abstract and the concrete data are present in the class and hence a modular view of data refinement is not provided. Other restrictions, such as the lack of support for method calls in the **satisfies** clause, puts further limits on the representation.

5.4 Motivation for a Two-Class Approach

The one-class approach combines the specification and the implementation in one class definition in Spec#. It differs from the normal Spec# approach in that the specification is written entirely in terms of one data type, and the implementation in terms of another. Although this separates the specification from the implementation, in terms of the data types used, the resulting class is cluttered due to its abstract and concrete components (section 5.1). Furthermore, a client can discover implementation details, since all invariants and abstraction functions are publicly available in the class.

Separating the specification and its implementation into two separate classes helps to solve these problems, because the client is presented with the specification while keeping the implementation details private.

The example of the implementation of the bag from Fig. 5.1 is presented, using the two-class approach in Fig. 5.2 and Fig. 5.3. This separation of the class specification from its implementation provides two views of the class. These are the client view provided by the class specification, and the supplier view which is provided by the implementation.

```
public class Bag{
        [SpecPublic] private int bagsum;
        [SpecPublic] private int bagcount;
        invariant 0 ≤ bagcount;  //(object  invariant)

        Bag()
        ensures bagcount == 0;
        ensures bagsum == 0;

        [Pure] public bool isEmpty()
        ensures result == (bagcount == 0);

        public void add(int x)
        requires 0 ≤ x;
        modifies bagsum, bagcount;
        ensures bagsum == old(bagsum) + x ;
        ensures bagcount == old(bagcount) + 1;
}
```

Figure 5.2: A specification class in the Two-Class Approach: A bag is specified in terms of two integer variables *BagSum* and *BagCount*, recording respectively, the sum of the elements in the bag, and the number of elements in the bag.

```
public class Bag{
        [Rep] private int []! elems;
        private int count;
        invariant 0 ≤ count && count ≤ elems.Length;

        public Bag(){
                this.count = 0
                this.elems = new int [100];
        }

        [Pure] public bool isEmpty(){
                if (count == 0) return true;
                else return false;
        }
        public void add(int x){
                expose (this){
                        if (count == elems.Length){
                                int []! b = new int[2*elems.Length+1];
                                Array.Copy(elems, 0, b, 0, elems.Length);
                                elems = b;
                        }
                        elems[this.count] = x;
                        count++;
                }
        }
}
```

Figure 5.3: An implementation class in the Two-Class Approach: A Spec# implementation of a bag, written in terms of an integer array *elems* which stores the bag contents and an integer *count* which records the number of elements in the bag.

We represent the two-class approach in Spec# using inheritance. This ensures that the implementation class is a subtype of the specification class and hence, its objects may be used anywhere that objects which satisfy the specification class are expected. This meets the expectation of data refinement, where a concrete class may be substituted anywhere that its abstract specification is expected.

The superclass is a specification class containing the abstract data fields, object invariant, and method specifications. All its class-level and method-level assertions are written in terms of abstract data, and therefore are independent of the object's implementation details. The subclass is the implementation class containing the concrete data, object invariant, and the method implementations. It inherits the method specifications and in doing so allows the Spec# tools to check that the implementation satisfies its specification. This verification is possible due to the abstraction invariant which links the abstract and the concrete data.

As the abstraction invariant would reveal implementation details to the client if it were visible in the specification, we record the abstraction invariant and any abstraction functions required for its definition in the subclass. This is permitted in an inheritance relation, since subclasses may add both fields and methods to a class.

5.5 Behavioural Subtyping

We now review how behavioural subtyping is supported in Spec#, before continuing our representation of data refinement. In Spec#, the class specification and the run-time checks associated with it, are inherited by the inheriting class. Spec# achieves behavioural subtyping by supporting the strengthening of object invariants and method postconditions through inheritance. However, this does not allow any changes in the method's pre-conditions as clients expect the method's specification at compile time (its static resolution) to agree with the specification at run-time (dynamic checking). Similarly, it does not allow any changes in the modifies clause, as extending the frame would be unsound, and shrinking the frame can be achieved with an added postcondition that states that a variable has not changed. Other method verification issues that arise from inheritance include:

Method Callbacks

A mechanism to prevent errors due to method callbacks in Spec# programs is presented in section 4.2.6, where **expose** blocks are used to control when an object is *valid* and when it is *mutable*. The invariant may be violated when the object is mutable, and must be satisfied when the object is valid.

Superclasses Breaking Subclass Invariants

Even when superclasses are modularly verified their impact on future subclasses is unknown. For example, object invariants may refer to fields that are defined in the superclass. However, superclass methods have permission to modify these, and in doing so may violate the invariant defined the subclass.

Spec# addresses this by restricting object invariants when they refer to superclass fields. Only fields annotated with [**Additive**] in the superclass, may be referred to in object invariants defined in their subclass[58]. There is a difficulty with this, as when the superclass is being defined, the invariants of its subclasses are unknown. Therefore, the supplier needs to predict the fields that should be annotated with [**Additive**].

Annotating a field as [**Additive**] means that subclasses are allowed to add further constraints on the value of the field. Subclass methods that are allowed to modify superclass fields must also be annotated with [**Additive**]. Figure 5.4 provides an example of using the [**Additive**] annotation where the *SetSpeed* method modifies the superclass field *speed*.

Overriding Methods Can Break Contracts

The Spec# methodology ensures that overriding methods do not break contracts by checking that postconditions and object invariants are strengthened, and that no changes occur in pre-conditions or frame conditions when a method is inherited.

Furthermore, an [**Additive**] **expose** is required on all methods in the superclass that are inherited. This is because the subclass must be exposed if this method is called on a subclass object.

```
public class Car{
        [Additive] [SpecPublic] protected int speed;
        invariant 0 ≤ speed;
        public Car(){
                speed = 0;
        }
        [Additive] public virtual void SetSpeed(int kmph)
        requires 0 ≤ kmph;
        ensures speed == kmph;
        {
                additive expose(this){
                        speed = kmph;
                }
        }
}
public class LuxuryCar:Car{
        [SpecPublic] private int cruiseControlSettings;
        invariant cruiseControlSettings == -1 || speed == cruiseControlSettings;
        public LuxuryCar(){
                cruiseControlSettings = -1;
        }
}
```

Figure 5.4: Car with Additive Expose blocks.

Overriding Methods and Modifies Clauses

Overriding methods should maintain the specification of the base method. If we were to list the fields that an overriding method can modify, we would need fore-sight regarding future fields that could be added to the class. If an overriding method modifies some field other than those listed in the modifies clause of its superclass methods, it will violate the original method specification.

5.6 Data Refinement: A Two-Class Approach

The support that Spec# provides for behavioural subtyping assists our two class representation of data refinement. In a data refinement, the client will request an object that satisfies the specification and will not be concerned with the implementation that they actually receive. Therefore, the implementing class must be a subtype of the specification class as all of the methods that are available in a specification must also be available in the implementation. As no code will be supplied in the specification class, there will be no need for expose blocks or additive expose blocks. However, we will include these in the implementing subclass for verification

purposes. The specification class contains data fields and method specifications, so our representation of data refinement is assisted by: using [**Additive**] to allow data fields to be referred to in invariants of subclasses; support for strengthening postconditions and invariants; and the maintenance of modifies clauses from specification methods.

5.6.0 Representing Data Refinement using Inheritance

We present the specification as an abstract class, since no implementation details are required. In Spec#, a class is described as *abstract* if it contains one or more abstract methods. An abstract class is annotated with the keyword **abstract** and its fields are usually declared as **protected** allowing them to be inherited by the subclass. Methods are publicly available to clients of the class as indicated by the keyword **public**. They are also annotated with **abstract** to indicate that only their specification needs to be supplied. [**Pure**] methods which are included to assist in writing the specification must also be publicly declared so that they can be referred to in the specification.

We represent the implementation class as a subclass that inherits the protected fields from the abstract class, provides additional concrete fields, and overrides the methods from the superclass providing their implementations. For simplicity, we assume that the fields declared in a class are distinct from other fields declared in the class or any class that it is associated with. Object invariants and method postconditions may be strengthened as outlined in section 5.5.

The Bag specification and implementation classes, represented as an abstract class and its implementation, are presented in Fig. 5.5 and Fig. 5.6. Note that the abstract class requires its data fields to be [**SpecPublic**] and [**Additive**] for reasons outlined above. Note also that no verification can be applied to the abstract class, as there is no implementation to check the specifications against. All assertion checking will be done in the subclass, where the method implementations are provided.

The implementation class presented in Fig. 5.6 provides both the invariant on the concrete data ($0 <= count \;\&\&\; count <= elems.Length$) and the abstraction invariant($bagcount == elems.Length \;\&\&\; bagsum == absSum()$). Note that they are not easily distinguished in the class definition, as they are both introduced us-

```
public abstract class Bag{
        [SpecPublic] [Additive] private int bagsum;
        [SpecPublic] [Additive] private int bagcount;
        invariant 0 ≤ bagcount;
        invariant 0 ≤ bagsum;

        Bag()
        ensures bagcount == 0;
        ensures bagsum == 0;

        [Pure] public bool isEmpty()
        ensures result == (bagcount == 0);

        public void add(int x)
        requires 0 ≤ x;
        modifies bagsum, bagcount;
        ensures bagsum == old(bagsum) + x ;
        ensures bagcount == old(bagcount) + 1;
}
```

Figure 5.5: An abstract class used to represent a specification class in the Two-Class Approach. Attributes are annotated as [**Additive**] since the abstraction invariant in the implementing class in Fig. 5.6 refers to them.

ing the keyword **invariant.** In fact, we could have written them as a conjunction, combining them into one object invariant. The abstraction invariant reveals implementation details and hence is more suitable for inclusion in the implementation, rather than in the specification. Likewise, the abstraction method $absSum()$ is specified and implemented in the implementation class. It reveals implementation details as it is used to define the abstraction invariant. Other methods inherit their specification from the abstract class while their implementation is provided here.

It is worthwhile drawing the readers attention to the constructor. The constructor has the responsibility of establishing the object invariant, which consists of the invariant on the concrete data and the abstraction invariant as mentioned above. It also has the responsibility of establishing the invariant on the abstract data, as this invariant is inherited from the specification class. The code provided in the constructor implementation only initialises the concrete data so the only invariant that is established is the invariant on the concrete data. The verification tools do not automatically know to use the link between the abstract and the concrete data, to verify the invariant on the abstract data. Rather than taking this abstraction invariant as an axiom, the tools try to verify the mapping as it is stated as a class invariant. As a result, the Spec# verification tools cannot verify the classes pro-

vided in our example. We solve this problem through the use of C# properties and Spec# model fields in sections 5.6.1 and 5.6.2 respectively.

Client and Supplier Views

This two-class approach provides separate views of the data refinement for the client and the supplier. As is desirable, there is no evidence of the specification in the implementation class (see Fig. 5.6). Due to the inheritance relationship, tools that enforce behavioural subtyping will try to verify that the implementation satisfies the abstract specification. This is what we require in a data refinement. However, in our representation, the only relationship specified between the abstract and concrete classes is a boolean expression that is the object invariant of the implementing class. Providing the abstraction invariant simply as an object invariant is not sufficient. The Spec# verifier tries to prove that this abstraction invariant (and the invariants on the abstract and concrete data) is established by the constructor and maintained by all methods. These verifications are not possible without the use of model or ghost fields which support the update of abstract fields in the specification when the concrete fields on which they depend are modified.

Extra specifications, written in terms of the concrete data, could be added to the implementation to provide a specification in terms of the concrete data. This would assist the supplier, as the specification would be written in terms of the data that is used in the implementation. It also assists the verification tools, since the proof that the implementation meets its specification is easier if both are written in terms of the same data. However, because we need to support behavioural subtyping, there are limitations to the specifications that we can add. Spec# permits strengthening object invariants and method postconditions in subclasses, but does not allow any changes in a method's pre-condition. Similarly, it does not allow any changes in the modifies clause, as extending the frame would be unsound as it would allow the method to establish its postcondition in more states than those permitted for the same method in the parent class. Shrinking the frame is achieved with an added postcondition that states that a variable has not changed. This means that both abstract and concrete fields that a method implementation can modify can be listed in the abstract specification. Likewise, the pre-conditions of methods

```
public class BagConcrete:Bag{
        [Rep] private int []! elems;
        private int count;
        invariant 0 ≤ count && count ≤ elems.Length;

        //abstraction invariant
        invariant bagcount == elems.Length && bagsum == absSum();

        public BagConcrete(){
                this.count = 0
                this.elems = new int[100];
        }
        [Pure] public bool isEmpty(){
                if (count == 0) return true;
                else return false;
        }
        public void add(int x){
                expose (this) {
                        if (count == elems.Length) {
                                int[]! b = new int [2*elems.Length+1];
                                Array.Copy(elems, 0, b, 0, elems.Length);
                                elems = b;
                        }
                        elems[this.count] = x;
                        count++;
                }
        }
        [Pure] public int absSum()
        ensures result == sum{int i in (0:elems.Length); elems[i]};
        {
                int s = 0;
                for(int j = 0; j < elems.Length; j ++)
                invariant j ≤ elems.Length && s ==
sum{int i in (0:j); elems[i]};
                {
                        s = s+elems[j];
                }
                return s;
        }
```

Figure 5.6: An implementation class in the Two-Class Approach where the Bag Implementation class inherits from the abstract Bag specification from Fig. 5.5.

in the abstract specification specifies the pre-conditions in terms of the concrete data. As a result, the abstract specification exposes implementation details, mixes the client and the supplier's view of the data, and requires the supplier to know the implementation that they were going use at the time that they are writing the specification. It also creates a dependency between the specification and a particular concrete implementation. This is not our intention, as a specification should be independent of its implementation. It should also be possible to provide many different implementations for one specification.

Relating Abstract and Concrete Data

We now discuss the relationship between the abstract and the concrete data in a data refinement. The first challenge concerns how to state the abstraction invariant. As inheritance allows object invariants to be strengthened and provides access to both the abstract and the concrete data, we added the abstraction invariant to the implementation class. Inheritance also permits new methods to be added, allowing abstraction functions that are required for the abstraction invariant's definition to be added, without making them visible in the class specification. One disadvantage off including the abstraction invariant here is that there is no distinction between the concrete object invariant and the abstraction invariant, since both appear as invariants of the subclass. In Spec#, this also generates problems for the verifier as the abstraction invariant needs to refer to fields that are defined in the superclass. Superclass methods have permission to modify these, and in doing so, may violate the abstraction invariant. Hence Spec# insists that we annotate the superclass fields with [**Additive**] so that the subclass is allowed to add further constraints on the value of the field.

A second challenge is establishing the object invariants. The constructor has the responsibility of establishing the object invariant when an object is created. In a data refinement, we never need to initialise the abstract data fields as they are not used in the class implementation. However, we do need to specify the initial values that these abstract fields should have in order to establish the abstract object invariant. This is achieved by specifying the constructor. The implementation class also requires a constructor. This constructor must establish the concrete invariant

by initialising the concrete data fields.

However, constructors that are declared within a superclass are not directly inherited by subclasses (although they can be called from within the subclass constructor). Therefore, the implementation class does not inherit the constructor specification from its superclass. Hence, the subclass constructor has no knowledge of the initialisations that are necessary to establish the invariant on the abstract data or the abstraction invariant.

We discuss the constructor implementation further in section 5.6.4. First, we discuss the use of C# properties and Spec# model fields to specify the invariants on the abstract and concrete data. These approaches also allow the elimination of the abstract data from the implementation, since abstract data is overridden with their concrete counterparts. An added advantage is that this override eliminates the necessity to use the [**Additive**] annotation on superclass fields. These model fields are overridden and hence no longer exist in their abstract form, in the subclass. The abstraction invariant will therefore be expressed as a combination of C# properties or Spec# model fields constraints that must be satisfied as the abstract data is overridden.

5.6.1 Using C# Properties to Specify and Implement Data Refinement

A C# property declares a private field and may declare methods which read, write, or compute the values of that field. They can be used as though they are public data members, but they are actually methods (called accessors). This enables data to be accessed easily while still providing the safety and flexibility of methods.

The only method that we require for an abstract data field is a method that reads its value. This method is declared as part of the property declaration using the keyword *get*. Property *get* methods are pure by default and hence may be used in specifications. Its specification, written as an **ensures** clause, corresponds to an invariant on that abstract data field.

Overridden versions of the property provide its concrete implementation. The value read in its overridden *get* method is the concrete representation of the abstract field. As the method is overridden, its postconditions can be strengthened and, in

doing so, the abstraction invariant that relates the abstract and concrete fields is provided in conjunction with the specification for the abstract field.

An example follows, where the property *Age* declares an abstract field which has the invariant property that *0 <= Age*. This is expressed by the property definition as follows:

```
public abstract int Age{
        get;
        ensures 0 ≤ result;
}
```

The sample implementation below, replaces the abstract field *Age* with the concrete field *count* by overriding the property so that the concrete field is read rather than the abstract field. The invariant relating the abstract and the concrete data is provided via the postconditions of the overriding property which yields the abstraction invariant *Age == count*.

```
private int count;
public override int Age{
        get
        ensures result == count;
        {
                return count;
        }
}
```

A larger example is shown in Fig. 5.7 and Fig. 5.8, where a Bag is specified as a pair of integers and implemented in terms of an array. The invariant on the abstract data is provided through the specifications of C# properties called *BagSum* and *BagCount*.

```
public abstract int BagSum{
        get
        ensures 0 ≤ result;
}
public abstract int BagCount{
        get
        ensures 0 ≤ result;
}
```

These properties express that both *BagSum* and *BagCount* are greater or equal to zero. The concrete version of these properties strengthen their postconditions via inheritance:

```
public override int BagSum{
        get
        ensures result == sum{int i in (0:count); elems[i]};
        {
                //Add implementation here
        }
}
public override int BagCount {
        get
        ensures result == count;
        {
                //Add implementation here
        }
}
```

The result is:

- the postconditions *result == sum{int i in (0: count); elems[i]}* is conjoined to the postconditions of the abstract *BagSum*; and
- the postconditions *result == count* is conjoined to the postconditions of the abstract *BagCount*.

It is through method overriding that we get the combined postconditions which allow us to deduce the following invariant on the concrete data:

```
(0 ≤ BagCount && BagCount == count) ⟹ 0 ≤ count &&
(0 ≤ BagSum && BagSum ==  sum{int i in (0: count); elems[i]})
                        ⟹ 0 ≤ sum {int i in (0:count); elems[i]}
```

Note that postconditions on the concrete fields are expressed as normal in the implementation class, and it is they that supply the abstraction invariant:

```
BagCount == count && BagSum == sum{int i in (0: count); elems[i]};
```

It is possible to write object invariants in an abstract class but it is not necessary in our example, because all invariants are expressed through the postconditions of

```
public abstract class BagAbstract{
        public abstract int BagSum{
                get;
                ensures 0 ≤ result;
        }
        public abstract int BagCount{
                get;
                ensures 0 ≤ result;
        }
        [Pure] public abstract bool isEmpty {
                get;
                ensures result == (BagCount == 0);
        }
        public abstract void add(int x);
        requires 0 ≤ x;
        modifies this.*;
        ensures BagSum == old(BagSum) + x;
        ensures BagCount == old(BagCount) + 1;
}
```

Figure 5.7: Specification of a Bag written as an abstract class using properties to specify the abstract and concrete data and their abstraction invariant.

```
public class BagConcrete:BagAbstract{

        [Rep][SpecPublic] private int []! elems;
        [SpecPublic] private int count;
        invariant 0 ≤ count && count ≤ elems.Length;
        invariant forall{int i in (0:count); 0 ≤ elems[i]}
        invariant 0 ≤ sum{int i in (0: count);elems[i]};

        public override int BagCount {
                get
                        ensures result == count;
                {
                        return count;
                }
        }
        public override int BagSum{
                get
                ensures result == sum{int i in (0: count);elems[i]};
                {
                        int s = 0;
                        for(int j = 0; j < count; j ++)
                        invariant j ≤ count;
                        invariant s == sum{int i in (0:j);elems[i]};
                        invariant 0 ≤ s;
                        {
                                s = s+elems[j];
                        }
                        return s;
                }
        }
}
```

Figure 5.8: An implementation of the bag specification using properties *BagCount* and *BagSum* from Fig. 5.7.

the property *get* methods. However, expressing a relationship between properties, such as

```
invariant BagSum ≤ BagCount;
```

is not currently supported in Spec#. (It causes a verification error, as the properties *BagSum* and *BagCount* are not "peer valid" and hence the invariant cannot be verified). This is a limitation of using properties. Using normal data fields rather than properties, overcomes this problem and allows invariants to express relationships between abstract fields.

However, the advantage of using properties is that we can override the abstract data, thus simplifying the specification, because an invariant that expresses properties of the abstract data is not necessary, and the abstraction invariant is inferred from the postconditions of overridden property methods.

5.6.2 Using Spec# Model Fields to Specify and Implement Data Refinement

Another option that allows us to support data refinement in Spec# is the use of model fields. Model fields, as discussed in chapter 4, are specification-only fields whose values are determined by mappings from an object's concrete state.

Rather than defining the abstract fields as properties, we define them as model fields which satisfy given assertions. This approach also offers the option of overriding the abstract data in the inheriting class. We illustrate by changing the example in Fig. 5.7 to specify its data fields using model fields rather than properties. The model fields representing *BagCount* and *BagSum* are specified as:

```
model int BagCount{
    satisfies 0 ≤ BagCount;
}
model int BagSum{
    satisfies 0 ≤ BagSum;
}
```

These model fields may be overridden in the subclass as follows:

```
override model int BagCount{
    satisfies BagCount == count;
}
override model int BagSum{
    satisfies BagSum == sum{int i in (0:count);elems[i]};
}
```

Note that the abstraction invariant is specified directly when we use model fields, rather than being implied from the postconditions of property *Get* methods in our previous representation. Another advantage of using model fields rather than properties is that model field values are updated when the concrete fields on which they depend are updated. As discussed in section 4.1.2, the methodology used in the Spec# verifier has been recently extended to update these model fields when the object invariant is known to be true.

Overriding model fields used in our example causes compilation errors in Spec#. These errors occur as the array *elems* and the variable *count* are "not admissible". Model fields are specification fields and our overrides relate them directly to variables from their concrete subclass. This is generally not allowed and hence causes (admissibility checking in) the Spec# compiler to report an error. When admissibility checking is turned off (using the command line switch /cc-), overriding model fields will not cause any errors.

Turning off admissibility checking is generally not desirable as it allows programs to be compiled when undesirable field access is present. However, in this case, the error reported concerns a field access that we want to allow so that we can express the relationship between the (abstract) model fields and their concrete representation (in the subclass). Overriding a model fields in the subclass so that its $satisfies$ clause relates the abstract fields and the concrete representation is how we express the abstraction invariant. Even with admissibility checking turned on, the unpermitted field access (according to the Spec# compiler) is in the concrete class, where, in a data refinement, both abstract and concrete variables can be referred. Hence, this is not an error that effects our representation of data refinement.

5.6.3 Specifying and Implementing Methods

Method specifications, in the form of its signature and contract, are provided in the **abstract** class as illustrated in Fig. 5.7. In general, an abstract class is not restricted to containing only abstract methods. However, we enforce this restriction, as a specification that is data refined will have no implementation details available.

We also restrict the subclass public methods to contain methods with the same signature as those in the superclass. All of these methods must be overridden in the subclass to provide their concrete implementations. Our choice of using C# properties or Spec# model fields to represent the abstract and the concrete data does not affect the specification and implementation of methods.

We implement methods using the inheritance mechanism provided in Spec#, with methods in the concrete class overriding their abstract equivalents. Through inheritance, a subclass may add extra methods that are not in the superclass. This does not cause any problems for representing data refinement using inheritance. Suppliers may add extra methods to assist with the implementation. As the client only sees the specification (superclass), they will never be aware of their existence.

Methods that do not change the object's state are prefixed with the keyword [**Pure**], and may be invoked in assertions such as pre-conditions, postconditions and invariants as they have no side effects. Methods of C# properties are pure by default. The methods *isEmpty* and *add* in Fig. 5.7 are implemented as follows, using the data refinement from Fig. 5.8:

```
[Pure] public override bool isEmpty{
    get{
        return count == 0;
    }
}
public override void add(int x){
    expose(this){
        if (count == elems.Length){
            int[]! b = new int[2*elems.Length+1];
            Array.Copy(elems, 0, b, 0, elems.Length);
            elems = b;
        }
        elems[this.count] = x;
        count++;
    }
}
```

The use of an **expose** block explicitly indicates when the object's invariant may be temporarily broken, so that method callback cannot occur as discussed in chapter 4. It is also used to identify where methods may modify sub-objects. In this case the array *elems* is a sub-object which is owned by the aggregate object of type `Bag` (as identified by the [**Rep**] field in Fig. 5.7. This subclass object is only mutable when its owner is mutable.

```
public class Super{
    int state;
    public void set(int v)
    modifies state;
    {
        state = v;
    }
}

public class Sub:Super{
    int oldState;
    public void set(int v){
        oldState = state;
        super.set(v);
    }
}
```

Figure 5.9: Overriding and **modifies** clauses.

5.6.4 Specifying and Implementing Constructors

As discussed in section 5.6.0, an abstract class cannot be instantiated directly. It can only be instantiated as a derived class. In a data refinement, we want to maintain the client's view that they create an instance of the specification when they create an object. The underlying implementation should be hidden from them, although the client will need to identify the implementation which they choose. We propose the use of a factory method to construct the concrete object.

To illustrate this, we continue the Bag example of Fig. 5.7. We provide the concrete class constructor *BagConcrete()* which initialises the fields of the concrete object:

```
protected BagConcrete()
ensures count == 0;
{
    count = 0;
    elems = new int[100];
}
```

We also provide a factory method *Make()*, which returns a concrete object masquerading as an abstract object.

```
public static BagAbstract!Make()
ensures result.IsNew;
ensures result.BagCount == 0;
ensures result.BagSum == 0;
{
    BagConcrete b = new BagConcrete();
    return b;
}
```

This is permitted as, due to subsumption, a subtype may be returned where a supertype is expected. In a data refinement, both of these methods will be provided in the implementation class. Every implementation class must provide a *Make()* method whose return type cannot be a null reference (note the ! annotation).

This method will return an object which satisfies the abstract specification. As far as a client is concerned, the object returned is an instance of the specification class. The object that is actually returned is an instance of the implementation class. Since the implementation class is a subtype of the specification class, this object can be substituted wherever an instance of the specification class is expected. The client chooses the concrete implementation that they want to use when they create an object in *main()*. In the case of the Bag example above, the client would write

```
BagAbstract b = BagConcrete.Make();
```

where *BagAbstract* is the name of the specification, and *BagConcrete* is the name of the chosen implementation.

5.7 Evaluation of the Two-Class Approach to Data Refinement in Spec#

Our two class approach to data refinement uses inheritance to link the abstract specification to its implementations, with C# properties or Spec# model fields representing the abstract data. We now summarise our findings with respect to the suite of proposals that we presented in chapter 4.

Proposal 1: *Separate the client and the supplier view to ensure a modular approach to data refinement.*
Evaluation: This separation is achieved via inheritance. Abstract classes provide the client view while their subclasses provide the supplier view.
Analysis: Abstract classes must be restricted, so that they do not contain method implementations (which traditionally they can). In addition, every method must be provided with an implementation in the implementing class. The client will call methods that are available in the specification, using the signature of the specification class. The actual method that is executed will be the method of the same name from the implementation class. When the method that the client calls has a parameter of type **this**, the type of the actual object that will be passed to the method will be a subtype of **this**. An error does not occur here, as the implementation object type is a subtype of the specification object type. Similarly, if a client expects to have a specification type object returned from a method call, but a subclass (implementation type) object is returned, no type error occurs.

When extra methods are provided in the implementation (for example, to assist the supplier) the client will not be aware of them, because they are not available in the public specification. When the client calls the constructor, a special $Make()$ method in the implementation class returns a subclass object (which masquerades as a superclass object). Suppliers must be aware of the inherited variables that are specification-only variables, and hence which should not be available for use in the implementation. In Spec#, these variables must be annotated with [**Additive**], so that the abstraction invariant in the implementing class can refer to them. Alternatively, specification-only fields are achieved by using model fields to declare the abstract data. The value of these fields change when the concrete fields on which they depend, change in the implementation.

Proposal 2: *Provide a version of the specification in terms of the concrete data to assist the supplier of the implementation and verification tools.*
Evaluation: This is partially achieved by our representation, through the presence of concrete data invariants in the supplier view. However, method specifications remain written in terms of the abstract data.
Analysis: Method specifications are inherited in our representation of data refine-

ment. This means that the suppliers view of the method is in terms of the abstract data, even though the implementation is written in terms of the concrete data. This is satisfactory for a data refinement, as the verification proceeds using the relationship between the abstract and the concrete data defined in the abstraction invariant.

Provision of specifications in terms of the concrete data would assist the supplier when reasoning about their implementations. This can be partially supported, as the supplier may add specifications that are written in terms of the concrete data. However, due to our use of the inheritance hierarchy, there are restrictions on the additions that are permitted (section 5.6.0). Spec# does not allow method pre-conditions or frame conditions to be changed in subclasses. Therefore, changing pre-conditions and frame conditions so that they refer to concrete fields is not permitted in our two-class representation of data refinement. Adding postconditions that refer to the concrete fields is permitted and simply adds an extra proof obligation to the verification.

Proposal 3: *Provide separate declarations of properties of abstract fields (as object invariants) and their mapping to concrete fields (abstraction invariants).*
Evaluation: The object invariant, which specifies properties of the abstract fields, is specified in the specification class, and the abstraction invariant is specified in the implementing class.
Analysis: Using properties or model fields to represent the abstract data, allows the separation of invariants that specify the properties of the abstract fields in the specification, and the specification of the abstraction invariant.

When using properties, the invariant on the abstract data is provided as a post-condition of the property *get* method. The abstraction invariant can be inferred from a combination of the property name (the abstract field name) and the postcondition of the concrete property *get* method in the specification class. The invariant on the concrete data is inferred automatically. When using model fields, the abstract invariant is provided through the **satisfies** clause of the model field declarations in the specification class. The abstraction invariant is explicitly stated in the **satisfies** clause of the overridden model fields, and the invariant on the concrete data is inferred automatically.

Proposal 4: *Provide support for an abstraction invariant which maps the abstract data type that is used in the specification to the concrete data type that is used in the implementation.*
Evaluation: This is achieved through the postconditions of property *get* methods or through the **satisfies** clauses of model fields.
Analysis: See Analysis of Proposal 3 above.

Proposal 5: *Provide an object invariant which specifies properties of the abstract fields (the abstract invariant).*
Evaluation: This is achieved in the specification class, through the postconditions of the property *get* methods or through the **satisfies** clause of the model field declarations.
Analysis: See Analysis of Proposal 3 above.

Proposal 6: *Provide an object invariant which specifies properties of the concrete fields (the concrete invariant).*
Evaluation: When properties are used to represent the abstract data, the concrete invariant is provided through a combination of the postconditions of the abstract fields and the concrete property *get* methods. When model fields are used to represent the abstract data, the concrete invariant is provided through a combination of the **satisfies** clauses of the model fields in the specification and the implementation.
Analysis: In both cases, extra invariants on the concrete data can be explicitly stated (as a conjunct of the **satisfies** clause, or as a conjunct of the postcondition of the property *get* method) for the overridden field. This is permitted, as it could be necessary to state properties of the concrete data. The client will not be aware of the extra constraints, and a stronger object invariant will not damage the specification in any way. Note that neither property *get* methods or model fields can modify the data types that they declare. This is because property *get* methods are pure by default, and model fields may not be used in assignments.

Proposal 7: *Provide verification support so that proving the concrete invariant holds implies that the abstract invariant also holds.*
Evaluation: This is achieved with both properties and model fields.
Analysis: In both cases, the concrete invariant is inferred from the abstract invariant and the abstraction invariant. Due to this dependency and the overriding of *get* methods or model fields, the abstract invariant holds whenever the concrete invariant holds. The concrete invariant must hold whenever the corresponding object is valid. If extra constraints are specified on the concrete data, they will not interfere with the inherited constraints that are specified on the abstract data.

Proposal 8: *Distinguish between* valid *and* mutable *objects, as in the Boogie methodology, and use this information to determine when it is safe to violate the object invariants.*
Evaluation: This is achieved, as we are using the Boogie methodology to verify that the implementation provided in the subclass satisfies the specification in the superclass.
Analysis: See Analysis of Proposal 7 above.

Proposal 9: *In the specification of aggregate objects provide aliasing control so that every sub-object has a unique owning object, and*
(a) that sub-object can only be modified through its owner
(b) the abstraction invariant and the invariant on the abstract data may refer to fields of sub-objects as long as they are "owned" by the aggregate object.
Evaluation: Aliasing control is achieved through use of the [**Rep**] annotation in Spec#. Sub-objects that are identified with [**Rep**] are owned by the aggregate object. Sub-objects can only be modified if its owner is in a mutable state.
Analysis: Spec# allows the modification of non-local variables that are listed in the modifies clause. In addition, the Boogie methodology implicitly allows each method to have a net effect on the state of committed objects. To modify an object, the method body must first obtain a reference to the object and must then follow the methodology rules to make that object mutable. In other words, the object must be a [**Rep**] object that is exposed. The **satisfies** clause can refer to the model field to which it belongs, other fields of the class in which the model field is declared,

or fields of any sub-objects which are owned by the class which declares the model field. Therefore, the abstraction invariant, and the invariant on the abstract data, may refer to fields of sub-objects as long as they are owned by the aggregate object.

Proposal 10: *In the client's view frame conditions should name the abstract fields that a method may modify.*
Evaluation: This is achieved in all of our representations.

Proposal 11: *Concrete fields should not be named in the client's view of the frame condition as this will reveal details about the implementation.*
Evaluation: Due to the Spec# implementation of behavioural subtyping, an overriding method cannot extend its frame conditions. Hence, concrete fields must be named in the client's view of the specification. This can be avoided if the specification is given ownership of the concrete fields when an implementation is provided. If an object owns another object, it can modify its owned object, when that owned object is in a committed state. This is achieved by using the [**Rep**] annotation.

Using model fields to represent the abstract data, and overriding them to give them the properties of the concrete also avoids this problem. In this case, the abstract and concrete fields have the same name so their names will be available in both the client and supplier views. However, only the abstract properties of the data will be available in the client's view. The concrete properties will remain in the supplier's view.
Analysis: One advantage of the use of [**Rep**] fields is that it requires only a modest set of abstraction features for use in modifies clauses. A disadvantage is that sometimes the over-approximation is too coarse.

Proposal 12: *Naming the abstract fields in a frame condition should grant permission for the modification of concrete fields on which they depend (via the abstraction).*
Evaluation: Naming the abstract fields in a frame condition does grant permission for the modification of concrete fields on which they depend when *model* fields are used to represent the abstract data and are overridden in the implementation class.
Analysis: See Analysis of Proposal 11 above.

Proposal 13: *Support inheritance between abstract specifications, ensuring that the rules of behavioural subtyping are adhered to.*
Evaluation: Inheritance between abstract classes can be used to generate a new specification.
Analysis: New methods and new object invariants may be added to specifications in an inheriting class. Existing method specifications, including properties and model fields, may be overridden to strengthen postconditions.

Proposal 14: *Support the overriding of abstractions to strengthen them in inheriting classes (this only applies to implementations).*
Evaluation: Abstractions that are expressed through properties and model fields may be overridden.
Analysis: postconditions of abstractions, expressed through properties and model fields, may be strengthened via overriding. The postconditions of overriding abstractions are conjoined to the postconditions of the overridden abstractions.

5.8 Conclusion

In this chapter, we explored the options for representing modular data refinement in the Spec# programming system. We used existing Spec# features in our representation, and completely decoupled the specification from its implementation. Our aim was to separate the client and supplier views of a program so that the client sees a specification that is written in terms of the abstract data. Any class that satisfies this specification can provide the implementation. The client should not be concerned with any implementation details. Their only concern is that the implementation provided should satisfy the specification.

Our conclusion is to represent data refinement in Spec# using a two-class approach to represent the specification and its implementation. The specification, written in terms of the abstract data is provided by an abstract class. The implementation, written in terms of the concrete data is provided by a class that inherits the specification and provides the code that satisfies this specification. Model fields are used to model the abstract data and its invariant properties. These model fields

can be overridden to define the abstraction invariant, and the invariant on the concrete fields that is used to implement the specification. We make use of the Boogie methodology, which is built into the Spec# programming system, to control aliasing and mutability.

Our representation meets all of the proposals in our proposal suite from section 4.3, albeit with some limitations. The provision of a specification that is written in terms of the concrete data (proposal 2) is not fully met, although enough concrete specification is provided so that our data refinement can be verified by the Spec# tools. Proposals 9, 11 and 12 rely on the use of Spec# annotations to assist the underlying verifier. If these annotations are not used correctly, the Spec# tools will not verify the data refinement as access to necessary data will not be granted.

Some drawbacks of using inheritance to relate the specification and the implementation classes are identified in section 5.1. These restrictions, together with the suite of proposals though which we evaluate our representation will be used in chapter 7 to analyse when two classes in an inheritance relationship meet the criteria for representing a data refinement. First, we build on our experience of representing data refinement in Spec# to design a framework for a modular approach to data refinement. In particular, this framework presents a language that could be implemented as an extension of existing specification languages so that they do not rely on inheritance as the mechanism for linking an implementation to its specification.

Chapter 6

A Framework for Modular Data Refinement

In this chapter we present an object-oriented programming framework to support a modular approach to data refinement. We focus on representing a single-step refinement process, where there are two views of a program: the view of the *supplier* who writes the implementation; and the view of the *client* who uses the program. The client need only be aware of the abstract specification, instantiating the specification to generate an object. The concrete object that is created satisfies the abstract specification, while its implementation details remain hidden from the client. The supplier has access to the abstract specification, and is responsible for providing a concrete implementation and the abstraction invariant which relates the concrete and the abstract data. When the client instantiates the specification, the object that is created is one that is implemented in terms of the concrete data.

We design DRSL, a data refinement specification language which supports modules to express the abstract specification, the concrete specification (which we refer to as the implementation), and the abstraction invariant. It also supports relating these modules via a **refines** relationship. The language is designed to meet the proposals discussed in chapter 4 where possible. We evaluate our language with respect to these proposals in section 4.3 and provide some examples of data refinements expressed in our framework in section 6.3.

6.1 A Data Refinement Specification Language

DRSL is a Java-like language with added support for data refinement. It is similar to JML and Spec# in that it contains a mixture of specification constructs and programming constructs. Object-oriented characteristics such as object references, classes, subclasses, methods with dynamic dispatch and covariant arrays are supported, while features like inheritance, interfaces, static members, access modifiers and generics are excluded.

A program is defined as a collection of classes as follows:

```
Prog    ::=  Class*
Class   ::=  Specification | Abstraction | Implementation
```

When a client instantiates a specification, an object type that satisfies the specification is created. The actual implementation that is used remains hidden from the client. Therefore, the client only needs to see the specification class whereas the supplier is concerned with all three types of class involved in the data refinement. Specification classes are linked to their implementation and the abstraction invariant through a refinement relation. We write the data refinement: `A` $\sqsubseteq_{AI}$ `C` using the syntax

```
impl C refines A via AI
```

where

- `A`, a specification class, specifies an object type in terms of abstract data types; its fields are abstract and it specifies method behaviour in terms of the abstract data.
- `C`, an implementation class, implements an object type in terms of concrete data types; its fields are concrete and its methods consists of executable code which is written in terms of the concrete data.
- `AI`, an abstraction class, provides the abstraction invariants that define the relationship between the abstract data in an object's specification and the concrete data in an object's implementation.

A specification is refined when any of its abstract data fields are replaced by a concrete data field. All references to that abstract data type must be refined

to refer to the concrete data type in the implementation, while maintaining the original specification contracts. The *refines* relationship links the specifications and implementations using the abstraction invariant and associated methods as described by the abstraction class.

6.1.0 Types

Throughout the text, the words term and expression are used synonymously. We write $T\ t$ when a term t is declared to have the type T. The types permitted in our language are booleans, integers, arrays and object types where the object types are either generated from a specification identifier (Id) or from the built-in *Object* type.

```
Type            ::=  bool | int | Type[] | ObjectType
ObjectType      ::=  Id | Object
```

Values of object types are called objects. They consist of a special value called *null*, and of references to a suite of class members (fields, invariants and methods). We write $Id\ o$ when an object o has the object type specified by the specification Id. Implementations provide the executable code that give these objects the behaviour which satisfies their specifications.

Object types and array types respect polymorphic subtyping. Therefore, if T is a subtype of S, then an expression of type T can be assigned to a designator of type S (but not vice-versa unless T and S are the same type). Every reference has a type which returns the actual type (the type or a subtype) stored in the reference, represented as an object. All objects are a subtype of *Object*. Arrays are references to sequences of values. Our arrays are covariant in their element type, support the usual indexing, and have a built in read only field called *Length*.

6.1.1 Specifications

Each specification specifies the behaviour of an object type. We identify specification classes using the *spec* keyword and define them as follows:

```
Specification   ::= spec Id {SMember*}
SMember         ::= Field | Inv | SMethod
Field           ::= Type Id;
Inv             ::= invariant BExpr
SMethod         ::= pure ReturnType Id (TypeId*) ≜ Contract
                          | public ReturnType Id (TypeId*) ≜ Contract
                          | Id(TypeId*) Contract
```

A specification has an identity and a set of member declarations in the form of fields, invariants and methods. We refer to these as model members, as they are written in terms of abstract data and therefore are independent of the object's implementation details. A client can create an object o which maintains the specified behaviour of a specification Id by writing $Id\ o$. A specification for a bag specified in terms of a pair of integers is presented in Figure 6.1.

```
spec BagAbstract{
        int bagsum;
        int bagcount;
        invariant 0 ≤ bagcount && 0 ≤ bagsum;

        BagAbstract() ≜ bagsum, bagcount:[true; bagcount == 0 && bagsum == 0)]
        pure bool isEmpty() ≜ result:[true; result == (bagcount == 0)]
        void add(int x) ≜
                bagsum, bagcount:[0 ≤ x;
                        bagsum == old(bagsum) + x && bagcount ==
old(bagcount) + 1]
}
```

Figure 6.1: DRSL specification of a bag of integers.

Model Fields and Model Invariants

Model fields define the abstract state space of the specified object type. They are specification-only fields that can be declared of any type, and which are used to specify the methods and properties of the object. Model invariants are boolean expressions which state properties of the object in terms of its abstract data. Hence, if an integer field c should be positive for all objects that satisfy its specification, then we write *invariant* $0 <= c$.

All fields and invariants in a specification are *public*. This allows the client to see the abstract data, its properties, and model method specifications which describe

the permitted behaviour of the abstract data. A sample specification is provided in Fig. 6.1.

Model Methods

A method is a parameterised operation that can be invoked on an object to modify or observe its state. Model methods provide the specifications of these methods and hence are publicly available to the client of a specification. We support specifications on pure methods, methods that have side effects and constructors:

```
SMethod        ::= pure ReturnType Id (TypeId*) Contract
                   | ReturnType Id (TypeId*) Contract | Id (TypeId*) Contract
ReturnType     ::= Type | void
TypeId         ::= Type Id
Contract       ::= Modfield*:[BExpr; BExpr]
```

Every specification must provide at least one constructor which has the same name as its specification. It has the form Id $(TypeId*)$ $Contract$ and its role is to specify the value of the fields so that the specification's invariant is established. A constructor's contract therefore consists of a frame condition which names the model fields, the pre-condition which cannot refer to the parameter *this* (as *this* does not exist before the constructor call), and a postcondition that implies the specification invariant.

Methods that do not change the object's state are referred to as pure. They are prefixed with the keyword *pure* and may be invoked in assertions such as pre-conditions, postconditions, and invariants as they have no side effects. Methods that may change the state of an object are not permitted for use in assertions.

Return Types

Methods which do not return a value have the return type *void*. The postcondition of methods which return a value may refer to the *result* variable (a variable which refers to the return type of a method). The type of the value stored in *result* must be a subtype of the method's return type.

Contracts

The contract $Modfield* : [BExpr; BExpr]$ provides the methods specification. The frame condition $ModField*$ provides the list of model fields that a method can modify (akin to *assignable* clauses in JML or $modifies$ clauses in Spec#). The contract's frame $[BExpr; BExpr]$ provides the method's pre-condition followed by the method's postcondition (akin to *requires* and *ensures* clauses in JML and Spec#). The specific fields that a method is allowed to modify is described using the following notation:

```
ModField        ::= ODesignator ModSuffix
ODesignator     ::= this | Designator
Designator      ::= Id | Expr.Id | Expr [Expr]
ModSuffix       ::= .Id | .* | [*]
```

These include any fields within the scope of *this* and cannot include the expression $E.Length$ (for any expression E). A frame condition designator may have different forms:

- $O.x$ permits the method to modify field x of object O,
- $O.*$ allows the modification of any field in the object O,
- $O[*]$ allows the modification of array O at any array index.

Assertions such as pre-conditions, postconditions and invariants are expressed as boolean expressions. Logical and arithmetic operations (&&, ||, !, $\Longrightarrow$ <=, >=, ==, etc) with standard precedences are permitted in our expressions. So too are atoms as listed below:

```
Atom       ::= Literal |Designator | (Type)Expr | Call | old(Expr) |
               Quantification |this| result
Literal    ::= false | true | null | 0 | 1 | 2 | 3 | 4 ...
Designator ::= Id | Expr.Id | Expr [Expr]
Call       ::= Expr.Id(Expr*)
```

In addition to the explicitly declared parameters, every method has an implicit parameter referred to by the keyword *this*. This parameter is within the scope of the specification and provides a reference to the object which can call a method. postconditions can also mention variables (annotated with *old*) which evaluate to the value of the variable in the pre-condition of the method.

Method calls in assertions are restricted to *pure* method calls as these have no side effects and are therefore allowed in expressions. Calls to constructors, calls to methods which have side effects, and the allocation of new types are not permitted in assertions. Universal and existential quantifiers may be used to quantify over a range of integer values. Bindings of the form *int i in* $(0 : N)$ are equivalent to the range $0 <= i < N$. Comprehensions, which apply an operator over a range of integer values, could be added to DRSL. We include the *sum* comprehension for illustration purposes and show its usage in Fig. 6.4.

```
Quantification  ::= Quantor{Binding;Expr}
Quantor         ::= forall | exists | sum
Binding         ::= int Id in (Expr:Expr)
```

Fig. 6.2 illustrates the use of quantification in assertions. The invariant states that all values in the array are greater than or equal to zero. The constructor is responsible for establishing this invariant by ensuring that all values in the array are set to zero, while the method *add*() has the responsibility of maintaining the invariant, ensuring that at least one value in the array is greater than zero.

```
spec CounterArray{
     int[] c;
     invariant forall {int i in (0:c.Length); 0 ≤ c[i]}
     Counter() ≙ c:[true; forall{int i in (0:c.Length); c[i] == 0}]
     void add() ≙ c:[0 ≤ c.Length; exists {int i in (0:c.Length); 0 < c[i]}]
}
```

Figure 6.2: DRSL specification illustrating quantifier syntax.

6.1.2 Implementations

A specification class specifies an object type in terms of its abstract data, while an implementation class implements an object type in terms of the concrete data. An implementation class has an identity, identifies the specification that it refines, and provides the abstraction class which defines the data refinement. It also contains fields, invariants and methods that provide the concrete implementation of the object. The syntax for implementation classes is as follows:

```
Implementation  ::= impl OID refines SId via AbstrId {IMember*}
OId             ::= Id
SId             ::= Id
AbstrId         ::= Id
IMember         ::= Field | IMethod| Inv
Field           ::= Type Id
Inv             ::= invariant BExpr
IMethod         ::= | Id (TypeId*) Block | Prefix ReturnType Id (TypeId*) Block
                    | Prefix ReturnType Id (TypeId*) Block
```

An example of an implementation class is provided in Fig. 6.3.

```
impl BagConcrete refines BagAbstract via BagAI{
        int [] elems;
        int count;
        invariant 0 ≤ count && count ≤ elems.Length;

        BagConcrete(){
                this.count = 0;
                elems = new int[100];
        }
        pure bool isEmpty(){
                if (count == 0) return true;
                else return false;
        }
        void add(int x){
                if (count == elems.Length){
                        int [] b  = new int [2*elems.Length+1];
                        Array.Copy(elems, 0, b, 0, elems.Length);
                        elems = b;
                }
                elems[this.count] = x;
                count++;
       }
}
```

Figure 6.3: DRSL implementation of the specification from Fig. 6.1

Implementation Fields and Invariants

Implementation fields define the concrete state space of an object and are used in the implementation of each of the methods. Each object has its own copy of the variables which are instantiated to give the object its state. Implementation fields are private to the class, as they provide the concrete data used in the implementation. Invariants in the implementation class specify properties of the concrete fields. These invariants are also private as they, like the data that they describe, should not be revealed outside the implementation. For simplicity, we assume that

the fields declared in a class are distinct from other fields declared in the class or any class that it is associated with.

Implementation Methods

The methods in an implementation class have the same type signature as the methods in the specification class that it refines. Methods are public as they must be available to the client of the specification. Every method has one implementation, consisting of a block of statements as defined below.

```
Stmt            ::= Block | Type Id; | Designator = Expr; | ifStmt |
                        Call | WhileStmt |  ForStmt | return Expr;
Block           ::= {Stmt*}
IfStmt          ::= if (Expr) Stmt else Stmt
WhileStmt       ::= while (Expr) invariant Block
ForStmt         ::= for(Stmt; Expr; Stmt) invariant Block
```

The statement $T\ x$ declares a variable named x with type T which is local to the method's implementation. The name of variables must be distinct in the method's scope. The assignment statement $x = E$ assigns the value of expression E to the designator x. The type of E must be the same type as (or a subtype of) the type of x. If x has the form $o.f$ where o is an object and f is its field, then the field f of object o is assigned the value of E. If x has the form $A[f]$ then element f of array A is updated to the value of E.

Expressions are as in the specification module with the following additions:

```
Expr            ::= Call | Allocation
Call            ::= Id(Expr*) | Expr.Id(Expr*)
Allocation      ::= new ObjectType(I, Expr*) | new Type[Expr]
```

We allow for two types of call statement: calls to constructors and calls to methods that have a *void* return type. These have the form $m(\vec{E})$ or $O.m(\vec{E})$ where O is an object that is not *null*, m is a method, and $\vec{E}$ is a possibly empty vector of expressions whose types must be subtypes of the formal parameters of the method m. Calls to methods whose return type is not *void* may be used in expressions.

The allocation of new object types and new arrays is supported. We write

$$S\ o\ =\ new\ I(\vec{E})$$

to allocate an object which has the object type specified in specification S and implemented by implementation I, where $\vec{E}$ is a possibly empty vector of expressions whose types must be subtypes of the formal parameters of the implementation's constructor. The expression new $T[x]$ allocates an array containing x values of type T, where x is an integer defining the length of the allocated array. The array type is $T[\,]$.

In the statement, $if\ (E)\ then\ S\ else\ T$, the expression E must be a boolean expression. If E evaluates to true, then statement S is executed. If E evaluates to false, then statement T is executed. Similarly, the guard of a *while* statement is a boolean expression. The statements that make up the body of the loop are executed if the guard evaluates to *true*. The value of the guard is checked again, and the loop body is repeated until the guard evaluates to $false$. The loop *invariant* is a boolean expression that states a property that is true on every iteration of the loop. It cannot contain any non-pure call or allocation expressions. The for statement is similar to the *while* statement and follows the standard syntax of for loops in Java.

6.1.3 Abstractions

An abstraction class contains the abstraction invariants that relate the abstract and the concrete data in the data refinement. No fields are present as the class can never be instantiated. The class may contain methods which help in defining the abstraction invariant. We do not need to provide an implementation for these methods as they are helper methods that will never be executed. Instead, we supply their specification contract which is required while discharging data refinement proof obligations. All members of the abstraction class are private, as the client should not be permitted access to these implementation details. The class is identified by the keyword *abstr* and defined as:

```
Abstraction     ::= abstr Id {AMember*}
AMember         ::= Inv | AMethod
Inv             ::= invariant BExpr
AMethod         ::= ReturnType Id(TypeId*)Contract
```

Fig. 6.3 presented an implementation of the *BagAbstract* specification in terms of an array. The same methods are supplied as those in its specification, but are implemented in terms of the concrete array. The abstraction invariant, relating the

concrete data from the implementation to the abstract data of the specification is provided in Fig. 6.4.

```
abstr BagAI{
 invariant bagcount == elems.Length && bagsum == absSum();
 int absSum() ≙ result:[true; result == sum {int i in (0:elems.Length); elems[i]}]
}
```

Figure 6.4: An abstraction class in DRSL.

6.2 Evaluating our Data Refinement Framework

In this section we evaluate our proposed framework for modular data refinement with respect to our proposals for the two-class approach to data refinement as presented in section 4.3.

Proposal 1: (section 4.2.3) *Separate the client and the supplier view to ensure a modular approach to data refinement.*

Our framework achieves the separation of client and supplier views by design. All data and specification relating to the client view is contained within the specification class where it is publicly available for the client to read. All concrete data and implementations are privately contained within two classes: the implementation class and the abstraction class. This achieves a deeper separation of the supplier view, so that methods and invariants involved in the abstraction relation are separated from the implementation of the specification. Public specifications from the specification class complete the view for the supplier.

Proposal 2: (section 4.2.4) *Provide a version of the specification in terms of the concrete data to assist the supplier of the implementation and verification tools.*

Our framework provides some support for specification in terms of the concrete data, as invariants on the concrete data in the implementation class, and specifications on methods in the abstraction classes are permitted. It does not currently provide specifications in terms of the concrete data for methods that are specified in the specification class. This support could be added by allowing contracts on methods in the implementation class. The automatic generation of these contracts should be possible through a combination of information in the specification class, invariants in the implementation class and invariants in the abstract class.

Proposal 3: (section 4.2.5) *Provide separate declarations of properties of abstract fields and their mapping to concrete fields.*
Our framework achieves this by requiring properties of abstract fields to be expressed as invariants in the specification class and their mapping to concrete fields to be expressed as invariants in the abstraction class.

Proposal 4: (section 4.2.5) *Provide support for an abstraction invariant which maps the abstract data type that is used in the specification to the concrete data type that is used in the implementation.*
Our framework provides this support in the form of the abstraction class.

Proposal 5: (section 4.2.6) *Provide an object invariant which specifies properties of the abstract fields (the abstract invariant).*
Our framework achieves this by requiring properties of abstract fields to be expressed as invariants in the specification class.

Proposal 6: (section 4.2.6) *Provide an object invariant which specifies properties of the concrete fields (the concrete invariant).*
Our framework achieves this by requiring properties of concrete fields to be expressed as invariants in the implementation class.

Proposal 7: (section 4.2.6) *Provide verification support so that proving the concrete invariant holds implies that the abstract invariant also holds.*
Combining the knowledge that the concrete invariant holds, with the information in the abstraction module will allow the abstract invariant to be verified.

Proposal 8: (section 4.2.6) *Distinguish between* valid *and* mutable *objects, as in the Boogie methodology, and use this information to determine when it is safe to violate the object invariant.*
Our framework does not enforce this at the same granularity as the Boogie methodology. Invariants must be established by the constructor, hold before a method call and be re-established when the method terminates. Therefore, an object must be in a valid state at the start and end of every method call, nested or otherwise. Finer granularity of support for this proposal depends on the verification tools in which our framework is implemented. Our framework is compatible with languages like Spec# where a distinction between valid and mutable objects is supported via *expose*() statements.

Proposal 9: (section 4.2.7) *In the specification of aggregate objects provide aliasing control so that every sub-object has a unique owning object, and*
(a) that sub-object can only be modified through its owner
(b) the abstraction invariant and the invariant on the abstract data may refer to fields of sub-objects as long as they are "owned" by the aggregate object.
This should be the case but enforcement depends on the underlying language in which our framework is implemented. For example, in Java[1] to get exclusive ownership of an object, a class could either construct an object (or make a deep clone of an existing object) and never expose that instance to the outside world. However, if our framework is implemented as an extension of the Spec# language, there is explicit support for ownership through the use of [*Rep*] annotations section 4.2.7.

Proposal 10: (section 4.2.8) *In the client's view frame conditions should name the abstract fields that a method may modify.*
Our framework achieves this by supporting frame conditions in method contracts in the abstract class.

Proposal 11: (section 4.2.8) *Concrete fields should not be named in the client's view of the frame condition as this will reveal details about the implementation.*
Our framework achieves this by only allowing frame conditions in method contracts to refer to fields in the scope of the specification class.

Proposal 12: (section 4.2.8) *Naming the abstract fields in a frame condition should grant permission for the modification of concrete fields on which they depend (via the abstraction).*
The abstraction class provides information about the dependencies between the abstract and the concrete fields so a rule could be easily implemented to ensure that this is the case.

Proposal 13: (section 4.2.9) *Support inheritance between abstract specifications, ensuring that the rules of behavioural subtyping are adhered to.*
This is currently not supported by our framework but could be permitted if the inherited specification adheres to the permitted structure for a specification class.

Proposal 14: (section 4.2.9) *Support the overriding of abstractions to strengthen them in inheriting classes (this only applies to implementations as abstractions will not be present in specifications).*

[1]There are also techniques in Java to ensure ownership using generics.

Our framework supports this by allowing a specification to have many implementations. Strengthening the abstraction invariant requires the expression of a new *refines* relationship between the specification class, the new implementation class and new abstraction class.

Summary: Our framework concentrates on what a programming language should provide in order to express a data refinement. This framework could be used to extend existing languages such as Spec# or JML so that their support for data refinement is improved. Building this framework into such languages would allow our framework to benefit from the underlying verification technologies that are already in place for these languages.

6.3 Data Refinement Examples in DRSL

Two further examples of data refinements expressed in DRSL are presented in this section. The first example, provided in Fig. 6.5, Fig. 6.6 and Fig. 6.7, illustrates the specification and implementation of a bank-account. The abstraction invariant is simply a one-to-one mapping between the abstract and the concrete data.

```
spec AccountAbstract{
        int bal;
        invariant 0 ≤ bal;

        AccountAbstract(int amt) ≙ bal:[0 ≤ amt; bal == amt]
        AccountAbstract(AccountAbstract acc)  ≙
                bal:[0 ≤ acc.getBalance(); bal == acc.bal]
        void transfer(int amt, AccountAbstract acc) ≙
                bal, acc.bal:[0 < amt && amt ≤ acc.getBalance();
                        bal == old(bal) + amt && acc.bal ==
old(acc.bal - amt)]
        void withdraw(int amt) ≙ bal:[0 < amt && amt ≤ bal; bal ==
old(bal) - amt]
        void deposit(int amt) ≙ bal:[0 < amt; bal == old(bal) + amt]
        pure int getBalance()≙ [true; result == bal]
}
```

Figure 6.5: DRSL specification of a bank account.

We note that constructor and method signatures take in formal parameters which have the specification type. The actual parameter types used in the implementation are the implementation types. This substitution is permitted as the object type

created by instantiating the concrete specification is a behavioural subtype of the abstract specification.

The second example, provided in Fig. 6.8, Fig. 6.9 and Fig. 6.10, illustrates the specification and implementation of a clock as discussed in section 4.2. Here, the abstraction invariant maps the three concrete data fields to a single abstract field.

6.4 Conclusion

In this chapter we presented DRSL, a framework for modular data refinement. This framework could be used to extend existing languages such as Spec# or JML so that their support for data refinement is improved and their underlying verification technology can be used to verify that a program conforms to our framework. In the next chapter we build a tool to demonstrate the viability and effectiveness of our proposed framework.

```
impl AccountConcrete refines AccountAbstract via AccountAI{
        int balance;
        invariant 0 ≤ balance;

        AccountConcrete(int amt) { this.balance = amt; }

        AccountConcrete(AccountConcrete acc){
                this.balance = acc.getBalance(); }
        }

        void transfer(int amt, AccountConcrete acc){
                acc.withdraw(amt);
                deposit(amt);
        }

        void withdraw(int amt){ balance -= amt; }
        void deposit(int amt) { balance += amt; }
        pure int getBalance() { return balance; }
}
```

Figure 6.6: DRSL implementation of the specification in Fig. 6.5.

```
abstr AccountAI{
 invariant bal = balance;
}
```

Figure 6.7: An abstraction class for Fig. 6.6.

```
spec ClockAbstract{

        long time ;
        invariant time == getSecond() + getMinute() * 60 + getHour() * 60 * 60;
        invariant 0 ≤ time && time < 24 * 60 * 60;

        ClockAbstract () ≜ time:[true; true]
        void tick() ≜ time:[true; time == old(time + 1) % (24 * 60 * 60)]
        pure int getHour() ≜ :[true; 0 ≤ result && result ≤ 23];
        pure int getMinute() ≜ :[true; 0 ≤ result && result ≤ 59];
        pure int getSecond() ≜ :[true; 0 ≤ result && result ≤ 59];
        void setTime(int hr, int min) ≜
                time:[0 ≤ hr && hr ≤ 23 && 0 ≤ min && min ≤ 59;
                                time == hr * 60 * 60 + min * 60]
}
```

Figure 6.8: DRSL specification of a clock.

```
impl ClockConcrete refines ClockAbstract via ClockAI{

        int hour, second, minute;

        invariant 0 ≤ hour && hour ≤ 23;
        invariant 0 ≤ minute && minute ≤ 59;
        invariant 0 ≤ second && second ≤ 59;

        Clock(){
                hour = 12; minute = 0; second = 0;
        }
        void tick() {
                second = second + 1;;
                if (second == 60){second = 0; minute = minute + 1;}
                if (minute == 60){minute = 0; hour = hour + 1;}
                if (hour == 24){hour = 0;}
        }
        pure int getHour(){return hour;}
        pure int getMinute(){return minute;}
        pure int getSecond(){return second;}
        void setTime(int hr, int min){
                this.hour = hr; this.minute = min; this.second = 0;
        }
}
```

Figure 6.9: DRSL implementation of the specification in Fig. 6.8

```
abstr ClockAI{
  invariant time = second + (minute* 60) + (hour * 60 * 60);
}
```

Figure 6.10: An abstraction class for Fig. 6.9.

Chapter 7

JIRRA: A JIR Refinement Analyser

As part of this dissertation, a proof-of-concept tool was developed to demonstrate the viability and effectiveness of our proposed framework. This tool takes the form of an application that checks whether or not a program conforms to our framework for the modular data refinement of object-oriented programs.

7.1 Tool Overview

Our JIR Refinement Analyser (JIRRA) is built using OpenJIR as a JML/Java parser in conjunction with the ObjectWeb bytecode manipulation and analysis framework. OpenJIR [53] is an experimental version of OpenJML [22] supporting the embedding of JML specifications into class files in the form of a JML Intermediate Representation (JIR) [81]. The OpenJIR library allows parsing JML programs and embedding their JML in into the Java bytecode. It also provides visitors to assist in later analysis of the JML specifications. The ObjectWeb[1] library provides visitors to permit analysis of the regular Java bytecode. The advantage of using these libraries is that they are both used in JMLEclipse, an Eclipse-based integrated development and verification environment for Java and JML which is under active development [19].

[1] http://asm.objectweb.org

7.1.0 Using JIRRA

The tool is run from the command line with the following options:
java -jar jirra <parameters> abstractClass concreteClass
where parameters are:

–classpath <paths> A semicolon separated list of folders where .class files are located

–sourcepath <paths> A semicolon separated list of folders where .java files are located

abstractClass The name of the abstract class. Must be in a package.

concreteClass The name of the concrete class. Must be in a package.

For an example of how to run JIRRA on the classes BagAbstract.java and BagConcrete.java see Figure 7.1.

```
C:\>jirra --classpath ./account/bin --sourcepath ./account/src
account.BagAbstract account.BagConcrete
Abstract class: account.BagAbstract Concrete class: account.BagConcrete
Working directory: ./jirembedded
```

Figure 7.1: Running JIRRA at the command-line.

7.1.1 Tool Requirements

The tool requires OpenJDK6 or later and has been tested under Windows XP, Windows 7 and Ubuntu 10.04.

7.1.2 Tool Input

The input to JIRRA is a program in the form of two Java classes with JML specifications. One class represents the client's view of a data refinement, while the other represents the supplier's view of a data refinement. Sample input for the bag data refinement from chapter 6 is presented in Fig. 7.2 and Fig. 7.3.

```
public abstract class BagAbstract{

        //-----------------Abstract Data ----------------------
        @SpecPublic protected int bagsum;
        @SpecPublic protected int bagcount;

        //-----------------Abstract Invariant -----------------
        //@ public invariant 0 ≤ bagcount && 0 ≤ bagsum;

        //--------Machinery to provide access to concrete ------
        /*@ assignable bagsum;
          @ assignable bagcount;
          @*/
        protected abstract void init();

        //-----------------Abstract Methods --------------------
        //@ ensures \result == (bagcount ==0);
        @Pure public abstract boolean isEmpty();

        /*@ requires 0 ≤ x;
          @ assignable bagsum,bagcount;
          @ ensures bagsum == \old(bagsum) + x;
          @ ensures bagcount == \old(bagcount) + 1;
          @*/
        public abstract void add(int x);
}
```

Figure 7.2: JIRRA Input: Client view of a data refinement.

7.1.3 Tool Output

JIRRA determines whether or not a program conforms to our framework for the modular data refinement of object-oriented programs by using a series of checks to analyse its input. The output is in the form of a list of questions and the tool's response. Sample output, resulting from the analysis of the classes in Fig. 7.2 and Fig. 7.3, is displayed in Fig. 7.4.

7.1.4 Refinement Checks

As the tool takes in programs written in JML, rather than programs written in DRSL, we formulate the JML programs so that they exhibit features of programs written in our framework. Modifications to the input are in keeping with our two-class approach to data refinement (section 5.6). The client view is represented by an abstract class which contains the abstract data, the invariant describing the abstract data and the method specifications. The supplier view is represented by a single class, which inherits the abstract data, invariants and the method specifications

```
public class BagConcrete extends BagAbstract{

        //------------------Concrete Data and Invariants------------
        private int count;
        private int[] elems;
        //@ private invariant 0 ≤ count && count ≤ elems.length;
        //@ private invariant coupling();

        //-----------Machinery to make Data Refinement work----------
        //@ensures coupling();
        private BagConcrete() {
                init();
        }

        public static BagAbstract getInstance(){
                return new BagConcrete();
        }

        //@ensures coupling();
        @Helper protected  void init(){
                this.count = 0;
                this.elems = new int[5];
                for(int i = 0; i < count; i++){
                        this.elems[i] = i;
                }
        }

        //----------------- Concrete Methods --------------------------
        @Pure public boolean isEmpty(){
                if (count == 0) return true;
                else return false;
        }

        //@ensures coupling();
        public void add(int x){
                if (count == elems.length){
                        int [] b = new int[2*elems.length+1];
                        System.arraycopy(elems,0,b,0,elems.length);
                        elems = b;
                }
                elems[this.count] = x;
                count++;
        }
        //-----------------------Coupling Invariant ---------------------
        @Pure protected boolean coupling(){
                bagsum = sum();
                bagcount = count;
                return true;
        }

        //@ensures \result == (\sum int i; 0 ≤ i && i < count; elems[i]);
        private int sum(){
                bagsum = 0;
                for(int i = 0; i < count; i++){
                        bagsum += elems[i];
                }
                return bagsum;
        }
}
```

Figure 7.3: JIRRA Input: Supplier view of a data refinement.

```
Calling OpenJirEmbedder: Made directory ./jirembedded/account
JIR embedding done on account.BagAbstract at./jirembedded/account/BagAbstract.class
JIR embedding done on account.BagConcret at./jirembedded/account/BagConcrete.class

Analysing Java bytecode on ./jirembedded/account/BagAbstract
Analysing embedded JML on ./jirembedded/account/BagAbstract
Analysing Java bytecode on ./jirembedded/account/BagConcrete
Analysing embedded JML on ./jirembedded/account/BagConcrete
Stored results for: account/BagConcrete and account/BagAbstract

------------------ Abstract Class Checks:--------------------------
is the super class object? yes
is the class abstract? yes
does the class contain at least one field? yes
does the class contain at least one factory initialiser method? yes
are all fields spec public and protected? yes
are all normal methods public and abstract? yes
have all normal methods a specification? yes
is only the default constructor present? yes
are all factory initialiser methods protected? yes
are all factory initialiser methods abstract? yes
do all factory initialiser methods return void? yes
do assignable clauses for each factory initialiser
method list all the abstract class fields? yes
are all invariants public? yes
are all invariant fields spec public? yes
are all requires fields owned or method parameters? yes
are all ensures fields owned or method parameters? yes
are all assignable fields owned? yes
------------------- Concrete Class Checks:--------------------------
does the class inherit from the abstract? yes
is the class concrete? yes
does the class contain at least one field? yes
does the class contain at least one factory initialiser? yes
does the class contain at least one factory method? yes
are all fields private or protected? yes
are no fields spec public? yes
are all constructors private or protected? yes
do all constructors call the factory initialiser? yes
do all factories call the constructor? yes
are all factory initialisers protected? yes
all factory initialiser methods return void? yes
all factory initialiser methods are helper methods? yes
each factory initialiser's first postcondition is 'ensures coupling()'? yes
all factory initialisers implemented in terms of concrete variables only?yes
are all factory methods public and static? yes
do all factory methods return an instance of the abstract super-class? yes
is there exactly one coupling() method present? yes
is the coupling method protected and pure? yes
does the coupling method return true? yes
is coupling() implemented in terms of both abstract and concrete variables?yes
shadowing of abstract fields absent? yes
are all normal non-inherited methods non-public ? yes
are normal methods implemented in terms of concrete variables only? yes
first post of all non-pure normal methods is 'ensures coupling()'? yes
are all invariants private? yes
is coupling() present as an invariant? yes
are all invariant fields local to the class (not inherited)? yes
All tests passed.
```

Figure 7.4: Sample JIRRA output.

from the abstract class. The concrete data, the invariant describing the concrete data, method implementations and the abstraction invariant are added to this class. The abstraction invariant is established by a method named *coupling*(), allowing us to distinguish it from the invariant on the concrete data. A preferred approach would use JML *model* variables with *represents* clauses (section 4.2.5) to define the abstraction invariant. However, these are not currently supported in OpenJIR.

When the client requests an object satisfying the specification, the implementation class creates an object that satisfies the specification. This effect is achieved in the two-class approach using factory methods (section 5.6.4). When a client requests an object satisfying the specification, a factory method from the implementation class is called. This factory method calls the concrete constructor, which calls an initialiser method, which initialises the created object. For example, when the client requests an object that satisfies the *BagAbstract* specification, a factory method of its concrete class is called. The result is that an object of type *BagAbstract* is created and returned.

```
BagAbstract bag = BagConcrete.getInstance();
```

In order to perform checks to determine if programs conform to our framework for modular data refinement, we assume the following conventions:

- factory methods are named *getInstance*(),
- initialiser methods are named *init*(), and
- abstraction invariants are defined in methods named *coupling*().

The checks performed on the input are listed below. The phrase "normal methods" refers to all methods apart from constructors, factory methods, initialiser methods and coupling methods.

Abstract Class Checks

A0 the super class is *Object*:

checks that nothing is inherited from other specification or implementation classes.

A1 the class is *abstract*:

ensures that the class is a specification class.

A2 the class contains at least one field:

ensures there is data in the class which can be represented in the concrete class.

A3 the class contains at least one initialiser method:

used to pass parameters to the concrete constructor and to specify the initial state of the object on creation.

A4 all fields are annotated with @*SpecPublic* :

ensures that the client can view the invariant and method specifications.

A5 all fields are *protected*:

ensures that fields can only be accessed through an inheriting class.

A6 all normal methods are *public*:

allows the client to call the methods when an object has been created.

A7 all non-constructor methods are *abstract*:

as they provide a specification view of the program.

A8 all normal methods have a specification:

to provide the method's behaviour.

A9 no constructors are present other than the default constructor:

ensures that all objects are constructed via the initialiser method.

A10 all initialiser methods are *protected*:

ensures that initialiser methods can only be called by objects of an inheriting class.

A11 all initialiser methods are *abstract*:

ensures that objects can only be created in an inheriting class.

A12 all initialiser methods have a return type of *void*:

ensures that no value is returned (initialisers simply initialise the object's state).

A13 the frame conditions of each initialiser method lists all of the abstract fields:

ensures that all fields corresponding to the abstract representation of the object can be initialised.

A14 all invariants are *public*:

ensures specification visibility to the client.

A15 all fields used in the invariants are @*SpecPublic*:

allows properties of the abstract fields to be publicly specified.

A16 all fields used in the method pre-conditions are declared in the class or provided

as method parameters:

ensures visibility to the client and does not reveal implementation details.

A17 all fields used in the method postconditions are declared in the class or provided as method parameters.

ensures visibility to the client and does not reveal implementation details.

A18 all fields in the method frame conditions are declared within the abstract class:

ensures visibility to the client and does not reveal implementation details.

Java specific comments: Abstract classes in Java provide a default constructor that is not *abstract* so the check **A7** is limited to non-constructor methods. Check **A9** ensures that no constructors, other than the default, are present. Check **A15** is true if checks #**A0** and #**A4** are true.

Concrete Class Checks

C0 the super class is the abstract class:

ensures that the concrete class inherits the specification.

C1 the class is not abstract:

ensures that the concrete class implements all methods in the specification.

C2 the class contains at least one field:

ensures there is data in the class which can be mapped to its abstract representation.

C3 the class contains at least one initialiser method:

needed to initialise the object on creation.

C4 the class contains at least one factory method:

needed to call the constructor and return the created object.

C5 the class contains at least one user-supplied constructor:

needed to create an object and call the initialiser method to initialise it.

C6 no fields are *public*:

ensures that no implementation details are revealed.

C7 no fields are annotated with @*SpecPublic*:

ensures that no implementation details are revealed in the specification.

C8 no constructors are *public*:

ensures that constructors cannot be called outside the class.

C9 all constructors call an initialiser method, only call an initialiser method and call it with the same parameter list (in the same order) as the initialiser method:

ensures that objects cannot be created without calling the initialiser method, establishing the specification inherited from the abstract class.

C10 all factory methods call a constructor, only call a constructor and call it with the same parameter list (in the same order) as the factory method:

ensures that objects cannot be created without calling the concrete constructor.

C11 all initialiser methods are *protected*:

ensures that initialiser methods cannot be called outside the class. They are inherited from the abstract class where they are required to be protected(**A10**).

C12 all initialiser methods have a return type of *void*:

ensures that no value is returned (initialisers simply initialise the object's state).

C13 all initialiser methods are annotated with @*Helper*:

avoids an invariant check at the pre-condition of initialiser methods, as the constructor will not have returned, and hence the invariant will not be established when the initialiser method is called. All initialiser methods are called by constructors, so avoiding this check on initialiser methods does not cause verification errors.

C14 all initialiser method specifications are strengthened to include the postcondition *ensures coupling*():

adds a check that the invariant is established by the initialiser methods.

C15 no initialiser methods are abstract:

ensures that all initialiser methods are implemented.

C16 all initialiser methods are implemented in terms of concrete fields only:

ensures that no abstract fields are used in the implementation.

C17 all factory methods are *public*:

ensures that they can be called by clients of the class.

C18 all factory methods are *static*:

ensures that an object can be created.

C19 all factory methods return an instance of the superclass:

ensures that objects that satisfy the specification are generated.

C20 exactly one *coupling*() method is present:

ensures that the abstraction invariant is defined in one unique place.

C21 the *coupling*() method is protected:

ensures that no implementation details are revealed outside the class.

C22 the *coupling*() method is annotated with @*Pure*:

ensures that we can call the *coupling*() method in specifications even though we know it has the (desirable) side effect of setting the abstract fields to values determined by the changes to their concrete representations.

C23 the *coupling*() method returns *true*:

allows us to express when the coupling invariant should be true.

C24 the *coupling*() method is implemented in terms of both the abstract and the concrete fields:

ensures that the coupling invariant refers to all of the abstract fields and at least one of the concrete fields (checking that all mappings between abstract and concrete fields are provided by the invariant is not checked automatically).

C25 all normal methods are not *abstract*:

ensures that a full implementation is provided by the class.

C26 all normal methods which are newly added to the class are not *public*:

allows the addition of methods to assist with the implementation. As these are not public, they cannot be called from outside the class.

C27 permitted changes to method parameters are in keeping with subtyping rules:

ensures no typing errors when a concrete implementation is substituted for an abstract specification.

C28 methods are implemented in terms of concrete fields only:

ensures that no abstract fields are used in the implementation.

C29 the first *ensures* clause of all non-pure normal methods is *ensures coupling*():

ensures that the coupling invariant is established by every non-pure normal method before its postcondition (written in terms of the abstract data) is checked.

C30 all invariants are *private*:

ensures that no implementation details are revealed outside the class.

C31 *invariant coupling*() is present:

checks that the abstraction invariant is established by the constructor and maintained by the public methods.

C32 all fields used in the invariant on the concrete data are declared within the implementation class:

ensures that the private invariant does not state any properties of the abstract fields.

Java specific comments: We check that no shadowing of abstract fields occurs in the concrete class. Check **C5** is given for free as we check that all constructors call the initialisation method. If the user does not supply a constructor, the presence of the default constructor will cause check **C9** to fail. Check **C15** is true if **C0** and **C1** are true, check **C25** is true if **C1** is true and check **C27** is automatically checked by Java.

7.2 Conclusion

We have developed JIRRA, a proof-of-concept tool to demonstrate the viability and effectiveness of the framework for modular data refinement as proposed in chapter 6. This tool takes the form of an application that checks whether or not a program conforms to our framework for the modular data refinement of object-oriented programs. A series of checks are performed on the program to determine whether or not the program conforms to our framework. Further examples of programs that have been tested in JIRRA are available in Appendix B.

Chapter 8

Conclusions

The major advantage of data refinement is that specifications can be written and reasoned about in a way that is independent of their implementations. Object-oriented verification tools provide some support for data abstraction, but fall-short of supporting the verification of data refinements. We summarise our achievements in providing support for data refinement in object-oriented verification tools and suggest some future work in this area.

8.1 Summary and Conclusions

Object-oriented program verification tools provide a behavioural interface specification language in which to express their specifications. These languages are typically a superset of the programming language, offering a richer set of expressions, which are used to provide the specification. The implementation is provided in terms of the programming language constructs. Verification tools are used to translate these specifications and their implementations into proof obligations that are discharged by a theorem prover if the implementation satisfies its specification. If the theorem prover cannot discharge the proof obligations, no guarantee that the implementation satisfies its specification is offered.

Data refinement is the process of transforming a specification written in terms of the abstract data type into a specification that is written in terms of the concrete data type. We say that the implementation *refines* the specification. The theory

of data refinement guarantees that this replacement does not adversely affect the behaviour of the programs that use these specifications. The key idea is to use an abstraction relation to describe the connection between the abstract and the concrete data types. In verifying a data refinement, we prove that the implementation, which is written in terms of one data type, satisfies the specification that is written in terms of another data type.

Techniques for data abstraction provide clear separation between the abstract properties of a program's specification, and the concrete properties of its implementation, while maintaining the relationship between the two data types that are used in the refinement. However, many specification languages allow the specification to be written in terms of fields, methods and types that are intended to be private to the implementation. This forces a coupling between the specification and the implementation, exposing implementation details in the specification, and requiring the complete rewriting of a specification when minor modifications to an implementation are made.

In this dissertation, we proposed a framework for object-oriented programming languages that supports modular data refinement and the separation of client and supplier views of these data refinements. This separation also ensures that implementation details are not exposed via a specification and that alternative implementations can be provided for a specification without concerning the software client. Our proposed framework can be used to extend existing behavioural interface specification languages, such as Spec# or JML, so that their support for data refinement is improved and their underlying verification technology can be used to verify that a program conforms to our framework. To prove the feasibility of these proposed extensions, a prototypical tool was developed to analyse JML programs to determine if they conform to our framework.

8.1.0 Achieving Our Goals

To achieve our goals, we examined existing specification languages and their support for data abstraction. The support offered for data abstraction in the JML and Spec# languages was thoroughly analysed. Our conclusion was to represent data refinement using a two-class approach to represent the specification and its

implementation. The specification, written in terms of the abstract data is provided by an abstract class. The implementation, written in terms of the concrete data, is provided by a class that inherits this specification and provides the code that satisfies it. Many restrictions on the inheritance relationship were imposed in order to achieve a true representation of a data refinement. For example, no implementation details are permitted in the abstract class, and extra public methods cannot be added to the inheriting concrete class. These are normal restrictions in data refinement, but are permitted in the inheritance hierarchy. We analysed this approach to data refinement in the Spec# language and showed that our approach could also be used in JML.

The first goal of our research was to provide and evaluate a framework for modular data refinement in behavioural interface specification languages for object-oriented programs. This framework is presented and evaluated in chapter 6. As our framework concentrates on what a programming language should provide in order to express a data refinement, it is designed to meet the proposals listed in chapter 4. Enforcement of proposals concerning the verification technology used (Proposals 8 and 9) depend on the underlying language in which our framework is implemented. For example, if our framework is implemented as an extension of the Spec# language, there is explicit support for ownership through the use of [*Rep*] annotations section 4.2.7. Inheritance between specifications is not directly supported as our framework concentrates on a one-step data refinement where one specification is refined by one implementation. We do not consider this a drawback, as providing alternative specifications and implementations is possible within our framework. This is achieved by defining alternative specification-implementation pairs which are related by an appropriate abstraction invariant.

The second goal of our research was to provide improved support for data refinement in existing verification tools for object-oriented programs, making use of existing language structure and verification techniques where possible. Our proposed framework could be integrated into existing languages such as Spec# or JML, so that their support for data refinement is improved and their underlying verification technology used to verify that a program conforms to a data refinement. To prove the feasibility of these proposed extensions, a prototypical tool was developed. This tool accepts JML programs and checks whether they conform to the rules governing

a data refinement, as defined by our framework. We implement our proposed extensions using a restricted form of inheritance to represent a data refinement. We show that, even without a critical abstraction component[1], a data refinement can still be expressed and verified satisfactorily in JML. At present, writing data refinements in JML in the format accepted by our tool is cumbersome (a checklist is provided in section 7.1.4). Therefore, the next phase of this research is to provide the language extensions for our framework in JML. We have already demonstrated that a similar representation of data refinement can be expressed in Spec# (chapter 5). Therefore we believe that these results are also applicable here.

8.2 Future Work

The refinement analysis tool which we developed is presented in chapter 7. This tool checks if a JML program conforms to our framework for modular data refinement. This is achieved by analysing the Java bytecode along with an embedded JML intermediate representation of the program. The OpenJIR library allows us to parse JML programs and embed their JML in their Java bytecode. It also provides visitors to assist in later analysis of the JML specifications. The ObjectWeb library also provides visitors to permit analysis of the regular Java bytecode.

The next phase in this work is to move from an analytical tool to a synthetic one. Both the OpenJIR and ObjectWeb libraries allow for program manipulation in addition to analysis. By providing the developer with carefully designed Java annotations, we can allow them to easily embed information about their program into their compiled bytecode. A second stage compiler (our data refinement compiler) can detect these annotations and, using the aforementioned bytecode libraries, transform the code automatically to generate a verifiable data refinement. The code generated by this tool will be JIR embedded code. This JIR approach is used by the current version of the JMLEclipse project so back-end tools should be readily available for manipulation of this generated code. Our proposed annotation-based approach also fits naturally with JML 4.5, where an annotation-based approach to JML has already been adopted. We can achieve a similar result on the .NET plat-

[1]model fields are currently not supported in JMLEclipse

form once the bytecode translator[2] for .NET programs is released. This translator takes Microsoft Intermediate Language (MSIL) programs from a .NET compiler and produces a Boogie program which existing verification tools can be used to verify. It is envisaged that bytecode manipulation capabilities, similar to those already available in Java will be supported.

In summary, we have shown in this dissertation that support for data refinement is both desirable and feasible in behavioural interface specification languages. Our future work will focus on integrating such support into both JML and Spec#.

[2]Due for release at http://boogie.codeplex.com

Appendix A

Refinement Examples

$S : [T, S = (+j : 0 <= j < 100 : A[j])]$
"**Introduce Local variable**" and "**Strengthen postcondition**"
$S, i : [T, S = (+j : 0 <= j < i : A[j]) \wedge i = 100]$
"**Sequential Composition**"
$1) S, i : [T, S = (+j : 0 <= j < i : A[j])]$
$2) S, i : [S = (+j : 0 <= j < i : A[j]), S = (+j : 0 <= j < i : A[j]) \wedge i = 100)]$

$1) S, i : [T, S = (+j : 0 <= j < i : A[j])]$
"**Following assignment**"
$S, i : [T, S = (+j : 0 <= j < i : A[j])]$
$i := 0$
$S, i : [T, S = 0]$
"**Contract Frame**"
$S : [T, S = 0]$
"**Introduce assignment**"
$S := 0$

$2) S, i : [S = (+j : 0 <= j < i : A[j]), S = (+j : 0 <= j < i : A[j]) \wedge i = 100)$
"**Introduce Iteration**"
$While\ i \neq 100$
DO
$\quad 4) S, i : [i \neq 100 \wedge S = (+j : 0 <= j < i : A[j]),$
$\quad S = (+j : 0 <= j < i : A[j]) \wedge 0 <= (100 - i) < (100 - i)]$
OD

$4) S, i : [i \neq 100 \wedge S = (+j : 0 <= j < i : A[j]),$
$\quad S = (+j : 0 <= j < i : A[j]) \wedge 0 <= (100 - i) < (100 - i)$
"**Following assignment**"
$i := i + 1$
"**Contract Frame and simplify**"
$S : [i \neq 100 \wedge S = (+j : 0 <= j < i : A[j]), S = (+j : 0 <= j < i + 1 : A[j])]$
$= S : [i \neq 100 \wedge S = (+j : 0 <= j < i : A[j]), S = (+j : 0 <= j < i : A[j] + A[i])$
"**Introduce assignment**"
$S := S + A[i]$

Finished Solution
$S := 0, i := 0;$
$while\ i \neq 100$
DO
$\quad S := S + A[i]; i := i + 1;$
OD

Figure A.1: The refinement of a specification to an implementation using refinement calculus.

$Fact : [T,\ Fact = N!]$
"**introduce Local Var**" **and** "**Strengthen pre – condition**"
$Fact, i : [t, Fact = i!I = N]$
"**Sequential Composition**"
$1) Fact, i : [t, Fact = i!]$
$2) Fact, i : [t, Fact = i!, Fact = i! \wedge i = N]$

$1) Fact, i : [t, Fact = i!]$
"**Following assignment**"
$Fact, i : [t, Fact = 0!]$
$i := 0$
"**Contract frame**"
$Fact : [t, Fact = 1]$
"**Introduce Assignment**"
$Fact := 1$

$2) Fact, i : [t, Fact = i!, Fact = i! \wedge i = N]$
"**Introduce Iteration**"
$while\ i \neq N$
DO
$3) Fact, i : [i \neq N \wedge Fact = i!, 0 = (N - i) \wedge (N - i_0)]$
OD
$3) Fact, i : [i \neq N \wedge Fact = i!, 0 = (N - i) \wedge (N - i_0)]$

"**Following assignment**"
$Fact, i : [i \neq N \wedge Fact = i!, Fact = (i + 1)!0 = N - (i + 1) \wedge (N - i_0)]$
$i := i + 1$
"**Contact frame and simplify**"
$Fact, i : [i \neq NFact = i!, Fact = (i + 1) * i!]$
"**Introduce Assignment**"
$Fact := Fact * (i + 1)$

Finished Solution
$Fact := 1$
$i := 0$
$whilei \neq 100$
DO
$Fact := fact * (i + 1)$
$i := i + 1$
OD

Figure A.2: The refinement of a specification to an implementation using refinement calculus.

Appendix B

JIRRA Inputs: Data Refinement Examples

```
public  abstract class  AccountAbstract {

        //-----------------Abstract Data and Invariants ---------------------

        @SpecPublic  protected int bal;
        //@ public invariant bal >= 0 ;

        //-------------Machinery to provide access to concrete --------------
        /*@ requires amt >= 0;
          @ assignable bal;
          @ ensures  bal == amt; @*/
        protected abstract void init(int amt); d

        /*@ requires acc.bal >= 0;
          @ assignable bal;
          @ ensures bal == acc.bal; @*/
        protected abstract void init(AccountAbstract acc);

        //-----------------Abstract Methods ----------------------------------
        /*@ requires amt > 0 && amt ≤ acc.getBalance();
          @ assignable bal, acc.bal;
          @ ensures bal == \old(bal) + amt && acc.bal == \old(acc.bal - amt);
          @*/
        public abstract void transfer(int amt, AccountAbstract acc);

        /*@ requires amt > 0 && amt ≤ bal;
          @ assignable bal;
          @ ensures bal == \old(bal) - amt; @*/
        public abstract void withdraw(int amt);

        /*@ requires amt > 0;
          @ assignable bal;
          @ ensures bal == amt + \old(bal); @*/
        public abstract void deposit(int amt);

        //@ensures \result == bal;
        @Pure public abstract int getBalance();
}
```

Figure B.1: JIRRA Input: Client view of the account data refinement.

```
public class AccountConcrete extends AccountAbstract{

        //-------------Concrete Data and Invariants---------------------
        private  int balance; //concrete data
        //@ private invariant balance >= 0; //concrete invariant
        //@ private invariant coupling(); //abstraction function

        //----------Machinery to make Data Refinement work--------------
        //@ensures coupling();
        private AccountConcrete(int amt){
                init(amt);
        }
        public static AccountAbstract getInstance(int amt){
                return new AccountConcrete(amt);
        }
        //@ensures coupling();
        private AccountConcrete(AccountAbstract acc){
                init(acc);
        }
        public static AccountAbstract getInstance(AccountAbstract acc){
                return new AccountConcrete(acc);
        }
       //@ensures coupling();
        @Helper protected  void init(int amt){
          this.balance=amt;
        }
        //@ensures coupling();
        @Helper protected void init(AccountAbstract acc){
          this.balance=acc.getBalance();
        }

        //----------------- Concrete Methods ----------------------------
        //@ensures coupling();
        public void transfer(int amt, AccountAbstract acc) {
                acc.withdraw(amt);
                deposit(amt);
        }
        //@ensures coupling();
        public void withdraw(int amt) {
                balance -= amt;
        }
        //@ensures coupling();
        public void deposit(int amt) {
                balance += amt;
        }
        //@ensures coupling();
        public /*@ pure @*/ int getBalance() {
                return balance;
        }

        //-----------------------Coupling Invariant -------------------
        //@ensures true;
        @Pure protected  boolean coupling(){
                bal=balance;
                return true;
        }
}
```

Figure B.2: JIRRA Input: Supplier view of the account data refinement.

```
public abstract class ClockAbstract{

        //-----------------Abstract Data and Invariants------------------
        @SpecPublic protected  long time;
        //@ public invariant time ==
                     getSecond() + getMinute() * 60 + getHour() * 60 * 60;

        //@ public invariant 0 ≤ time && time < 24 * 60 * 60;

        //-----------------Abstract Methods -----------------------------
        //@ assignable time;
        //@ ensures time == (\old(time + 1)) % (24 * 60 * 60);
        public abstract void tick();

        //@ ensures 0 ≤ \result && \result ≤ 23;
        @Pure public abstract int getHour();

        //@ ensures 0 ≤ \result && \result ≤ 59;
        @Pure public abstract int getMinute();

        //@ ensures 0 ≤ \result && \result ≤ 59;
        @Pure public abstract int getSecond();

        /*@ requires 0 ≤ hr && hr ≤ 23 && 0 ≤ min && min ≤ 59;
          @ assignable time;
          @ ensures (time == hr * 60 * 60 + min * 60);
          @*/
        public abstract void setTime(int hr, int min);

        //--------Machinery to provide access to concrete ----------
        //@ assignable time;
        protected abstract void init();
}
```

Figure B.3: JIRRA Input: Client view of the clock data refinement.

```
public class ClockConcrete extends ClockAbstract{

        //------------------Concrete Data and Invariants------------------
        private int hour;
        private int minute;
        private int second;
        //@ private invariant 0 ≤ hour && hour ≤ 23;
        //@ private invariant 0 ≤ minute && minute ≤ 59;
        //@ private invariant 0 ≤ second && second ≤ 59;
        //@ private invariant coupling();

        //-----------Machinery to make Data Refinement work--------------
        //@ensures coupling();
        private ClockConcrete(){
                init();
        }

        public static ClockAbstract getInstance() {
                return new ClockConcrete();
        }

        //@ensures coupling();
        @Helper protected void init(){
                hour = 12;
                minute = 0;
                second = 0;
        }

        //----------------- Concrete Methods ---------------------------
        @Pure public int getHour(){return hour; }
        @Pure public int getMinute(){return minute; }
        @Pure public int getSecond(){return second;}

        //@ensures coupling();
        public void setTime(int hr, int min){
                this.hour = hr; this.minute = min; this.second = 0;
        }

        //@ensures coupling();
        public void tick(){
                second = second + 1;;
                if (second == 60) second = 0; minute = minute + 1;
                if (minute == 60) minute = 0; hour = hour + 1;
                if (hour == 24) hour = 0;
        }

        //-----------------------Coupling Invariant ------------------
        //@ ensures \result==true;
        @Pure protected  boolean coupling(){
                time = second + (minute * 60) + (hour * 60 * 60);
                return  true;
        }
}
```

Figure B.4: JIRRA Input: Supplier view of the clock data refinement.

Bibliography

[1] J. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, Aug. 1996.

[2] Ádám Darvas and K. R. M. Leino. Practical reasoning about invocations and implementations of pure methods. In M. B. Dwyer and A. Lopes, editors, *Fundamental Approaches to Software Engineering (FASE)*, volume 4422 of *LNCS*, pages 336–351. Springer-Verlag, 2007.

[3] P. Amey and R. Chapman. Static verification and extreme programming. *Ada Lett.*, XXIV(1):4–9, 2004.

[4] K. R. Apt. Ten years of hoare's logic. *ACM Transactions on Programming Languages and Systems*, 3:431–483, 1981.

[5] R. J. Back. *On the Correctness of Refinement Steps in Program Development*. PhD thesis, Åbo Akademi, Department of Computer Science, Helsinki, Finland, 1978.

[6] R. J. Back. Correctness preserving program refinements: Proof theory and applications. In *Tract 131, Mathematisch Centrum*, 1980.

[7] R. J. Back, A. Akademi, and J. V. Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[8] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. Technical Report MSR-TR-2004-08, Microsoft Research, Microsoft Corporation, Redmond, WA, January 28 2004.

[9] A. Banerjee, M. Barnett, and D. A. Naumann. Boogie meets regions: A verification experience report. In *VSTTE '08: Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments*, pages 177–191, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison Wesley, New York, NY, 2003.

[11] M. Barnett, B. E. Chang, R. Deline, B. Jacobs, and K. R. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, volume 4111 of Lecture Notes in Computer Science*, pages 364–387. Springer, 2006.

[12] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2006.

[13] M. Barnett, R. DeLine, M. Fahndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, June 2004.

[14] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.

[15] B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of Object-Oriented Software. The KeY Approach: Foreword by K. Rustan M. Leino (Lecture Notes in Computer Science).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[16] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[17] G. Carter, R. Monahan, and J. Morris. Software refinement with perfect developer. In *3rd IEEE International Conference on Software Engineering and Formal Methods*, Koblenz, Germany, September 5-9 2005.

[18] P. Chalin, J. Kiniry, G. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *FMCO'2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 342–363. Springer, 2005.

[19] P. Chalin, Robby, P. R. James, J. Lee, and G. Karabotsos. Towards an industrial grade IVE for Java and next generation research platform for JML. Technical Report TR2009-10-01, Kansas State University, Jan. 2010.

[20] Y. Cheon, G. Leavens, M. Sitaraman, and S. Edwards. Model variables: cleanly supporting abstraction in design by contract: Research articles. *Softw. Pract. Exper.*, 35(6):583–599, 2005.

[21] Y. Cheon and G. T. Leavens. The Larch/Smalltalk interface specification language. *ACM Trans. Softw. Eng. Methodol.*, 3(3):221–153, 1994.

[22] D. Cok. OpenJML, 2009.

[23] D. R. Cok. Reasoning with specifications containing method calls and model fields. *Journal of Object Technology*, 4(8):77–103, 2005.

[24] D. Crocker. Perfect Developer: A tool for object-oriented formal specification and refinement. In *12th International FME Symposium, Tools Exhibition Notes*, Pisa, Italy, September 8-14 2003.

[25] Á. Darvas and P. Müller. Reasoning About Method Calls in Interface Specifications. *Journal of Object Technology (JOT)*, 5(5):59–85, June 2006.

[26] A. Darvas and P. Müller. Faithful mapping of model classes to mathematical structures. In *SAVCBS '07: Proceedings of the 2007 conference on Specification and verification of component-based systems*, pages 31–38, New York, NY, USA, 2007. ACM.

[27] L. de Moura and N. Björner. Z3: An efficient SMT solver. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS*, 2008.

[28] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.

[29] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 258–267, Washington, DC, USA, 1996. IEEE Computer Society.

[30] E. W. Dijkstra. Notes on Structured Programming. circulated privately, apr 1970.

[31] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[32] G. Dromey. *Program Derivation: The Development of Programs From Specifications*. Addison-Wesley, 1989.

[33] J. Filliâtre. *The WHY verification tool (Tutorial and Reference Manual)*. INRIA TeamProject Proval, France, version 2.19 edition.

[34] J. Filliâtre and C. Marché. Multi-prover verification of C programs. In *Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29, Berlin, 2004. Springer-Verlag.

[35] R. B. Findler, M. Latendresse, and M. Felleisen. Behavioral contracts and behavioral subtyping. *SIGSOFT Softw. Eng. Notes*, 26(5):229–236, 2001.

[36] P. H. B. Gardiner and C. Morgan. A single complete rule for data refinement. *Formal Asp. Comput.*, 5(4):367–382, 1993.

[37] J. Guttag and J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, Berlin, 1993.

[38] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson. Behavioral interface specification languages. Technical Report CS-TR-09-01, University of Central Florida, School of EECS, Orlando, FL, Mar. 2009.

[39] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[40] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1, 1972.

[41] C. A. R. Hoare, J. He, and J. W. Sanders. Prespecification in data refinement. *Inf. Process. Lett.*, 25(2):71–76, 1987.

[42] T. Hoare and J. Misra. Verified software: Theories, tools, experiments vision of a grand challenge project. In B. Meyer and J. Woodcock, editors, *VSTTE*, volume 4171 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2005.

[43] D. Jackson. Lightweight formal methods. In *Formal Methods Europe*, page 1, Berlin, Germany, March 12-16 2001.

[44] B. Jonsson. Simulations between specifications of distributed systems. In *CONCUR '91: Proceedings of the 2nd International Conference on Concurrency Theory*, pages 346–360, London, UK, 1991. Springer-Verlag.

[45] J. Kavanagh and W. Hall. Grand Challenges in Computing Research - GCCR '08 - report, 2008.

[46] KindSoftware. Extended Static Checker for Java version 2 (ESC/Java2). http://secure.ucd.ie/products/opensource/ESCJava2/, April 25 2006.

[47] L. Lamport. A simple approach to specifying concurrent systems. *Commun. ACM*, 32(1):32–45, 1989.

[48] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program.*, 55(1-3):185–208, 2005.

[49] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Form. Asp. Comput.*, 19(2):159–189, 2007.

[50] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *Object Oriented Programming Systems Languages and Applications*, pages 105–106, Minneapolis, Minnesota, October 15-19 2000.

[51] G. T. Leavens and D. A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. Technical Report 06-20b, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, Sept. 2006.

[52] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. JML reference manual, 2004.

[53] J. Lee, Robby, and P. Chalin. Tutorial on JIR, 2010.

[54] K. R. M. Leino. Specification and verification of object-oriented software (marktoberdorf international summer school 2008, lecture notes), 2008.

[55] K. R. M. Leino. *This is Boogie 2 (working draft of the Boogie 2 language reference manual)*. Microsoft Research, One Microsoft Way Redmond, WA 98052 USA, 2008.

[56] K. R. M. Leino and R. Middelkoop. Proving consistency of pure methods and model fields. In M. Chechik and M. Wirsing, editors, *FASE*, volume 5503 of *Lecture Notes in Computer Science*, pages 231–245. Springer, 2009.

[57] K. R. M. Leino and R. Monahan. Automatic verification of textbook programs that use comprehensions. In *Workshop on Formal Techniques for Java-like Programs (FTfJP)*, July 2007.

[58] K. R. M. Leino and R. Monahan. Program Verification Using the Spec# Programming System, ECOOP 2009 Tutorial. http://www.rosemarymonahan.com/specsharp/papers/ECOOPTutorial.pdf, July 2009.

[59] K. R. M. Leino and R. Monahan. Reasoning about comprehensions with first-order smt solvers. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 615–622, New York, NY, USA, 2009. ACM.

[60] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer Verlag, June 2004.

[61] K. R. M. Leino and P. Müller. A verification methodology for model fields. In P. Sestoft, editor, *ESOP*, volume 3924 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2006.

[62] K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, Sept. 2002.

[63] B. Liskov. Data abstraction and hierarchy. *SIGPLAN Notices*, pages 17–34, May 1988.

[64] B. Liskov and J. Guttag. *Abstraction and specification in program development / Barbara Liskov and John Guttag.* MIT Press; McGraw-Hill, Cambridge, Mass. New York, 1986.

[65] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

[66] B. H. Liskov and J. M. Wing. *Behavioural subtyping using invariants and constraints.* Cambridge University Press, New York, NY, USA, 2001.

[67] C. Marché, C. P. Mohring, and X. Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004.

[68] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, second edition, 1997.

[69] C. Morgan. The specification statement. *ACM Trans. Program. Lang. Syst.*, 10(3):403–419, 1988.

[70] C. Morgan. *Programming from specifications (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994.

[71] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comput. Program.*, 9(3):287–306, 1987.

[72] J. M. Morris. Laws of data refinement. *Acta Informatica*, 26, 1989.

[73] J. M. Morris. Programs from specifications. In E. Dijkstra, editor, *Formal Development of Programs and Proofs*. Addison-Wesley, 1990.

[74] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer, 2002.

[75] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Program.*, 62(3):253–286, 2006.

[76] G. Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, Oct. 1989.

[77] M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 75–86, New York, NY, USA, 2008. ACM.

[78] M. Potet and Y. Rouzaud. Composition and refinement in the BMethod. In *B '98: Proceedings of the Second International B Conference on Recent Advances in the Development and Use of the B Method*, pages 46–65, London, UK, 1998. Springer-Verlag.

[79] S. Prehn and W. J. Toetenel, editors. *VDM '91 - Formal Software Development, 4th International Symposium of VDM Europe, Noordwijkerhout, The Netherlands, October 21-25, 1991, Proceedings, Volume 2: Tutorials*, volume 552 of *Lecture Notes in Computer Science*. Springer, 1991.

[80] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[81] Robby and P. Chalin. Preliminary design of a unified JML representation and software infrastructure. Technical Report TR2009-04-01, Kansas State University, July 2009.

[82] C. D. Ruby. Safely creating correct subclasses without seeing superclass code. In *OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-Oriented Programming, Systems, Languages, and Applications (Addendum)*, pages 155–156, New York, NY, USA, 2000. ACM.

[83] A. Rudich, Á. Darvas, and P. Müller. Checking well-formedness of pure-method specifications. In J. Cuellar and T. Maibaum, editors, *Formal Methods (FM)*, volume 5014 of *Lecture Notes in Computer Science*, pages 68–83. Springer-Verlag, 2008.

[84] J. Smans, B. Jacobs, and F. Piessens. VeriCool: An automatic verifier for a concurrent object-oriented language. In G. Barthe and F. S. de Boer, editors, *FMOODS*, volume 5051 of *Lecture Notes in Computer Science*, pages 220–239. Springer, 2008.

[85] J. Smans, B. Jacobs, F. Piessens, and W. Schulte. An automatic verifier for Java-like programs based on dynamic frames. In *Fundamental Approaches to Software Engineering*, volume 4961 of *Lecture Notes in Computer Science*, pages 261–275, Berlin, Apr. 2008. Springer-Verlag.

[86] J. Wing. Writing larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, Jan. 1987.

[87] J. Wing. A specifier's introduction to formal methods. *IEEE Computer*, pages 8–24, Sept. 1990.

[88] J. Woodcock. The Refinement Calculus. In Prehn and Toetenel [79], pages 80–95.

[89] J. Woodcock. A Tutorial on the Refinement Calculus. In Prehn and Toetenel [79], pages 79–140.

[90] J. Woodcock. Two Refinement Case Studies. In Prehn and Toetenel [79], pages 118–140.